LINUX API

ИСЧЕРПЫВАЮЩЕЕ РУКОВОДСТВО

МАЙКЛ КЕРРИСК

ПИТЕР®

no starch press

THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch
press

San Francisco

МАЙКЛ КЕРРИСК

LINUX API
ИСЧЕРПЫВАЮЩЕЕ РУКОВОДСТВО

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2018

ББК 32.973.2-018.2
УДК 004.451
К36

Керриск Майкл

К36 Linux API. Исчерпывающее руководство. — СПб.: Питер, 2018. — 1248 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02689-5

Linux Programming Interface — исчерпывающее руководство по программированию приложений для Linux и UNIX. Описанный здесь интерфейс применяется практически с любыми приложениями, работающими в операционных системах Linux или UNIX.

В этой авторитетной книге эксперт по Linux Майкл Керриск подробно описывает библиотечные вызовы и библиотечные функции, которые понадобятся вам при системном программировании. Вся теория сопровождается объяснениями на примерах четких и понятных полнофункциональных программ.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1593272203 англ.
ISBN 978-5-496-02689-5

© 2010 by Michael Kerrisk.
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Библиотека программиста», 2018

Краткое содержание

Предисловие **26**
Глава 1. История и стандарты **37**
Глава 2. Основные понятия **57**
Глава 3. Общее представление о системном программировании **79**
Глава 4. Файловый ввод-вывод: универсальная модель ввода-вывода **105**
Глава 5. Файловый ввод-вывод: дополнительные сведения **123**
Глава 6. Процессы **147**
Глава 7. Выделение памяти **172**
Глава 8. Пользователи и группы **186**
Глава 9. Идентификаторы процессов **200**
Глава 10. Время **220**
Глава 11. Системные ограничения и возможности **246**
Глава 12. Информация о системе и процессе **259**
Глава 13. Буферизация файлового ввода-вывода **268**
Глава 14. Файловые системы **287**
Глава 15. Атрибуты файла **315**
Глава 16. Расширенные атрибуты **346**
Глава 17. Списки контроля доступа **354**
Глава 18. Каталоги и ссылки **371**
Глава 19. Мониторинг событий файлов **406**
Глава 20. Сигналы: фундаментальные концепции **418**
Глава 21. Сигналы: обработчики сигналов **447**
Глава 22. Сигналы: дополнительные возможности **472**
Глава 23. Таймеры и переход в режим сна **502**
Глава 24. Создание процессов **533**
Глава 25. Завершение работы процесса **549**
Глава 26. Мониторинг дочерних процессов **557**
Глава 27. Выполнение программы **576**
Глава 28. Подробнее о создании процесса и выполнении программы **601**
Глава 29. Потоки выполнения: введение **627**
Глава 30. Потоки выполнения: синхронизация **641**
Глава 31. Потоки выполнения: потоковая безопасность и локальное хранилище **662**
Глава 32. Потоки выполнения: отмена потока **677**
Глава 33. Потоки выполнения: дальнейшие подробности **686**
Глава 34. Группы процессов, сессии и управление заданиями **702**
Глава 35. Приоритеты процессов и их планирование **733**
Глава 36. Ресурсы процессов **752**

Глава 37. Демоны **764**
Глава 38. Написание безопасных программ с повышенными привилегиями **780**
Глава 39. Система возможностей **793**
Глава 40. Учет входа в систему **811**
Глава 41. Основы разделяемых библиотек **825**
Глава 42. Продвинутые возможности разделяемых библиотек **849**
Глава 43. Краткий обзор межпроцессного взаимодействия **866**
Глава 44. Каналы и очереди FIFO **876**
Глава 45. Отображение в память **906**
Глава 46. Операции с виртуальной памятью **932**
Глава 47. Введение в межпроцессное взаимодействие стандарта POSIX **942**
Глава 48. Очереди сообщений стандарта POSIX **947**
Глава 49. Семафоры стандарта POSIX **969**
Глава 50. Разделяемая память POSIX **983**
Глава 51. Блокировка файлов **991**
Глава 52. Сокеты: введение **1021**
Глава 53. Сокеты: домен UNIX **1035**
Глава 54. Сокеты: основы сетей TCP/IP **1046**
Глава 55. Сокеты: домены сети Интернет **1062**
Глава 56. Сокеты: архитектура сервера **1095**
Глава 57. Сокеты: углубленный материал **1108**
Глава 58. Терминалы **1142**
Глава 59. Альтернативные модели ввода/вывода **1174**
Глава 60. Псевдотерминалы **1221**
Список используемых источников **1241**

Оглавление

Предисловие 26
Цель книги 26
Для кого эта книга 26
Linux и UNIX 27
Структура книги 27
Примеры программ 28
Упражнения 29
Стандарты и портируемость 29
Ядро Linux и версии библиотеки C 29
Использование программного интерфейса других языков программирования 30
Об авторе 30
Благодарности 31
Разрешения 36
Обратная связь 36

Глава 1. История и стандарты 37
1.1. Краткая история UNIX и языка C 37
1.2. Краткая история Linux 41
1.2.1. Проект GNU 41
1.2.2. Ядро Linux 42
1.3. Стандартизация 46
1.3.1. Язык программирования C 46
1.3.2. Первые стандарты POSIX 47
1.3.3. X/Open Company и Open Group 49
1.3.4. SUSv3 и POSIX.1-2001 49
1.3.5. SUSv4 и POSIX.1-2008 51
1.3.6. Этапы развития стандартов UNIX 52
1.3.7. Стандарты реализаций 53
1.3.8. Linux, стандарты и нормативная база Linux 54
1.4. Резюме 55

Глава 2. Основные понятия 57
2.1. Основа операционной системы: ядро 57
2.2. Оболочка 60
2.3. Пользователи и группы 61
2.4. Иерархия одного каталога. Что такое каталоги, ссылки и файлы 62
2.5. Модель файлового ввода-вывода 65
2.6. Программы 66
2.7. Процессы 67
2.8. Отображение в памяти 71
2.9. Статические и совместно используемые библиотеки 71
2.10. Межпроцессное взаимодействие и синхронизация 72
2.11. Сигналы 73
2.12. Потоки 74
2.13. Группы процессов и управление заданиями в оболочке 74
2.14. Сессии, управляющие терминалы и управляющие процессы 75
2.15. Псевдотерминалы 75
2.16. Дата и время 76
2.17. Клиент-серверная архитектура 76
2.18. Выполнение действий в реальном масштабе времени 77
2.19. Файловая система /proc 78
2.20. Резюме 78

Глава 3. Общее представление о системном программировании 79

3.1. Системные вызовы 79
3.2. Библиотечные функции 82
3.3. Стандартная библиотека языка C; GNU-библиотека C (glibc) 82
3.4. Обработка ошибок, возникающих при системных вызовах и вызовах библиотечных функций 84
3.5. Пояснения по поводу примеров программ, приводимых в книге 86
3.5.1. Ключи и аргументы командной строки 87
3.5.2. Типовые функции и заголовочные файлы 87
3.6. Вопросы переносимости 96
3.6.1. Макросы проверки возможностей 96
3.6.2. Типы системных данных 98
3.6.3. Прочие вопросы, связанные с портированием 102
3.7. Резюме 104
3.8. Упражнение 104

Глава 4. Файловый ввод-вывод: универсальная модель ввода-вывода 105

4.1. Общее представление 105
4.2. Универсальность ввода-вывода 107
4.3. Открытие файла: open() 108
4.3.1. Аргумент flags системного вызова open() 109
4.3.2. Ошибки, возвращаемые из системного вызова open() 113
4.3.3. Системный вызов creat() 114
4.4. Чтение из файла: read() 114
4.5. Запись в файл: write() 115
4.6. Закрытие файла: close() 116
4.7. Изменение файлового смещения: lseek() 117
4.8. Операции, не вписывающиеся в модель универсального ввода-вывода: ioctl() 121
4.9. Резюме 122
4.10. Упражнения 122

Глава 5. Файловый ввод-вывод: дополнительные сведения 123

5.1. Атомарность и состояние гонки 123
5.2. Операции управления файлом: fcntl() 126
5.3. Флаги состояния открытого файла 127
5.4. Связь файловых дескрипторов с открытыми файлами 128
5.5. Дублирование дескрипторов файлов 129
5.6. Файловый ввод-вывод по указанному смещению: pread() и pwrite() 133
5.7. Ввод-вывод по принципу фрагментации-дефрагментации: readv() и writev() 134
5.8. Усечение файла: truncate() и ftruncate() 137
5.9. Неблокирующий ввод-вывод 138
5.10. Ввод-вывод, осуществляемый в отношении больших файлов 138
5.11. Каталог /dev/fd 142
5.12. Создание временных файлов 143
5.13. Резюме 144
5.14. Упражнения 145

Глава 6. Процессы 147

6.1. Процессы и программы 147
6.2. Идентификатор процесса и идентификатор родительского процесса 148
6.3. Структура памяти процесса 149
6.4. Управление виртуальной памятью 153
6.5. Стек и стековые фреймы 155

6.6. Аргументы командной строки (argc, argv) ... 156
6.7. Список переменных среды ... 158
6.8. Выполнение нелокального перехода: setjmp() и longjmp() ... 165
6.9. Резюме ... 170
6.10. Упражнения ... 171

Глава 7. Выделение памяти ... 172

7.1. Выделение памяти в куче ... 172
7.1.1. Установка крайней точки программы: brk() и sbrk() ... 172
7.1.2. Выделение памяти в куче: malloc() и free() ... 173
7.1.3. Реализация функций malloc() и free() ... 177
7.1.4. Другие методы выделения памяти в куче ... 180
7.2. Выделение памяти в стеке: alloca() ... 183
7.3. Резюме ... 184
7.4. Упражнения ... 185

Глава 8. Пользователи и группы ... 186

8.1. Файл паролей: /etc/passwd ... 186
8.2. Теневой файл паролей: /etc/shadow ... 187
8.3. Файл групп: /etc/group ... 188
8.4. Извлечение информации о пользователях и группах ... 189
8.5. Шифрование пароля и аутентификация пользователя ... 195
8.6. Резюме ... 198
8.7. Упражнения ... 199

Глава 9. Идентификаторы процессов ... 200

9.1. Реальный идентификатор пользователя и реальный идентификатор группы ... 200
9.2. Действующий идентификатор пользователя и действующий идентификатор группы ... 200
9.3. Программы с установленным идентификатором пользователя и установленным идентификатором группы ... 201
9.4. Сохраненный set-user-ID и сохраненный set-group-ID ... 203
9.5. Пользовательские и групповые ID файловой системы ... 204
9.6. Дополнительные групповые идентификаторы ... 205
9.7. Извлечение и модификация идентификаторов процессов ... 205
9.7.1. Извлечение и изменение реальных, действующих и сохраненных установленных идентификаторов ... 206
9.7.2. Извлечение и изменение идентификаторов файловой системы ... 212
9.7.3. Извлечение и изменение дополнительных групповых идентификаторов ... 213
9.7.4. Сводный обзор вызовов, предназначенных для изменения идентификаторов процесса ... 214
9.7.5. Пример: вывод на экран идентификаторов процесса ... 216
9.8. Резюме ... 218
9.9. Упражнения ... 218

Глава 10. Время ... 220

10.1. Календарное время ... 220
10.2. Функции преобразования представлений времени ... 222
10.2.1. Преобразование значений типа time_t к виду, подходящему для устройств вывода информации ... 223
10.2.2. Преобразования между time_t и разделенным календарным временем ... 223
10.2.3. Преобразования между разделенным календарным временем и временем в печатном виде ... 225

10.3. Часовые пояса 232
10.4. Локали 235
10.5. Обновление системных часов 239
10.6. Программные часы (мгновения) 240
10.7. Время процесса 241
10.8. Резюме 244
10.9. Упражнение 245

Глава 11. Системные ограничения и возможности 246

11.1. Системные ограничения 247
11.2. Извлечение в ходе выполнения программы значений ограничений (и возможностей) системы 251
11.3. Извлечение в ходе выполнения программы значений ограничений (и возможностей), связанных с файлами 253
11.4. Неопределенные ограничения 254
11.5. Системные возможности 255
11.6. Резюме 257
11.7. Упражнения 258

Глава 12. Информация о системе и процессе 259

12.1. Файловая система /proc 259
12.1.1. Получение информации о процессе: /proc/PID 259
12.1.2. Системная информация, находящаяся в /proc 262
12.1.3 Доступ к файлам, находящимся в /proc 263
12.2. Идентификация системы: uname() 264
12.3. Резюме 266
12.4. Упражнения 267

Глава 13. Буферизация файлового ввода-вывода 268

13.1. Буферизация файлового ввода-вывода при работе в режиме ядра: буферная кэш-память 268
13.2. Буферизация в библиотеке stdio 272
13.3. Управление буферизацией файлового ввода-вывода, осуществляемой в ядре 274
13.4. Обзор буферизации ввода-вывода 278
13.5. Уведомление ядра о схемах ввода-вывода 279
13.6. Обход буферной кэш-памяти: непосредственный ввод/вывод 281
13.7. Смешивание библиотечных функций и системных вызовов для файлового ввода-вывода 284
13.8. Резюме 285
13.9. Упражнения 285

Глава 14. Файловые системы 287

14.1. Специальные файлы устройств 287
14.2. Диски и разделы 289
14.3. Файловые системы 290
14.4. Индексные дескрипторы 292
14.5. Виртуальная файловая система 294
14.6. Журналируемые файловые системы 295
14.7. Иерархия одиночного каталога и точки монтирования 297
14.8. Монтирование и размонтирование файловых систем 298
14.8.1. Монтирование файловой системы: mount() 299
14.8.2. Размонтирование файловой системы: системные вызовы umount() и umount2() 305

14.9. Дополнительные функции монтирования ... 306
14.9.1. Монтирование файловой системы в нескольких точках монтирования ... 306
14.9.2. Создание стека монтирования в одной точке ... 307
14.9.3. Флаги монтирования, которые являются параметрами конкретной точки монтирования ... 307
14.9.4. Связанные (синонимичные) точки монтирования ... 308
14.9.5. Рекурсивное связанное монтирование ... 309
14.10. Файловая система виртуальной памяти: tmpfs ... 310
14.11. Получение информации о файловой системе: statvfs() ... 311
14.12. Резюме ... 313
14.13. Упражнение ... 314

Глава 15. Атрибуты файла ... 315

15.1. Извлечение информации о файле: stat() ... 315
15.2. Файловые метки времени ... 320
15.2.1. Изменение меток времени файла с помощью системных вызовов utime() и utimes() ... 323
15.2.2. Изменение меток времени файла с помощью системного вызова utimensat() и функции futimens() ... 325
15.3. Принадлежность файла ... 326
15.3.1. Принадлежность новых файлов ... 327
15.3.2. Изменение принадлежности файла: системные вызовы chown(), fchown() и lchown() ... 327
15.4. Права доступа к файлу ... 330
15.4.1. Права доступа к обычным файлам ... 330
15.4.2. Права доступа к каталогам ... 332
15.4.3. Алгоритм проверки прав доступа ... 333
15.4.4. Проверка доступности файла: системный вызов access() ... 335
15.4.5. Биты set-user-ID, set-group-ID и закрепляющий ... 336
15.4.6. Маска режима создания файла процесса: umask() ... 337
15.4.7. Изменение прав доступа к файлу: системные вызовы chmod() и fchmod() ... 339
15.5. Флаги индексного дескриптора (расширенные атрибуты файла в файловой системе ext2) ... 340
15.6. Резюме ... 344
15.7. Упражнения ... 344

Глава 16. Расширенные атрибуты ... 346

16.1. Обзор ... 346
16.2. Подробности реализации расширенных атрибутов ... 348
16.3. Системные вызовы для манипуляции расширенными атрибутами ... 349
16.4. Резюме ... 353
16.5. Упражнение ... 353

Глава 17. Списки контроля доступа ... 354

17.1. Обзор ... 354
17.2. Алгоритм проверки прав доступа с помощью списков контроля доступа ... 356
17.3. Длинная и краткая текстовые формы списков контроля доступа ... 357
17.4. Запись ACL_MASK и класс группы для ACL-списка ... 358
17.5. Команды getfacl и setfacl ... 359
17.6. ACL-списки по умолчанию и создание файла ... 361
17.7. Границы реализации списка контроля доступа ... 362
17.8. API для ACL-списков ... 363

17.9. Резюме 370
17.10. Упражнение 370

Глава 18. Каталоги и ссылки 371

18.1. Каталоги и (жесткие) ссылки 371
18.2. Символические (мягкие) ссылки 374
18.3. Создание и удаление (жестких) ссылок: системные вызовы link() и unlink() 377
18.4. Изменение имени файла: системный вызов rename() 380
18.5. Работа с символическими ссылками: системные вызовы symlink() и readlink() 381
18.6. Создание и удаление каталогов: системные вызовы mkdir() и rmdir() 382
18.7. Удаление файла или каталога: функция remove() 384
18.8. Чтение каталогов: функции opendir() и readdir() 384
18.9. Обход дерева файлов: функция nftw() 390
18.10. Текущий рабочий каталог процесса 394
18.11. Работа с использованием файлового дескриптора каталога 396
18.12. Изменение корневого каталога процесса: системный вызов chroot() 398
18.13. Анализ имени пути: функция realpath() 400
18.14. Синтаксический разбор строк с именем пути: функции dirname() и basename() 402
18.15. Резюме 404
18.16. Упражнения 404

Глава 19. Мониторинг событий файлов 406

19.1. Обзор 406
19.2. Интерфейс inotify 407
19.3. События inotify 409
19.4. Чтение событий inotify 410
19.5. Ограничения очереди и файлы /proc 416
19.6. Старая система мониторинга событий файлов: dnotify 416
19.7. Резюме 417
19.8. Упражнение 417

Глава 20. Сигналы: фундаментальные концепции 418

20.1. Концепции и общие сведения 418
20.2. Типы сигналов и действия по умолчанию 420
20.3. Изменение диспозиций сигналов: signal() 426
20.4. Введение в обработчики сигналов 427
20.5. Отправка сигналов: kill() 430
20.6. Проверка существования процесса 432
20.7. Другие способы отправки сигналов: raise() и killpg() 433
20.8. Отображение описаний сигналов 434
20.9. Наборы сигналов 435
20.10. Сигнальная маска (блокирование доставки сигналов) 438
20.11. Ожидающие сигналы 439
20.12. Сигналы не ставятся в очередь 440
20.13. Изменение диспозиций сигналов: sigaction() 443
20.14. Ожидание сигнала: pause() 445
20.15. Резюме 445
20.16. Упражнения 446

Глава 21. Сигналы: обработчики сигналов 447

21.1. Проектирование обработчиков сигналов 447
21.1.1. Сигналы не ставятся в очередь (еще раз о...) 447
21.1.2. Реентерабельные функции и функции, безопасные для асинхронных сигналов 448

21.1.3. Глобальные переменные и тип данных sig_atomic_t ... 453
21.2. Другие методы завершения работы обработчика сигнала ... 454
21.2.1. Выполнение нелокального перехода из обработчика сигнала ... 454
21.2.2. Аварийное завершение процесса: abort() ... 458
21.3. Обработка сигнала на альтернативном стеке: signalstack() ... 459
21.4. Флаг SA_SIGINFO ... 462
21.5. Прерывание и повторный запуск системных вызовов ... 467
21.6. Резюме ... 470
21.7. Упражнение ... 471

Глава 22. Сигналы: дополнительные возможности ... 472

22.1. Файлы дампа ядра ... 472
22.2. Частные случаи доставки, диспозиции и обработки ... 474
22.3. Прерываемые и непрерываемые состояния сна процесса ... 475
22.4. Аппаратно генерируемые сигналы ... 476
22.5. Синхронная и асинхронная генерация сигнала ... 477
22.6. Тайминг и порядок доставки сигнала ... 478
22.7. Реализация и переносимость функции signal() ... 479
22.8. Сигналы реального времени ... 481
22.8.1. Отправка сигналов реального времени ... 483
22.8.2. Обработка сигналов реального времени ... 485
22.9. Ожидание сигнала с использованием маски: sigsuspend() ... 488
22.10. Синхронное ожидание сигнала ... 492
22.11. Получение сигналов через файловый дескриптор ... 496
22.12. Межпроцессное взаимодействие посредством сигналов ... 498
22.13. Ранние API сигналов ... 499
22.14. Резюме ... 500
22.15. Упражнения ... 501

Глава 23. Таймеры и переход в режим сна ... 502

23.1. Интервальные таймеры ... 502
23.2. Планирование и точность таймеров ... 507
23.3. Установка времени ожидания для блокирующих операций ... 508
23.4. Приостановка выполнения на определенный отрезок времени (переход в режим сна) ... 509
23.4.1. Переход в режим сна (низкая точность): вызов sleep() ... 509
23.4.2. Переход в режим сна (высокая точность): вызов nanosleep() ... 510
23.5. Часы стандарта POSIX ... 512
23.5.1. Получение текущего значения часов: вызов clock_gettime() ... 513
23.5.2. Изменение значения часов: вызов clock_settime() ... 514
23.5.3. Получение идентификатора часов для определенного процесса или потока ... 514
23.5.4. Улучшенный переход в режим сна (высокая точность): вызов clock_nanosleep() ... 515
23.6. Интервальные таймеры POSIX ... 516
23.6.1. Создание таймера: вызов timer_create() ... 517
23.6.2. Запуск и остановка таймера: вызов timer_settime() ... 519
23.6.3. Получение текущего значения таймера: вызов timer_gettime() ... 520
23.6.4. Удаление таймера: вызов timer_delete() ... 520
23.6.5. Уведомление с помощью сигнала ... 521
23.6.6. Дополнительные срабатывания таймера ... 524
23.6.7. Уведомление с помощью потока ... 525
23.7. Таймеры, которые уведомляют с помощью файловых дескрипторов: интерфейс timerfd ... 528

23.8. Резюме 532
23.9. Упражнения 532

Глава 24. Создание процессов 533

24.1. Обзор вызовов fork(), exit(), wait() и execve() 533
24.2. Создание нового процесса: fork() 534
24.2.1. Совместный доступ к файлу родителя и потомка 537
24.2.2. Семантика памяти вызова fork() 540
24.3. Системный вызов vfork() 542
24.4. Состояние гонки после вызова fork() 544
24.5. Синхронизация с помощью сигналов как способ избежать состояния гонки 546
24.6. Резюме 548
24.7. Упражнения 548

Глава 25. Завершение работы процесса 549

25.1. Завершение процесса: вызовы _exit() и exit() 549
25.2. Завершение процесса в подробностях 550
25.3. Обработчики выхода 551
25.4. Взаимодействие между буферами stdio и вызовами fork() и _exit() 554
25.5. Резюме 556
25.6. Упражнение 556

Глава 26. Мониторинг дочерних процессов 557

26.1. Ожидание дочернего процесса 557
26.1.1. Системный вызов wait() 557
26.1.2. Системный вызов waitpid() 559
26.1.3. Статус ожидания 560
26.1.4. Завершение процесса из обработчика сигнала 564
26.1.5. Системный вызов waitid() 565
26.1.6. Системные вызовы wait3() и wait4() 567
26.2. Процессы-«сироты» и процессы-«зомби» 567
26.3. Сигнал SIGCHLD 569
26.3.1. Установка обработчика сигнала SIGCHLD 570
26.3.2. Доставка сигнала SIGCHLD для остановленных потомков 573
26.3.3. Игнорирование завершенных дочерних процессов 573
26.4. Резюме 574
26.5. Упражнения 575

Глава 27. Выполнение программы 576

27.1. Выполнение новой программы: execve() 576
27.2. Библиотечные функции семейства exec() 579
27.2.1. Переменная среды PATH 580
27.2.2. Задание аргументов программы в виде списка 582
27.2.3. Передача переменных среды вызывающего процесса новой программе 582
27.2.4. Выполнение файла через ссылку на его дескриптор: fexecve() 583
27.3. Интерпретируемые скрипты 583
27.4. Дескрипторы файлов и вызовы exec() 587
27.5. Сигналы и вызов exec() 589
27.6. Выполнение консольных команд: system() 590
27.7. Реализация функции system() 593
27.8. Резюме 599
27.9. Упражнения 599

Глава 28. Подробнее о создании процесса и выполнении программы 601
28.1. Учет ресурсов, используемых процессом 601
28.2. Системный вызов clone() 607
28.2.1. Аргумент flags вызова clone() 612
28.2.2. Расширения к вызову waitpid() для клонированных потомков 619
28.3. Скорость создания процессов 619
28.4. Влияние вызовов exec() и fork() на атрибуты процесса 621
28.5. Резюме 625
28.6. Упражнение 626

Глава 29. Потоки выполнения: введение 627
29.1. Краткий обзор 627
29.2. Общие сведения о программном интерфейсе Pthreads 630
29.3. Создание потоков 631
29.4. Завершение потоков 633
29.5. Идентификаторы потоков 633
29.6. Присоединение к завершенному потоку 635
29.7. Отсоединение потока 637
29.8. Атрибуты потоков 637
29.9. Сравнение потоков и процессов 638
29.10. Резюме 639
29.11. Упражнения 640

Глава 30. Потоки выполнения: синхронизация 641
30.1. Защита доступа к разделяемым переменным: мьютексы 641
30.1.1. Статически выделяемые мьютексы 645
30.1.2. Закрытие и открытие мьютекса 645
30.1.3. Производительность мьютексов 647
30.1.4. Взаимное блокирование мьютексов 648
30.1.5. Динамическая инициализация мьютексов 649
30.1.6. Атрибуты мьютексов 650
30.1.7. Типы мьютексов 650
30.2. Оповещение об изменении состояния: условные переменные 651
30.2.1. Статически выделяемые условные переменные 652
30.2.2. Оповещение и ожидание условных переменных 652
30.2.3. Проверка предиката условной переменной 656
30.2.4. Пример программы: подсоединение любого завершенного потока 657
30.2.5. Динамически выделяемые условные переменные 660
30.3. Резюме 661
30.4. Упражнения 661

Глава 31. Потоки выполнения: потоковая безопасность и локальное хранилище 662
31.1. Потоковая безопасность (и новый взгляд на реентерабельность) 662
31.2. Единовременная инициализация 665
31.3. Данные уровня потока 666
31.3.1. Данные уровня потока с точки зрения библиотечной функции 666
31.3.2. Обзор программного интерфейса для работы с данными уровня потока 667
31.3.3. Подробности о программном интерфейсе для работы с данными уровня потока 667
31.3.4. Использование программного интерфейса для работы с данными уровня потока 670
31.3.5. Ограничения реализации данных уровня потока 674

31.4. Локальное хранилище потока ... 674
31.5. Резюме ... 676
31.6. Упражнения ... 676

Глава 32. Потоки выполнения: отмена потока ... 677

32.1. Отмена потока ... 677
32.2. Состояние и тип отмены ... 677
32.3. Точки отмены ... 678
32.4. Проверка возможности отмены потока ... 681
32.5. Обработчики, освобождающие ресурсы ... 681
32.6. Асинхронная отмена ... 685
32.7. Резюме ... 685

Глава 33. Потоки выполнения: дальнейшие подробности ... 686

33.1. Стеки потоков ... 686
33.2. Потоки и сигналы ... 687
33.2.1. Как модель сигналов в UNIX соотносится с потоками ... 687
33.2.2. Изменение масок сигналов потока ... 688
33.2.3. Отправка сигнала потоку ... 689
33.2.4. Разумная обработка асинхронных сигналов ... 689
33.3. Потоки и управление процессами ... 690
33.4. Модели реализации потоков ... 692
33.5. Разные реализации POSIX-потоков в Linux ... 693
33.5.1. LinuxThreads ... 694
33.5.2. Библиотека NPTL ... 696
33.5.3. Выбор между разными реализациями многопоточности ... 698
33.6. Продвинутые возможности программного интерфейса Pthreads ... 700
33.7. Резюме ... 700
33.8. Упражнения ... 701

Глава 34. Группы процессов, сессии и управление заданиями ... 702

34.1. Краткий обзор ... 702
34.2. Группы процессов ... 703
34.3. Сессии ... 707
34.4. Контролирующие терминалы и контролирующие процессы ... 708
34.5. Активные и фоновые группы процессов ... 710
34.6. Сигнал SIGHUP ... 711
34.6.1. Обработка сигнала SIGHUP командной оболочкой ... 712
34.6.2. Сигнал SIGHUP и завершение контролирующего процесса ... 714
34.7. Управление заданиями ... 716
34.7.1. Управление заданиями в рамках командной оболочки ... 716
34.7.2. Реализация управления заданиями ... 718
34.7.3. Обрабатываем сигналы, связанные с управлением заданиями ... 723
34.7.4. Осиротевшие группы процессов (и новый взгляд на сигнал SIGHUP) ... 727
34.8. Резюме ... 731
34.9. Упражнения ... 732

Глава 35. Приоритеты процессов и их планирование ... 733

35.1. Приоритеты процессов (значение nice) ... 733
35.2. Обзор планирования в режиме реального времени ... 736
35.2.1. Политика SCHED_RR ... 738
35.2.2. Политика SCHED_FIFO ... 739
35.2.3. Политики SCHED_BATCH и SCHED_IDLE ... 739

35.3. Программный интерфейс планирования в режиме реального времени 739
35.3.1. Диапазон приоритетов реального времени....740
35.3.2. Изменение и получение политик и приоритетов 740
35.3.3. Освобождение ресурсов процессора 746
35.3.4. Временной отрезок в политике SCHED_RR....746
35.4. Привязка к процессору 747
35.5. Резюме 749
35.6. Упражнения....750

Глава 36. Ресурсы процессов....752

36.1. Ресурсы, использующиеся процессом....752
36.2. Ограничения на ресурсы для отдельных процессов....754
36.3. Подробности об отдельных ограничениях на ресурсы....759
36.4. Резюме 763
36.5. Упражнения....763

Глава 37. Демоны 764

37.1. Краткий обзор 764
37.2. Создание демона 764
37.3. Рекомендации по написанию демонов....768
37.4. Использование сигнала SIGHUP для повторной инициализации демона....769
37.5. Запись в журнал сообщений и ошибок с помощью системы syslog....771
37.5.1. Краткий обзор 772
37.5.2. Программный интерфейс syslog 773
37.5.3. Файл /etc/syslog.conf....778
37.6. Резюме 779
37.7. Упражнение 779

Глава 38. Написание безопасных программ с повышенными привилегиями 780

38.1. Нужно ли программе устанавливать идентификаторы пользователя или группы?....780
38.2. Работайте с минимальными привилегиями....781
38.3. Будьте осторожны при выполнении программы....783
38.4. Избегайте раскрытия деликатной информации 785
38.5. Изоляция процесса....785
38.6. Не забывайте о сигналах в состоянии гонки....786
38.7. Подводные камни, связанные с файловыми операциями и вводом/выводом файлов 787
38.8. Не доверяйте внешнему вводу или среде выполнения 788
38.9. Остерегайтесь переполнений буфера 789
38.10. Остерегайтесь DoS-атак....790
38.11. Проверяйте результаты выполнения и предусматривайте безопасное завершение в случае неудачи....791
38.12. Резюме....791
38.13. Упражнения....792

Глава 39. Система возможностей 793

39.1. Зачем нужна система возможностей....793
39.2. Система возможностей в Linux....794
39.3. Возможности, связанные с процессами и файлами....794
39.3.1. Возможности процесса....794
39.3.2. Возможности файлов....795
39.3.3. Назначение разрешенных и действующих возможностей процесса 798

39.3.4. Для чего нужны разрешенные и действующие возможности файла 799
39.3.5. Для чего нужны наследуемые возможности процесса и файла 799
39.3.6. Назначение и просмотр возможностей файла из командной оболочки 800
39.4. Современная реализация системы возможностей 801
39.5. Изменение возможностей процесса во время выполнения exec() 801
39.5.1. Ограничивающий набор возможностей 802
39.5.2. Сохранение традиционной для администратора семантики 802
39.6. Как изменение пользовательского идентификатора влияет на возможности процесса 803
39.7. Программное изменение возможностей процесса 803
39.8. Создание среды, в которой возможности являются единственным механизмом повышения привилегий 807
39.9. Определение возможностей, которые требуются программе 810
39.10. Резюме 810
39.11. Упражнение 810

Глава 40. Учет входа в систему 811

40.1. Краткий обзор файлов utmp и wtmp 811
40.2. Программный интерфейс utmpx 811
40.3. Структура utmpx 812
40.4. Извлечение информации из файлов utmp и wtmp 814
40.5. Получение имени текущего пользователя: getlogin() 817
40.6. Запись в файлы utmp и wtmp данных о пребывании в системе 818
40.7. Файл lastlog 822
40.8. Резюме 824
40.9. Упражнения 824

Глава 41. Основы разделяемых библиотек 825

41.1. Библиотека объектов 825
41.2. Статические библиотеки 826
41.3. Краткий обзор разделяемых библиотек 827
41.4. Создание и использование разделяемых библиотек. Первые шаги 828
41.4.1. Создание разделяемой библиотеки 829
41.4.2. Адресно-независимый код 829
41.4.3. Использование статической библиотеки 830
41.4.4. Разделяемые библиотеки и имя soname 832
41.5. Полезные инструменты для работы с разделяемыми библиотеками 834
41.6. Версии и соглашение об именовании разделяемых библиотек 835
41.7. Установка разделяемых библиотек 838
41.8. Совместимые и несовместимые библиотеки 841
41.9. Обновления разделяемых библиотек 841
41.10. Задание каталогов для поиска библиотеки в объектном файле 842
41.11. Поиск разделяемых библиотек на этапе выполнения 845
41.12. Разрешение символов на этапе выполнения 845
41.13. Использование статической библиотеки вместо динамической 847
41.14. Резюме 847
41.15. Упражнение 848

Глава 42. Продвинутые возможности разделяемых библиотек 849

42.1. Динамически загружаемые библиотеки 849
42.1.1. Открытие разделяемой библиотеки: dlopen() 850
42.1.2. Анализ ошибок: dlerror() 852
42.1.3. Получение адреса символа: dlsym() 852

42.1.4. Закрытие разделяемой библиотеки: dlclose() ... 855
42.1.5. Получение информации о загруженных символах: dladdr() ... 855
42.1.6. Получение доступа к символам главной программы ... 856
42.2. Управление видимостью символов ... 856
42.3. Версионные сценарии компоновщика ... 857
42.3.1. Управление видимостью символов с помощью версионных сценариев ... 858
42.3.2. Версионирование символов ... 859
42.4. Инициализация и финализация функций ... 861
42.5. Предварительная загрузка разделяемых библиотек ... 862
42.6. Мониторинг работы динамического компоновщика: LD_DEBUG ... 863
42.7. Резюме ... 864
42.8. Упражнения ... 865

Глава 43. Краткий обзор межпроцессного взаимодействия ... 866

43.1. Классификация IPC-механизмов ... 866
43.2. Средства взаимодействия ... 866
43.3. Средства синхронизации ... 869
43.4. Сравнение IPC-механизмов ... 870
43.5. Резюме ... 875
43.6. Упражнения ... 875

Глава 44. Каналы и очереди FIFO ... 876

44.1. Краткий обзор ... 876
44.2. Создание и использование каналов ... 878
44.3. Каналы как средство синхронизации процессов ... 883
44.4. Использование каналов для соединения фильтров ... 885
44.5. Взаимодействие с консольными командами с помощью канала: popen() ... 888
44.6. Каналы и буферизация стандартного ввода/вывода ... 891
44.7. Очереди FIFO ... 892
44.8. Клиент-серверные приложения на основе очередей FIFO ... 894
44.9. Неблокирующий ввод/вывод ... 901
44.10. Семантика вызовов read() и write() в контексте каналов и очередей FIFO ... 903
44.11. Резюме ... 904
44.12. Упражнения ... 905

Глава 45. Отображение в память ... 906

45.1. Краткий обзор ... 906
45.2. Создание отображения: mmap() ... 908
45.3. Удаление отображения с участка памяти: munmap() ... 911
45.4. Отображение файлов ... 912
45.4.1. Приватные файловые отображения ... 913
45.4.2. Разделяемые файловые отображения ... 914
45.4.3. Крайние случаи ... 917
45.4.4. Взаимодействие защиты памяти и режима доступа к памяти ... 919
45.5. Синхронизация отображенного участка памяти: msync() ... 919
45.6. Дополнительные флаги вызова mmap() ... 920
45.7. Анонимные отображения ... 922
45.8. Изменение отображенного участка памяти: mremap() ... 924
45.9. Флаг MAP_NORESERVE и перерасход пространства подкачки ... 925
45.10. Флаг MAP_FIXED ... 927
45.11. Нелинейные отображения: remap_file_pages() ... 928
45.12. Резюме ... 930
45.13. Упражнения ... 931

Глава 46. Операции с виртуальной памятью 932
46.1. Изменение защиты памяти: mprotect() 932
46.2. Блокирование памяти: mlock() и mlockall() 934
46.3. Определение местонахождения памяти: mincore() 937
46.4. Предсказание модели использования памяти в будущем: madvise() 940
46.5. Резюме 941
46.6. Упражнения 941

Глава 47. Введение в межпроцессное взаимодействие стандарта POSIX 942
47.1. Краткий обзор программных интерфейсов 942
47.2. Резюме 946

Глава 48. Очереди сообщений стандарта POSIX 947
48.1. Краткий обзор 947
48.2. Открытие, закрытие и удаление очереди сообщений 948
48.3. Связь между дескрипторами и очередями сообщений 950
48.4. Атрибуты очередей сообщений 951
48.5. Обмен сообщениями 956
48.5.1. Отправка сообщений 956
48.5.2. Получение сообщений 957
48.5.3. Отправка и получение сообщений с ограниченным временем ожидания 959
48.6. Оповещение о сообщении 960
48.6.1. Получение оповещения в виде сигнала 962
48.6.2. Получение уведомлений в отдельном потоке 964
48.7. Возможности, характерные для Linux 965
48.8. Ограничения, относящиеся к очередям сообщений 967
48.9. Резюме 968
48.10. Упражнения 968

Глава 49. Семафоры стандарта POSIX 969
49.1. Краткий обзор 969
49.2. Именованные семафоры 969
49.2.1. Открытие именованного семафора 970
49.2.2. Закрытие семафора 972
49.2.3. Удаление именованного семафора 972
49.3. Операции с семафорами 973
49.3.1. Декрементация семафора 973
49.3.2. Инкрементация семафора 975
49.3.3. Получение текущего значения семафора 976
49.4. Анонимные семафоры 977
49.4.1. Инициализация анонимного семафора 978
49.4.2. Уничтожение анонимного семафора 981
49.5. Сравнение POSIX-семафоров с мьютексами из библиотеки Pthreads 981
49.6. Ограничения, связанные с семафорами 982
49.7. Резюме 982
49.8. Упражнение 982

Глава 50. Разделяемая память POSIX 983
50.1. Краткий обзор 983
50.2. Создание объектов разделяемой памяти 984
50.3. Использование объектов разделяемой памяти 986
50.4. Удаление объектов разделяемой памяти 989
50.5. Сравнение программных интерфейсов для работы с разделяемой памятью 989
50.6. Резюме 990

Глава 51. Блокировка файлов 991
51.1. Краткий обзор 991
51.2. Блокировка файла с помощью вызова flock() 993
51.2.1. Семантика наследования и снятия блокировок 996
51.2.2. Ограничения вызова flock() 997
51.3. Блокировка записей с помощью вызова fcntl() 997
51.3.1. Структура flock 999
51.3.2. Аргумент cmd 1000
51.3.3. Подробности об установке и снятии блокировок 1001
51.3.4. Взаимная блокировка 1002
51.3.5. Пример: программа для интерактивной блокировки 1002
51.3.6. Пример: библиотека функций для установки блокировок 1006
51.3.7. Производительность блокировок и их ограничения 1008
51.3.8. Семантика наследования и снятия блокировок 1009
51.3.9. Зависание блокировок и приоритет отложенных запросов на их получение 1010
51.4. Строгая блокировка 1011
51.5. Файл /proc/locks 1014
51.6. Выполнение только одного экземпляра программы 1015
51.7. Устаревшие способы блокировки 1017
51.8. Резюме 1019
51.9. Упражнения 1019

Глава 52. Сокеты: введение 1021
52.1. Краткий обзор 1021
52.2. Создание сокета: socket() 1024
52.3. Привязывание сокета к адресу: bind() 1025
52.4. Универсальные структуры для хранения адресов сокетов: struct sockaddr 1025
52.5. Потоковые сокеты 1026
52.5.1. Ожидание входящих соединений: listen() 1028
52.5.2. Прием соединения: accept() 1029
52.5.3. Соединение с удаленным сокетом: connect() 1029
52.5.4. Операции ввода/вывода с потоковыми сокетами 1030
52.5.5. Закрытие соединения: close() 1030
52.6. Датаграммные сокеты 1031
52.6.1. Обмен датаграммами: recvfrom() и sendto() 1031
52.6.2. Использование вызова connect() в сочетании с датаграммными сокетами 1033
52.7. Резюме 1033

Глава 53. Сокеты: домен UNIX 1035
53.1. Адреса сокетов в домене UNIX: struct sockaddr_un 1035
53.2. Потоковые сокеты в домене UNIX 1037
53.3. Датаграммные сокеты в домене UNIX 1040
53.4. Права доступа к сокетам домена UNIX 1043
53.5. Создание соединенной пары сокетов: socketpair() 1043
53.6. Абстрактное пространство имен сокетов в Linux 1044
53.7. Резюме 1045
53.8. Упражнения 1045

Глава 54. Сокеты: основы сетей TCP/IP 1046
54.1. Интерсети 1046
54.2. Сетевые протоколы и уровни 1047
54.3. Канальный уровень 1050

54.4. Сетевой уровень: IP ... 1050
54.5. IP-адреса ... 1052
54.6. Транспортный уровень ... 1054
54.6.1. Номера портов ... 1055
54.6.2. Протокол пользовательских датаграмм (UDP) ... 1056
54.6.3. Протокол управления передачей (TCP) ... 1056
54.7. Документы, выносимые на рассмотрение (RFC) ... 1060
54.8. Резюме ... 1060

Глава 55. Сокеты: домены сети Интернет ... 1062

55.1. Сокеты интернет-домена ... 1062
55.2. Порядок байтов в сети ... 1062
55.3. Представление данных ... 1064
55.4. Адреса интернет-сокетов ... 1066
55.5. Краткий обзор функций для преобразования сетевых адресов и имен служб ... 1068
55.6. Функции inet_pton() и inet_ntop() ... 1070
55.7. Пример клиент-серверного приложения (на основе датаграммных сокетов) ... 1071
55.8. Система доменных имен (DNS) ... 1073
55.9. Файл /etc/services ... 1076
55.10. Преобразование имен узлов и служб, не зависящее от протокола ... 1077
55.10.1. Функция getaddrinfo() ... 1077
55.10.2. Удаление списков со структурами addrinfo: freeaddrinfo() ... 1080
55.10.3. Выявление ошибок: gai_strerror() ... 1080
55.10.4. Функция getnameinfo() ... 1081
55.11. Пример клиент-серверного приложения на основе потоковых сокетов ... 1082
55.12. Библиотека для работы с сокетами интернет-домена ... 1088
55.13. Сравнение сокетов в UNIX- и интернет-доменах ... 1092
55.14. Дополнительная информация ... 1093
55.15. Резюме ... 1093
55.16. Упражнения ... 1094

Глава 56. Сокеты: архитектура сервера ... 1095

56.1. Итерационные и параллельные серверы ... 1095
56.2. Итерационный UDP-сервер echo ... 1095
56.3. Параллельный TCP-сервер echo ... 1098
56.4. Другие разновидности архитектуры параллельного сервера ... 1100
56.5. Демон inetd ... 1102
56.6. Резюме ... 1106
56.7. Упражнения ... 1107

Глава 57. Сокеты: углубленный материал ... 1108

57.1. Частичное чтение и запись в контексте потоковых сокетов ... 1108
57.2. Системный вызов shutdown() ... 1110
57.3. Специальные системные вызовы для работы с сокетами: recv() и send() ... 1113
57.4. Системный вызов sendfile() ... 1114
57.5. Получение адреса сокета ... 1117
57.6. Подробности реализации протокола TCP ... 1119
57.6.1. Формат TCP-сегментов ... 1119
57.6.2. Порядковые номера и подтверждения в протоколе TCP ... 1121
57.6.3. Машина состояний и диаграмма перехода состояний в протоколе TCP ... 1122
57.6.4. Установка TCP-соединения ... 1124
57.6.5. Разрыв TCP-соединения ... 1125

57.6.6. Вызов shutdown() для TCP-сокета ... 1126
57.6.7. Состояние TIME_WAIT ... 1127
57.7. Мониторинг сокетов: утилита netstat ... 1128
57.8. Мониторинг данных, проходящих по протоколу TCP, с помощью утилиты tcpdump ... 1130
57.9. Параметры сокета ... 1131
57.10. Параметр сокета SO_REUSEADDR ... 1132
57.11. Наследование флагов и параметров сокета при выполнении вызова accept() ... 1134
57.12. Выбор между TCP и UDP ... 1135
57.13. Продвинутые возможности ... 1136
57.13.1. Внеканальные данные ... 1136
57.13.2. Системные вызовы sendmsg() и recvmsg() ... 1136
57.13.3. Передача файловых дескрипторов ... 1137
57.13.4. Получение учетных данных отправителя ... 1137
57.13.5. Последовательный обмен пакетами ... 1137
57.13.6. Протоколы транспортного уровня SCTP и DCCP ... 1138
57.14. Резюме ... 1139
57.15. Упражнения ... 1140

Глава 58. Терминалы ... 1142

58.1. Краткий обзор ... 1143
58.2. Извлечение и изменение атрибутов терминала ... 1143
58.3. Команда stty ... 1146
58.4. Специальные символы терминала ... 1148
58.5. Флаги терминала ... 1153
58.6. Режимы ввода/вывода терминала ... 1159
58.6.1. Канонический режим ... 1159
58.6.2. Неканонический режим ... 1159
58.6.3. Режимы с обработкой, без обработки и cbreak ... 1161
58.7. Скорость передачи данных в терминале ... 1167
58.8. Управление последовательным портом ... 1169
58.9. Размер окна терминала ... 1170
58.10. Идентификация терминала ... 1172
58.11. Резюме ... 1172
58.12. Упражнения ... 1173

Глава 59. Альтернативные модели ввода/вывода ... 1174

59.1. Краткий обзор ... 1174
59.1.1. Уведомления, срабатывающие по уровню или фронту ... 1177
59.1.2. Применение неблокирующего режима в сочетании с альтернативными моделями ввода/вывода ... 1178
59.2. Мультиплексирование ввода/вывода ... 1179
59.2.1. Системный вызов select() ... 1179
59.2.2. Системный вызов poll() ... 1185
59.2.3. Условия готовности файлового дескриптора ... 1189
59.2.4. Сравнение вызовов select() и poll() ... 1192
59.2.5. Проблемы, присущие вызовам select() и poll() ... 1193
59.3. Ввод/вывод на основе сигналов ... 1194
59.3.1. Установка владельца файлового дескриптора ... 1196
59.3.2. Когда генерируется сигнал о возможности ввода/вывода? ... 1198
59.3.3. Эффективное использование ввода/вывода на основе сигналов ... 1199
59.4. Программный интерфейс epoll ... 1201

59.4.1. Создание экземпляра epoll: вызов epoll_create() ... 1202
59.4.2. Редактирование списка интереса epoll: вызов epoll_ctl() ... 1203
59.4.3. Ожидание событий: вызов epoll_wait() ... 1204
59.4.4. Подробности семантики интерфейса epoll ... 1209
59.4.5. Производительность интерфейса epoll по сравнению с мультиплексированным вводом/выводом ... 1211
59.4.6. Уведомления, срабатывающие по фронту ... 1212
59.5. Ожидание сигналов и готовности файловых дескрипторов ... 1213
59.5.1. Системный вызов pselect() ... 1214
59.5.2. Трюк с зацикленным каналом ... 1216
59.6. Резюме ... 1218
59.7. Упражнения ... 1220

Глава 60. Псевдотерминалы ... 1221

60.1. Краткий обзор ... 1221
60.2. Псевдотерминалы стандарта UNIX 98 ... 1225
60.2.1. Открытие неиспользуемого первичного устройства: вызов posix_openpt() ... 1225
60.2.2. Изменение владельца и прав доступа к вторичному устройству: вызов grantpt() ... 1226
60.2.3. Разблокировка вторичного устройства: вызов unlockpt() ... 1227
60.2.4. Получение имени вторичного устройства: вызов ptsname() ... 1227
60.3. Открытие первичного устройства: вызов ptyMasterOpen() ... 1228
60.4. Соединение процессов с помощью псевдотерминала: вызов ptyFork() ... 1229
60.5. Ввод/вывод псевдотерминала ... 1232
60.6. Реализация программы script(1) ... 1234
60.7. Атрибуты терминала и размер окна ... 1238
60.8. Резюме ... 1239
60.9. Упражнения ... 1239

Список используемых источников ... 1241

Приветствую вас, читателей русскоязычного издания моей книги The Linux Programming Interface.

В этой книге представлено практически полное описание API системного программирования под управлением Linux. Ее содержимое применимо к широкому диапазону Linux-платформ, начиная с обычных серверов, универсальных компьютеров и настольных систем и заканчивая большим разнообразием встроенных устройств (в том числе работающих под управлением операционной системы Android), на которых в настоящее время запускается ОС Linux.

Англоязычное издание этой книги вышло в конце 2010 года. С того времени было выпущено несколько обновлений ядра Linux (их было примерно по пять за год). Несмотря на это, содержимое оригинала книги, а следовательно, и данного перевода, не утратило актуальности и сохранит ее еще на долгие годы. Тому есть две причины.

Во-первых, несмотря на стремительность разработки ядра Linux, API, связанный с пользовательским пространством ядра, изменяется гораздо медленнее. Такая консервативность — естественное следствие того факта, что ядро разработано с целью обеспечить *стабильную* основу для приложений, выполняемых в пространстве пользователя. Скоротечность развития API пространства пользователя неприемлема для тех программ, которым следует запускаться на нескольких версиях ядра.

Во-вторых, изменения вносятся в виде *дополнений* к интерфейсам, рассматриваемым в книге, а не *модификаций* уже существующих функциональных свойств, описанных в ней же. (Хочу еще раз отметить, что это вполне естественный ход разработки ядра Linux: специалисты прилагают большие усилия к тому, чтобы ничего не нарушить в *уже существующем* API пользовательского пространства.) Со дня выхода оригинала книги в данный API были внесены изменения. Их перечень (на английском) дотошные читатели могут увидеть на моем сайте по адресу http://man7.org/tlpi/api_changes/.

В заключение хочу отметить: я очень горжусь тем, что моя книга будет переведена на другой язык. Перевод на русский стал для меня особенно приятным сюрпризом, поскольку в детстве я пытался выучить этот язык по книгам. (К сожалению, отсутствие педагога или русскоговорящего окружения не позволили мне существенно преуспеть в этом начинании.) Перевод текста объемом 1250 страниц — задача не из легких, и я благодарен издателю и команде переводчиков. Надеюсь, что результаты нашей усердной работы и усилий множества других людей, помогавших выпустить в свет оригинал, окажутся весьма полезными для вас, читателей этого русскоязычного издания.

Майкл Керриск (Michael Kerrisk)

Предисловие

Цель книги

В этой книге я описываю программный интерфейс операционной системы Linux: системные вызовы, библиотечные функции и другие низкоуровневые интерфейсы, которые есть в Linux — свободно распространяемой реализации UNIX. Эти интерфейсы прямо или косвенно используются каждой программой, работающей в Linux. Они позволяют приложениям выполнять следующие операции:

- файловый ввод/вывод;
- создание и удаление файлов и каталогов;
- создание новых процессов;
- запуск программ;
- установку таймеров;
- взаимодействие между процессами и потоками на одном компьютере;
- взаимодействие между процессами, запущенными на разных компьютерах, объединенных посредством сети.

Такой набор низкоуровневых интерфейсов иногда называют *интерфейсом системного программирования*.

Несмотря на то что основное внимание уделяется операционной системе Linux, подробно рассмотрены также стандарты и вопросы, связанные с портируемостью. Я четко разграничиваю темы, специфичные для Linux, и функциональные возможности, типичные для большинства реализаций UNIX и описанные в стандарте POSIX, а также в спецификации Single UNIX Specification. Таким образом, эта книга предлагает всеобъемлющее рассмотрение программного интерфейса UNIX/POSIX и ее могут использовать программисты, которые создают приложения, предназначенные для других UNIX-систем, или портируемые программы.

Для кого эта книга

Книга предназначена главным образом для такой аудитории, как:

- программисты и разработчики программного обеспечения, создающие приложения для Linux, других UNIX-систем или иных систем, совместимых со стандартом POSIX;
- программисты, выполняющие портирование приложений из Linux в другие реализации UNIX или из Linux в другие операционные системы (ОС);
- преподаватели и заинтересованные студенты, которые преподают или изучают программирование для Linux или для других UNIX-систем;
- системные администраторы и «продвинутые» пользователи, которые желают тщательнее изучить программный интерфейс Linux/UNIX и понять, каким образом реализованы различные части системного ПО.

Предполагается, что у вас есть какой-либо опыт программирования, при этом опыт системного программирования необязателен. Я также рассчитываю на то, что вы разбираетесь в языке программирования C и знаете, как работать в оболочке и пользоваться основными командами Linux или UNIX. Если вы впервые сталкиваетесь с Linux или UNIX, вам будет полезно прочесть главу 2 — в ней приводится обзор основных понятий Linux и UNIX, ориентированный на программиста.

Стандартным справочным руководством по языку C является книга [Kernighan & Ritchie, 1988]. В книге [Harbison & Steele, 2002] этот язык рассмотрен более подробно, а также приведены изменения, появившиеся в стандарте C99. Издание [van der Linden, 1994] дает альтернативное рассмотрение языка C, очень увлекательное и толковое. В книге [Peek et al., 2001] содержится хорошее краткое введение в работу с системой UNIX.

Linux и UNIX

Эта книга могла бы быть полностью посвящена системному программированию в стандарте UNIX (то есть POSIX), поскольку многие функции, которые можно найти в других реализациях UNIX, присутствуют также в Linux и наоборот. Тем не менее, поскольку создание портируемых приложений — одна из основных задач, важно также описать и особенности Linux, которые расширяют стандартный программный интерфейс UNIX. Одной из причин является популярность Linux. Другая причина состоит в том, что иногда необходимо применять нестандартные расширения: либо из соображений производительности, либо для достижения функциональности, недоступной в стандартном программном интерфейсе UNIX. (По этим причинам все реализации UNIX снабжены нестандартными расширениями.)

Таким образом, хотя я задумал эту книгу так, чтобы она была полезна для программистов, работающих со всеми реализациями UNIX, я также привожу полное описание программных средств, характерных для Linux. К ним относятся следующие:

- интерфейс `epoll`, который позволяет получать уведомления о событиях файлового ввода-вывода;
- интерфейс `inotify`, позволяющий отслеживать изменения файлов и каталогов;
- мандаты (возможности) (capabilities) — позволяют предоставить какому-либо процессу часть прав суперпользователя;
- расширенные атрибуты;
- флаги индексного дескриптора;
- системный вызов `clone()`;
- файловая система `/proc`;
- характерные для Linux особенности реализации файлового ввода-вывода, сигналов, таймеров, потоков, совместно используемых (общих) библиотек, межпроцессного взаимодействия и сокетов.

Структура книги

Вы можете использовать эту книгу по меньшей мере двумя способами.

- В качестве вводного руководства в программный интерфейс Linux/UNIX. Можно читать книгу от начала до конца. Главы из второй половины издания основаны на материале, изложенном в главах первой половины; ссылки на более поздний материал по возможности сведены к минимуму.
- В качестве всеобъемлющего справочника по программному интерфейсу Linux/UNIX. Расширенное оглавление и обилие перекрестных ссылок позволяют читать книгу в произвольном порядке.

Главы этой книги сгруппированы следующим образом.

1. *Предварительные сведения и понятия*. История UNIX, языка C и Linux, а также обзор стандартов UNIX (глава 1); ориентированное на программиста описание тем,

относящихся к Linux и UNIX (глава 2); фундаментальные понятия системного программирования в Linux и UNIX (глава 3).

2. *Фундаментальные функции интерфейса системного программирования*. Файловый ввод/вывод (главы 4 и 5); процессы (глава 6); выделение памяти (глава 7); пользователи и группы (глава 8); идентификаторы процесса (глава 9); время (глава 10); системные ограничения и возможности (глава 11); получение информации о системе и процессе (глава 12).
3. *Более сложные функции интерфейса системного программирования*. Буферизация файлового ввода-вывода (глава 13); файловые системы (глава 14); атрибуты файла (глава 15); расширенные атрибуты (глава 16); списки контроля доступа (глава 17); каталоги и ссылки (глава 18); отслеживание файловых событий (глава 19); сигналы (главы 20–22); таймеры (глава 23).
4. *Процессы, программы и потоки*. Создание процесса, прекращение процесса, отслеживание дочерних процессов и выполнение программ (главы 24–28); потоки POSIX (главы 29–33).
5. *Более сложные темы, относящиеся к процессам и программам*. Группы процессов, сессии и управление задачами (глава 34); приоритет процессов и диспетчеризация (глава 35); ресурсы процессов (глава 36); демоны (глава 37); написание программ, привилегированных в плане безопасности (глава 38); возможности (глава 39); учетные записи (глава 40); совместно используемые библиотеки (главы 41 и 42).
6. *Межпроцессное взаимодействие (IPC)*. Обзор IPC (глава 43); каналы и очереди FIFO (глава 44); отображение в память (глава 45); операции с виртуальной памятью (глава 46); IPC в стандарте POSIX: очереди сообщений, семафоры и совместно используемая память (главы 47–50); блокировка файлов (глава 51).
7. *Сокеты и сетевое программирование*. IPC и сетевое программирование с помощью сокетов (главы 52–57).
8. *Углубленное рассмотрение вопросов ввода-вывода*. Терминалы (глава 58); альтернативные модели ввода-вывода (глава 59); псевдотерминалы (глава 60).

Примеры программ

Как использовать большинство интерфейсов, описанных в этой книге, я поясняю с помощью коротких готовых программ. Многие из них позволят вам с легкостью поэкспериментировать с командной строкой, чтобы увидеть, как работают различные системные вызовы и библиотечные функции. Таким образом, книга содержит довольно много программного кода с примерами — около 15 000 строк на языке C — а также фрагменты сессий работы в оболочке.

Для начала неплохо разобрать примеры программ и поэкспериментировать с ними, но усвоить понятия, изложенные в этой книге, эффективнее всего можно, написав код. Либо изменяя предложенные примеры для реализации ваших идей, либо создавая новые программы.

Весь исходный код доступен для загрузки с сайта англоязычного издания этой книги: http://www.man7.org/tlpi.

В архив включены также дополнительные программы, которых нет в издании. Назначение и подробности работы этих программ описаны в комментариях к программному коду. Для сборки программ приложены файлы `makefiles`, а в сопроводительных файлах `README` приводятся дополнительные подробности о программах.

Предоставляемый исходный код является свободно распространяемым, и его можно изменять при условии соблюдения лицензии GNU Affero General Public License (Affero

GPL) version 3 («Общедоступная лицензия GNU Affero, 3-я версия»), текст которой присутствует в архиве с исходным кодом.

На указанном сайте вы также найдете дополнительную информацию об этой книге.

Упражнения

Большинство глав завершаются набором упражнений. В одних из них мы предлагаем вам поэкспериментировать с приведенными примерами программ. Другие упражнения — это вопросы по темам, рассматриваемым в главе. Среди них также есть указания по написанию программ, которые могли бы помочь вам в усвоении материала.

Стандарты и портируемость

В этой книге особое внимание я уделил вопросам портируемости. Вы обнаружите немало ссылок на соответствующие стандарты, в особенности на объединенный стандарт POSIX.1-2001 и Single UNIX Specification version 3 (SUSv3). Кроме того, я привожу подробные сведения об изменениях в недавней версии — объединенном стандарте POSIX.1-2008 и SUSv4. (Поскольку стандарт SUSv3 гораздо обширнее и является стандартом UNIX с наибольшим влиянием (на момент написания книги), в этом руководстве при рассмотрении стандартов я, как правило, опираюсь на SUSv3, добавляя примечания об отличиях в SUSv4. Тем не менее можно рассчитывать на то, что, за исключением указанных случаев, утверждения о спецификациях в стандарте SUSv3 действуют и в SUSv4.)

Рассказывая о функциях, которые не стандартизированы, я привожу перечень отличий для других реализаций UNIX. Я также особо отмечаю те главные функции Linux, которые характерны именно для этой ОС, а заодно выделяю небольшие различия в реализации системных вызовов и библиотечных функций Linux и других UNIX-систем. В тех случаях, когда о какой-либо функции не говорится как о специфичной для Linux, можете считать, что она является стандартной и присутствует в большинстве или во всех реализациях UNIX.

Я протестировал работу большинства программ, приведенных в этой книге (за исключением тех, что используют функции, отмеченные как специфичные для Linux), в некоторых или во всех этих системах: Solaris, FreeBSD, Mac OS X, Tru64 UNIX и HP-UX. Для улучшения портируемости программ в некоторые из этих систем на сайте http://www.man7.org/tlpi приводятся альтернативные версии большинства примеров с дополнительным кодом, которого нет в этой книге.

Ядро Linux и версии библиотеки C

Основной акцент этой книги сделан на версии Linux 2.6.x, ядро которой было наиболее популярно на момент написания книги. Подробности версии 2.4 также описаны, причем отмечено, чем различаются функции в версиях Linux 2.4 и 2.6. В тех случаях, когда новые функции введены в серии релизов Linux 2.6.x, указана точная версия ядра, в которой они появились (например, 2.6.34).

Что касается библиотеки C, основное внимание уделено GNU-библиотеке C (glibc) 2-й версии. Там, где это важно, приведены различия между версиями glibc 2.x.

Когда это издание готовилось к печати, было выпущено ядро Linux версии 2.6.35, а версия glibc 2.12 уже появилась. Книга написана применительно к этим версиям программного обеспечения. Изменения, которые появились в интерфейсах Linux и в библиотеке glibc после публикации этой книги, будут отмечены на сайте.

Использование программного интерфейса других языков программирования

Хотя примеры программ написаны на языке C, вы можете применять рассмотренные в этой книге интерфейсы из других языков программирования, в частности компилируемых языков C++, Pascal, Modula, Ada, FORTRAN, D, а также таких языков сценариев, как Perl, Python и Ruby. (Для Java необходим другой подход; см., например, работу [Rochkind, 2004].) Понадобятся иные приемы для того, чтобы добиться необходимых определений констант и объявлений функций (за исключением языка C++). Может также потребоваться дополнительная работа для передачи аргументов функции в том стиле, которого требуют соглашения о связях в языке C. Но, несмотря на эти различия, основные понятия не меняются, и вы обнаружите, что информация из этого руководства применима даже при работе с другим языком программирования.

Об авторе

Я начал работать в UNIX и на языке C в 1987 году, когда провел несколько недель за рабочей станцией HP Bobcat, имея при себе первое издание книги Марка Рохкинда «Эффективное UNIX-программирование» (*Marc Rochkind*, Advanced UNIX Programming) и распечатку руководства по командной оболочке C shell shell (она же csh) (в конечном итоге у нее был довольно потрепанный вид). Подход, который я применял тогда, я стараюсь применять и теперь. Рекомендую его также всем, кто приступает к работе с новой технологией в разработке ПО: потратьте время на чтение документации (если она есть) и пишите небольшие (но постепенно увеличивающиеся) тестовые программы до тех пор, пока не начнете уверенно понимать программное обеспечение. Я обнаружил, что в конечном итоге такой вариант самообучения хорошо окупает себя в плане сэкономленного времени. Многие примеры программ в книге сконструированы так, чтобы подтолкнуть вас к применению этого подхода.

Сначала я работал инженером-программистом и разработчиком ПО. В то же время мне очень нравится преподавание, и несколько лет я занимался им как в академической, так и в бизнес-среде. Я провел множество недельных курсов по обучению системному программированию UNIX, и этот опыт вдохновил меня на написание книги.

Я проработал в Linux почти в два раза меньше, чем в UNIX, но за это время мои интересы еще больше сфокусировались на границе между ядром и пространством пользователя — на программном интерфейсе Linux. Это вовлекло меня в несколько взаимосвязанных видов деятельности. Время от времени я предоставлял исходную информацию и отчеты об ошибках для стандарта POSIX/SUS, выполнял тестирование и экспертную оценку новых интерфейсов пространства пользователя, добавленных к ядру Linux (и помог обнаружить и исправить множество ошибок в коде и дизайне этих интерфейсов). Я также регулярно выступал на конференциях, посвященных интерфейсам и связанной с ними документации. Меня приглашали на несколько ежегодных совещаний Linux Kernel Developers Summit (саммит разработчиков ядра Linux). Общей нитью, которая связывает все эти виды деятельности воедино, является мой наиболее заметный вклад в мир Linux — работа над проектом man-pages (http://www.kernel.org/doc/man-pages/).

Названный проект лежит в основе страниц руководства Linux в разделах 2–5 и 7. Эти страницы описывают программные интерфейсы, которые предоставляются ядром Linux и GNU-библиотекой C, — тот же охват тем, что и в этой книге. Я занимался проектом man-pages более десяти лет. Начиная с 2004 года я сопровождаю его. В эту задачу входят приблизительно в равных долях написание документации, изучение исходного кода ядра

и библиотеки, а также создание программ для проверки деталей. (Документирование интерфейса — прекрасный способ обнаружить ошибки в этом интерфейсе.) Кроме того, я самый продуктивный участник проекта man-pages: он содержит около 900 страниц, 140 из них написал я один и 125 — в соавторстве. Поэтому весьма вероятно, что вы уже читали что-либо из моих публикаций еще до того, как приобрели эту книгу. Надеюсь, что информация вам пригодилась, и также надеюсь, что эта книга окажется еще более полезной.

Благодарности

Без поддержки огромного количества людей эта книга не стала бы такой, какая она есть. С великим удовольствием благодарю их.

Научные редакторы из разных стран, как одна большая команда, читали черновые варианты, отыскивали ошибки, указывали на нечеткие объяснения, предлагали перефразированные варианты и схемы, тестировали программы, снабжали упражнениями, выявляли особенности работы Linux и других реализаций UNIX, которые были мне неизвестны, а также поддерживали и воодушевляли меня. Многие эксперты щедро поделились ценной информацией, которую мне удалось включить в эту книгу. Благодаря этим дополнениям возникает впечатление о моей большой осведомленности, хотя на самом деле это не так. Все ошибки, которые присутствуют в книге, конечно же, остаются на моей совести.

Особо благодарю следующих специалистов, которые развернуто прокомментировали различные фрагменты рукописи.

- Кристоф Блэсс (Christophe Blaess) является программистом-консультантом и профессиональным преподавателем. Специализируется на производственных (в реальном времени и встраиваемых) приложениях на основе Linux. Он автор замечательной книги на французском языке *Programmation systume en C sous Linux* («Системное программирование на языке C в Linux»), в которой охвачены многие из тем, изложенных в данной книге. Он прочитал и щедро прокомментировал многие главы моей книги.
- Дэвид Бутенхоф (David Butenhof) из компании Hewlett-Packard — участник самой первой рабочей группы по потокам POSIX, а также по расширениям стандарта Single UNIX Specification для потоков. Он автор книги *Programming with POSIX Threads* («Программирование с помощью потоков POSIX»). Он написал исходную базовую реализацию потоков DCE Threads для проекта Open Software Foundation и был ведущим проектировщиком реализации потоков для проектов OpenVMS и Digital UNIX. Дэвид проверил главы о потоках, предложил множество улучшений и терпеливо помогал мне лучше разобраться в некоторых особенностях API для потоков POSIX.
- Джеф Клэр (Geoff Clare) занят в проекте The Open Group разработкой комплексов тестирования на соответствие стандартам UNIX. Он уже более 20 лет принимает участие в разработке стандартов UNIX и является одним из немногих ключевых участников группы Austin Group, которая создает объединенный стандарт, образующий спецификацию POSIX.1 и основные части спецификации Single UNIX Specification. Джеф тщательно проверил части рукописи, относящиеся к стандартным интерфейсам UNIX, терпеливо и вежливо предлагая свои исправления и улучшения. Он выявил малозаметные ошибки в коде и помог сосредоточиться на важности следования стандартам при создании портируемых программ.
- Лоик Домэнье (Loïc Domaigné), работавший в немецкой авиадиспетчерской службе, — разработчик программных комплексов, который проектирует и создает распределенные параллельные отказоустойчивые встроенные системы с жесткими требованиями работы в реальном времени. Он предоставил обзорный вводный материал

для спецификации потоков в стандарте SUSv3. Лоик — замечательный преподаватель и эрудированный участник различных технических онлайн-форумов. Он тщательно проверил главы о потоках, а также многие другие разделы книги. Он также написал несколько хитроумных программ для проверки особенностей реализации потоков в Linux, а также предложил множество идей по улучшению общей подачи материала.

- Герт Деринг (Gert Döring) написал программы mgetty и sendfax — наиболее популярные свободно распространяемые пакеты для работы с факсами в UNIX и Linux. В настоящее время он главным образом работает над созданием обширных сетей на основе протоколов IPv4 и IPv6 и управлением ими. Эта деятельность включает в себя сотрудничество с коллегами по всей Европе с целью определения рабочих политик, которые обеспечивают надежную работу инфраструктуры Интернета. Герт дал немало ценных советов по главам, описывающим терминалы, учетные записи, группы процессов, сессии и управление задачами.
- Вольфрам Глоджер (Wolfram Gloger) — ИТ-консультант, который последние 15 лет нередко участвовал в различных проектах FOSS (Free and Open Source Software, свободно распространяемое ПО и ПО с открытым исходным кодом). Среди прочего, Вольфрам является разработчиком пакета malloc, который используется в GNU-библиотеке C. В настоящее время он разрабатывает веб-сервисы для дистанционного обучения, но иногда все так же занимается ядром и системными библиотеками. Вольфрам проверил несколько глав и особенно помог мне при рассмотрении вопросов, относящихся к памяти.
- Фернандо Гонт (Fernando Gont) — сотрудник центра CEDI (Centro de Estudios de Informática) при аргентинском университете Universidad Tecnológica Nacional. Он занимается в основном интернет-разработками и активно участвует в работе сообщества IETF (Internet Engineering Task Force, Инженерный совет Интернета), для которого он написал несколько рабочих предложений. Фернандо также занят оценкой безопасности коммуникационных протоколов в британском центре CPNI (Centre for the Protection of National Infrastructure, Центр защиты государственной инфраструктуры). Он впервые выполнил всестороннюю оценку безопасности протоколов TCP и IP. Фернандо очень тщательно проверил главы, посвященные сетевому программированию, объяснил множество особенностей протоколов TCP/IP, а также предложил немало улучшений.
- Андреас Грюнбахер (Andreas Grünbacher) — специалист по ядру и автор реализации расширенных атрибутов в Linux, а также списков контроля доступа в стандарте POSIX. Андреас тщательно проверил многие главы, оказал существенную поддержку, а один из его комментариев значительно повлиял на структуру книги.
- Кристоф Хельвиг (Christoph Hellwig) является консультантом по хранению данных в Linux и по файловым системам, а также экспертом по ядру, многие части которого он сам и разрабатывал. Кристоф любезно согласился на некоторое время отвлечься от написания и проверки обновлений для ядра Linux, чтобы просмотреть пару глав этой книги и дать много ценных советов.
- Андреас Егер (Andreas Jaeger) руководил портированием Linux в архитектуру x86-64. Будучи разработчиком GNU-библиотеки C, он портировал эту библиотеку и сумел добиться ее соответствия стандартам в различных областях, особенно в ее математической части. В настоящее время он является менеджером проектов openSUSE в компании Novell. Андреас проверил намного больше глав, чем я рассчитывал, предложил множество улучшений и воодушевил на дальнейшую работу с книгой.
- Рик Джонс (Rick Jones), который известен также как «Мистер Netperf» (Networked Systems Performance Curmudgeon (буквально — «старый ворчун на тему производительности сетевых систем») в компании Hewlett-Packard), дотошно проверил главы о сетевом программировании.

- Энди Клин (Andi Kleen) (работавший тогда в компании SUSE Labs) — знаменитый профессионал, который работал над различными характеристиками ядра Linux, включая сетевое взаимодействие, обработку ошибок, масштабируемость и программный код низкоуровневой архитектуры. Энди досконально проверил материал о сетевом программировании, расширил мое представление о большинстве особенностей реализации протоколов TCP/IP в Linux и дал немало советов, позволивших улучшить подачу материала.
- Мартин Ландерс (Martin Landers) (компания Google) был еще студентом, когда мне посчастливилось познакомиться с ним в колледже. За короткий срок он успел добиться многого, поработав и проектировщиком архитектуры ПО, и ИТ-преподавателем, и профессиональным программистом. Мне повезло, что Мартин оказался в числе моих экспертов. Его многочисленные язвительные комментарии и исправления значительно улучшили многие главы этой книги.
- Джейми Лоукир (Jamie Lokier) — известный специалист, который в течение 15 лет участвует в разработке Linux. В настоящее время он характеризует себя как «консультант по решению сложных проблем, в которые каким-либо образом вовлечена Linux». Джейми тщательнейшим образом проверил главы об отображении в память, совместно используемой памяти POSIX и об операциях в виртуальной памяти. Благодаря его комментариям я гораздо лучше стал разбираться в этих темах и смог существенно улучшить структуру глав книги.
- Барри Марголин (Barry Margolin) за время своей 25-летней карьеры работал системным программистом, системным администратором и инженером службы поддержки. В настоящее время он является ведущим инженером по производительности в компании Akamai Technologies. Он популярный и уважаемый автор сообщений в онлайн-форумах об UNIX и Интернете, а также рецензент множества книг на эти темы. Барри проверил несколько глав моей книги и предложил много улучшений.
- Павел Плужников (Paul Pluzhnikov) (компания Google) в прошлом был техническим руководителем и ключевым разработчиком инструмента для отладки памяти Insure++. Он отлично разбирается в отладчике gdb и часто отвечает посетителям форумов об отладке, выделении памяти, совместно используемых библиотеках и состоянии переменных среды в момент работы программы. Павел просмотрел многие главы и внес несколько ценных предложений.
- Джон Рейзер (John Reiser) (совместно с Томом Лондоном (Tom London)) осуществил одно из самых первых портирований UNIX в 32-битную архитектуру: вариант VAX-11/780. Он создал системный вызов `mmap()`. Джон проверил многие главы (включая, разумеется, и главу о системном вызове `mmap()`) и привел множество исторических подробностей.
- Энтони Робинс (Anthony Robins) (адъюнкт-профессор по информатике в университете города Отаго в Новой Зеландии), мой близкий друг вот уже более трех десятилетий, стал первым читателем черновиков некоторых глав. Он предложил немало ценных замечаний на раннем этапе и оказал поддержку по мере развития проекта.
- Михаэль Шрёдер (Michael Schröder) (компания Novell) — один из главных авторов GNU-программы screen. Работа над ней научила его хорошо разбираться в тонкостях и различиях в реализации драйверов терминалов. Михаэль проверил главы о терминалах и псевдотерминалах, а также главу о группах процессов, сессиях и управлении задачами, дав ценные отзывы.
- Манфред Спрол (Manfred Spraul), разрабатывавший среди прочего код межпроцессного взаимодействия (IPC) в ядре Linux, тщательно проверил некоторые главы о нем и предложил множество улучшений.

- Том Свигг (Tom Swigg), в прошлом преподаватель UNIX в компании Digital, выступил в роли эксперта на ранних стадиях. Том уже более 25 лет работает инженером-программистом и ИТ-преподавателем и в настоящее время трудится в лондонском университете South Bank, где занимается программированием и поддержкой Linux и среды VMware.
- Йенс Томс Тёрринг (Jens Thoms Törring) является представителем поколения физиков, которые превратились в программистов. Он разработал множество драйверов устройств с открытым кодом, а также другое ПО. Йенс прочитал на удивление разнородные главы и поделился исключительно ценными соображениями о том, в чем можно улучшить каждую из них.

Многие другие технические эксперты также прочитали различные части этой книги и предложили ценные комментарии. Благодарю вас: Джордж Анцингер (George Anzinger) (компания MontaVista Software), Стефан Бечер (Stefan Becher), Кшиштоф Бенедичак (Krzysztof Benedyczak), Дэниэл Бранеборг (Daniel Brahneborg), Эндрис Брауэр (Andries Brouwer), Анабел Черч (Annabel Church), Драган Цветкович (Dragan Cvetkovič), Флойд Л. Дэвидсон (Floyd L. Davidson), Стюарт Дэвидсон (Stuart Davidson) (компания Hewlett-Packard Consulting), Каспер Дюпон (Kasper Dupont), Петер Феллингер (Peter Fellinger) (компания jambit GmbH), Мел Горман (Mel Gorman) (компания IBM), Нильс Голлеш (Niels Gqllesch), Клаус Гратцл (Claus Gratzl), Серж Халлин (Serge Hallyn) (компания IBM), Маркус Хартингер (Markus Hartinger) (компания jambit GmbH), Ричард Хендерсон (Richard Henderson) (компания Red Hat), Эндрю Джоузи (Andrew Josey) (компания The Open Group), Дэн Кегел (Dan Kegel) (компания Google), Давид Либенци (Davide Libenzi), Роберт Лав (Robert Love) (компания Google), Х. Дж. Лу (H. J. Lu) (компания Intel Corporation), Пол Маршалл (Paul Marshall), Крис Мэйсон (Chris Mason), Майкл Матц (Michael Matz) (компания SUSE), Тронд Майклбаст (Trond Myklebust), Джеймс Пич (James Peach), Марк Филлипс (Mark Phillips) (компания Automated Test Systems), Ник Пиггин (Nick Piggin) (компании SUSE Labs и Novell), Кай Йоханнес Поттхофф (Kay Johannes Potthoff), Флориан Рампп (Florian Rampp), Стефен Ротвелл (Stephen Rothwell) (компании Linux Technology Centre и IBM), Маркус Швайгер (Markus Schwaiger), Стефен Твиди (Stephen Tweedie) (компания Red Hat), Бритта Варгас (Britta Vargas), Крис Райт (Chris Wright), Михал Вронски (Michal Wronski) и Умберто Замунер (Umberto Zamuner).

Помимо технических рецензий, я получал разнообразную поддержку от множества людей и организаций.

Спасибо следующим людям за их ответы на технические вопросы: Яну Кара (Jan Kara), Дейву Клайкампу (Dave Kleikamp) и Джону Снейдеру (Jon Snader). Благодарю Клауса Гратцла и Пола Маршалла за помощь в системном менеджменте.

Спасибо компании Linux Foundation (LF), которая на протяжении 2008 года оплачивала мою полную занятость в качестве стипендианта при работе над проектом man-pages, а также при тестировании и экспертной оценке программного интерфейса Linux. И хотя компания не оказывала непосредственную финансовую поддержку работы над этой книгой, она все же финансово поддерживала меня и мою семью, а возможность сконцентрироваться на документировании и тестировании программного интерфейса Linux благоприятно отразилась на моем «частном» проекте. На индивидуальном уровне — спасибо Джиму Землину (Jim Zemlin) за то, что он оказался в роли моего «интерфейса» при работе в LF, а также членам Технического совета компании (LF Technical Advisory Board), которые поддержали мою заявку на принятие в число стипендиантов.

Благодарю Алехандро Фореро Куэрво (Alejandro Forero Cuervo), который предложил название для этой книги!

Более 25 лет назад, когда я получал первую ученую степень, Роберт Биддл (Robert Biddle) заинтриговал меня рассказами об UNIX и языках C и Ratfor. Искренне благодарю его.

Спасибо следующим людям, которые не были непосредственно связаны с этим проектом, но воодушевили меня на получение второй ученой степени в университете Кентербери в Новой Зеландии. Это Майкл Ховард (Michael Howard), Джонатан Мэйн-Уиоки (Jonathan Mane-Wheoki), Кен Стронгман (Ken Strongman), Гарт Флетчер (Garth Fletcher), Джим Поллард (Jim Pollard) и Брайан Хейг (Brian Haig).

Уже довольно давно Ричард Стивенс (Richard Stevens) написал несколько превосходных книг о UNIX-программировании и протоколах TCP/IP. Для меня, а также для многих других программистов, эти издания стали прекрасным источником технической информации на долгие годы.

Спасибо следующим людям и организациям, которые обеспечили меня UNIX-системами, позволили проверить тестовые программы и уточнить детали для других реализаций UNIX: Энтони Робинсу (Anthony Robins) и Кэти Чандра (Cathy Chandra) — за системы тестирования в Университете Отаго в Новой Зеландии; Мартину Ландерсу (Martin Landers), Ральфу Эбнеру (Ralf Ebner) и Клаусу Тилку (Klaus Tilk) — за системы тестирования в Техническом университете Мюнхена в Германии; компании Hewlett-Packard за то, что сделала свободно доступными в Интернете свои системы testdrive; Полю де Веерду (Paul de Weerd) — за доступ к системе OpenBSD.

Искренне признателен двум мюнхенским компаниям и их владельцам, которые не только предоставили мне гибкий график работы и приветливых коллег, но и оказались исключительно великодушны, позволив мне пользоваться их офисами на время написания книги. Спасибо Томасу Кахабке (Thomas Kahabka) и Томасу Гмельху (Thomas Gmelch) из компании exolution GmbH, а отдельное спасибо — Петеру Феллингеру и Маркусу Хартингеру из компании jambit GmbH.

Спасибо за разного рода помощь, полученную от вас, Дэн Рэндоу (Dan Randow), Карен Коррел (Karen Korrel), Клаудио Скалмацци (Claudio Scalmazzi), Майкл Шюпбах (Michael Schüpbach) и Лиз Райт (Liz Wright). Благодарю Роба Суистеда (Rob Suisted) и Линли Кук (Lynley Cook) за фотографии, которые использованы на обложке.

Спасибо следующим людям, которые всевозможными способами подбадривали и поддерживали меня при работе над этим проектом: это Дебора Черч (Deborah Church), Дорис Черч (Doris Church) и Энни Карри (Annie Currie).

Спасибо сотрудникам издательства No Starch Press за все виды содействия этому внушительному проекту. Благодарю Билла Поллока (Bill Pollock) за то, что удалось договориться с ним с самого начала, за его незыблемую уверенность в проекте и за терпеливое сопровождение. Спасибо моему первому выпускающему редактору Меган Дунчак (Megan Dunchak). Спасибо корректору Мэрилин Смит (Marilyn Smith), которая обязательно найдет множество огрехов, несмотря на то что я изо всех сил стремлюсь к ясности и согласованности. Райли Хоффман (Riley Hoffman) всецело отвечал за макет и дизайн этой книги, а также взял на себя роль выпускающего редактора, когда мы вышли на финишную прямую. Райли милостиво вытерпел мои многочисленные запросы, касающиеся подходящего макета, и в итоге выдал превосходный результат. Спасибо!

Теперь я понимаю значение избитой фразы о том, что семья писателя также вносит свою лепту в его работу. Спасибо Бритте и Сесилии за поддержку и за долгие часы ожидания, пока мне приходилось быть вдали от семьи, чтобы завершить книгу.

Разрешения

Институт инженеров электротехники и электроники (IEEE) и компания The Open Group любезно предоставили право цитировать фрагменты текста из документов IEEE Std 1003.1, 2004 Edition (Стандарт IEEE 1003.1, версия 2004 года), Standard for Information Technology — Portable Operating System Interface (POSIX) (Стандарт информационных технологий — портируемый интерфейс операционной системы) и The Open Group Base Specifications Issue 6 (Базовые спецификации Open Group. Выпуск 6). Полную версию стандарта можно прочитать на сайте http://www.unix.org/version3/online.html.

Обратная связь

Буду признателен за сообщения об ошибках в программах, предложения по улучшению кода, а также за исправления, которые позволят повысить портируемость кода. Приветствуются также сообщения об опечатках и предложения по улучшению материала книги. Поскольку изменения в программном интерфейсе Linux разнообразны и иногда происходят слишком часто, чтобы за ними мог уследить один человек, буду рад вашим сообщениям о новых или измененных функциях, о которых следует рассказать в будущем издании этой книги.

Майкл Тимоти Керриск
Мюнхен, Германия —
Крайстчерч, Новая Зеландия
Август 2010 г.
mtk@man7.org

1 История и стандарты

Linux относится к семейству операционных систем UNIX. По компьютерным меркам у UNIX весьма длинная история, краткий обзор которой дается в первой половине этой главы. Рассказ начнется с обзора истоков UNIX и языка программирования C и продолжится рассмотрением двух направлений, приведших систему Linux к ее теперешнему виду: проекта GNU и разработки ядра Linux.

Одна из примечательных особенностей UNIX состоит в том, что она не создавалась под контролем какого-то одного разработчика или организации. Вклад в ее развитие внесли многие группы: как коммерческие, так и некоммерческие. Такое развитие событий привело к добавлению в UNIX множества инновационных свойств, но наряду с этим способствовало появлению и негативных последствий. В частности, со временем обнаруживались расхождения в реализациях UNIX, все более затруднявшие написание приложений, способных работать во всех вариантах реализации системы. Возникла потребность в стандартизации реализаций UNIX, и она рассматривается во второй половине главы.

> Что касается самого понятия UNIX, то в мире бытуют два определения. Одно из них указывает на те операционные системы, которые прошли официальную проверку на совместимость с единой спецификацией под названием Single UNIX Specification и в результате этого получили от владельца торговой марки UNIX, The Open Group, официальное право называться UNIX. На момент написания книги это право не было получено ни одной из свободно распространяемых реализаций UNIX (например, Linux и FreeBSD).
>
> Согласно другому общепринятому значению определение UNIX распространяется на те системы, которые по внешнему виду и поведению похожи на классические UNIX-системы (например, на исходную версию Bell Laboratories UNIX и ее более поздние ветки — System V и BSD). В соответствии с этим определением Linux, как правило, считается UNIX-системой (как и современные BSD-системы). Хотя в этой книге спецификации Single UNIX Specification уделяется самое пристальное внимание, мы последуем второму определению UNIX и поэтому позволим себе довольно частое использование фраз вроде «Linux, как и другие реализации UNIX...».

1.1. Краткая история UNIX и языка C

Первая реализация UNIX была разработана в 1969 году (в год рождения Линуса Торвальдса (Linus Torvalds)) Кеном Томпсоном (Ken Thompson) в компании Bell Laboratories, являвшейся подразделением телефонной корпорации AT&T. Эта реализация была написана на ассемблере для мини-компьютера Digital PDP-7. Название UNIX было выбрано из-за созвучия с *MULTICS (Multiplexed Information and Computing Service)*, названием более раннего проекта операционной системы (ОС), разрабатываемой AT&T в сотрудничестве с институтом Massachusetts Institute of Technology (MIT) и компанией General Electric. (К тому времени AT&T уже была выведена из проекта из-за срыва первоначальных планов по разработке экономически пригодной системы.) Томпсон позаимствовал у MULTICS ряд идей для своей новой операционной системы, включая древовидную структуру файловой системы, отдельную программу для интерпретации команд (оболочки) и понятие файлов как неструктурированных потоков байтов.

В 1970 году UNIX была переписана на языке ассемблера для только что приобретенного мини-компьютера Digital PDP-11, который в то время считался новой и довольно мощной машиной. Следы PDP-11 до сих пор могут обнаруживаться в большинстве реализаций UNIX, включая Linux, под различными названиями.

Некоторое время спустя один из коллег Томпсона по Bell Laboratories, с которым он на ранней стадии сотрудничал при создании UNIX, Деннис Ритчи (Dennis Ritchie), разработал и реализовал язык программирования C. Процесс создания носил эволюционный характер; C был последователем более раннего языка программирования B, код которого выполнялся в режиме интерпретации. Язык B был изначально реализован Томпсоном и впитал в себя множество его идей, позаимствованных из еще более раннего языка программирования под названием BCPL. К 1973 году язык C уже был доведен до состояния, позволившего почти полностью переписать на нем ядро UNIX. Таким образом, UNIX стала одной из самых ранних ОС, написанных на языке высокого уровня, что позволило в дальнейшем портировать ее на другие аппаратные архитектуры.

Весьма широкая востребованность языка C и его потомка C++ в качестве языков системного программирования обусловлена их предысторией. Предыдущие широко используемые языки разрабатывались с другими предопределяемыми целями: FORTRAN предназначался для решения инженерных и научных математических задач, COBOL был рассчитан на работу в коммерческих системах обработки потоков, ориентированных на записи данных. Язык C заполнил пустующую нишу, и, в отличие от FORTRAN и COBOL (которые были разработаны крупными рабочими группами), конструкция языка C возникла на основе идей и потребностей нескольких отдельных личностей, стремящихся к достижению единой цели: разработке высокоуровневого языка для реализации ядра UNIX и связанных с ним программных систем. Подобно самой операционной системе UNIX, язык C был разработан профессиональными программистами для их собственных нужд. В результате получился весьма компактный, эффективный, мощный, лаконичный, прагматичный и последовательный в своей конструкции модульный язык.

UNIX от первого до шестого выпуска

В период с 1969 по 1979 год вышло несколько выпусков UNIX, называемых *редакциями*. По сути, они были текущими вариантами развивающейся версии, которая разрабатывалась в компании AT&T. В издании [Salus, 1994] указываются следующие даты первых шести редакций UNIX.

- Первая редакция, ноябрь 1971 года. К этому времени UNIX работала на PDP-11 и уже имела компилятор FORTRAN и версии множества программ, используемых по сей день, включая ar, cat, chmod, chown, cp, dc, ed, find, ln, ls, mail, mkdir, mv, rm, sh, su и who.
- Вторая редакция, июнь 1972 года. К этому моменту UNIX была установлена на десяти машинах компании AT&T.
- Третья редакция, февраль 1973 года. В эту редакцию был включен компилятор языка C и первая реализация конвейеров (pipes).
- Четвертая редакция, ноябрь 1973 года. Это была первая версия, практически полностью написанная на языке C.
- Пятая редакция, июнь 1974 года. К этому времени UNIX была установлена более чем на 50 системах.
- Шестая редакция, май 1975 года. Это была первая редакция, широко использовавшаяся вне компании AT&T.

За время выхода этих редакций система UNIX стала активнее использоваться, а ее репутация — расти, сначала в рамках компании AT&T, а затем и за ее пределами. Важным

вкладом в эту популярность была публикация статьи о UNIX в журнале *Communications of the ACM* [Ritchie & Thompson, 1974].

К этому времени компания AT&T владела санкционированной правительством монополией на телефонные системы США. Условия соглашения AT&T с правительством США не позволяли компании заниматься продажей программного обеспечения, а это означало, что она не могла продавать UNIX. Вместо этого начиная с 1974 года, с выпуском пятой и особенно с выпуском шестой редакции, AT&T за символическую плату организовала лицензированное распространение UNIX для использования в университетах. Распространяемые для университетов пакеты включали документацию и исходный код ядра (на то время около 10 000 строк кода).

Эта кампания стала существенным вкладом в популяризацию использования операционной системы, и к 1977 году UNIX работала примерно в 500 местах, включая 125 университетов в США и некоторых других странах. UNIX была для университетов весьма дешевой, но при этом мощной интерактивной многопользовательской операционной системой, в то время как коммерческие операционные системы стоили очень дорого. Кроме того, факультеты информатики получали исходный код реальной операционной системы, который они могли изменять и предоставлять своим студентам для изучения и проведения экспериментов. Одни студенты, вооружившись знаниями операционной системы UNIX, превратились в ее ярых приверженцев. Другие пошли еще дальше, основав новые компании или присоединившись к таким компаниям для продажи недорогих компьютерных рабочих станций с запускаемой на них легко портируемой операционной системой UNIX.

Рождение BSD и System V

В январе 1979 года вышла седьмая редакция UNIX. Она повысила надежность системы и предоставила усовершенствованную файловую систему. Этот выпуск также содержал несколько новых инструментальных средств, включая awk, make, sed, tar, uucp, Bourne shell и компилятор языка FORTRAN 77. Значимость седьмой редакции обуславливалась также тем, что, начиная с этого выпуска, UNIX разделилась на два основных варианта: BSD и System V, истоки которых мы сейчас кратко рассмотрим.

Кен Томпсон (Ken Thompson) в 1975/1976 учебном году был приглашенным профессором Калифорнийского университета в Беркли, откуда он в свое время выпустился. Там он работал с несколькими студентами выпускного курса, добавляя к UNIX множество новых свойств. (Один из этих студентов, Билл Джой (Bill Joy), впоследствии стал сооснователем компании Sun Microsystems, которая вскоре заявила о себе на рынке рабочих станций UNIX.) Со временем в Беркли было разработано множество новых инструментов и функций, включая C shell, редактор vi. Кроме того, были усовершенствованы файловая система (Berkeley Fast File System), почтовый агент sendmail, компилятор языка Pascal и система управления виртуальной памятью на новой архитектуре Digital VAX.

Эта версия UNIX, включавшая свой собственный исходный код, получила весьма широкое распространение под названием Berkeley Software Distribution (BSD). Первым полноценным дистрибутивом, появившимся в декабре 1979 года, стал 3BSD. (Ранее выпущенные в Беркли дистрибутивы BSD и 2BSD представляли собой не полные дистрибутивы UNIX, а пакеты новых инструментов, разработанных в Беркли.)

В 1983 году группа исследования компьютерных систем — Computer Systems Research Group — из Калифорнийского университета в Беркли выпустила 4.2BSD. Этот выпуск был примечателен тем, что в нем содержалась полноценная реализация протокола TCP/IP, включая интерфейс прикладного программирования (API) сокетов, и множество различных средств для работы в сети. Выпуск 4.2BSD и его предшественник 4.1BSD стали активно распространяться в университетах по всему миру. Они также легли в основу SunOS (впервые выпущенную в 1983 году) — UNIX-вариант, продаваемый компанией

Sun. Другими примечательными выпусками BSD были 4.3BSD в 1986 году и последний выпуск — 4.4BSD — в 1993 году.

> Самое первое портирование (перенос) системы UNIX на оборудование, отличное от PDP-11, произошло в 1977–1978 годах, когда Деннис Ритчи и Стив Джонсон (Steve Johnson) портировали ее на Interdata 8/32, а Ричард Миллер (Richard Miller) из Воллонгонского университета в Австралии одновременно с ними портировал ее на Interdata 7/32. Портированная версия Berkeley Digital VAX базировалась на более ранней (1978 года), также портированной версии, созданной Джоном Рейзером (John Reiser) и Томом Лондоном (Tom London). Она называлась 32V и была по сути тем же самым, что и седьмая редакция для PDP-11, за исключением более обширного адресного пространства и более емких типов данных.

В то же время принятое в США антимонопольное законодательство привело к разделу компании AT&T (юридический процесс начался в середине 1970-х годов, а сам раздел произошел в 1982 году), за которым последовали утрата монополии на телефонные системы и приобретение компанией права вывода UNIX на рынок. В результате в 1981 году состоялся выпуск System III (три). Эта версия была создана организованной в компании AT&T группой поддержки UNIX (UNIX Support Group, USG). В ней работали сотни специалистов, занимавшихся усовершенствованием UNIX и созданием приложений для этой системы (в частности, созданием пакетов подготовки документов и средств разработки ПО). В 1983 году последовал первый выпуск System V (пять). Несколько последующих выпусков привели к тому, что в 1989 году состоялся окончательный выпуск System V Release 4 (SVR4), ко времени которого в System V было перенесено множество свойств из BSD, включая сетевые объекты. Лицензия на System V была выдана множеству коммерческих поставщиков, использовавших эту версию как основу своих собственных реализаций UNIX.

Таким образом, вдобавок к различным дистрибутивам BSD, распространявшимся через университеты в конце 1980-х годов, UNIX стала доступна в виде коммерческих реализаций на различном оборудовании. Они включали:

- разработанную в компании Sun операционную систему SunOS, а позже и Solaris;
- созданные в компании Digital системы Ultrix и OSF/1 (в настоящее время, после нескольких переименований и поглощений, HP Tru64 UNIX);
- AIX компании IBM;
- HP-UX компании Hewlett-Packard (HP);
- NeXTStep компании NeXT;
- A/UX для Apple Macintosh;
- XENIX для архитектуры Intel x86-32 компаний Microsoft и SCO. (В данной книге реализация Linux для x86-32 будет упоминаться как Linux/x86-32.)

Такая ситуация резко контрастировала с типичными для того времени сценариями создания собственного оборудования и разработки под него ОС, когда каждый производитель создавал одну или от силы несколько собственных архитектур компьютерных микросхем, для которых он продавал операционную систему (или системы) собственной разработки.

Специализированный характер большинства поставляемых систем означал ориентацию покупателей только на одного поставщика. Переход на другую специализированную ОС и аппаратную платформу мог оказаться слишком дорогим удовольствием, поскольку для этого требовалось портирование имеющихся приложений и переучивание рабочего персонала. Этот фактор в совокупности с появлением дешевых однопользовательских рабочих станций под UNIX от различных производителей делал портируемую UNIX-систему все более привлекательной с коммерческой точки зрения.

1.2. Краткая история Linux

Говоря «Linux», обычно подразумевают полноценную UNIX-подобную операционную систему, часть которой формируется ядром Linux. Но такое толкование не совсем верно, поскольку многие ключевые компоненты, содержащиеся в коммерческих дистрибутивах Linux, фактически берутся из проекта, появившегося несколькими годами раньше самой Linux.

1.2.1. Проект GNU

В 1984 году весьма талантливый программист Ричард Столлман (Richard Stallman), работавший в Массачусетском технологическом институте, приступил к созданию «свободно распространяющейся» реализации UNIX. Работа была затеяна Столлманом из этических соображений, и принцип свободного распространения был определен в юридическом, а не в финансовом смысле (см. статью по адресу http://www.gnu.org/philosophy/free-sw.html). Но, как бы то ни было, под сформулированной Столлманом правовой свободой подразумевалось, что такие программные средства, как операционные системы, должны быть доступны на бесплатной основе или поставляться по весьма скромной цене.

Столлман боролся против правовых ограничений, накладываемых на фирменные операционные системы поставщиками компьютерных продуктов. Эти ограничения означали, что покупатели компьютерных программ, как правило, не могли видеть исходный код купленной ими программы и, конечно же, не могли ее копировать, изменять или распространять. Он отметил, что такие нормы порождают конкуренцию между программистами и вызывают у них стремление припрятывать свои проекты, вместо того чтобы сотрудничать и делиться ими.

В ответ на это Столлман запустил проект GNU (рекурсивно определяемый акроним, взятый из фразы GNU's not UNIX). Он хотел разработать полноценную, находящуюся в свободном доступе UNIX-подобную систему, состоящую из ядра и всех сопутствующих программных пакетов, и призвал присоединиться к нему всех остальных программистов. В 1985 году Столлман основал Фонд свободного программного обеспечения — Free Software Foundation (FSF), некоммерческую организацию для поддержки проекта GNU, а также для разработки совершенно свободного ПО.

> Когда был запущен проект GNU, в понятиях, введенных Столлманом, версия BSD не была свободной. Для использования BSD по-прежнему требовалось получить лицензию от AT&T, и пользователи не могли свободно изменять и распространять дальше код AT&T, формирующий часть BSD.

Одним из важных результатов появления проекта GNU была разработка общедоступной лицензии — *GNU General Public License (GPL)*. Она стала правовым воплощением представления Столлмана о свободном программном обеспечении. Большинство программных средств в дистрибутиве Linux, включая ядро, распространяются под лицензией GPL или одной из нескольких подобных лицензий. Программное обеспечение, распространяемое под лицензией GPL, должно быть доступно в форме исходного кода и должно предоставлять право дальнейшего распространения в соответствии с положениями GPL. Внесение изменений в программы, распространяемые под лицензией, не запрещено, но любое распространение такой измененной программы должно также производиться в соответствии с положениями о GPL-лицензировании. Если измененное программное средство распространяется в исполняемом виде, автор также должен дать всем получателям возможность приобрести измененные исходные коды с затратами, не дороже носителя, на котором они находятся. Первая версия GPL была выпущена в 1989 году. Текущая, третья

версия этой лицензии, выпущена в 2007 году. До сих пор используется и вторая версия, выпущенная в 1991 году: именно она применяется для ядра Linux. (Различные лицензии свободно распространяемого программного обеспечения рассматриваются в источниках [St. Laurent, 2004] и [Rosen, 2005].)

В рамках проекта GNU так и не было создано работающее ядро UNIX. Но под эгидой этого проекта разработано множество других разнообразных программ. Поскольку эти программы были созданы для работы под управлением UNIX-подобных операционных систем, они могут использоваться и используются на существующих реализациях UNIX и в некоторых случаях даже портируются на другие ОС. Среди наиболее известных программ, созданных в рамках проекта GNU, можно назвать текстовый редактор Emacs, пакет компиляторов GCC (изначально назывался компилятором GNU C, но теперь переименован в пакет GNU-компиляторов, содержащий компиляторы для C, C++ и других языков), оболочка bash и glibc (GNU-библиотека C).

В начале 1990-х годов в рамках проекта GNU была создана практически завершенная система, за исключением одного важного компонента: рабочего ядра UNIX. Проект GNU и Фонд свободного программного обеспечения начали работу над амбициозной конструкцией ядра, известной как GNU Hurd и основанной на микроядре Mach. Но ядро Hurd до сих пор находится не в том состоянии, чтобы его можно было выпустить. (На время написания этой книги работа над Hurd продолжалась и это ядро могло запускаться только на машинах с архитектурой x86-32.)

> Значительная часть программного кода, составляющего то, что обычно называют системой Linux, фактически была взята из проекта GNU, поэтому при ссылке на всю систему Столлман предпочитает использовать термин GNU/Linux. Вопрос, связанный с названием (Linux или GNU/Linux) стал причиной дебатов в сообществе разработчиков свободного программного обеспечения. Поскольку данная книга посвящена в основном API ядра Linux, в ней чаще всего будет использоваться термин Linux.

Начало было положено. Чтобы соответствовать полноценной UNIX-системе, созданной в рамках проекта GNU, требовалось только рабочее ядро.

1.2.2. Ядро Linux

В 1991 году Линус Торвальдс (Linus Torvalds), финский студент хельсинкского университета, задумал создать операционную систему для своего персонального компьютера с процессором Intel 80386. Во время учебы он имел дело с Minix, небольшим UNIX-подобным ядром операционной системы, разработанным в середине 1980-х годов Эндрю Таненбаумом (Andrew Tanenbaum), профессором голландского университета. Таненбаум распространял Minix вместе с исходным кодом как средство обучения проектированию ОС в рамках университетских курсов. Ядро Minix могло быть собрано и запущено в системе с процессором Intel 80386. Но, поскольку оно в первую очередь рассматривалось в качестве учебного пособия, ядро было разработано с прицелом на максимальную независимость от архитектуры аппаратной части и не использовало все преимущества, предоставляемые процессорами Intel 80386.

По этой причине Торвальдс приступил к созданию эффективного полнофункционального ядра UNIX для работы на машине с процессором Intel 80386. Через несколько месяцев он спроектировал основное ядро, позволявшее компилировать и запускать различные программы, разработанные в рамках проекта GNU. Затем, 5 октября 1991 года, Торвальдс обратился за помощью к другим программистам, анонсировав версию своего ядра под номером 0.02 в следующем, теперь уже широко известном (многократно процитированном) сообщении в новостной группе Usenet:

«Вы скорбите о тех временах, когда мужчины были настоящими мужчинами и сами писали драйверы устройств? У вас нет хорошего проекта и вы мечтаете вонзить свои зубы в какую-нибудь ОС, чтобы модифицировать ее для своих нужд? Вас раздражает то, что все работает под Minix? И не требуется просиживать ночи, чтобы заставить программу работать? Тогда это послание адресовано вам. Месяц назад я уже упоминал, что работаю над созданием свободной версии Minix-подобной операционной системы для компьютеров семейства AT-386. И вот наконец моя работа достигла той стадии, когда системой уже можно воспользоваться (хотя, может быть, и нет, все зависит от того, что именно вам нужно), и у меня появилось желание обнародовать исходный код для его свободного распространения. Пока это лишь версия 0.02..., но под ее управлением мне уже удалось вполне успешно запустить такие программные средства, как bash, gcc, gnu-make, gnu-sed, compress и так далее».

По сложившейся со временем традиции присваивать клонам UNIX имена, оканчивающиеся на букву X, ядро в конечном итоге получило название Linux. Изначально оно было выпущено под более ограничивающую лицензию, но вскоре Торвальдс сделал его доступным под лицензией GNU GPL.

Призыв к поддержке оказался эффективным. Для разработки Linux к Торвальдсу присоединились другие программисты. Они начали добавлять новую функциональность: усовершенствованную файловую систему, поддержку сетевых технологий, использование драйверов устройств и поддержку многопроцессорных систем. К марту 1994 года разработчики смогли выпустить версию 1.0. В марте 1995 года появилась версия Linux 1.2, в июне 1996 года — Linux 2.0, затем, в январе 1999 года, вышла версия Linux 2.2, а в январе 2001 года была выпущена версия Linux 2.4. Работа над созданием ядра версии 2.5 началась в ноябре 2001 года, что в декабре 2003 года привело к выпуску версии Linux 2.6.

Отступление: версии BSD

Следует заметить, что в начале 1990-х годов уже была доступна еще одна свободная версия UNIX для машин с архитектурой x86-32. Портированную на архитектуру x86-32 версию вполне состоявшейся к тому времени системы BSD под названием 386/BSD разработали Билл (Bill) и Линн Джолиц (Lynne Jolitz). Она была основана на выпуске BSD Net/2 (июнь 1991 года) — версии исходного кода 4.3BSD. В нем весь принадлежавший AT&T исходный код был либо заменен, либо удален, как в случае с шестью файлами, которые не так-то просто было переписать. При портировании кода Net/2 в код для архитектуры x86-32 Джолицы заново написали недостающие исходные файлы, и первый выпуск (версия 0.0) системы 386/BSD состоялся в феврале 1992 года.

После первой волны успеха и популярности работа над 386/BSD по различным причинам замедлилась. Вскоре появились две альтернативные группы разработчиков, которые создавали собственные выпуски на основе 386/BSD. Это были NetBSD, где основной упор был сделан на возможность портирования на широкий круг аппаратных платформ, и FreeBSD, созданный с прицелом на высокую производительность и получивший наиболее широкое распространение из всех современных версий BSD. Первый выпуск NetBSD под номером 0.8 состоялся в апреле 1993 года. Первый компакт-диск с FreeBSD (версии 1.0) появился в декабре 1993 года. Еще одна версия BSD под названием OpenBSD была выпущена в 1996 году (исходная версия вышла под номером 2.0) после ответвления от проекта NetBSD. В OpenBSD основное внимание уделялось безопасности. В середине 2003 года, после отделения от FreeBSD 4.x, появилась новая версия BSD — DragonFly BSD. Подход к ее разработке отличался от применявшегося при создании FreeBSD 5.x. Теперь особое внимание было уделено проектированию под архитектуры симметричной многопроцессорности (SMP).

Наверное, рассказ об истории BSD в начале 1990-х годов будет неполным без упоминания о судебных процессах между UNIX System Laboratories (USL, дочерней компании, принадлежащей AT&T и занимавшейся разработкой и рыночным продвижением UNIX) и командой из Беркли. В начале 1992 года компания Berkeley Software Design, Incorporated (BSDi, в настоящее время входит в состав Wind River) приступила к распространению сопровождаемых на коммерческой основе версий BSD UNIX под названием BSD/OS (на базе выпуска Net/2) и добавлений, разработанных Джолицами под названием 386/BSD. Компания BSDi распространяла двоичный и исходный код по цене $995 и советовала потенциальным клиентам пользоваться телефонным номером 1-800-ITS-UNIX.

В апреле 1992 года компания USL предъявила иск компании BSDi, пытаясь воспрепятствовать продаже этих проектов. Как заявлялось в USL, они по-прежнему представляли собой исходный код, который был защищен патентом, полученным USL, и составлял коммерческую тайну. Компания USL также потребовала, чтобы BSDi прекратила использовать вводящий в заблуждение телефонный номер. Со временем иск был выдвинут еще и Калифорнийскому университету. Суд в конечном итоге отклонил все, кроме двух претензий USL, а также встречный иск Калифорнийского университета к USL, в котором утверждалось, что USL не упомянула о том, что в System V содержится код BSD.

В ходе рассмотрения иска в суде USL была куплена компанией Novell, чей руководитель, ныне покойный Рэй Нурда (Ray Noorda), публично заявил, что он предпочел бы конкурировать на рынке, а не в суде. Спор окончательно был урегулирован в январе 1994 года. В итоге от Калифорнийского университета потребовали удалить из выпуска Net/2 три из 18 000 файлов, внести незначительные изменения в несколько файлов и добавить упоминание об авторских правах USL в отношении примерно 70 других файлов, которые университет тем не менее мог продолжать распространять на свободной основе. Эта измененная система была выпущена в июне 1994 года под названием 4.4BSD-Lite. (Последним выпуском университета в июне 1995 года был 4.4BSD-Lite, выпуск 2.) На данный момент по условиям правового урегулирования требуется, чтобы в BSDi, FreeBSD и NetBSD их база Net/2 была заменена исходным кодом 4.4BSD-Lite. Как отмечено в публикации [McKusick et al., 1996], хотя эти обстоятельства привели к замедлению процесса разработки версий, производных от BSD, был и положительный эффект. Он заключался в том, что эти системы были повторно синхронизированы с результатами трехлетней работы, проделанной университетской группой Computer Systems Research Group со времени выпуска Net/2.

Номера версий ядра Linux

Подобно большинству свободно распространяемых продуктов, для Linux практикуется модель ранних (release-early) и частых (release-often) выпусков, поэтому новые исправленные версии ядра появляются довольно часто (иногда чуть ли не каждый день). По мере расширения круга пользователей Linux каждая модель выпуска была настроена так, чтобы не влиять на тех, кто уже пользуется этой системой. В частности, после выпуска Linux 1.0 разработчики ядра приняли систему нумерации версий ядра *x.y.z*, где *x* обозначала номер основной версии, *y* — номер второстепенной версии в рамках основной версии, а *z* — номер пересмотра второстепенной версии (с незначительными улучшениями и исправлениями).

Согласно этой модели в разработке всегда находятся две версии ядра. Это *стабильная* ветка для использования в производственных системах, у которой имеется четный номер второстепенной версии, и более изменчивая *дорабатываемая* ветка, которая носит следующий более высокий нечетный номер второстепенной версии. По теории, которой

не всегда четко придерживаются на практике, все новые функции должны добавляться в текущие дорабатываемые серии ядра, а в новых редакциях стабильных серий нужно ограничиваться лишь незначительными улучшениями и исправлениями. Когда текущая дорабатываемая ветка оказывается подходящей для выпуска, она становится новой стабильной веткой и ей присваивается четный номер второстепенной версии. Например, дорабатываемая ветка ядра с номером 2.3.*z* в результате становится стабильной веткой ядра с номером 2.4.

После выпуска версии ядра с номером 2.6 модель разработки была изменена. Главной причиной для этого изменения послужили проблемы и недовольства, вызванные длительными периодами между выпусками стабильных версий ядра[1]. Вокруг доработки этой модели периодически возникали споры, но основными остались следующие характеристики[2].

- Версии ядер перестали делить на стабильные и дорабатываемые. Каждый новый выпуск 2.6.*z* может содержать новые функции. У выпуска есть жизненный цикл, начинающийся с добавления функций, которые затем стабилизируются в течение нескольких версий-кандидатов. Когда такие версии признают достаточно стабильными, их выпускают в качестве ядра 2.6.*z*. Между циклами выпуска обычно проходит около трех месяцев.
- Иногда в стабильный выпуск с номером 2.6.*z* требуется внести небольшие исправления для устранения недостатков или решения проблем безопасности. Если эти исправления важны и кажутся достаточно простыми, то разработчики не ждут следующего выпуска с номером 2.6.*z*, а вносят их, выпуская версию с номером вида 2.6.*z.r*. Здесь *r* является следующим номером для второстепенной редакции ядра, имеющего номер 2.6.*z*.
- Дополнительная ответственность за стабильность ядра, поставляемого в дистрибутиве, перекладывается на поставщиков этого дистрибутива.

В следующих главах иногда будут упоминаться версии ядра, в которых встречаются конкретные изменения API (например, новые или измененные системные вызовы). Хотя до выпуска серии 2.6.*z* большинство изменений ядра происходило в дорабатываемых ветвях с нечетной нумерацией, я буду в основном ссылаться на следующую стабильную версию ядра, в которой появились эти изменения. Ведь большинство разработчиков приложений, как правило, пользуются стабильной версией ядра, а не одним из ядер дорабатываемой версии. Во многих случаях на страницах руководств указывается именно то дорабатываемое ядро, в котором конкретная функция появилась или изменилась.

Для изменений, появившихся в серии ядра с номерами 2.6.*z*, я указываю точную версию ядра. Когда говорится, что функция является новой для ядра версии 2.6, без указания номера редакции *z*, имеется в виду функция, которая была реализована в дорабатываемых сериях ядра с номером 2.5 и впервые появилась в стабильной версии ядра 2.6.0.

---

[1] Между выпусками Linux 2.4.0 и 2.6.0 прошло почти три года.

[2] В результате перенумерации ядра Linux с 2.6.x на 3.x в июле 2011 года, а затем (в апреле 2015 года) в 4.x, обсуждение нумерации ядра на этой странице теперь устарело. Однако изменения коснулись лишь схемы нумерации: она была упрощена (инвариант 2.6 был заменен на 3, а впоследствии на 4). Модель разработки ядра остается неизменной. Как Линус Торвальдс отметил в версии 3.0, в релизе нет ничего особенного (то есть никаких более значительных изменений, чем изменения в Linux 2.6.39 и в каждом из предыдущих выпусков 2.6.x).

В представленном здесь списке каждый из экземпляров 2.6.*z* может быть просто заменен на 4.*z* и описание будет по-прежнему актуальным для текущей модели разработки ядра.

Портирование на другие аппаратные архитектуры

В начале разработки Linux главной целью было не достижение возможности портирования системы на другие вычислительные архитектуры, а создание работоспособной реализации под архитектуру Intel 80386. Но с ростом популярности Linux стала портироваться на другие архитектуры. Список аппаратных архитектур, на которые была портирована Linux, продолжает расти и включает в себя x86-64, Motorola/IBM PowerPC и PowerPC64, Sun SPARC и SPARC64 (UltraSPARC), MIPS, ARM (Acorn), IBM zSeries (бывшая System/390), Intel IA-64 (Itanium; см. публикацию [Mosberger & Eranian, 2002]), Hitachi SuperH, HP PA-RISC и Motorola 68000.

Дистрибутивы Linux

Если называть вещи своими именами, то название Linux относится лишь к ядру, разработанному Линусом Торвальдсом. И тем не менее, сам термин Linux обычно используется для обозначения ядра, а также широкого ассортимента других программных средств (инструментов и библиотек), которые в совокупности составляют полноценную операционную систему. На самых ранних этапах существования Linux пользователю требовалось собрать все эти инструменты воедино, создать файловую систему и правильно разместить и настроить в ней все программные средства. На это уходило довольно много времени и требовался определенный уровень квалификации. В результате появился рынок для распространителей Linux. Они проектировали *пакеты (дистрибутивы)* для автоматизации основной части процесса установки, создания файловой системы и установки ядра, а также других требуемых системе программных средств.

Самые первые дистрибутивы появились в 1992 году. Это были MCC Interim Linux (Manchester Computing Centre, UK), TAMU (Texas A&M University) и SLS (SoftLanding Linux System). Самый старый из выживших до сих пор коммерческих дистрибутивов, Slackware, появился в 1993 году. Примерно в то же время появился и некоммерческий дистрибутив Debian, за которым вскоре последовали SUSE и Red Hat. В настоящее время весьма большой популярностью пользуется дистрибутив Ubuntu, который впервые появился в 2004 году. Теперь многие компании-распространители также нанимают программистов, которые активно обновляют существующие проекты по разработке свободного ПО или инициируют новые проекты.

1.3. Стандартизация

В конце 1980-х годов начали проявляться негативные последствия имеющегося широкого разнообразия доступных реализаций UNIX. Одни реализации основывались на BSD, в то время как другие были созданы на основе System V, а у третьих функционал был позаимствован из обоих вариантов. Кроме того, каждый коммерческий распространитель добавлял к своей собственной реализации дополнительные функции. Все это привело к постепенному усложнению портирования программных продуктов и перехода людей с одной реализации UNIX на другую. Эта ситуация показала, что требовалась стандартизация языка программирования C и системы UNIX, чтобы упростить портирование приложений с одной системы на другую. Рассмотрим выработанные в итоге стандарты.

1.3.1. Язык программирования C

К началу 1980-х годов язык C существовал уже в течение 10 лет и был реализован во множестве разнообразных UNIX-систем, а также в других операционных системах. В некоторых

реализациях отмечались незначительные различия. В частности, это произошло из-за того, что определенные аспекты требуемого функционионала языка не были подробно описаны в существующем де-факто стандарте С. Этот стандарт приводился в вышедшей в 1978 году книге Кернигана (Kernighan) и Ритчи (Ritchie) «Язык программирования Си». (Синтаксис языка С, описанный в этой книге, иногда называют *традиционным С*, или *K&R C*.) Кроме того, с появлением в 1985 году языка C++ проявились конкретные улучшения и дополнения, которые могли быть привнесены в С без нарушения совместимости с существующими программами. В частности, сюда можно отнести прототипы функций, присваивание структур, спецификаторы типов (`const` и `volatile`), перечисляемые типы и ключевое слово `void`.

Эти факторы побудили к стандартизации С. Ее кульминацией в 1989 году стало утверждение Американским институтом национальных стандартов (ANSI) стандарта языка С (X3.159-1989), который в 1990 году был принят в качестве стандарта (ISO/IEC 9899:1990) Международной организацией по стандартизации (ISO). Наряду с определением синтаксиса и семантики языка С в этом стандарте давалось описание стандартной библиотеки С, включающей возможности stdio, функции обработки строк, математические функции, различные файлы заголовков и т. д. Эту версию С обычно называют C89 или (значительно реже) ISO C90, и она полностью рассмотрена во втором издании (1988 года) книги Кернигана и Ритчи «Язык программирования Си».

Пересмотренное издание стандарта языка С было принято ISO в 1999 году (ISO/IEC 9899:1999; см. http://www.open-std.org/jtc1/sc22/wg14/www/standards). Его обычно называют C99, и он включает несколько изменений языка и его стандартной библиотеки. В частности, там описаны добавление типов данных long long и логического (булева), присущий C++ стиль комментариев (`//`), ограниченные указатели и массивы переменной длины. Новый стандарт для языка Си (ISO/IEC 9899:2011) опубликован 8 декабря 2011. В нем описаны поддержка многопоточности, обобщенные макросы, анонимные структуры и объединения, статичные утверждения, функция `quick_exit`, новый режим эксклюзивного открытия файла и др.

Стандарты языка С не зависят от особенностей операционной системы, то есть они не привязаны к UNIX-системе. Это означает, что программы на языке С, для написания которых использовалась только стандартная библиотека С, должны быть портированы на любой компьютер и операционную систему, предоставляющую реализацию языка С.

> Исторически C89 часто называли ANSI C, и это название до сих пор иногда употребляется в таком значении. Например, оно используется в gcc, где спецификатор -ansi означает «поддерживать все программы ISO C90». Но мы будем избегать этого названия, поскольку теперь оно звучит несколько двусмысленно. После того как комитет ANSI принял пересмотренную версию C99, будет правильным считать, что стандартом ANSI C следует называть C99.

1.3.2. Первые стандарты POSIX

Термин POSIX (аббревиатура от Portable Operating System Interface) обозначает группу стандартов, разработанных под руководством Института инженеров электротехники и электроники — Institute of Electrical and Electronic Engineers (IEEE), а точнее, его комитета по стандартам для портируемых приложений — Portable Application Standards Committee (PASC, http://www.pasc.org/). Цель PASC-стандартов — содействие портируемости приложений на уровне исходного кода.

> Название POSIX было предложено Ричардом Столлманом (Richard Stallman). Последняя буква X появилась потому, что названия большинства вариантов UNIX заканчивались на X. В стандарте указывалось, что название должно произноситься как pahzicks, наподобие слова positive («положительный»).

Для нас в стандартах POSIX наибольший интерес представляет первый стандарт POSIX, который назывался POSIX.1 (или в более полном виде POSIX 1003.1), и последующий стандарт POSIX.2.

POSIX.1 и POSIX.2

POSIX.1 стал IEEE-стандартом в 1988 году и после небольшого количества пересмотров был принят как стандарт ISO в 1990 году (ISO/IEC 9945-1:1990). (Полных версий POSIX нет в свободном доступе, но их можно приобрести у IEEE на сайте http://www.ieee.org/.)

> POSIX.1 сначала основывался на более раннем (1984 года) неофициальном стандарте, выработанном объединением поставщиков UNIX под названием /usr/group.

В POSIX.1 документируется интерфейс прикладного программирования (API) для набора сервисов, которые должны быть доступны программам из соответствующей операционной системы. ОС, способная справиться с этой задачей, может быть сертифицирована в качестве совместимой с POSIX.1.

POSIX.1 основан на системном вызове UNIX и API библиотечных функций языка C, но при этом не требует, чтобы с этим интерфейсом была связана какая-либо конкретная реализация. Это означает, что интерфейс может быть реализован любой операционной системой и не обязательно UNIX. Фактически некоторые поставщики добавили API к своим собственным операционным системам, сделав их совместимыми с POSIX.1, в то же время оставив сами ОС в практически неизменном виде.

Большое значение приобрели также расширения исходного стандарта POSIX.1. Стандарт IEEE POSIX 1003.1b (POSIX.1b, ранее называвшийся POSIX.4 или POSIX 1003.4), одобренный в 1993 году, содержит расширения базового стандарта POSIX для работы в режиме реального времени. Стандарт IEEE POSIX 1003.1c (POSIX.1c), одобренный в 1995 году, содержит определения потоков в POSIX. В 1996 году была разработана пересмотренная версия стандарта POSIX.1 (ISO/IEC 9945-1:1996), основной текст которой остался без изменений, но в него были внесены дополнения, касающиеся работы в режиме реального времени и использования потоков. Стандарт IEEE POSIX 1003.1g (POSIX.1g) определил API для работы в сети, включая сокеты. Стандарт IEEE POSIX 1003.1d (POSIX.1d), одобренный в 1999 году, и POSIX.1j, одобренный в 2000 году, определили дополнительные расширения основного стандарта POSIX для работы в режиме реального времени.

> Расширения POSIX.1b для работы в режиме реального времени включают файловую синхронизацию, асинхронный ввод/вывод, диспетчеризацию процессов, высокоточные часы и таймеры, а также обмен данными между процессами с применением семафоров, совместно используемой памяти и очереди сообщений. Префикс POSIX часто применяется для трех методов обмена данными между процессами, чтобы их можно было отличить от похожих, но более старых методов реализации семафоров, совместного использования памяти и очередей сообщений из System V.

Родственный стандарт POSIX.2 (1992, ISO/IEC 9945-2:1993) затрагивает оболочку и различные утилиты UNIX, включая интерфейс командной строки компилятора кода языка C.

FIPS 151-1 и FIPS 151-2

FIPS является аббревиатурой от федерального стандарта обработки информации — Federal Information Processing Standard. Это название набора стандартов, разработанных правительством США и используемых гражданскими правительственными учреждениями. В 1989 году был опубликован стандарт FIPS 151-1, основанный на стандарте 1988 года IEEE

POSIX.1 и предварительной версии стандарта ANSI C. Основное отличие FIPS 151-1 от POSIX.1 (1988 года) заключалось в том, что по стандарту FIPS требовалось наличие кое-каких функций, которые в POSIX.1 оставались необязательными. Поскольку основным покупателем компьютерных систем было правительство США, большинство поставщиков обеспечили совместимость своих UNIX-систем с FIPS 151-1-версией стандарта POSIX.1.

Стандарт FIPS 151-2 совмещен с редакцией 1990 ISO стандарта POSIX.1, но в остальном остался без изменений. Уже устаревший FIPS 151-2 был отменен в феврале 2000 года.

1.3.3. X/Open Company и Open Group

X/Open Company представляла собой консорциум, основанный международной группой поставщиков UNIX. Он предназначался для принятия и внедрения существующих стандартов с целью выработки всеобъемлющего согласованного набора стандартов открытых систем. Им было выработано руководство *X/Open Portability Guide*, состоящее из серий руководств по обеспечению портируемости на базе стандартов POSIX. Первым весомым выпуском этого руководства в 1989 году стал документ под названием Issue 3 (XPG3), за которым в 1992 году последовал документ XPG4. Последний был пересмотрен в 1994 году, в результате чего появился XPG4 версии 2, стандарт, который также включал в себя важные части документа AT&T's System V Interface Definition Issue 3. Эта редакция также была известна как *Spec 1170*, где число 1170 соответствует количеству *интерфейсов* (функций, файлов заголовков и команд), определенных стандартом.

Когда компания Novell, которая в начале 1993 года приобрела у AT&T бизнес, связанный с системами UNIX, позже самоустранилась от него, она передала права на торговую марку UNIX консорциуму X/Open. (Планы по этой передаче были анонсированы в 1993 году, но в силу юридических требований передача прав была отложена до начала 1994 года.) Стандарт XPG4 версии 2 был перекомпонован в *единую UNIX-спецификацию — Single UNIX Specification* (SUS, иногда встречается вариант SUSv1), которая также известна под названием UNIX 95. Эта перекомпоновка включала XPG4 версии 2, спецификацию X/Open Curses Issue 4 версии 2 и спецификацию X/Open Networking Services (XNS) Issue 4. Версия 2 Single UNIX Specification (SUSv2, http://www.unix.org/version2/online.html) появилась в 1997 году, а реализация UNIX, сертифицированная на соответствие требованиям этой спецификации, может называть себя UNIX 98. (Данный стандарт иногда также называют XPG5.)

В 1996 году консорциум X/Open объединился с *Open Software Foundation* (OSF), в результате чего был сформирован консорциум *The Open Group*. В настоящее время в The Open Group, где продолжается разработка стандартов API, входят практически все компании или организации, имеющие отношение к системам UNIX.

> OSF был одним из двух консорциумов поставщиков, сформировавшихся в ходе UNIX-войн в конце 1980-х годов. Кроме прочих, в него входили Digital, IBM, HP, Apollo, Bull, Nixdorf и Siemens. OSF был сформирован главным образом в ответ на угрозы, вызванные бизнес-альянсом AT&T (изобретателей UNIX) и Sun (наиболее мощного игрока на рынке рабочих станций под управлением UNIX). В свою очередь, AT&T, Sun и другие компании сформировали конкурирующий консорциум UNIX International.

1.3.4. SUSv3 и POSIX.1-2001

Начиная с 1999 года IEEE, Open Group и ISO/IEC Joint Technical Committee 1 объединились в *Austin Common Standards Revision Group* (CSRG, http://www.opengroup.org/austin/) с целью пересмотра и утверждения стандартов POSIX и Single UNIX Specification. (Свое

название Austin Group получила потому, что ее первое заседание состоялось в городе Остин, штат Техас, в сентябре 1998 года.) В результате этого в декабре 2001 года был одобрен стандарт POSIX 1003.1-2001, иногда называемый просто POSIX.1-2001 (который впоследствии был утвержден в качестве ISO-стандарта ISO/IEC 9945:2002).

POSIX 1003.1-2001 заменил собой SUSv2, POSIX.1, POSIX.2 и ряд других более ранних стандартов POSIX. Этот стандарт также известен как Single UNIX Specification версии 3, и ссылки на него в книге будут в основном иметь вид *SUSv3*.

Базовая спецификация SUSv3 состоит почти из 3700 страниц, разбитых на следующие четыре части.

- *Base Definitions (XBD)*. Включает в себя определения, термины, положения и спецификации содержимого файлов заголовков. Всего предоставляются спецификации 84 файлов заголовков.
- *System Interfaces (XSH)*. Начинается с различной полезной справочной информации. В основном в ней содержатся спецификации разных функций (реализуемых либо в виде системных вызовов, либо в виде библиотечных функций в конкретной реализации UNIX). Всего в нее включено 1123 системных интерфейса.
- *Shell and Utilities (XCU)*. В этой части определяются возможности оболочки и различные команды (утилиты) UNIX. Всего в ней представлено 160 утилит.
- *Rationale (XRAT)*. Включает в себя текстовые сведения и объяснения, касающиеся предыдущих частей.

Кроме того, в SUSv3 входит спецификация X/Open CURSES Issue 4 версии 2 (XCURSES), в которой определяются 372 функции и три файла заголовков для API *curses*, предназначенного для управления экраном.

Всего в SUSv3 описано 1742 интерфейса. Для сравнения, в POSIX.1-1990 (с FIPS 151-2) определено 199 интерфейсов, а в POSIX.2-1992 — 130 утилит.

Спецификация SUSv3 доступна по адресу http://www.unix.org/version3/online.html. Реализации UNIX, сертифицированные в соответствии с требованиями SUSv3, имеют право называться *UNIX 03*.

В результате проблем, обнаруженных с момента одобрения исходного текста SUSv3, в него были внесены различные незначительные правки и уточнения. В итоге появилось техническое исправление номер 1 (Technical Corrigendum Number 1), уточнения из которого были внесены в редакцию SUSv3 от 2003 года, и техническое исправление номер 2 (Technical Corrigendum Number 2), уточнения из которого добавлены в редакцию 2004 года.

POSIX-соответствие, XSI-соответствие и XSI-расширение

Исторически стандарты SUS (и XPG) полагались на соответствующие стандарты POSIX и были структурированы как их функциональные расширенные варианты. Поскольку в стандартах SUS определялись дополнительные интерфейсы, эти стандарты сделали обязательными многие интерфейсы и особенности поведения, считавшиеся необязательными в POSIX.

С некоторыми нюансами эти различия сохраняются в POSIX 1003.1-2001, являющемся одновременно стандартом IEEE и Open Group Technical Standard (то есть, как уже было отмечено, он представляет собой объединение раннего POSIX и SUS). Этот документ определяет два уровня соответствия.

- *Соответствие POSIX:* задает основной уровень интерфейсов, который должен предоставляться реализацией, претендующей на соответствие. Допускает предоставление реализацией других необязательных интерфейсов.

- *Соответствие X/Open System Interface (XSI):* чтобы соответствовать XSI, реализация должна отвечать всем требованиям соответствия POSIX, а также предоставлять ряд интерфейсов и особенностей поведения, которые считаются для него необязательными. Реализация должна достичь этого уровня, чтобы получить от Open Group право называться UNIX 03.

Дополнительные интерфейсы и особенности поведения, требуемые для XSI-соответствия, обобщенно называются *XSI-расширением*. В их число входит поддержка потоков, функций `mmap()` и `munmap()`, API dlopen, ограничений ресурсов, псевдотерминалов, System V IPC, API syslog, функции `poll()`, учетных записей пользователей.

В дальнейшем, когда речь пойдет о SUSv3-соответствии, мы будем иметь в виду XSI-соответствие.

> Поскольку теперь POSIX и SUSv3 относятся к одному и тому же документу, дополнительные интерфейсы и перечень обязательных возможностей, требуемых для SUSv3, выделяются в тексте документа особым образом.

Неопределенные и слабо определенные интерфейсы

Временами вам будут попадаться ссылки на неопределенные или слабо определенные в SUSv3 интерфейсы.

Под *неопределенным* будет пониматься интерфейс, который не определяется в официальном стандарте, хотя упоминается в имеющихся справочных заметках или в тексте пояснений.

Когда говорится, что интерфейс *слабо определен*, подразумевается, что, хотя интерфейс включен в стандарт, важные подробности не определены (зачастую по причине того, что в комитете не достигли согласия из-за различий в существующих реализациях).

При использовании неопределенных или слабо определенных интерфейсов нельзя на 100 % гарантировать успешную портируемость приложений на другие реализации UNIX, а портируемые приложения не должны полагаться на поведение конкретной реализации. И все же в некоторых случаях подобные интерфейсы в различных реализациях достаточно согласованы, и о таких случаях я, как правило, буду писать отдельно.

Средства с пометкой LEGACY

Иногда какое-то средство в SUSv3 имеет пометку *LEGACY*. Она означает, что это средство оставлено для сохранения совместимости со старыми приложениями, а в новых приложениях его лучше не использовать. Во многих случаях существуют другие API, предоставляющие эквивалентные функциональные возможности.

1.3.5. SUSv4 и POSIX.1-2008

В 2008 году Austin Group завершила пересмотр объединенной спецификации POSIX.1 и Single UNIX. Как и предшествующая версия стандарта, она состоит из основной спецификации, дополненной XSI-расширением. Эту редакцию мы будем называть SUSv4.

Изменения в SUSv4 не столь масштабные, как в SUSv3. Из наиболее существенных можно выделить следующие.

- Добавлены новые спецификации для некоторых функций. Из их числа в книге упоминаются `dirfd()`, `fdopendir()`, `fexecve()`, `futimens()`, `mkdtemp()`, `psignal()`,

`strsignal()` и `utimensat()`. Другие новые функции предназначены для работы с файлами (например, `openat()`, рассматриваемая в разделе 18.11) и практически являются аналогами существующих функций (например, `open()`). Они отличаются лишь тем, что относительный путь к файлу разрешается относительно каталога, на который ссылается дескриптор открытого файла, а не относитльно текущего рабочего каталога процесса.

- Некоторые функции, указанные в SUSv3 как необязательные, становятся обязательной частью стандарта в SUSv4. Например, отдельные функции, составлявшие в SUSv3 часть XSI-расширения, в SUSv4 стали частью базового стандарта. Среди функций, ставших обязательными в SUSv4, можно назвать функции, входящие в API сигналов режима реального времени (раздел 22.8), в API POSIX-таймеров (раздел 23.6), в API dlopen (раздел 42.1) и в API POSIX-семафоров (глава 48).
- Кое-какие функции из SUSv3 в SUSv4 помечены как устаревшие. К их числу относятся `asctime()`, `ctime()`, `ftw()`, `gettimeofday()`, `getitimer()`, `setitimer()` и `siginterrupt()`.
- Спецификации некоторых функций, помеченных в SUSv3 как устаревшие, из SUSv4 удалены. Среди них `gethostbyname()`, `gethostbyaddr()` и `vfork()`.
- Различные особенности существующих в SUSv3 спецификаций претерпели изменения в SUSv4. Например, к списку функций, от которых требуется обеспечение безопасной обработки асинхронных сигналов, добавились дополнительные функции (см. табл. 21.1).

Далее в книге изменения в SUSv4, относящиеся к рассматриваемым вопросам, будут оговариваться специально.

1.3.6. Этапы развития стандартов UNIX

На рис. 1.1, где рассмотренные в предыдущих разделах стандарты расположены в хронологическом порядке, показано обобщенное представление об их взаимосвязи. Сплошными линиями на этой схеме обозначено прямое наследование стандартов, а прерывистыми стрелками показаны случаи, когда один стандарт, повлиявший на другой, был включен в качестве его части или же просто перенесен в него.

Ситуация с сетевыми стандартами была несколько сложнее. Действия по стандартизации в этой области начали предприниматься в конце 1980-х годов. В то время был образован комитет POSIX 1003.12 для стандартизации API сокетов, API X/Open Transport Interface (XTI) (альтернативный API программирования сетевых приложений на основе интерфейса транспортного уровня System V Transport Layer Interface) и различных API, связанных с работой в сети. Становление стандарта POSIX 1003.12 заняло несколько лет, в течение которых он был переименован в POSIX 1003.1g. Этот стандарт был одобрен в 2000 году.

Параллельно с разработкой POSIX 1003.1g в X/Open велась разработка спецификации X/Open Networking Specification (XNS). Первая ее версия, XNS, выпуск 4, была частью первой версии Single UNIX Specification. За ней последовала спецификация XNS, выпуск 5, которая составила часть SUSv2. По сути, XNS, выпуск 5, была такой же, как и текущая на то время предварительная версия (6.6) POSIX.1g. Затем последовала спецификация XNS, выпуск 5.2, отличавшаяся от XNS, выпуск 5, и был одобрен стандарт POSIX.1g. В нем был помечен устаревшим API XTI и включен обзор протокола Internet Protocol version 6 (IPv6), разработанного в середине 1990-х годов. XNS, выпуск 5.2, заложил основу для документации, относящейся к работе в сети и включенной в замененный нынче стандарт SUSv3. По аналогичным причинам POSIX.1g был отозван в качестве стандарта вскоре после своего одобрения.

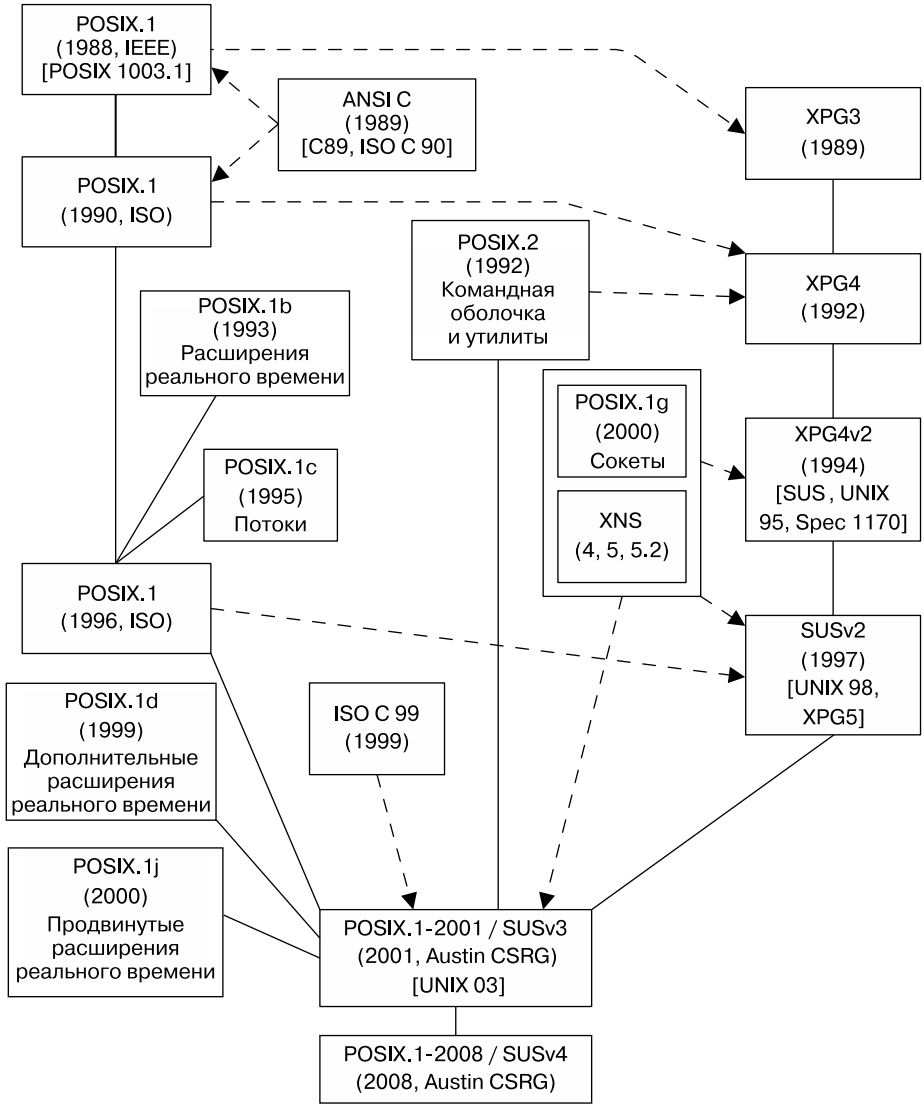


Рис. 1.1. Взаимоотношения между различными стандартами UNIX и C

1.3.7. Стандарты реализаций

В дополнение к стандартам, разработанным независимыми компаниями, иногда даются ссылки на два стандарта реализаций, определенных финальным выпуском BSD (4.4BSD) и AT&T's System V, выпуск 4 (SVR4). Последний стандарт реализации был документально оформлен публикацией System V Interface Definition (SVID) (компания AT&T). В 1989 году AT&T опубликовала выпуск 3 стандарта SVID, определявшего интерфейс, который должна предоставлять реализация UNIX, чтобы называться System V, выпуск 4. (В Интернете SVID можно найти по адресу http://www.sco.com/developers/devspecs/.)

Поскольку поведение некоторых системных вызовов и библиотечных функций в SVR4 и BSD различается, во многих реализациях UNIX предусмотрены библиотеки совместимости и средства условной компиляции. Они эмулируют поведение тех особенностей, которые не были включены в конкретную реализацию UNIX (см. подраздел 3.6.1). Тем самым облегчается портирование приложений из другой реализации UNIX.

1.3.8. Linux, стандарты и нормативная база Linux

В первую очередь разработчики Linux (то есть ядра, библиотеки glibc и инструментария) стремятся соответствовать различным стандартам UNIX, особенно POSIX и Single UNIX Specification. Но на время написания этих строк ни один из распространителей Linux не получил от Open Group права называться *UNIX*. Проблемы — во времени и средствах. Чтобы получить такое право, каждому дистрибутиву от поставщика необходимо пройти тестирование на соответствие, и с выпуском каждого нового дистрибутива требуется повторное тестирование. Тем не менее близкое соблюдение различных стандартов позволяет Linux успешно оставаться на рынке UNIX.

Что касается большинства коммерческих реализаций UNIX, разработкой и распространением операционной системы занимается одна и та же компания. А вот с Linux картина иная — реализация отделена от распространения, и распространением Linux занимаются многие организации: как коммерческие, так и некоммерческие.

Линус Торвальдс не занимается распространением или поддержкой какого-либо конкретного дистрибутива Linux. Но в отношении других отдельных разработчиков Linux ситуация иная. Многие разработчики, занимающиеся ядром Linux и другими проектами свободного программного обеспечения, являются сотрудниками различных компаний-распространителей Linux или работают на компании (такие как IBM и HP), испытывающие большой интерес к Linux. Хотя эти компании могут влиять на направление, в котором развивается Linux, выделяя программистам время на разработку конкретных проектов, ни одна из них не управляет операционной системой Linux как таковой. И конечно же, многие другие разработчики ядра Linux и GNU-проектов трудятся на добровольной основе.

На время написания этих строк Торвальдс числился сотрудником фонда Linux Foundation (http://www.linux-foundation.org/), бывшей лаборатории Open Source Development Laboratory, OSDL, некоммерческого консорциума организаций, уполномоченных оказывать содействие развитию Linux.

Из-за наличия нескольких распространителей Linux, а также из-за того, что реализаторы ядра не контролируют содержимое дистрибутивов, «стандартной» коммерческой версии Linux не существует. Ядро отдельного дистрибутива Linux обычно базируется на некоторой версии ядра Linux из основной ветки разработки (которую ведет Торвальдс) с набором необходимых изменений.

В этих изменениях (патчах) предоставляются функции, которые в той или иной степени считаются коммерчески востребованными и способными тем самым обеспечить конкурентные преимущества на рынке. Иногда эти исправления принимаются в качестве основной ветви разработки. Фактически некоторые новые функции ядра изначально были разработаны компаниями-распространителями и, прежде чем стать частью основной ветки, появились в их дистрибутивах. Например, версия 3 журналируемой файловой системы Reiserfs была частью ряда дистрибутивов Linux задолго до того, как была принята в качестве основной ветки версии 2.4.

Итогом всех ранее упомянутых обстоятельств стали в основном незначительные различия в системах, предлагаемых разными компаниями-распространителями Linux. Это напоминает расхождения в реализациях в начальные годы существования UNIX, но в существенно меньших масштабах. В результате усилий по обеспечению совместимости между разными дистрибутивами Linux появился стандарт под названием Linux Standard Base (LSB) (http://www.linux-foundation.org/en/LSB). В рамках LSB был разработан и внедрен набор стандартов для систем Linux. Они обеспечивают возможность запуска двоичных приложений (то есть скомпилированных программ) на любой LSB-совместимой системе.

Двоичная портируемость, внедренная с помощью LSB, отличается от портируемости исходного кода, внедренной стандартом POSIX. Портируемость исходного кода означает возможность написания программы на языке C с ее последующей успешной компиляцией и запуском на любой POSIX-совместимой системе. Двоичная совместимость имеет куда более привередливый характер, и, как правило, она недостижима на различных аппаратных платформах. Она позволяет осуществлять однократную компиляцию на конкретной аппаратной платформе, после чего запускать откомпилированную программу в любой совместимой реализации, запущенной на этой аппаратной платформе. Двоичная портируемость является весьма важным требованием для коммерческой жизнеспособности приложений, созданных под Linux независимыми поставщиками программных продуктов — independent software vendor (ISV).

1.4. Резюме

Впервые система UNIX была введена в эксплуатацию в 1969 году на мини-компьютере Digital PDP-7 Кеном Томпсоном из Bell Laboratories (подразделения AT&T). Множество идей было привнесено из ранее созданной системы MULTICS. К 1973 году UNIX была перенесена на мини-компьютер PDP-11 и переписана на C, языке программирования, разработанном и реализованном в Bell Laboratories Деннисом Ритчи (Dennis Ritchie). По закону не имея возможности продавать UNIX, компания AT&T за символическую плату распространяла полноценную систему среди университетов. Дистрибутив включал исходный код и стал весьма популярен в университетской среде. Это была недорогая операционная система, код которой можно было изучать и изменять как преподавателям, так и студентам, изучающим компьютерные технологии.

Ключевую роль в разработке UNIX сыграл Калифорнийский институт в Беркли. Там операционная система была расширена Кеном Томпсоном и несколькими студентами-выпускниками. К 1979 году университет создал собственный UNIX-дистрибутив под названием BSD. Он получил широкое распространение в академических кругах и стал основой для нескольких коммерческих реализаций.

Тем временем компания AT&T лишилась своего монопольного положения на рынке и занялась продажами системы UNIX. В результате появился еще один основной вариант UNIX под названием System V, который также послужил базой для нескольких коммерческих реализаций.

К разработке (GNU) Linux привели два разных проекта. Одним из них был GNU-проект, основанный Ричардом Столлманом. В конце 1980-х годов в рамках GNU была создана практически завершенная, свободно распространяемая реализация UNIX. Недоставало только работоспособного ядра. В 1991 году Линус Торвальдс, вдохновленный ядром Minix, придуманным Эндрю Таненбаумом, создал работоспособное ядро UNIX для архитектуры Intel x86-32. Торвальдс обратился за помощью к другим программистам для усовершенствования ядра. На его призыв откликнулось множество программистов,

и со временем система Linux была расширена и портирована на большое количество разнообразных аппаратных архитектур.

Проблемы портирования, возникшие из-за наличия разных реализаций UNIX и C, существовавших к концу 1980-х годов, сильно повлияли на решение вопросов стандартизации. Язык C прошел стандартизацию в 1989 году (C89), а пересмотренный стандарт вышел в 1999 году (C99). Первая попытка стандартизации интерфейса операционной системы привела к выпуску POSIX.1, одобренному в качестве стандарта IEEE в 1988 году и в качестве стандарта ISO в 1990 году. В 1990-е годы были разработаны дополнительные стандарты, включая различные версии спецификации Single UNIX Specification. В 2001 году был одобрен объединенный стандарт POSIX 1003.1-2001 и SUSv3. Этот стандарт собрал воедино и расширил различные более ранние стандарты POSIX и более ранние версии спецификации Single UNIX Specification. В 2008 году был завершен менее масштабный пересмотр стандарта, что привело к объединенному стандарту POSIX 1003.1-2008 и SUSv4.

В отличие от большинства коммерческих реализаций UNIX, в Linux реализация отделена от дистрибутива. Следовательно, не существует какого-либо «официального» дистрибутива Linux. Предложения каждого распространителя Linux состоят из какого-то варианта текущей стабильной версии ядра с добавлением различных усовершенствований. В рамках LSB разрабатывается и внедряется набор стандартов для систем Linux с целью обеспечения совместимости двоичных приложений в разных дистрибутивах Linux. Эти стандарты позволяют запускать откомпилированные приложения в любой LSB-совместимой системе, запущенной на точно таком же аппаратном оборудовании.

Дополнительная информация

Дополнительные сведения об истории и стандартах UNIX можно найти в публикациях [Ritchie, 1984], [McKusick et al., 1996], [McKusick & Neville-Neil, 2005], [Libes & Ressler, 1989], [Garfinkel et al., 2003], [Stevens & Rago, 2005], [Stevens, 1999], [Quartermann & Wilhelm, 1993], [Goodheart & Cox, 1994] и [McKusick, 1999].

В публикации [Salus, 1994] изложена подробная история UNIX, откуда и были почерпнуты основные сведения, приведенные в начале главы. В публикации [Salus, 2008] предоставлена краткая история Linux и других проектов по созданию свободных программных продуктов. Многие подробности истории UNIX можно также найти в выложенной в Интернет книге Ронды Хобен (Ronda Hauben) History of UNIX (http://www.dei.isep.ipp.pt/~acc/docs/unix.html). Весьма подробную историческую справку, относящуюся к выпускам различных реализаций UNIX, вы найдете по адресу http://www.levenez.com/unix/.

В публикации [Josey, 2004] дается обзорная информация по истории систем UNIX и разработке SUSv3, а также приводится руководство по использованию спецификации, сводные таблицы имеющихся в SUSv3 интерфейсов и пособие по переходу от SUSv2 к SUSv3 и от C89 к C99.

Наряду с предоставлением программных продуктов и документации, на сайте GNU (http://www.gnu.org/) содержится подборка философских статей, касающихся свободного программного обеспечения. А в публикации [Williams, 2002] дается биография Ричарда Столлмана.

Собственные взгляды Торвальдса на развитие Linux можно найти в публикации [Torvalds & Diamond, 2001].

2 Основные понятия

В этой главе вводится ряд понятий, имеющих отношение к системному программированию Linux. Она предназначена для тех читателей, которые работали в основном с другими операционными системами или имеют весьма небогатый опыт работы с Linux либо иными реализациями UNIX.

2.1. Основа операционной системы: ядро

Понятие *«операционная система»* зачастую употребляется в двух различных значениях.

- Для обозначения всего пакета, содержащего основные программные средства управления ресурсами компьютера и все сопроводительные стандартные программные инструменты: интерпретаторы командной строки, графические пользовательские интерфейсы, файловые утилиты и редакторы.
- В более узком смысле — для обозначения основных программных средств, управляющих ресурсами компьютера (например, центральным процессором, оперативной памятью и устройствами) и занимающихся их распределением.

В качестве синонима второго значения зачастую используется такое понятие, как *«ядро»*. Именно в этом смысле операционная система и будет рассматриваться в данной книге.

Хотя запуск программ на компьютере возможен и без ядра, его наличие существенно упрощает написание других программ и работу с ними, а также повышает доступную программистам эффективность и гибкость. Ядро выполняет эту задачу, предоставляя слой программного обеспечения для управления ограниченными ресурсами компьютера.

> Исполняемая программа ядра Linux обычно находится в каталоге с путевым именем /boot/vmlinuz или же в другом подобном ему каталоге. Происхождение этого имени имеет исторические корни. В ранних реализациях UNIX ядро называлось unix. В более поздних реализациях UNIX, работающих с виртуальной памятью, ядро было переименовано в vmunix. В Linux в имени файла отобразилось название системы, а вместо последней буквы x использована буква z. Это говорит о том, что ядро является сжатым исполняемым файлом.

Задачи, выполняемые ядром

Кроме всего прочего, в круг задач, выполняемых ядром, входят следующие.

- *Диспетчеризация процессов.* У компьютера имеется один или несколько центральных процессоров (CPU), выполняющих инструкции программ. Как и другие UNIX-системы, Linux является *многозадачной операционной системой с вытеснением*. *Многозадачность* означает, что несколько процессов (например, запущенные программы) могут одновременно находиться в памяти и каждая может получить в свое распоряжение центральный процессор (процессоры). *Вытеснение* означает, что правила, определяющие, какие именно процессы получают в свое распоряжение центральный

процессор (ЦП) и на какой срок, устанавливает имеющийся в ядре диспетчер процессов (а не сами процессы).

- *Управление памятью.* По меркам конца прошлого века объем памяти современного компьютера огромен, но и объем программ также соответственно увеличился. При этом физическая (оперативная) память осталась в разряде ограниченных ресурсов, которые ядро должно распределять между процессами справедливым и эффективным образом. Как и в большинстве современных операционных систем, в Linux используется управление виртуальной памятью (см. раздел 6.4) — технология, дающая два основных преимущества.
  - Процессы изолированы друг от друга и от ядра, поэтому один процесс не может читать или изменять содержимое памяти другого процесса или ядра.
  - В памяти требуется хранить только часть процесса, снижая таким образом объем памяти, требуемый каждому процессу и позволяя одновременно содержать в оперативной памяти большее количество процессов. Вследствие этого повышается эффективность использования центрального процессора, так как в результате увеличивается вероятность того, что в любой момент времени есть по крайней мере один процесс, который может быть выполнен центральным процессором (процессорами).
- *Предоставление файловой системы.* Ядро предоставляет файловую систему на диске, позволяя создавать, считывать обновлять, удалять файлы, выполнять их выборку и производить с ними другие действия.
- *Создание и завершение процессов.* Ядро может загрузить новую программу в память, предоставить ей ресурсы (например, центральный процессор, память и доступ к файлам), необходимые для работы. Такой экземпляр запущенной программы называется *процессом*. Как только выполнение процесса завершится, ядро обеспечивает высвобождение используемых им ресурсов для дальнейшего применения другими программами.
- *Доступ к устройствам.* Устройства (мыши, мониторы, клавиатуры, дисковые и ленточные накопители и т. д.), подключенные к компьютеру, позволяют обмениваться информацией между компьютером и внешним миром — осуществлять ввод/вывод данных. Ядро предоставляет программы с интерфейсом, упрощающим доступ к устройствам. Этот доступ происходит в рамках определенного стандарта. Одновременно с этим ядро распределяет доступ к каждому устройству со стороны нескольких процессов.
- *Работа в сети.* Ядро от имени пользовательских процессов отправляет и принимает сетевые сообщения (пакеты). Эта задача включает в себя маршрутизацию сетевых пакетов в направлении целевой операционной системы.
- *Предоставление интерфейса прикладного программирования (API) системных вызовов.* Процессы могут запрашивать у ядра выполнение различных задач с использованием точек входа в ядро, известных как *системные вызовы*. API системных вызовов Linux — главная тема данной книги. Этапы выполнения процессом системного вызова подробно описаны в разделе 3.1.

Кроме перечисленных выше свойств, такая многопользовательская операционная система, как Linux, обычно предоставляет пользователям абстракцию *виртуального персонального компьютера*. Иначе говоря, каждый пользователь может зайти в систему и работать в ней практически независимо от других. Например, у каждого пользователя имеется собственное дисковое пространство (домашний каталог). Кроме этого, пользователи могут запускать программы, каждая из которых получает свою долю времени центрального процессора и работает со своим виртуальным адресным пространством. Эти программы, в свою очередь, могут независимо друг от друга получать доступ к устрой-

ствам и передавать информацию по сети. Ядро занимается разрешением потенциальных конфликтов при доступе к ресурсам оборудования, поэтому пользователи и процессы обычно даже ничего о них не знают.

Режим ядра и пользовательский режим

Современные вычислительные архитектуры обычно позволяют центральному процессору работать как минимум в двух различных режимах: *пользовательском* и *режиме ядра* (который иногда называют *защищенным*). Аппаратные инструкции позволяют переключаться из одного режима в другой. Соответственно области виртуальной памяти могут быть помечены в качестве части *пользовательского пространства* или *пространства ядра*. При работе в пользовательском режиме ЦП может получать доступ только к той памяти, которая помечена в качестве памяти пользовательского пространства. Попытки обращения к памяти в пространстве ядра приводят к выдаче аппаратного исключения. При работе в режиме ядра центральный процессор может получать доступ как к пользовательскому пространству памяти, так и к пространству ядра.

Некоторые операции могут быть выполнены только при работе процессора в режиме ядра. Сюда можно отнести выполнение инструкции `halt` для остановки системы, обращение к оборудованию, занимающемуся управлением памятью, и инициирование операций ввода-вывода на устройствах. Используя эту конструктивную особенность оборудования для размещения операционной системы в пространстве ядра, разработчики ОС могут обеспечить невозможность доступа пользовательских процессов к инструкциям и структурам данных ядра или выполнения операций, которые могут отрицательно повлиять на работу системы.

Сравнение взглядов на систему со стороны процессов и со стороны ядра

Решая множество повседневных программных задач, мы привыкли думать о программировании, ориентируясь на процессы. Но, прежде чем рассматривать различные темы, освещаемые далее в этой книге, может быть полезно переориентировать свои взгляды на систему, став на сторону ядра. Чтобы контраст стал заметнее, рассмотрим, как все выглядит, сначала с точки зрения процесса, а затем с точки зрения ядра.

В работающей системе обычно выполняется множество процессов. Для процесса многое происходит асинхронно. Выполняемый процесс не знает, когда он будет приостановлен в следующий раз, для каких других процессов будет спланированно время центрального процессора (и в каком порядке) или когда в следующий раз это время будет спланировано для него. Передача сигналов и возникновение событий обмена данными между процессами осуществляются через ядро и могут произойти в любое время. Многое происходит незаметно для процесса. Он не знает, где находится в памяти, размещается ли конкретная часть его пространства памяти в самой оперативной памяти или же в области подкачки (в выделенной области дискового пространства, используемой для дополнения оперативной памяти компьютера). Точно так же процесс не знает, где на дисковом накопителе хранятся файлы, к которым он обращается, — он просто ссылается на файлы по имени. Процесс работает изолированно, он не может напрямую обмениваться данными с другим процессом. Он не может сам создать новый процесс или даже завершить свое собственное существование. И наконец, процесс не может напрямую обмениваться данными с устройствами ввода-вывода, подключенными к компьютеру.

С другой стороны, у работающей системы имеется всего одно ядро, которое обо всем знает и всем управляет. Ядро содействует выполнению всех процессов в системе. Оно решает, какой из процессов следующим получит доступ к центральному процессору, когда это произойдет и сколько продлится. Ядро обслуживает структуры данных, содержащие информацию обо всех запущенных процессах, и обновляет их по мере

создания процессов, изменения их состояния и прекращения их выполнения. Ядро обслуживает все низкоуровневые структуры данных, позволяющие преобразовывать имена файлов, используемые программами, в физические местоположения файлов на диске. Ядро также обслуживает структуры данных, которые отображают виртуальную память каждого процесса в физическую память компьютера и в область (области) подкачки на диске. Весь обмен данными между процессами осуществляется через механизмы, предоставляемые ядром. Отвечая на запросы процессов, ядро создает новые процессы и прекращает работу существующих. И наконец, ядро (в частности, драйверы устройств) выполняет весь непосредственный обмен данными с устройствами ввода-вывода, осуществляя по требованию перемещение информации в пользовательские процессы и из них в устройства.

Далее в книге будут встречаться фразы вроде «процесс может создавать другой процесс», «процесс может создать конвейер», «процесс может записывать данные в файл» и «процесс может останавливать свою работу путем вызова функции `exit()`». Но вам следует запомнить, что посредником во всех этих действиях является ядро, а такие утверждения — всего лишь сокращения фраз типа «процесс может *запросить у ядра* создание другого процесса» и т. д.

Дополнительная информация

В число современных публикаций, охватывающих концепции и конструкции операционных систем с конкретными ссылками на системы UNIX, входят труды [Tanenbaum, 2007], [Tanenbaum & Woodhull, 2006] и [Vahalia, 1996]. В последнем подробно описаны архитектуры виртуальной памяти. Издание [Goodheart & Cox, 1994] предоставляет подробную информацию, касающуюся System V Release 4. Публикация [Maxwell, 1999] содержит аннотированный перечень избранных частей ядра Linux 2.2.5. В издании [Lions, 1996] представлен детально разобранный исходный код Sixth Edition UNIX, который и сегодня остается полезным источником информации о внутреннем устройстве UNIX. В публикации [Bovet & Cesati, 2005] дается описание реализации ядра Linux 2.6.

2.2. Оболочка

Оболочка — это специальная программа, разработанная для чтения набранных пользователем команд и выполнения соответствующих программ в ответ на эти команды. Иногда такую программу называют *командным интерпретатором*.

Оболочкой входа в систему обозначают процесс, создаваемый для запуска оболочки при первом входе пользователя в систему.

В некоторых операционных системах командный интерпретатор является составной частью ядра, но в системах UNIX оболочка представляет собой пользовательский процесс. Существует множество различных оболочек, и несколько различных пользователей (или один пользователь) могут одновременно работать на одном компьютере с несколькими разными оболочками. Со временем выделились основные оболочки.

- *Bourne shell (sh)*. Эта оболочка, написанная Стивом Борном (Steve Bourne), является старейшей из широко используемых оболочек. Она была стандартной оболочкой для Seventh Edition UNIX. Bourne shell характеризуется множеством особенностей, актуальных для всех оболочек: перенаправление ввода-вывода, организация конвейеров, генерация имен файлов (подстановка), использование переменных, работа с переменными среды, подстановка команд, фоновое выполнение команд и функций. Все последующие реализации UNIX включали Bourne shell в дополнение к любым другим оболочкам, которые они могли предоставлять.

- *C shell (csh)*. Была написана Биллом Джоем (Bill Joy) из Калифорнийского университета в Беркли. Такое имя она получила из-за схожести многих конструкций управления выполнением этой оболочки с конструкциями языка программирования C. Оболочка C shell предоставляет ряд полезных интерактивных средств, недоступных в Bourne shell, включая историю команд, управление заданиями и использование псевдонимов. Оболочка C shell не имеет обратной совместимости с Bourne shell. Хотя стандартной интерактивной оболочкой на BSD была C shell, сценарии оболочки (которые вскоре будут рассмотрены) обычно создавались для Bourne shell, дабы сохранялась их портируемость между всеми реализациями UNIX.
- *Korn shell (ksh)*. Оболочка была написана в качестве преемника Bourne shell Дэвидом Корном (David Korn) из AT&T Bell Laboratories. Кроме поддержки обратной совместимости с Bourne shell, в нее были включены интерактивные средства, подобные предоставляемым оболочкой C shell.
- *Bourne again shell (bash)*. Была разработана в рамках проекта GNU в качестве усовершенствованной реализации Bourne shell. Она предоставляет интерактивные средства, подобные тем, что доступны при работе с оболочками C и Korn. Основными создателями bash являются Брайан Фокс (Brian Fox) и Чет Рэми (Chet Ramey). Bash, наверное, наиболее популярная оболочка Linux. (Фактически в Linux Bourne shell, sh, предоставляется посредством имеющейся в bash наиболее приближенной к оригиналу эмуляции оболочки sh.)

  В POSIX.2-1992 определяется стандарт для оболочки, которая была основана на актуальной в ту пору версии оболочки Korn. В наши дни стандарту POSIX соответствуют обе оболочки: и Korn shell и bash, но при этом они предоставляют несколько расширений стандарта и отличаются друг от друга многими из этих расширений.

Оболочки разработаны не только для использования в интерактивном режиме, но и для выполнения в режиме интерпретации *сценариев оболочки*. Эти сценарии представляют собой текстовые файлы, содержащие команды оболочки. Для этого каждая из оболочек имеет элементы, обычно присущие языкам программирования: переменные, циклы, условные инструкции, команды ввода-вывода и функции.

Все оболочки выполняют схожие задачи, хотя и имеют отличающийся синтаксис. При описании в этой книге операций оболочки, как правило, будет подразумеваться, что таким образом работают все оболочки, если отдельно не встретится ссылка на операцию конкретной оболочки. В большинстве примеров, требующих применения оболочки, используется bash, но, пока не будет утверждаться обратное, считайте, что эти примеры работают точно так же и на других оболочках Bourne-типа.

2.3. Пользователи и группы

Для каждого пользователя системы предусмотрена уникальная идентификация. Кроме того, пользователи могут принадлежать к группам.

Пользователи

У каждого пользователя имеется уникальное *имя для входа в систему* (имя пользователя) и соответствующий *числовой идентификатор пользователя* — numeric user ID (UID). Каждому пользователю соответствует своя строка в *файле паролей системы*, `/etc/passwd`, где прописаны эти сведения, а также следующая дополнительная информация.

- *Идентификатор группы (Group ID, GID)* — числовой идентификатор группы, к которой принадлежит пользователь.
- *Домашний каталог* — исходный каталог, в который пользователь попадает после входа в систему.
- *Оболочка входа в систему* — имя программы, выполняемой для интерпретации команд пользователя.

Парольная запись может также включать в закодированном виде пароль пользователя. Но в целях безопасности пароль зачастую хранится в отдельном *теневом файле паролей*, прочитать который могут только привилегированные пользователи.

Группы

В целях администрирования, в частности для управления доступом к файлам и другим системным ресурсам, есть смысл собрать пользователей в *группы*. Например, всех специалистов в команде, работающей над одним проектом и пользующейся по этой причине одним и тем же набором файлов, можно свести в одну группу. В ранних реализациях UNIX пользователь мог входить только в одну группу. В версии BSD пользователю позволялось одновременно принадлежать сразу нескольким группам, и эта идея была подхвачена создателями других реализаций UNIX, а также поддержана стандартом POSIX.1-1990. Каждая группа обозначается одной строкой в *системном файле групп*, `/etc/group`, включающем следующую информацию.

- *Название группы* — уникальное имя группы.
- *Идентификатор группы (Group ID, GID)* — числовой идентификатор, связанный с данной группой.
- *Список пользователей* — список с запятыми в качестве разделителей, содержащий имена пользователей, входящих в группу (которые не идентифицированы как участники группы в поле идентификатора группы в своей записи в файле паролей).

Привилегированный пользователь

Один из пользователей, называемый *привилегированным (superuser)*, имеет в системе особые привилегии. У учетной записи привилегированного пользователя UID содержит значение `0`, и, как правило, в качестве имени пользователя применяется слово `root`. В обычных системах UNIX привилегированный пользователь обходит в системе все разрешительные проверки. Таким образом, к примеру, привилегированный пользователь может получить доступ к любому файлу в системе независимо от требуемых для этого разрешений и может отправлять сигналы любому имеющемуся в системе пользовательскому процессу. Системный администратор пользуется учетной записью привилегированного пользователя для выполнения различных задач администрирования системы.

2.4. Иерархия одного каталога. Что такое каталоги, ссылки и файлы

Для организации всех файлов в системе ядро поддерживает структуру одного иерархического каталога. (В отличие от таких операционных систем, как Microsoft Windows, где своя собственная иерархия каталогов имеется у каждого дискового устройства.) Основу этой иерархии составляет *корневой каталог* по имени / (слеш). Все файлы и каталоги являются дочерними или более отдаленными потомками корневого каталога. Пример такой иерархической файловой структуры показан на рис. 2.1.

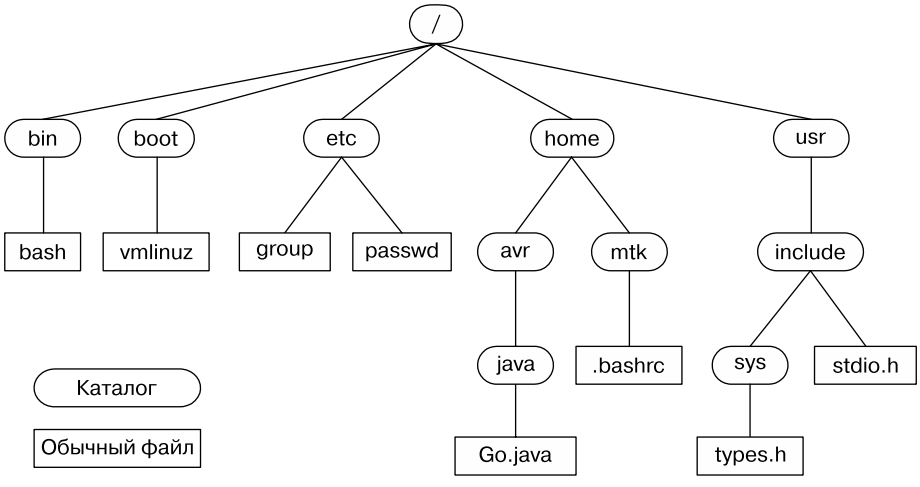


Рис. 2.1. Пример иерархии одного каталога Linux

Типы файлов

Внутри файловой системы каждый файл имеет метку, указывающую, к какому *типу* файлов он относится. Один из этих типов файлов обозначает стандартные файлы данных, которые чаще всего называют *обычными* или *простыми* файлами, чтобы отличить их от файлов других типов. Другие типы файлов включают в себя устройства, конвейеры, сокеты, каталоги и символьные ссылки.

Термин «*файл*» обычно используется для обозначения файла любого типа, а не только обычного файла.

Каталоги и ссылки

Каталог — это особый файл, чье содержимое принимает форму таблицы из имен файлов в совокупности с указателями на соответствующие файлы. Эта связка из имени файла и указателя на него называется *ссылкой*, и у файлов в одном и том же или в разных каталогах может быть несколько ссылок, а следовательно, и несколько имен.

Каталоги могут содержать ссылки как на файлы, так и на другие каталоги. С помощью ссылок между каталогами устанавливается иерархия каталогов, показанная на рис. 2.1.

Каждый каталог содержит как минимум две записи: `.` (точка), которая представляет собой ссылку на сам каталог, и `..` (точка-точка), которая является ссылкой на его родительский каталог — тот каталог, что расположен над ним в иерархии. Каждый каталог, за исключением корневого, имеет свой *родительский каталог*. Для корневого каталога запись `..` является ссылкой на него самого (таким образом, обозначение `/..` — то же самое, что и `/`).

Символьные ссылки

Подобно обычной ссылке, *символьная ссылка* предоставляет альтернативное имя для файла. Но, в отличие от обычной ссылки, представляющей собой в списке каталога запись вида «имя файла плюс указатель», символьная ссылка — это специально помеченный файл, содержащий имя другого файла. (Иными словами, у символьной ссылки в каталоге есть запись вида «имя файла плюс указатель», и файл, на который ссылается указатель, содержит строку с именем другого файла.) Этот последний файл часто называют *целью* символьной ссылки, и зачастую говорится, что символьная ссылка «указывает» или

«ссылается» на целевой файл. Когда в системном вызове указывается путевое имя, в большинстве случаев ядро автоматически снимает *косвенность* каждой символьной ссылки в путевом имени (также говорят «*следует* по ним»), заменяя ее именем того файла, на который она ведет. Этот процесс может происходить рекурсивно, если цель символьной ссылки сама по себе является символьной ссылкой. (Ядро накладывает ограничение на количество ссылок, чтобы предотвратить возможность появления замкнутых цепочек символьных ссылок.) Если символьная ссылка указывает на несуществующий файл, то говорится, что это *битая ссылка*.

В качестве альтернативных названий для обычной и символьной ссылки зачастую используются выражения *«жесткая ссылка»* и *«мягкая ссылка»*. Смысл наличия двух разных типов ссылок объясняется в главе 18.

Имена файлов

В большинстве файловых систем Linux длина имен файлов может составлять до 255 символов. Имена файлов могут содержать любые символы, за исключением слешей (/) и символа с нулевым кодом (\0). Но желательно использовать только буквы и цифры, а также символы точки (.), подчеркивания (_) и дефиса (-). Этот 65-символьный набор, [-._a-zA-Z0-9], в SUSv3 называется *портируемым набором символов для имен файлов*.

Следует избегать использования в именах файлов символов, не входящих в портируемый набор, поскольку эти символы могут иметь специальное значение в оболочке, внутри регулярных выражений или в других контекстах. Если имя файла с символами, имеющими специальное значение, появляется в таких контекстах, эти символы должны быть *экранированы*, то есть специально помечены. Для этого обычно перед ними добавляют обратный слеш (\), который показывает, что они не должны быть интерпретированы в их специальном значении. В контекстах, где механизм экранирования недоступен, такое имя файла применять нельзя.

Кроме того, следует избегать ситуаций, когда в связке *команда имя_файла* имя файла начинается с -, так как оно может быть ошибочно принято за ключ *команды*.

Путевые имена

Путевое имя представляет собой строку, содержащую символ слеша (/) (опционально), за которым следуют серии имен файлов, также отделенных друг от друга слешами. Все эти компоненты имен файлов, за исключением последнего, идентифицируют каталог (или символьную ссылку, указывающую на каталог). Последний компонент путевого имени может идентифицировать любой тип файла, включая каталог. Серия компонентов из имен файлов, предшествующая завершающему слешу, иногда называется *каталожной* частью путевого имени, а имя, следующее за последним слешем, обычно называют файлом или *базовой* частью путевого имени.

Путевое имя считывается слева направо, каждое имя файла находится в каталоге, указанном в предыдущей части путевого имени. Строка `..` может использоваться в любом месте путевого имени для указания на родительский каталог того места, которое до этих пор было задано в путевом имени.

Путевое имя описывает местоположение файла в иерархии одного каталога и является либо абсолютным, либо относительным.

- *Абсолютное путевое имя* начинается со слеша и указывает на местоположение файла относительно корневого каталога. Примеры абсолютного путевого имени для файлов, показанных на рис. 2.1: `/home/mtk/.bashrc`, `/usr/include` и `/` (путевое имя корневого каталога).

- *Относительное путевое имя* указывает местоположение файла относительно рабочего каталога текущего запущенного процесса (см. ниже) и отличается от абсолютного путевого имени отсутствием начального слеша. На рис. 2.1 из каталога `usr` на файл `types.h` можно указать, используя относительное путевое имя `include/sys/types.h`, а из каталога `avr` доступ к файлу `.bashrc` можно получить с помощью относительного путевого имени `../mtk/.bashrc`.

Текущий рабочий каталог

У каждого процесса есть свой *текущий рабочий каталог* (который иногда называют просто *рабочим* или *текущим*). Это «текущее местоположение» процесса в иерархии одного каталога, и именно с данного каталога для процесса интерпретируются относительные путевые имена.

Процесс наследует свой текущий рабочий каталог от родительского процесса. В случае входа в систему для оболочки рабочим каталогом является домашний каталог пользователя, который указан в его записи в файле паролей. Текущий рабочий каталог оболочки может быть изменен с помощью команды `cd`.

Владение файлами и права доступа

С каждым файлом связаны UID и GID, определяющие владельца этого файла и группу, к которой он принадлежит. Понятие «владение файлом» применяется для определения прав доступа пользователей к файлу.

Для организации доступа к файлу система делит пользователей на три категории: на *владельца* файла (иногда называемого *пользователем* файла), пользователей, входящих в группу, соответствующую идентификатору *группы (группу) файла*, и всех остальных *(других пользователей)*. Для каждой из этих категорий пользователей могут быть установлены три бита прав доступа (что всего составляет девять бит прав доступа):

- *права доступа на чтение* позволяют считывать содержимое файла;
- *права доступа на запись* дают возможность вносить изменения в содержимое файла;
- *права доступа на выполнение* позволяют выполнять файл, который является либо программой, либо сценарием, обрабатываемым каким-нибудь интерпретатором (обычно, но не всегда им оказывается одна из оболочек).

Эти права доступа могут быть установлены и для каталогов, хотя их значения несколько отличаются:

- *права доступа на чтение* позволяют выводить список содержимого каталога (например, список имен файлов);
- *права доступа на запись* дают возможность изменять содержимое каталога (например, можно добавлять имена файлов, заниматься их перемещением и изменением);
- *права доступа на выполнение* (иногда называемые *поиском*) позволяют получить доступ к файлам внутри каталога (с учетом прав доступа к самим файлам).

2.5. Модель файлового ввода-вывода

Одной из отличительных черт модели ввода-вывода в системах UNIX является понятие *универсальности ввода-вывода*. Это означает, что одни и те же системные вызовы (`open()`, `read()`, `write()`, `close()` и т. д.) используются для выполнения ввода-вывода во всех типах файлов, включая устройства. (Ядро преобразует запросы приложений на ввод/вывод в соответствующие операции файловой системы или драйверов устройств, выполняющие

ввод/вывод в отношении целевого файла или устройства.) Из этого следует, что программа, использующая эти системные вызовы, будет работать с любым типом файлов.

По сути, ядро выдает один тип файла: последовательный поток байтов, к которому, в случае файлов на дисках, дисков и ленточных накопителей, можно получить произвольный доступ с помощью системного вызова `lseek()`.

Многие приложения и библиотеки интерпретируют *символ новой строки*, или *символ конца строки* (имеющий десятичный ASCII-код 10) как завершающий одну строку текста и начинающий другую строку. В системах UNIX отсутствует *символ конца файла*, и конец файла определяется при чтении, не возвращающем данные.

Файловый дескриптор

Системные вызовы ввода-вывода ссылаются на открытый файл с использованием *файлового дескриптора*, представляющего собой неотрицательное (обычно небольшое) целое число. Файловый дескриптор обычно возвращается из выдачи системного вызова `open()`, который получает в качестве аргумента путевое имя, а оно, в свою очередь, указывает на файл, в отношении которого будут выполняться операции ввода-вывода.

При запуске оболочкой процесс наследует, как правило, три дескриптора открытых файлов:

- дескриптор 0 является *стандартным вводом* — файлом, из которого процесс получает свой ввод;
- дескриптор 1 является *стандартным выводом* — файлом, в который процесс записывает свой вывод;
- дескриптор 2, являющийся *стандартной ошибкой*, — файлом, в который процесс записывает сообщения об ошибках и уведомления об исключительных и нештатных условиях.

В интерактивной оболочке или программе эти три дескриптора подключены, как правило, к терминалу. В библиотеке stdio они соответствуют файловым потокам `stdin`, `stdout` и `stderr`.

Библиотека stdio

Для выполнения файлового ввода-вывода программы обычно используют функции ввода-вывода, содержащиеся в стандартной библиотеке языка C. Этот набор функций, известный как библиотека stdio, включает функции `fopen()`, `fclose()`, `scanf()`, `printf()`, `fgets()`, `fputs()` и т. д. Функции stdio наслаиваются поверх системных вызовов ввода-вывода (`open()`, `close()`, `read()`, `write()` и т. д.).

> Предполагается, что читатель уже знаком со стандартными функциями ввода-вывода (stdio) языка C, поэтому мы не рассматриваем их в данной книге. Дополнительные сведения о библиотеке stdio можно найти в изданиях [Kernighan & Ritchie, 1988], [Harbison & Steele, 2002], [Plauger, 1992] и [Stevens & Rago, 2005].

2.6. Программы

Программы обычно существуют в двух формах. Первая форма представляет собой *исходный код* — понятный человеку текст, состоящий из серий инструкций, написанных на языке программирования, например на C. Чтобы стать исполняемым, исходный код должен быть преобразован во вторую форму: двоичные (бинарные) инструкции на языке машины, понятные для компьютера. (В отличие от *сценария*, являющегося текстовым файлом с командами, напрямую обрабатываемыми программой, такой как оболочка или

интерпретатор команд.) Два значения понятия *«программы»* обычно считаются синонимами, так как в процессе компиляции и сборки исходный код преобразуется в семантически эквивалентный двоичный машинный код.

Фильтры

Понятие *«фильтр»* часто обозначает программу, которая считывает вводимые в нее данные из stdin, выполняет преобразования этого ввода и записывает преобразованные данные на stdout. Примеры фильтров: cat, grep, tr, sort, wc, sed и awk.

Аргументы командной строки

В языке C программы могут получать доступ к *аргументам командной строки* — словам, введенным в командную строку при запуске программы. Для доступа к аргументам командной строки глобальная функция `main()` программы объявляется следующим образом:

```
int main(int argc, char *argv[])
```

Переменная `argc` содержит общее количество аргументов командной строки, а отдельные аргументы доступны в виде строковых значений, которые нужно указать в качестве элементов массива `argv`. Первая из этих строк, `argv[0]`, соответствует имени самой программы.

2.7. Процессы

Говоря простым языком, *процесс* представляет собой экземпляр выполняемой программы. Когда программа выполняется, ядро загружает ее код в виртуальную память, выделяет память под переменные программы и определяет учетные структуры данных ядра для записи различной информации о процессе (имеются в виду идентификатор процесса, код завершения, пользовательские и групповые идентификаторы).

С точки зрения ядра процессы являются объектами, между которыми ядро должно делить различные ресурсы компьютера. В случае с ограниченными ресурсами, например памятью, ядро изначально выделяет некоторый их объем процессу и регулирует это выделение в ходе жизненного цикла процесса, реагируя на потребности процесса и общие потребности системы в этом ресурсе. Когда процесс завершается, все такие ресурсы высвобождаются для повторного использования другими процессами. Другие ресурсы, такие как время центрального процессора и сетевой трафик, являются возобновляемыми, но должны быть поровну поделены между всеми процессами.

Модель памяти процесса

Процесс логически делится на следующие части, известные как *сегменты*.

- *Текст* — инструкции программы.
- *Данные* — статические переменные, используемые программой.
- *Динамическая память (куча)* — область, из которой программа может динамически выделять дополнительную память.
- *Стек* — часть памяти, которая может расширяться и сжиматься по мере вызова функций и возвращения из них и которая используется для выделения хранилища под локальные переменные и информацию о взаимосвязанности вызовов функций.

Создание процесса и выполнение программы

Процесс может создать новый процесс с помощью системного вызова `fork()`. Процесс, вызывающий `fork()`, известен как *родительский процесс*, а новый процесс называется *дочерним процессом*. Ядро создает дочерний процесс путем изготовления дубликата родительского

процесса. Дочерний процесс наследует копии родительских сегментов данных, стека и кучи, которые затем могут изменяться независимо от своих родительских копий. (Текст программы размещается в области памяти с пометкой «только для чтения» и совместно используется двумя процессами.)

Дочерний процесс запускается либо для выполнения другого набора функций в том же самом коде, что и у родительского процесса, либо зачастую для использования системного вызова `execve()` с целью загрузки и выполнения совершенно новой программы. Вызов `execve()` удаляет существующие сегменты текста, данных, стека и кучи, заменяя их новыми сегментами, основываясь на коде новой программы.

У вызова `execve()` есть ряд надстроек в виде родственных функций библиотеки языка C с несколько отличающимся интерфейсом, но сходной функциональностью. У всех этих функций имена начинаются со строки `exec`. (В тех случаях, когда разница между ними неважна, мы будем для общей ссылки на эти функции использовать обозначение `exec()`. И все же следует иметь в виду, что на самом деле функции по имени `exec()` не существует.)

В основном глагол «выполнять» (exec) будет употребляться для описания операций, выполняемых `execve()` и ее библиотечными функциями-надстройками.

Идентификатор процесса и идентификатор родительского процесса

У каждого процесса есть уникальный целочисленный *идентификатор процесса (PID)*. У каждого процесса также есть атрибут *идентификатора родительского процесса (PPID)*, идентифицирующий процесс, запросивший у ядра создание данного процесса.

Завершение процесса и код завершения

Процесс может быть завершен двумя способами: запросом своего собственного завершения с использованием системного вызова `_exit()` (или родственной ему библиотечной функции `exit()`) или путем его уничтожения извне с помощью сигнала. В любом случае процесс выдает *код завершения*, небольшое неотрицательное целое число, которое может быть проверено родительским процессом с использованием системного вызова `wait()`. В случае вызова `_exit()` процесс явным образом указывает свой собственный код завершения. Если процесс уничтожается сигналом, код завершения устанавливается по типу сигнала, уничтожившего процесс. (Иногда мы будем называть аргумент, передаваемый `_exit()`, *кодом выхода* процесса, чтобы отличить его от кода завершения, который является либо значением, переданным `_exit()`, либо указателем на сигнал, уничтоживший процесс.)

По соглашению, код завершения `0` служит признаком успешного завершения процесса, а ненулевое значение служит признаком возникновения какой-то ошибки. Большинство оболочек позволяют получить код завершения последней выполненной программы с помощью переменной оболочки по имени `$?`.

Принадлежащие процессу идентификаторы пользователя и группы (учетные данные)

У каждого процесса имеется несколько связанных с ним идентификаторов пользователей (UID) и групп (GID). К ним относятся следующие.

- *Реальный идентификатор пользователя* и *реальный идентификатор группы*. Они идентифицируют пользователя и группу, которым принадлежит процесс. Новый процесс наследует эти идентификаторы (ID) от своего родительского процесса. Оболочка входа в систему получает свой реальный UID и реальный GID от соответствующих полей в системном файле паролей.

- *Действующий идентификатор пользователя* и *действующий идентификатор группы*. Эти два идентификатора (в сочетании с рассматриваемыми сразу после них дополнительными идентификаторами групп) используются при определении прав доступа, имеющихся у процесса при доступе к защищенным ресурсам, таким как файлы и объекты обмена данными между процессами. Обычно имеющиеся у процессов действующие идентификаторы содержат те же значения, что и соответствующие им реальные ID. При изменении действующих идентификаторов процессу можно присваивать права доступа другого пользователя или группы, в порядке, который вскоре будет рассмотрен.
- *Дополнительные идентификаторы группы*. Они позволяют определить дополнительные группы, которым принадлежит процесс. Новый процесс наследует свои дополнительные идентификаторы групп от своего родительского процесса. Оболочка входа в систему получает свои дополнительные идентификаторы групп от системного файла групп.

Привилегированные процессы

Традиционно в системах UNIX *привилегированным* считается процесс, чей *действующий* идентификатор пользователя имеет значение `0` (привилегированный пользователь, суперпользователь). Такой процесс обходит ограничения прав доступа, обычно применяемые ядром. И наоборот, *непривилегированным* называется процесс, запущенный другими пользователями. Такие процессы имеют ненулевой действующий UID и должны соблюдать навязываемые ядром правила разрешения доступа.

Процесс может быть привилегированным из-за того, что был создан другим привилегированным процессом, например оболочкой входа в систему, запущенной суперпользователем *(root)*. Еще один способ получения процессом привилегированности связан с механизмом установки идентификатора пользователя (set-user-ID), который позволяет присвоить процессу такой же действующий идентификатор пользователя, как и идентификатор пользователя файла выполняемой программы.

Мандаты (возможности)

Начиная с ядра версии 2.2, Linux делит привилегии, традиционно предоставляемые суперпользователю, на множество отдельных частей, называемых *возможностями*. Каждая привилегированная операция связана с конкретной возможностью, и процесс может выполнить операцию, только если у него имеется соответствующая возможность. Традиционный привилегированный процесс (с действующим идентификатором пользователя, равным `0`) соответствует процессу со всеми включенными возможностями.

Предоставление процессу некоторого набора возможностей позволяет ему выполнять часть операций, обычно разрешенных суперпользователю, не позволяя ему выполнять другие операции такого вида.

Возможности подробно рассматриваются в главе 39. Далее в книге при упоминании конкретной операции, которая может выполняться только привилегированным процессом, в скобках будет указываться конкретная возможность. Названия возможностей начинаются с префикса `CAP` (например, `CAP_KILL`).

Процесс init

При загрузке системы ядро создает особый процесс, который называется `init`, «родитель всех процессов». Он ведет свое происхождение от программного файла `/sbin/init`. Все процессы в системе создаются (используя `fork()`) либо процессом `init`, либо одним из его потомков. Процесс `init` всегда имеет идентификатор процесса `1` и запускается с правами доступа суперпользователя. Процесс `init` не может быть уничтожен (даже привилегированным

пользователем) и завершается только при завершении работы системы. Основной задачей `init` является создание и слежение за процессами, требуемыми работающей системе. (Подробности можно найти на странице руководства `init(8)`.)

Процессы-демоны

Демоном называется процесс специального назначения, создаваемый и управляемый системой точно так же, как и другие процессы, но отличающийся от них следующими характеристиками.

- Он долгоживущий. Процесс-демон зачастую запускается при загрузке системы и продолжает свое существование до тех пор, пока работа системы не будет завершена.
- Он запускается в фоновом режиме, и у него нет управляющего терминала, с которого он мог бы считывать ввод или на который он мог бы записывать вывод.

К примерам процессов-демонов относятся `syslogd`, который записывает сообщения в системный журнал, и `httpd`, который обслуживает веб-страницы посредством протокола передачи гипертекста — Hypertext Transfer Protocol (HTTP).

Список переменных среды

У каждого процесса имеется *список переменных среды*, являющийся набором *переменных среды*, который содержится в памяти пользовательского пространства процесса. Каждый элемент этого списка состоит из имени и связанного с ним значения. При создании нового процесса с помощью `fork()` он наследует копию среды своего родителя. Таким образом, среда предоставляет родительскому процессу механизм для обмена информацией с дочерним процессом. Когда процесс заменяет программу, запуская новую программу с помощью `exec()`, последняя либо наследует среду, используемую старой программой, либо получает новую среду, указанную как часть вызова `exec()`.

Переменные среды, как в следующем примере, создаются в большинстве оболочек командой `export` (или командой `setenv` в оболочке C shell):

```
$ export MYVAR='Hello world'
```

> При предоставлении сессии командной оболочки, показывающей интерактивный ввод и вывод, текст ввода будет всегда выделяться полужирным шрифтом. Иногда в сессию будет включаться комментарий, выделенный курсивом, — в нем содержатся пояснения, касающиеся введенных команд или произведенного вывода.

Программы на языке C могут получать доступ к среде, используя внешнюю переменную (`char **environ`) и различные библиотечные функции, позволяющие процессу извлекать и изменять значения в его среде.

Переменные среды предназначены для различных целей. Например, оболочка определяет и использует ряд переменных, к которым можно получить доступ из сценариев и программ, выполняемых из оболочки. В число таких переменных входят `HOME`, указывающая путевое имя пользовательского каталога входа в систему, и `PATH`, указывающая список каталогов, в которых оболочка будет вести поиск программ, соответствующих введенным пользователем командам.

Ограничения ресурсов

Каждый процесс потребляет ресурсы, например открытые файлы, память и время центрального процессора. Используя системный вызов `setrlimit()`, процесс может установить верхний предел своего потребления различных ресурсов. Каждый такой *предел* имеет два связанных с ним значения: *мягкое ограничение*, ограничивающее тот объем ресурса,

который процесс может задействовать, и *жесткое ограничение*, представляющее собой верхний предел значения, которое может быть отрегулировано мягким ограничением. Непривилегированный процесс может изменить свое мягкое ограничение для конкретного ресурса на любое значение в диапазоне от нуля и до соответствующего жесткого ограничения, но свое жесткое ограничение он может только понизить.

Когда с помощью `fork()` создается новый процесс, он наследует копии настроек ограничений ресурсов от своего родительского процесса.

Ограничения ресурсов оболочки могут быть отрегулированы с использованием команды `ulimit` (`limit` в оболочке C shell). Эти настройки ограничений наследуются дочерними процессами, создаваемыми оболочкой для выполнения команд.

2.8. Отображение в памяти

Системный вызов `mmap()` создает в виртуальном адресном пространстве вызывающего процесса новое *отображение в памяти*.

Отображения делятся на две категории.

- *Файловое отображение*, которое отображает область файла на виртуальную память вызывающего процесса. После отображения содержимое файла может быть доступно с помощью операций над байтами в соответствующей области памяти. Страницы отображения автоматически загружаются из файла по мере надобности.
- В противоположность первой категории, *анонимное отображение* не имеет соответствующего файла. Вместо этого страницы отображения получают начальное значение `0`.

Отображение в памяти одного процесса может совместно использоваться отображениями в других процессах. Это может произойти либо из-за того, что два процесса отображают в памяти одну и ту же область файла, либо по причине наследования дочерним процессом, созданным с помощью `fork()`, отображения в памяти от своего родительского процесса.

Когда два и более процесса совместно используют одни и те же страницы, каждый из них может видеть изменения, внесенные в содержимое страниц другим процессом, в зависимости от того, каким именно было создано отображение — закрытым или совместно используемым. Когда отображение является *закрытым*, изменения содержимого отображения невидимы другим процессам и не доводятся до базового файла. Когда отображение является *совместно используемым*, изменения содержимого отображения видны другим процессам, использующим совместно то же самое отображение в памяти, и доводятся до базового файла.

Отображения в памяти служат для различных целей, включая инициализацию текстового сегмента процесса из соответствующего сегмента выполняемого файла, выделения новой (заполненной нулями) памяти, файлового ввода-вывода (ввода-вывода с отображением в памяти), обмена данными между процессами (через общее отображение в памяти).

2.9. Статические и совместно используемые библиотеки

Объектная библиотека представляет собой файл, содержащий откомпилированный объектный код для (обычно логически связанных) наборов функций, которые могут быть вызваны из прикладных программ. Помещение кода для набора функций в единую объектную библиотеку упрощает выполнение задач по созданию и сопровождению программ. Современные системы UNIX предоставляют два типа объектных библиотек: *статические* и *совместно используемые библиотеки*.

Статические библиотеки

Статические библиотеки (которые также иногда называют *архивами*) в ранних системах UNIX были единственным типом библиотек. Статическая библиотека, по сути, является структурированной связкой откомпилированных объектных модулей. Для того чтобы в программе можно было пользоваться функциями статической библиотеки, при компоновке программы надо указать имя нужной библиотеки. После того как будет определено, в каких именно объектных модулях статической библиотеки находятся нужные для основной программы функции, компоновщик извлекает из библиотеки копии этих модулей и копирует их в получаемый в результате исполняемый файл (иногда его называют результирующим).

После разрешения из основной программы различных ссылок на функции в модули статической библиотеки сборщик извлекает из библиотеки копии требуемых объектных модулей и копирует их в получаемый в результате исполняемый файл. Такая программа называется *статически скомпонованной*.

Тот факт, что каждая статически скомпонованная программа включает свою собственную копию требуемых из библиотеки объектных модулей, создает массу неудобств. Одно из них заключается в том, что дублирование объектного кода в различных исполняемых файлах впустую тратит дисковое пространство. Соответственно, впустую также расходуется и память, когда статически скомпонованным программам, выполняющимся одновременно, необходима одна и та же библиотечная функция. Каждой программе требуется, чтобы в памяти размещалась отдельная копия функции. Кроме того, если библиотечная функция требует изменения, то после ее перекомпиляции и добавления в статическую библиотеку все приложения, нуждающиеся в использовании обновленной функции, должны быть перекомпонованы с библиотекой.

Совместно используемые библиотеки

Совместно используемые библиотеки были разработаны для решения проблем, связанных со статическими библиотеками.

Если программа скомпонована с совместно используемой библиотекой, то вместо копирования объектного модуля из библиотеки в исполняемый файл компоновщик просто делает запись в этот файл. Запись показывает, что во время выполнения исполняемому файлу необходимо обратиться к совместно используемой библиотеке. Когда исполняемый файл в процессе выполнения загружается в память, программа, называемая *динамическим компоновщиком*, обеспечивает поиск общих библиотек, требуемых исполняемому файлу, и их загрузку в память. Во время выполнения нужно, чтобы в памяти резидентно находилась только одна копия кода совместно используемой библиотеки. Этой копией могут воспользоваться все запущенные программы. Тот факт, что совместно используемая библиотека содержит единственную скомпилированную версию функции, экономит дисковое пространство. Кроме того, существенно упрощается задача обеспечения использования программами самой свежей версии функции. Простая перекомпоновка совместно используемой библиотеки с новым определением функции приведет к тому, что существующие программы станут автоматически применять новое определение при своем следующем выполнении.

2.10. Межпроцессное взаимодействие и синхронизация

Работающая система Linux состоит из большого количества процессов, многие из которых работают независимо друг от друга. Но некоторые процессы для достижения своих намеченных целей сотрудничают друг с другом, и им необходимы методы обмена данными и синхронизация их действий.

Одним из способов обмена данными между процессами является чтение информации с дисковых файлов и ее запись в эти файлы. Но для многих приложений этот способ является слишком медленным и негибким. Поэтому в Linux, как и во всех современных реализациях UNIX, предоставляется обширный набор механизмов для *межпроцессного взаимодействия* (Interprocess Communication, IPC), включая следующие:

- *сигналы*, которые используются в качестве признака возникновения события;
- *конвейеры* (известные пользователям оболочек в виде оператора |) и FIFO-буферы, которые могут применяться для передачи данных между процессами;
- *сокеты*, которые могут использоваться для передачи данных от одного процесса к другому; данные при этом находятся на одном и том же базовом компьютере либо на различных хостах, связанных по сети;
- *файловая блокировка*, позволяющая процессу блокировать области файла с целью предотвращения их чтения или обновления содержимого файла другими процессами;
- *очереди сообщений*, которые используются для обмена сообщениями (пакетами данных) между процессами;
- *семафоры*, которые применяются для синхронизации действий процессов;
- *совместно используемая память*, позволяющая двум и более процессам совместно использовать часть памяти. Когда один процесс изменяет содержимое совместно используемой области памяти, изменения тут же могут быть видимы всем остальным процессам.

Широкое разнообразие IPC-механизмов в системах UNIX с иногда перекрываемыми функциональными возможностями частично объясняется их различием в отдельных вариантах UNIX-систем и требованиями со стороны различных стандартов. Например, FIFO-буферы и доменные сокеты UNIX, по сути, выполняют одну и ту же функцию, позволяющую неродственным процессам в одной и той же системе осуществлять обмен данными. Их совместное существование в современных системах UNIX объясняется тем, что FIFO-буферы пришли из System V, а сокеты были взяты из BSD.

2.11. Сигналы

Хотя в предыдущем разделе сигналы были перечислены в качестве методов IPC, чаще всего они используются в широком разнообразии других контекстов, поэтому заслуживают более подробного рассмотрения.

Сигналы зачастую описываются как «программные прерывания». Поступление сигнала информирует процесс о том, что случилось какое-то событие или возникли исключительные условия. Существует множество разнообразных сигналов, каждый из которых идентифицирует событие или условие. Каждый тип сигнала идентифицируется с помощью целочисленного значения, определяемого в символьном имени, имеющем форму `SIGxxxx`.

Сигналы отправляются процессу ядром, другим процессом (с соответствующими разрешениями) или самим процессом. Например, ядро может отправить сигнал процессу, когда произойдет что-нибудь из следующего перечня:

- пользователь набрал на клавиатуре команду *прерывания* (обычно это Ctrl+C);
- завершился один из дочерних процессов данного процесса;
- истекло время таймера (будильника), установленного процессом;
- процесс попытался получить доступ к неверному адресу в памяти.

В оболочке сигнал процессу можно отправить с помощью команды `kill`. Внутри программ ту же возможность может предоставить системный вызов `kill()`.

Когда процесс получает сигнал, он, в зависимости от сигнала, выполняет одно из следующих действий:

- игнорирует сигнал;
- прекращает свою работу по сигналу;
- приостанавливается, чтобы впоследствии возобновить свое выполнение с получением сигнала специального назначения.

Для большинства типов сигналов вместо выполнения исходного действия по сигналу программа может либо проигнорировать сигнал (что пригодится, если игнорирование не является исходной реакцией на сигнал), либо установить *обработчик сигнала*. Последний представляет собой функцию, определенную программистом, которая автоматически вызывается при доставке сигнала процессу. Эта функция выполняет некоторые действия, из-за которых был сгенерирован сигнал.

В период времени между генерированием сигнала и его доставкой сигнал для процесса считается *ожидающим*. Обычно ожидающий сигнал доставляется сразу же, как только получающий его процесс будет спланирован следующим для выполнения, или немедленно, если процесс уже выполняется. Но можно также *заблокировать* сигнал, добавив его в *маску сигналов* процесса. Если сигнал был сгенерирован после блокировки, он остается ожидающим до тех пор, пока в последующем блокировка не будет снята (например, удалена из маски сигналов).

2.12. Потоки

В современных реализациях UNIX у каждого процесса может быть несколько потоков выполнения. Потоки можно представить себе в качестве набора процессов, совместно использующих одну и ту же виртуальную память, а также ряд других атрибутов. Каждый поток выполняет один и тот же программный код и совместно с другими потоками использует одну и ту же область данных и кучу. Но каждый поток имеет свой собственный стек, содержащий локальные переменные и информацию о связанности вызовов функций.

Потоки могут осуществлять взаимный обмен данными через совместно используемые глобальные переменные. API для работы с потоками предоставляет *условные переменные* и *мьютексы*, являющиеся примитивами, позволяющими потокам процесса обмениваться данными и синхронизировать свои действия, в частности их использование общих переменных. Потоки могут также обмениваться друг с другом данными с применением IPC и механизмов синхронизации, рассмотренных в разделе 2.10. Основным преимуществом использования потоков является упрощение обмена данными (через глобальные переменные) между сотрудничающими потоками. Кроме того, некоторые алгоритмы более естественно преобразуются в многопоточные реализации, чем в варианты использования нескольких процессов. Помимо этого, многопоточные приложения могут легко воспользоваться преимуществами параллельной обработки на многопроцессорном оборудовании.

2.13. Группы процессов и управление заданиями в оболочке

Каждая программа, выполняемая оболочкой, запускается в новом процессе. Например, оболочка создает три процесса для выполнения следующего конвейера команд, который выводит на экран список файлов в текущем рабочем каталоге (список отсортирован по размеру файлов):

```
$ ls -l | sort -k5n | less
```

Все основные оболочки, за исключением Bourne shell, предоставляют интерактивные возможности, называемые *управлением заданиями*. Они позволяют пользователю одновременно выполнять несколько команд или конвейеров и манипулировать ими. В оболочках, допускающих управление заданиями, все процессы в конвейере помещаются в новую *группу процессов* или в *задание*. (В простейшем случае, когда командная строка оболочки содержит только одну команду, создается новая группа процессов, включающая только один процесс.) Каждый процесс в группе процессов имеет одинаковый целочисленный *идентификатор группы процессов*. Он совпадает с идентификатором процесса одного из процессов группы, который называется *лидером группы процессов*.

Ядро позволяет всем процессам, входящим в группу, выполнять различные действия, в особенности доставку сигналов. Оболочки, допускающие управление заданиями, применяют эту функцию, чтобы позволить пользователю, как показано в следующем разделе, приостанавливать или возобновлять все процессы в конвейере.

2.14. Сессии, управляющие терминалы и управляющие процессы

Сессией называется коллекция групп процессов (заданий). У всех имеющихся в сессии процессов будет один и тот же *идентификатор сессии*. *Ведущим в сессии* является процесс, создающий сессию, а идентификатор этого процесса становится идентификатором сессии.

Сессии в основном используются оболочками, допускающими управление заданиями. Все группы процессов, созданные такой оболочкой, принадлежат той же сессии, что и оболочка, являющаяся ведущим процессом сессии.

У сессий обычно имеется связанный с ними *управляющий терминал*, который устанавливается, когда ведущий процесс сессии первый раз открывает терминальное устройство. Для сессии, созданной интерактивной оболочкой, это терминал, с которого пользователь вошел в систему. Терминал может быть управляющим для нескольких сессий.

Вследствие открытия управляющего терминала ведущий процесс сессии становится для него *управляющим процессом*. Если происходит отключение от терминала (например, если закрыто окно терминала), управляющий процесс получает сигнал `SIGHUP`.

В любой момент времени одна из групп процессов в сессии является *приоритетной группой (приоритетным заданием)*, которая может считывать ввод с терминала и отправлять на него вывод. Если пользователь набирает на управляющем терминале символ *прерывания* (обычно это Ctrl+C) или символ *приостановки* (обычно это Ctrl+Z), драйвер терминала отправляет сигнал, уничтожающий или приостанавливающий приоритетную группу процессов. У сессии может быть любое количество фоновых групп процессов (фоновых заданий), создаваемых с помощью символа амперсанда (&) в конце командной строки.

Оболочки, допускающие управление заданиями, предоставляют команды для просмотра списка всех заданий, отправки заданиями сигналов и перемещением заданий между режимом первого плана и фоновым режимом.

2.15. Псевдотерминалы

Псевдотерминалом называется пара подключенных виртуальных устройств, называемых *ведущим (master)* и *ведомым (slave)*. Эта пара устройств предоставляет IPC-канал, позволяющий перемещать данные в обоих направлениях между двумя устройствами.

Важной особенностью псевдотерминала является то, что ведомое устройство предоставляет интерфейс, который ведет себя как терминал. Он позволяет подключить к ведомому устройству программу, ориентированную на работу с терминалом, а затем воспользоваться другой программой, подключенной к ведущему устройству, для управления первой программой. Вывод, записанный программой-драйвером, проходит обычную обработку ввода, выполняемую драйвером терминала (например, в исходном режиме символ возврата каретки преобразуется в новую строку), а затем передается в качестве ввода ориентированной на работу с терминалом программе, подключенной к ведомому устройству. Все, что эта программа записывает в ведомое устройство, передается (после выполнения всей обычной обработки, проводимой на терминале) в качестве ввода программе-драйверу. Иными словами, программа-драйвер выполняет функцию, которую на традиционном терминале выполняет сам пользователь.

Псевдотерминалы используются в различных приложениях, в первую очередь в реализациях окон терминала, предоставляемых при входе в систему X Window, и в приложениях, предоставляющих сервисы входа в сеть, например telnet и ssh.

2.16. Дата и время

Для процесса интерес представляют два типа времени.

- *Реальное время*, которое измеряется либо относительно некоторой стандартной точки (*календарного* времени), либо относительно какой-то фиксированной точки, обычно от начала жизненного цикла процесса (*истекшее* или *физическое* время). В системах UNIX календарное время измеряется в секундах, прошедших с полуночи 1 января 1970 года всемирного координированного времени —Universal Coordinated Time (обычно сокращаемого до UTC), и координируется на базовой точке часовых поясов, определяемой линией долготы, проходящей через Гринвич, Великобритания. Эта дата, близкая к дате появления системы UNIX, называется *началом отсчета времени (Epoch)*.
- *Время процесса*, также называемое *временем центрального процессора*, которое является общим количеством времени центрального процессора, использованным процессом с момента старта. Время ЦП далее делится на *системное время центрального процессора*, то есть время, потраченное на выполнение кода в *режиме ядра* (например, на выполнение системных вызовов и работу других служб ядра от имени процесса), и *пользовательское время центрального процессора*, потраченное на выполнение кода в *пользовательском режиме* (например, на выполнение обычного программного кода).

Команда `time` выводит реальное время, системное и пользовательское время центрального процессора, потраченное на выполнение процессов в конвейере.

2.17. Клиент-серверная архитектура

Проектирование и разработка клиент-серверных приложений будут подробно рассматриваться в нескольких местах этой книги.

Клиент-серверное приложение разбито на два составляющих процесса:

- *клиент*, который просит сервер о какой-либо *услуге*, отправив ему сообщение с запросом;
- *сервер*, который изучает запрос клиента, выполняет соответствующие действия, а затем отправляет назад клиенту сообщение с ответом.

Иногда клиент и сервер могут быть вовлечены в расширенный диалог из запросов и ответов.

Обычно клиентское приложение взаимодействует с пользователем, а серверное приложение предоставляет доступ к некоторому совместно используемому ресурсу. Чаще всего обменом данными с одним или несколькими серверными процессами занимается несколько клиентских процессов.

Клиент и сервер могут находиться на одном и том же ведущем компьютере или на отдельных хостах, соединенных по сети. Для взаимного обмена сообщениями клиент и сервер используют IPC-механизмы, рассмотренные в разделе 2.10.

Серверы могут реализовывать различные сервисы, например:

- предоставление доступа к базе данных или другому совместно используемому информационному ресурсу;
- предоставление доступа к удаленному файлу по сети;
- инкапсуляция какой-нибудь бизнес-логики;
- предоставление доступа к совместно используемым аппаратным ресурсам (например, к принтеру);
- обслуживание веб-страниц.

Инкапсуляция сервиса на отдельном сервере имеет смысл по нескольким причинам, в числе которых следующие.

- *Рентабельность*. Предоставление одного экземпляра ресурса (например, принтера), управляемого сервером, может быть проще предоставления того же самого ресурса локально каждому компьютеру.
- *Управление, координация и безопасность*. При содержании ресурса (особенно информационного) в одном месте сервер может координировать доступ к ресурсу (например, так, чтобы два клиента не могли одновременно обновлять один и тот же блок информации) или обеспечить его безопасность таким образом, чтобы он был доступен только избранным клиентам.
- *Работа в разнородной среде*. В сети различные клиенты и сервер могут быть запущены на различном оборудовании и на разных платформах операционных систем.

2.18. Выполнение действий в реальном масштабе времени

Приложения, работающие *в реальном масштабе времени*, должны своевременно откликаться на ввод. Зачастую такой ввод поступает от внешнего датчика или специализированного устройства ввода, и вывод принимает форму управления каким-нибудь внешним оборудованием. Примерами приложений, требующих реакции в реальном масштабе времени, могут служить автоматизированные сборочные линии, банкоматы, авиационные навигационные системы.

Хотя многие приложения реального масштаба времени требуют быстрых откликов на ввод, определяющим фактором является то, что ответ гарантированно должен быть предоставлен к конкретному конечному сроку после возникновения запускающего события.

Обеспечение быстроты реагирования в реальном масштабе времени, особенно когда важно сохранить короткое время отклика, требует поддержки от базовой операционной системы. Большинство операционных систем в силу присущих им особенностей не в состоянии предоставить такую поддержку, поскольку требования быстроты реагирования в реальном масштабе времени могут конфликтовать с требованиями, предъявляемыми к многопользовательским операционным системам с разделением времени. Традиционные

реализации UNIX не являются операционными системами реального масштаба времени, хотя и были разработаны их версии с подобными характеристиками. Кроме того, были созданы варианты Linux, отвечающие требованиям, предъявляемым к системам реального масштаба времени, и самые новые ядра Linux разрабатываются так, чтобы полноценно поддерживать приложения реального масштаба времени.

В POSIX.1b определено несколько расширений к POSIX.1 для поддержки приложений реального масштаба времени. В их числе асинхронный ввод/вывод, совместно используемая память, отображаемые в памяти файлы, блокировка памяти, часы и таймеры реального масштаба времени, альтернативные политики диспетчеризации, сигналы, очереди сообщений и семафоры реального масштаба времени. Но даже притом, что большинство современных реализаций UNIX еще не могут называться системами реального масштаба времени, они поддерживают некоторые или даже все эти расширения. (По ходу повествования вам еще встретятся описания этих особенностей POSIX.1b, поддерживаемых Linux.)

> Понятие реального времени используется в данной книге при обращении к концепции календарного или прошедшего времени, а понятие реального масштаба времени используется для обозначения операционной системы или приложения, предоставляющего тот тип реагирования в реальном масштабе времени, который рассмотрен в текущем разделе.

2.19. Файловая система /proc

Как и в некоторых других реализациях UNIX, в Linux предоставляется файловая система `/proc`, состоящая из набора каталогов и файлов, смонтированных в каталоге `/proc`.

`/proc` — виртуальная файловая система, предоставляющая интерфейс структуре данных ядра в форме, похожей на файлы и каталоги файловой системы. Тем самым предоставляется простой механизм для просмотра и изменения различных системных атрибутов. Кроме того, набор каталогов с именами в форме `/proc/`*`PID`*, где *`PID`* является идентификатором процесса, позволяет нам просматривать информацию о каждом процессе, запущенном в системе.

Содержимое файлов в каталоге `/proc` в основном представлено в форме текста, доступного для прочтения человеком, и может быть разобрано сценариями оболочки. Программа может просто открыть нужный файл и считать из него данные или записать их в него. В большинстве случаев для изменения содержимого файлов в каталоге `/proc` процесс должен быть привилегированным.

По мере рассмотрения различных частей интерфейса программирования Linux будут также рассматриваться и относящиеся к ним файлы каталога `/proc`. Дополнительная общая информация по этой файловой системе приводится в разделе 12.1. Файловая система `/proc` не определена никакими стандартами, и рассматриваемые здесь детали относятся только к системе Linux.

2.20. Резюме

В этой главе был перечислены основные понятия, относящиеся к системному программированию Linux. Усвоение этих понятий должно предоставить пользователям с весьма скромным опытом работы с Linux или UNIX теоретическую базу, вполне достаточную для того, чтобы приступить к изучению системного программирования.

3 Общее представление о системном программировании

В текущей главе рассматриваются различные темы, без изучения которых невозможно перейти к системному программированию. Сначала будут описаны системные вызовы и подробно рассмотрены этапы их выполнения. Затем будет уделено внимание библиотечным функциям и их отличиям от системных вызовов, после чего все это будет увязано с описанием GNU-библиотеки C.

При осуществлении системного вызова или вызова библиотечной функции обязательно нужно проверять код возврата, чтобы определить, насколько успешно прошел вызов. Поэтому в главе описан порядок проведения таких проверок и представлен набор функций, которые используются в большинстве приводимых здесь примеров программ для диагностики ошибок, возвращаемых системными вызовами и библиотечными функциями.

В завершение будут рассмотрены различные вопросы, относящиеся к программированию портируемых программных средств, в частности использование макросов проверки возможностей и определенных в SUSv3 стандартных типов системных данных.

3.1. Системные вызовы

Системный вызов представляет собой управляемую точку входа в ядро, позволяющую процессу запрашивать у ядра осуществления некоторых действий в интересах процесса. Ядро дает возможность программам получать доступ к некоторым сервисам с помощью интерфейса прикладного программирования (API) системных вызовов. К таким сервисам, к примеру, относятся создание нового процесса, выполнение ввода-вывода и создание конвейеров для межпроцессного взаимодействия. (Системные вызовы Linux перечисляются на странице руководства `syscalls(2)`.)

Перед тем как перейти к подробностям работы системных вызовов, следует упомянуть о некоторых их общих характеристиках.

- Системный вызов изменяет состояние процессора, переводя его из пользовательского режима в режим ядра, позволяя таким образом центральному процессору получать доступ к защищенной памяти ядра.
- Набор системных вызовов не изменяется. Каждый системный вызов идентифицируется по уникальному номеру. (Обычно программам эта система нумерации неизвестна, они идентифицируют системные вызовы по именам.)
- У каждого системного вызова может быть набор аргументов, определяющих информацию, которая должна быть передана из пользовательского пространства (то есть из виртуального адресного пространства процесса) в пространство ядра и наоборот.

С точки зрения программирования, инициирование системного вызова во многом похоже на вызов функции языка C. Но при выполнении системного вызова многое происходит закулисно. Чтобы пояснить, рассмотрим все последовательные этапы происходящего на конкретной аппаратной реализации — x86-32.

1. Прикладная программа осуществляет системный вызов, вызвав функцию-оболочку из библиотеки C.
2. Функция-оболочка должна обеспечить доступность всех аргументов системного вызова подпрограмме его перехвата и обработки (которая вскоре будет рассмотрена). Эти аргументы передаются функции-оболочке через стек, но ядро ожидает их появления в конкретных регистрах центрального процессора. Функция-оболочка копирует аргументы в эти регистры.
3. Поскольку вход в ядро всеми системными вызовами осуществляется одинаково, ядру нужен какой-нибудь метод идентификации системного вызова. Для обеспечения такой возможности функция-оболочка копирует номер системного вызова в конкретный регистр (`%eax`).
4. В функции-оболочке выполняется машинный код *системного прерывания* (`int 0x80`), заставляющий процессор переключиться из пользовательского режима в режим ядра и выполнить код, указатель на который расположен в векторе прерывания системы `0x80` (в десятичной системе счисления — 128).

   В более современных архитектурах x86-32 реализуется инструкция sysenter, предоставляющая более быстрый способ входа в режим ядра, по сравнению с обычной инструкцией системного прерывания int 0x80. Использование sysenter поддерживается в версии ядра 2.6 и в glibc, начиная с версии 2.3.2.

5. В ответ на системное прерывание `0x80` ядро для его обработки инициирует свою подпрограмму `system_call()` (которая находится в ассемблерном файле arch/x86/kernel/entry.S). Обработчик прерывания делает следующее;
   1) сохраняет значения регистров в стеке ядра (см. раздел 6.5);
   2) проверяет допустимость номера системного вызова;
   3) вызывает соответствующую подпрограмму обслуживания системного вызова. Ее поиск ведется по номеру системного вызова: в таблице всех подпрограмм обслуживания системных вызовов в качестве индекса используется номер системного вызова (переменная ядра `sys_call_table`). Если у подпрограммы обслуживания системного вызова имеются аргументы, то она сначала проверяет их допустимость. Например, она проверяет, что адреса указывают на места, допустимые в пользовательской памяти. Затем подпрограмма обслуживания системного вызова выполняет требуемую задачу, которая может предполагать изменение значений адресов, указанных в переданных аргументах, и перемещение данных между пользовательской памятью и памятью ядра (например, в операциях ввода-вывода). И наконец, подпрограмма обслуживания системного вызова возвращает подпрограмме `system_call()` код возврата;
   4) восстанавливает значения регистров из стека ядра и помещает в стек возвращаемое значение системного вызова;
   5) возвращает управление функции-оболочке, одновременно переводя процессор в пользовательский режим.
6. Если возвращаемое значение подпрограммы обслуживания системного вызова свидетельствует о возникновении ошибки, то функция-оболочка присваивает это значение глобальной переменной `errno` (см. раздел 3.4). Затем функция-оболочка возвращает управление вызывавшему ее коду, предоставляя ему целочисленное значение, указывающее на успех или неудачу системного вызова.

В Linux подпрограммы обслуживания системных вызовов следуют соглашению о том, что для указания успеха возвращается неотрицательное значение. При ошибке подпрограмма возвращает отрицательное число, являющееся значением одной из errno-констант с противоположным знаком. Когда возвращается отрицательное значение, функция-оболочка библиотеки C меняет его знак на противоположный (делая его положительным), копирует результат в errno и возвращает значение –1, чтобы указать вызывающей программе на возникновение ошибки.

Это соглашение основано на предположении, что подпрограммы обслуживания системных вызовов в случае успеха не возвращают отрицательных значений. Но для некоторых таких подпрограмм это предположение неверно. Обычно это не вызывает никаких проблем, поскольку диапазон превращенных в отрицательные числа значений errno не пересекается с допустимыми отрицательными возвращаемыми значениями. Тем не менее в одном случае все же появляется проблема — когда дело касается операции F_GETOWN системного вызова fcntl(), рассматриваемого в разделе 59.3.

На рис. 3.1 на примере системного вызова `execve()` показана описанная выше последовательность. В Linux/x86-32 `execve()` является системным вызовом под номером 11 (`__NR_execve`). Следовательно, 11-я запись в векторе `sys_call_table` содержит адрес `sys_execve()`, подпрограммы, обслуживающей этот системный вызов. (В Linux подпрограммы обслуживания системных вызовов обычно имеют имена в формате `sys_`*`xyz`*`()`, где *`xyz`*`()` является соответствующим системным вызовом.)

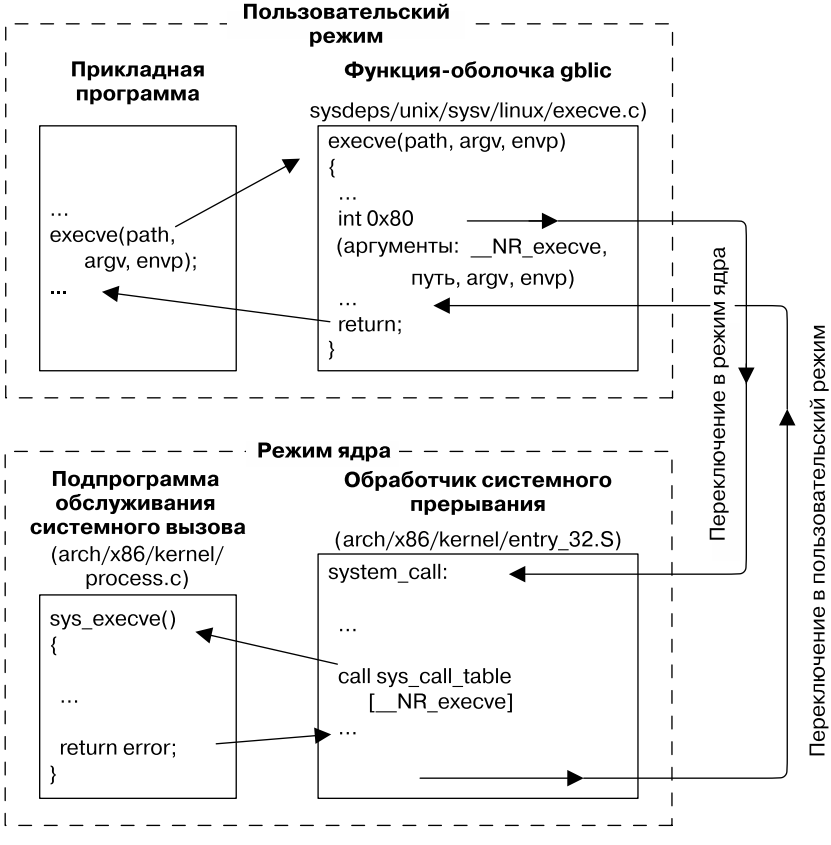


Рис. 3.1. Этапы выполнения системного вызова

Информации, изложенной в предыдущих абзацах, даже больше, чем нужно для усвоения всего остального материала книги. Но она поясняет весьма важное обстоятельство: даже для простого системного вызова должно быть проделано немало работы. Следовательно, у системных вызовов есть хотя и незначительные, но все же заметные издержки.

> В качестве примера издержек на осуществление системного вызова рассмотрим системный вызов getppid(). Он просто возвращает идентификатор родительского процесса, которому принадлежит вызывающий процесс. В одной из принадлежащих автору книги x86-32-систем с запущенной Linux 2.6.25 на совершение 10 миллионов вызовов getppid() ушло приблизительно 2,2 секунды. То есть на каждый вызов ушло около 0,3 микросекунды. Для сравнения, на той же системе на 10 миллионов вызовов функции языка C, которая просто возвращает целое число, ушло 0,11 секунды, или около 1/12 времени, затраченного на вызовы getppid(). Разумеется, большинство системных вызовов имеет более существенные издержки, чем getppid().

Поскольку с точки зрения программы на языке C вызов функции-оболочки библиотеки C является синонимом запуска соответствующей подпрограммы обслуживания системного вызова, далее в книге для обозначения действия «вызов функции-оболочки, которая запускает системный вызов *`xyz()`*» будет использоваться фраза «запуск системного вызова *`xyz()`*».

Дополнительные сведения о механизме системных вызовов Linux можно найти в изданиях [Love, 2010], [Bovet & Cesati, 2005] и [Maxwell, 1999].

3.2. Библиотечные функции

Библиотечная функция — одна из множества функций, составляющих стандартную библиотеку языка C. (Для краткости далее в книге при упоминании о конкретной функции вместо словосочетания *«библиотечная функция»* будем просто использовать слово *«функция»*.) Эти функции предназначены для решения широкого круга разнообразных задач: открытия файлов, преобразования времени в формат, понятный человеку, сравнения двух символьных строк и т. д.

Многие библиотечные функции вообще не используют системные вызовы (например, функции для работы со сроками). С другой стороны, некоторые библиотечные функции являются надстройками над системными вызовами. Например, библиотечная функция `fopen()` использует для открытия файла системный вызов `open()`. Зачастую библиотечные функции разработаны для предоставления более удобного интерфейса вызова по сравнению с тем, что имеется у исходного системного вызова. Например, функция `printf()` предоставляет форматирование вывода и буферизацию данных, а системный вызов `write()` просто выводит блок байтов. Аналогично этому функции `malloc()` и `free()` выполняют различные вспомогательные задачи, существенно облегчающие выделение и высвобождение оперативной памяти по сравнению с использованием исходного системного вызова `brk()`.

3.3. Стандартная библиотека языка C; GNU-библиотека C (glibc)

В различных реализациях UNIX существуют разные версии стандартной библиотеки языка C. Наиболее часто используемой реализацией в Linux является GNU-библиотека языка C (glibc, http://www.gnu.org/software/libc/).

Первоначально основным разработчиком и специалистом по обслуживанию GNU-библиотеки C был Роланд Макграт (Roland McGrath). До 2012 года этим занимался Ульрих Дреппер (Ulrich Drepper), после чего его полномочия были переданы сообществу разработчиков, многие из которых перечислены на странице https://sourceware.org/glibc/wiki/MAINTAINERS.

Для Linux доступны и другие реализации/версии библиотеки C, среди которых есть и такие, которым требуется (относительно) небольшой объем памяти, а предназначены они для встраиваемых приложений. В качестве примера можно привести uClibc (http://www.uclibc.org/) и diet libc (http://www.fefe.de/dietlibc/). В данной книге мы ограничиваемся рассмотрением glibc, поскольку именно эта библиотека языка C используется большинством приложений, разработанных под Linux.

Определение версии glibc в системе

Иногда требуется определить версию имеющейся в системе библиотеки glibc. Из оболочки это можно сделать, запустив совместно используемый библиотечный файл glibc, как будто он является исполняемой программой. Когда библиотека запускается как исполняемый файл, она выводит на экран разнообразную информацию, включая номер версии glibc:

```
$ /lib/libc.so.6
GNU C Library stable release version 2.10.1, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.0 20090506 (Red Hat 4.4.0-4).
Compiled on a Linux >>2.6.18-128.4.1.el5<< system on 2009-08-19.
Available extensions:
    The C stubs add-on version 2.1.2.
    crypt add-on version 2.1 by Michael Glad and others
    GNU Libidn by Simon Josefsson
    Native POSIX Threads Library by Ulrich Drepper et al
    BIND-8.2.3-T5B
    RT using linux kernel aio
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

В некоторых дистрибутивах Linux GNU-библиотека C находится в другом месте, путь к которому отличается от /lib/libc.so.6. Один из способов определить местоположение библиотеки — выполнить программу `ldd` (list dynamic dependencies — список динамических зависимостей) в отношении исполняемого файла, имеющего динамические ссылки на glibc (ссылки такого рода имеются у большинства исполняемых файлов). Затем можно изучить полученный в результате этого список библиотечных зависимостей, чтобы найти местоположение совместно используемой библиотеки glibc:

```
$ ldd myprog | grep libc
    libc.so.6 => /lib/tls/libc.so.6 (0x4004b000)
```

Есть два средства, с помощью которых прикладная программа может определить версию библиотеки GNU C в системе: тестирование констант или вызов библиотечной функции. Начиная с версии 2.0, в glibc определяются две константы, `__GLIBC__` и `__GLIBC_MINOR__`, которые могут быть протестированы в ходе компиляции (в инструкциях `#if`). В системе с установленной glibc 2.12 эти константы будут иметь значения `2` и `12`. Но использование этих констант не принесет пользы в программе, скомпилированной в одной системе, но запущенной в другой системе с отличающейся по версии библиотекой glibc.

Учитывая вышесказанное, можно воспользоваться функцией gnu_get_libc_version() для определения версии glibc доступной во время исполнения программы.

```
#include <gnu/libc-version.h>

const char *gnu_get_libc_version(void);
```

Возвращает указатель на заканчивающуюся нулевым байтом статически размещенную строку, содержащую номер версии библиотеки GNU C

Функция gnu_get_libc_version() возвращает указатель на строку вида 2.12.

Информацию о версии можно также получить, воспользовавшись функцией confstr() для извлечения значения конфигурационной переменной (относящегося к конкретной glibc-библиотеке) _CS_GNU_LIBC_VERSION. В результате вызова функции будет возвращена строка вида glibc 2.12.

3.4. Обработка ошибок, возникающих при системных вызовах и вызовах библиотечных функций

Почти каждый системный вызов и вызов библиотечной функции завершаются возвратом какого-либо значения, показывающего, чем все завершилось — успехом или неудачей. Надо *всегда* проверять код завершения, чтобы можно было убедиться в успешности вызова. Если успех вызову не сопутствовал, нужно предпринимать соответствующее действие — как минимум программа должна вывести на экран сообщение об ошибке, свидетельствующее о том, что случилось нечто неожиданное.

Несмотря на стремление сэкономить на наборе текста путем исключения подобных проверок (особенно после просмотра примеров программ, написанных под UNIX и Linux, где коды завершения не проверяются), это «экономия на спичках». Из-за отсутствия проверки кода, возвращенного системным вызовом или вызовом библиотечной функции, которая «в принципе не должна дать сбой», можно впустую потратить многие часы на отладку.

Есть несколько системных вызовов, никогда не дающих сбоев. Например, getpid() всегда успешно возвращает идентификатор процесса, а _exit() всегда прекращает процесс. Проверять значения, возвращаемые такими системными вызовами, нет никакого смысла.

Обработка ошибок системных вызовов

Возможные возвращаемые вызовом значения документируются на странице руководства по каждому системному вызову, и там показывается значение (или значения), свидетельствующее об ошибке. Обычно ошибка выявляется возвращением значения –1. Следовательно, системный вызов может быть проверен с помощью такого кода:

```
fd = open(pathname, flags, mode); /* Системный вызов для открытия файла */
if (fd == -1) {
  /* Код для обработки ошибки */
}
```

```
...
if (close(fd) == -1) {
  /* Код для обработки ошибки */
}
```

При неудачном завершении системного вызова для глобальной целочисленной переменной errno устанавливается положительное значение, позволяющее идентифицировать конкретную ошибку. Объявление errno, а также набора констант для различных номеров ошибок предоставляется за счет включения заголовочного файла <errno.h>. Все относящиеся к ошибкам символьные имена начинаются с E. Список возможных значений errno, которые могут быть возвращены каждым системным вызовом, предоставляется на каждой странице руководства в разделе заголовочного файла ERRORS. Простой пример использования errno для обнаружения ошибки системного вызова имеет следующий вид:

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
  if (errno == EINTR)
    fprintf(stderr, "read was interrupted by a signal\n");
                    // чтение было прервано сигналом
  else {
    /* Произошла какая-то другая ошибка */
  }
}
```

Успешно завершенные системные вызовы и вызовы библиотечных функций никогда не сбрасывают errno в 0, следовательно, эта переменная может иметь ненулевое значение из-за ошибки предыдущего вызова. Более того, SUSv3 разрешает успешно завершающей свою работу функции устанавливать для errno ненулевое значение (хотя это делают всего несколько функций). Поэтому при проверке на ошибку нужно всегда сперва проверить, не вернула ли функция значение, свидетельствующее о возникновении ошибки, и только потом исследовать errno для определения причины ошибки.

Некоторые системные вызовы (например, getpriority()) могут вполне законно возвращать при успешном завершении значение –1. Чтобы определить, не возникла ли при таких вызовах ошибка, перед самим вызовом нужно установить errno в 0 и проверить ее значение после вызова. Если вызов возвращает –1, а errno имеет ненулевое значение, значит, произошла ошибка. (Это же правило применимо к некоторым библиотечным функциям.)

Общая линия поведения после неудачного системного вызова заключается в выводе сообщения об ошибке на основе значения переменной errno. Для этой цели предоставляются библиотечные функции perror() и strerror().

Функция perror() выводит строку, указываемую с помощью аргумента msg. За строкой следует сообщение, соответствующее текущему значению переменной errno.

```
#include <stdio.h>

void perror(const char *msg);
```

Простой способ обработки ошибок из системных вызовов будет выглядеть следующим образом:

```
fd = open(pathname, flags, mode);
if (fd == -1) {
  perror("open");
  exit(EXIT_FAILURE);
}
```

Функция strerror() возвращает строку описания ошибки, соответствующую номеру ошибки, который задан в ее аргументе errnum.

```
#include <string.h>

char *strerror(int errnum);
```

Возвращает указатель на строку с описанием ошибки, соответствующую значению errnum

Строка, возвращенная strerror(), может быть размещена статически, что означает, что она может быть переписана последующими вызовами strerror().

Если в errnum указан нераспознаваемый номер ошибки, то strerror() возвращает строку вида Unknown error *nnn* (неизвестная ошибка с таким-то номером). В некоторых других реализациях strerror() в таких случаях возвращает значение NULL.

Поскольку функции perror() и strerror() чувствительны к настройкам локали (см. раздел 10.4), описания ошибок выводятся на языке локали.

Обработка ошибок из библиотечных функций

Различные библиотечные функции для обозначения ошибок возвращают разные типы данных и разные значения. (По каждой функции нужно обращаться к странице руководства.) В рамках этого раздела библиотечные функции могут быть разбиты на несколько категорий.

- Некоторые библиотечные функции возвращают информацию об ошибке точно таким же образом, что и системные вызовы: возвращают значение -1, а значение переменной errno указывает на конкретную ошибку. Примером такой функции может послужить remove(), удаляющая файл (используя системный вызов unlink()) или каталог (используя системный вызов rmdir()). Ошибки из этих функций могут определяться точно так же, как и ошибки из системных вызовов.
- Некоторые библиотечные функции возвращают при ошибке значение, отличное от -1, но все-таки устанавливают значение для переменной errno, чтобы указать на конкретные условия возникновения ошибки. Например, функция fopen() в случае ошибки возвращает нулевой указатель и устанавливает для переменной errno значение в зависимости от того, какой из положенных в основу ее работы системных вызовов завершился неудачно. Для определения типа таких ошибок могут применяться функции perror() и strerror().
- Другие библиотечные функции вообще не используют переменную errno. Метод определения наличия и причин ошибок зависит от конкретной функции и задокументирован на посвященной ей странице руководства. Для таких функций применение errno, perror() или strerror() с целью определения типа ошибок будет неприемлемо.

3.5. Пояснения по поводу примеров программ, приводимых в книге

В этом разделе будут рассмотрены различные соглашения и характерные особенности, которые обычно применяются к примерам программ, приводимым в книге.

3.5.1. Ключи и аргументы командной строки

Многие примеры программ в книге основаны на использовании ключей и аргументов командной строки, определяющих их поведение.

Традиционные ключи командной строки UNIX состоят из начального дефиса, буквы, идентифицирующей ключ, и необязательного аргумента. (Утилиты GNU предоставляют расширенный синтаксис ключей, состоящий из двух начальных дефисов, за которыми следует строка, идентифицирующая ключ, и необязательный аргумент.) Для анализа этих ключей используется стандартная библиотечная функция `getopt()`.

Каждый из наших примеров, где есть неочевидный синтаксис командной строки, снабжен простым вспомогательным средством для пользователя. При вызове с ключом `--help` программа выводит сообщение о порядке своей работы, показывая синтаксис для ключей и аргументов командной строки.

3.5.2. Типовые функции и заголовочные файлы

Большинство примеров программ включают заголовочный файл, содержащий необходимые в большинстве случаев определения, и в них также используется набор типовых функций. В этом разделе будут рассмотрены заголовочный файл и функции.

Типовой заголовочный файл

В листинге 3.1 приведен заголовочный файл, используемый практически в каждой программе, показанной в книге. Он включает различные другие заголовочные файлы, используемые во многих примерах программ, определяет тип данных `Boolean` и определяет макрос для вычисления минимума и максимума двух числовых значений. Применение данного файла позволяет немного сократить размеры примеров программ.

Листинг 3.1. Заголовочный файл, используемый в большинстве примеров программ

lib/tlpi_hdr.h

```
#ifndef TLPI_HDR_H
#define TLPI_HDR_H     /* Предотвращает случайное двойное включение */

#include <sys/types.h> /* Определения типов, используемые
                          многими программами */
#include <stdio.h>     /* Стандартные функции ввода-вывода */
#include <stdlib.h>    /* Прототипы наиболее востребованных библиотечных
                          функций плюс константы EXIT_SUCCESS
                          и EXIT_FAILURE */
#include <unistd.h>    /* Прототипы многих системных вызовов */
#include <errno.h>     /* Объявление errno и определение констант ошибок */
#include <string.h>    /* Наиболее используемые функции обработки строк */
#include "get_num.h"   /* Объявление наших функций для обработки числовых
                          аргументов (getInt(), getLong()) */
#include "error_functions.h" /* Объявление наших функций обработки ошибок */
typedef enum { FALSE, TRUE } Boolean;

#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))
#endif
```

lib/tlpi_hdr.h

Функции определения типа ошибок

Чтобы упростить обработку ошибок в наших примерах программ, мы используем функции определения типа ошибок. Объявление такой функции показано в листинге 3.2.

Листинг 3.2. Объявление для наиболее востребованных функций обработки ошибок

lib/error_functions.h

```
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H

void errMsg(const char *format, ...);

#ifdef __GNUC__
/* Этот макрос блокирует предупреждения компилятора при использовании
   команды 'gcc -Wall', жалующиеся, что "control reaches end of non-void
   function", то есть что управление достигло конца функции, которая
   должна вернуть значение, если мы используем следующие функции для
   прекращения выполнения main() или какой-нибудь другой функции,
   которая должна вернуть значение определенного типа (не void) */
#define NORETURN __attribute__ ((__noreturn__))
#else
#define NORETURN
#endif

void errExit(const char *format, ...) NORETURN ;
void err_exit(const char *format, ...) NORETURN ;
void errExitEN(int errnum, const char *format, ...) NORETURN ;
void fatal(const char *format, ...) NORETURN ;
void usageErr(const char *format, ...) NORETURN ;
void cmdLineErr(const char *format, ...) NORETURN ;

#endif
```

lib/error_functions.h

Для определения типа ошибок системных вызовов и библиотечных функций используются функции `errMsg()`, `errExit()`, `err_exit()` и `errExitEN()`.

```
#include "tlpi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

Функция `errMsg()` выводит сообщение на стандартное устройство вывода ошибки. Ее список аргументов совпадает со списком для функции `printf()`, за исключением того, что в строку вывода автоматически добавляется символ конца строки. Функция `errMsg()` выводит текст ошибки, соответствующий текущему значению переменной `errno`. Этот текст состоит из названия ошибки, например `EPERM`, дополненного описанием ошибки в том виде, в котором его возвращает функция `strerror()`, а затем следует вывод, отформатированный согласно переданным агрументам.

По своему действию функция `errExit()` похожа на `errMsg()`, но она также прекращает выполнение программы, либо вызвав функцию `exit()`, либо, если переменная среды `EF_DUMPCORE` содержит непустое строковое значение, вызвав функцию `abort()`, чтобы

создать файл дампа ядра для его использования отладчиком. (Файлы дампа ядра будут рассмотрены в разделе 22.1.)

Функция `err_exit()` похожа на `errExit()`, но имеет два отличия:

- не сбрасывает стандартный вывод перед выводом в него сообщения об ошибке;
- завершает процесс путем вызова `_exit()`, а не `exit()`. Это приводит к тому, что процесс завершается без сброса буферов stdio или вызова обработчиков выхода.

Подробности этих различий в работе `err_exit()` станут понятнее при изучении главы 25, где рассматривается разница между `_exit()` и `exit()`, а также обработка буферов stdio и обработчики выхода в дочернем процессе, созданном с помощью `fork()`. А пока мы просто возьмем на заметку, что функция `err_exit()` будет особенно полезна при написании нами библиотечной функции, создающей дочерний процесс, который следует завершить по причине возникновения ошибки. Это завершение должно произойти без сброса дочерней копии родительских буферов stdio (то есть буферов вызывающего процесса) и без вызова обработчиков выхода, созданных родительским процессом.

Функция `errExitEN()` представляет собой практически то же самое, что и `errExit()`, за исключением того, что вместо сообщения об ошибке, характерного текущему значению `errno`, она выводит текст, соответствующий номеру ошибки (отсюда и суффикс `EN`), заданному в аргументе `errnum`.

В основном функция `errExitEN()` применяется в программах, использующих API потоков стандарта POSIX. В отличие от традиционных системных вызовов UNIX, возвращающих при возникновении ошибки `-1`, функции потоков стандарта POSIX позволяют определить тип ошибки по ее номеру, возвращенному в качестве результата их выполнения (то есть в `errno`, как правило, помещается положительный номер типа). (В случае успеха функции потоков стандарта POSIX возвращают `0`.)

Определить типы ошибок из функции потоков стандарта POSIX можно с помощью следующего кода:

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
  errExit("pthread_create");
```

Но такой подход неэффективен, поскольку в программе, выполняемой в нескольких потоках, `errno` определяется в качестве макроса. Этот макрос расширяется в вызов функции, возвращающий левостороннее выражение (lvalue). Соответственно, каждое использование `errno` приводит к вызову функции. Функция `errExitEN()` позволяет создавать более эффективный эквивалент показанного выше кода:

```
int s;
s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
  errExitEN(s, "pthread_create");
```

> Согласно терминологии языка C левостороннее выражение (lvalue) — это выражение, ссылающееся на область хранилища[3]. Наиболее характерным его примером является идентификатор для переменной. Некоторые операторы также выдают такие выражения. Например, если p является указателем на область хранилища, то *p является левосторонним выражением. Согласно API потоков стандарта POSIX, errno переопределяется в функцию, возвращающую указатель на область хранилища, относящуюся к отдельному потоку (см. раздел 31.3).

[3] Подробнее о нем вы можете прочитать по адресу http://microsin.net/programming/arm/lvalue-rvalue.html. — *Примеч. пер.*

Для определения других типов ошибок используются функции `fatal()`, `usageErr()` и `cmdLineErr()`.

```
#include "tlpi_hdr.h"

void fatal(const char *format, ...);
void usageErr(const char *format, ...);
void cmdLineErr(const char *format, ...);
```

Функция `fatal()` применяется для определения типа ошибок общего характера, включая ошибки библиотечных функций, не устанавливающих значения для `errno`. У нее точно такой же список аргументов, что и у функции `printf()`, за исключением того, что к строке вывода автоматически добавляется символ конца строки. Она выдает отформатированный вывод на стандартное устройство вывода ошибки, а затем завершает выполнение программы с помощью `errExit()`.

Функция `usageErr()` предназначена для определения типов ошибок при использовании аргументов командной строки. Она принимает список аргументов в стиле `printf()` и выводит строку `Usage:`, за которой следует отформатированный вывод на стандартное устройство вывода ошибки, после чего она завершает выполнение программы путем вызова `exit()`. (Некоторые примеры программ в этой книге предоставляют свою собственную расширенную версию функции `usageErr()` под именем `usageError()`.)

Функция `cmdLineErr()` похожа на `usageErr()`, но предназначена для определения типов ошибок в переданных программе аргументах командной строки.

Реализации функций определения типов ошибок показаны в листинге 3.3.

Листинг 3.3. Функции обработки ошибок, используемые всеми программами

lib/error_functions.c

```
#include <stdarg.h>
#include "error_functions.h"
#include "tlpi_hdr.h"
#include "ename.c.inc"   /* Определяет ename и MAX_ENAME */

#ifdef __GNUC__
__attribute__ ((__noreturn__))
#endif

static void
terminate(Boolean useExit3)
{
  char *s;

  /* Сохраняет дамп ядра, если переменная среды EF_DUMPCORE определена
     и содержит непустую строку; в противном случае вызывает exit(3)
     или _exit(2), в зависимости от значения 'useExit3'. */
  s = getenv("EF_DUMPCORE");
  if (s != NULL && *s != '\0')
    abort();
  else if (useExit3)
    exit(EXIT_FAILURE);
  else
    _exit(EXIT_FAILURE);
}

static void
```

```
outputError(Boolean useErr, int err, Boolean flushStdout,
  const char *format, va_list ap)
{
#define BUF_SIZE 500
  char buf[BUF_SIZE], userMsg[BUF_SIZE], errText[BUF_SIZE];
  vsnprintf(userMsg, BUF_SIZE, format, ap);

  if (useErr)
    snprintf(errText, BUF_SIZE, " [%s %s]",
      (err > 0 && err <= MAX_ENAME) ?
      ename[err] : "?UNKNOWN?", strerror(err));
  else
    snprintf(errText, BUF_SIZE, ":");
  snprintf(buf, BUF_SIZE, "ERROR%s %s\n", errText, userMsg);
  if (flushStdout)
    fflush(stdout);   /* Сброс всего ожидающего стандартного вывода */
  fputs(buf, stderr);
  fflush(stderr);    /* При отсутствии построчной буферизации в stderr */
}

void
errMsg(const char *format, ...)
{
  va_list argList;
  int savedErrno;
  savedErrno = errno;  /* В случае ее изменения на следующем участке */
  va_start(argList, format);
  outputError(TRUE, errno, TRUE, format, argList);
  va_end(argList);
  errno = savedErrno;
}

void
errExit(const char *format, ...)
{
  va_list argList;
  va_start(argList, format);
  outputError(TRUE, errno, TRUE, format, argList);
  va_end(argList);
  terminate(TRUE);
}

void
err_exit(const char *format, ...)
{
  va_list argList;
  va_start(argList, format);
  outputError(TRUE, errno, FALSE, format, argList);
  va_end(argList);
  terminate(FALSE);
}

void
errExitEN(int errnum, const char *format, ...)
{
  va_list argList;
  va_start(argList, format);
  outputError(TRUE, errnum, TRUE, format, argList);
```

```
  va_end(argList);
  terminate(TRUE);
}

void
fatal(const char *format, ...)
{
  va_list argList;
  va_start(argList, format);
  outputError(FALSE, 0, TRUE, format, argList);
  va_end(argList);
  terminate(TRUE);
}

void
usageErr(const char *format, ...)
{
  va_list argList;
  fflush(stdout);  /* Сброс всего ожидающего стандартного вывода */
  fprintf(stderr, "Usage: ");
  va_start(argList, format);
  vfprintf(stderr, format, argList);
  va_end(argList);
  fflush(stderr);  /* При отсутствии построчной буферизации в stderr */
  exit(EXIT_FAILURE);
}

void
cmdLineErr(const char *format, ...)
{
  va_list argList;
  fflush(stdout); /* Сброс всего ожидающего стандартного вывода */
  fprintf(stderr, "Command-line usage error: ");
  va_start(argList, format);
  vfprintf(stderr, format, argList);
  va_end(argList);
  fflush(stderr); /* При отсутствии построчной буферизации в stderr */
  exit(EXIT_FAILURE);
}
```

lib/error_functions.c

Файл `ename.c.inc`, подключенный в листинге 3.3, показан в листинге 3.4. В этом файле определен массив строк `ename`, содержащий символьные имена, соответствующие каждому возможному значению `errno`. Наши функции обработки ошибок используют этот массив для вывода символьного имени, соответствующего конкретному номеру ошибки. Это выход из ситуации, при которой, с одной стороны, строка, возвращенная `strerror()`, не идентифицирует символьную константу, соответствующую ее сообщению об ошибке, в то время как, с другой стороны, на страницах руководства дается описание ошибок с использованием их символьных имен. По символьному имени на страницах руководства можно легко найти причину возникновения ошибки.

Содержимое файла ename.c.inc конкретизировано под архитектуру, поскольку значения errno в различных аппаратных архитектурах Linux несколько различаются. Версия, показанная в листинге 3.4, предназначена для системы Linux 2.6/x86-32. Этот файл был создан с использованием сценария (lib/Build_ename.sh), включенного в исходный код дистрибутива для данной книги. Сценарий можно использовать для создания версии ename.c.inc, которая должна подойти для конкретной аппаратной платформы и версии ядра.

Обратите внимание, что некоторые строки в массиве `ename` не заполнены. Они соответствуют неиспользуемым значениям ошибок. Кроме того, отдельные строки в `ename` состоят из двух названий ошибок, разделенных слешем. Они соответствуют тем случаям, когда у двух символьных имен ошибок имеется одно и то же числовое значение.

В файле ename.c.inc мы можем увидеть, что у ошибок EAGAIN и EWOULDBLOCK одно и то же значение. (В SUSv3 на этот счет есть явно выраженное разрешение, и значения этих констант одинаковы в большинстве, но не во всех других системах UNIX.) Эти ошибки возвращаются системным вызовом в тех случаях, когда он должен быть заблокирован (то есть вынужден находиться в режиме ожидания, прежде чем завершить свою работу), но вызывающий код потребовал, чтобы системный вызов вместо входа в режим блокировки вернул ошибку. Ошибка EAGAIN появилась в System V и возвращалась системными вызовами, выполняющими ввод/вывод, операции с семафорами, операции с очередями сообщений и блокировку файлов (fcntl()). Ошибка EWOULDBLOCK появилась в BSD и возвращалась блокировкой файлов (flock()) и системными вызовами, связанными с сокетами.

В SUSv3 ошибка EWOULDBLOCK упоминается только в спецификациях различных интерфейсов, связанных с сокетами. Для этих интерфейсов в SUSv3 разрешается возвращение при неблокируемых вызовах либо EAGAIN, либо EWOULDBLOCK. Для всех других неблокируемых вызовов в SUSv3 указана только ошибка EAGAIN.

Листинг 3.4. Имена ошибок Linux (для версии x86-32)

lib/ename.c.inc

```
static char *ename[] = {
  /*   0 */ "",
  /*   1 */ "EPERM", "ENOENT", "ESRCH", "EINTR", "EIO", "ENXIO", "E2BIG",
  /*   8 */ "ENOEXEC", "EBADF", "ECHILD", "EAGAIN/EWOULDBLOCK", "ENOMEM",
  /*  13 */ "EACCES", "EFAULT", "ENOTBLK", "EBUSY", "EEXIST", "EXDEV",
  /*  19 */ "ENODEV", "ENOTDIR", "EISDIR", "EINVAL", "ENFILE", "EMFILE",
  /*  25 */ "ENOTTY", "ETXTBSY", "EFBIG", "ENOSPC", "ESPIPE", "EROFS",
  /*  31 */ "EMLINK", "EPIPE", "EDOM", "ERANGE", "EDEADLK/EDEADLOCK",
  /*  36 */ "ENAMETOOLONG", "ENOLCK", "ENOSYS", "ENOTEMPTY", "ELOOP", "",
  /*  42 */ "ENOMSG", "EIDRM", "ECHRNG", "EL2NSYNC", "EL3HLT", "EL3RST",
  /*  48 */ "ELNRNG", "EUNATCH", "ENOCSI", "EL2HLT", "EBADE", "EBADR",
  /*  54 */ "EXFULL", "ENOANO", "EBADRQC", "EBADSLT", "", "EBFONT",
           "ENOSTR",
  /*  61 */ "ENODATA", "ETIME", "ENOSR", "ENONET", "ENOPKG", "EREMOTE",
  /*  67 */ "ENOLINK", "EADV", "ESRMNT", "ECOMM", "EPROTO", "EMULTIHOP",
  /*  73 */ "EDOTDOT", "EBADMSG", "EOVERFLOW", "ENOTUNIQ", "EBADFD",
  /*  78 */ "EREMCHG", "ELIBACC", "ELIBBAD", "ELIBSCN", "ELIBMAX",
  /*  83 */ "ELIBEXEC", "EILSEQ", "ERESTART", "ESTRPIPE", "EUSERS",
  /*  88 */ "ENOTSOCK", "EDESTADDRREQ", "EMSGSIZE", "EPROTOTYPE",
  /*  92 */ "ENOPROTOOPT", "EPROTONOSUPPORT", "ESOCKTNOSUPPORT",
  /*  95 */ "EOPNOTSUPP/ENOTSUP", "EPFNOSUPPORT", "EAFNOSUPPORT",
  /*  98 */ "EADDRINUSE", "EADDRNOTAVAIL", "ENETDOWN", "ENETUNREACH",
  /* 102 */ "ENETRESET", "ECONNABORTED", "ECONNRESET", "ENOBUFS", "EISCONN",
  /* 107 */ "ENOTCONN", "ESHUTDOWN", "ETOOMANYREFS", "ETIMEDOUT",
  /* 111 */ "ECONNREFUSED", "EHOSTDOWN", "EHOSTUNREACH", "EALREADY",
  /* 115 */ "EINPROGRESS", "ESTALE", "EUCLEAN", "ENOTNAM", "ENAVAIL",
  /* 120 */ "EISNAM", "EREMOTEIO", "EDQUOT", "ENOMEDIUM", "EMEDIUMTYPE",
  /* 125 */ "ECANCELED", "ENOKEY", "EKEYEXPIRED", "EKEYREVOKED",
  /* 129 */ "EKEYREJECTED", "EOWNERDEAD", "ENOTRECOVERABLE", "ERFKILL"
};

#define MAX_ENAME 132
```

lib/ename.c.inc

Функции для анализа числовых аргументов командной строки

Заголовочный файл в листинге 3.5 содержит объявление двух функций, часто используемых для анализа целочисленных аргументов командной строки: `getInt()` и `getLong()`. Главное преимущество использования этих функций вместо `atoi()`, `atol()` и `strtol()` заключается в том, что они предоставляют основные средства проверки на допустимость числовых аргументов.

```
#include "tlpi_hdr.h"

int getInt(const char *arg, int flags, const char *name);
long getLong(const char *arg, int flags, const char *name);
```

Обе возвращают значение *arg*, преобразованное в число

Функции `getInt()` и `getLong()` преобразуют строку, на которую указывает параметр `arg`, в значение типа `int` или `long` соответственно. Если `arg` не содержит допустимый строковый образ целого числа (то есть не состоит только лишь из цифр и символов `+` и `-`), эта функция выводит сообщение об ошибке и завершает выполнение программы.

Если аргумент `name` не содержит значение `NULL`, в нем должна находиться строка, идентифицирующая аргумент в параметре `arg`. Эта строка становится частью любого выводимого этими функциями сообщения об ошибке.

Аргумент `flags` предоставляет возможность управления работой функций `getInt()` и `getLong()`. Изначально они ожидают получения строк, содержащих десятичные целые числа со знаком. Путем логического сложения (`|`) в аргументе `flags` нескольких констант вида `GN_*`, определенных в листинге 3.5, можно выбрать иную основу для преобразования и ограничить диапазон чисел неотрицательными значениями или значениями больше нуля.

Листинг 3.5. Заголовочный файл для get_num.c

lib/get_num.h

```
#ifndef GET_NUM_H
#define GET_NUM_H
#define GN_NONNEG       01   /* Значение должно быть >= 0 */
#define GN_GT_0         02   /* Значение должно быть > 0 */

                             /* По умолчанию целые числа являются десятичными */
#define GN_ANY_BASE   0100   /* Можно использовать любое основание –
                                наподобие strtol(3) */
#define GN_BASE_8     0200   /* Значение выражено в виде восьмеричного числа */
#define GN_BASE_16    0400   /* Значение выражено в виде шестнадцатеричного числа */
long getLong(const char *arg, int flags, const char *name);

int getInt(const char *arg, int flags, const char *name);

#endif
```

lib/get_num.h

Реализации функций `getInt()` и `getLong()` показаны в листинге 3.6.

Хотя аргумент flags позволяет принудительно проверять диапазон допустимых значений, рассмотренный в основном тексте, в некоторых случаях в примерах программ такие провер-

ки не запрашиваются, даже если этот запрос кажется вполне логичным. Отказ от проверки диапазона в подобных случаях позволяет не только проводить эксперименты с правильным использованием системных вызовов и вызовов библиотечных функций, но и наблюдать, что произойдет, если будут предоставлены недопустимые аргументы. В приложениях, созданных для реальной работы, обычно вводятся более строгие проверки аргументов командной строки.

Листинг 3.6. Функции для анализа числовых аргументов командной строки

lib/get_num.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include "get_num.h"

static void
gnFail(const char *fname, const char *msg, const char *arg, const char *name)
{
  fprintf(stderr, "%s error", fname);
  if (name != NULL)
    fprintf(stderr, " (in %s)", name);
  fprintf(stderr, ": %s\n", msg);
  if (arg != NULL && *arg != '\0')
    fprintf(stderr, " offending text: %s\n", arg);
  exit(EXIT_FAILURE);
}

static long
getNum(const char *fname, const char *arg, int flags, const char *name)
{
  long res;
  char *endptr;
  int base;
  if (arg == NULL || *arg == '\0')
    gnFail(fname, "null or empty string", arg, name);
  base = (flags & GN_ANY_BASE) ? 0 : (flags & GN_BASE_8) ? 8 :
         (flags & GN_BASE_16) ? 16 : 10;
  errno = 0;
  res = strtol(arg, &endptr, base);
  if (errno != 0)
    gnFail(fname, "strtol() failed", arg, name);
  if (*endptr != '\0')
    gnFail(fname, "nonnumeric characters", arg, name);
  if ((flags & GN_NONNEG) && res < 0)
    gnFail(fname, "negative value not allowed", arg, name);
  if ((flags & GN_GT_0) && res <= 0)
    gnFail(fname, "value must be > 0", arg, name);
  return res;
}

long
getLong(const char *arg, int flags, const char *name)
{
  return getNum("getLong", arg, flags, name);
}
```

```
int
getInt(const char *arg, int flags, const char *name)
{
  long res;
  res = getNum("getInt", arg, flags, name);
  if (res > INT_MAX || res < INT_MIN)
    gnFail("getInt", "integer out of range", arg, name);
  return (int) res;
}
```

lib/get_num.c

3.6. Вопросы переносимости

В этом разделе мы рассмотрим, как создавать портируемые системные программы. В нем будут представлены макросы проверки возможностей и стандартные типы данных системы, определенные спецификацией SUSv3, а затем рассмотрены некоторые другие вопросы портируемости.

3.6.1. Макросы проверки возможностей

Поведение API системных вызовов и вызовов библиотечных функций регулируется различными стандартами (см. раздел 1.3). Одни стандарты определены организациями стандартизации, такими как Open Group (Single UNIX Specification), а другие — двумя исторически важными реализациями UNIX: BSD и System V, выпуск 4 (и объединенным System V Interface Definition).

Иногда при создании портируемого приложения могут понадобиться различные заголовочные файлы для предоставления только тех значений (констант, прототипов функций и т. д.), которые отвечают конкретному стандарту. Для этого определены несколько *макросов проверки возможностей*, которые доступны при компиляции программы.

Как один из вариантов, можно задать макрос в исходном коде программы до включения каких-либо заголовочных файлов:

```
#define _BSD_SOURCE 1
```

В качестве альтернативы можно воспользоваться ключом `-D` компилятора языка C:

```
$ cc -D_BSD_SOURCE prog.c
```

> Название «макрос проверки возможностей» может показаться странным, но, если взглянуть на него с точки зрения реализации, можно найти вполне определенный смысл. В реализации решается, какие свойства, доступные в каждом заголовке, нужно сделать видимыми путем проверки (с помощью #if), какие значения приложение определило для этих макросов.

Соответствующими стандартами установлены следующие макросы проверки возможностей (то есть их можно портировать на все системы, поддерживающие эти стандарты).

- `_POSIX_SOURCE` — если он задан (с любым значением), то предоставляет определения, соответствующие POSIX.1-1990 и ISO C (1990). Этот макрос заменен макросом `_POSIX_C_SOURCE`.

- _POSIX_C_SOURCE — если он определен со значением 1, то он производит такой же эффект, что и макрос _POSIX_SOURCE. Если он задан со значением большим или равным 199309, то также предоставляет определения для POSIX.1b (работа с системами реального времени). Если он приведен со значением, большим или равным 199506, то он также предоставляет определения для POSIX.1c (работа с потоками). Если он задан со значением 200112, то также предоставляет определения для базовой спецификации POSIX.1-2001 (то есть с включением XSI-расширения). (До выхода версии 2.3.3 заголовки glibc не интерпретировали значение 200112 для _POSIX_C_SOURCE.) Если макрос приведен со значением 200809, то он также предоставляет определения для базовой спецификации POSIX.1-2008. (До выхода версии 2.10 заголовочные файлы glibc не интерпретировали значение 200809 для _POSIX_C_SOURCE.)
- _XOPEN_SOURCE — если он задан (с любым значением), то предоставляет определения, соответствующие POSIX.1, POSIX.2 и X/Open (XPG4). Если он приведен со значением 500 или выше, то также предоставляет определения расширений SUSv2 (UNIX 98 и XPG5). Присвоение значения 600 или выше дополнительно приводит к предоставлению определения расширений SUSv3 XSI (UNIX 03) и расширений C99. (До выхода версии 2.2 заголовки glibc не интерпретировали значение 600 для _XOPEN_SOURCE.) Задание значения 700 или выше также приводит к предоставлению определения расширений SUSv4 XSI. (До выхода версии 2.10 заголовки glibc не интерпретировали значение 700 для _XOPEN_SOURCE.) Значения 500, 600 и 700 для _XOPEN_SOURCE были выбраны потому, что SUSv2, SUSv3 и SUSv4 являются соответственно выпусками Issues 5, 6 и 7 спецификаций X/Open.

Для glibc предназначены следующие макросы проверки возможностей.

- _BSD_SOURCE — если он задан (с любым значением), то предоставляет определения, соответствующие BSD. Явная установка одного лишь этого макроса приводит к тому, что в случае редких конфликтов стандартов предпочтение отдается определениям, соответствующим BSD.
- _SVID_SOURCE — если макрос приведен (с любым значением), то он предоставляет определения System V Interface Definition (SVID).
- _GNU_SOURCE — если он задан (с любым значением), то предоставляет все определения, предусмотренные предыдущими макросами, а также определения различных GNU-расширений.

Когда компилятор GNU C вызывается без специальных ключей, то по умолчанию определяются _POSIX_SOURCE, _POSIX_C_SOURCE=200809 (200112 с glibc версий от 2.5 до 2.9 или 199506 с glibc версии ниже 2.4), _BSD_SOURCE и _SVID_SOURCE.

Если определены отдельные макросы или компилятор вызван в одном из стандартных режимов (например, cc -ansi или cc -std=c99), то предоставляются только запрошенные определения. Существует одно исключение: если _POSIX_C_SOURCE не задан каким-либо другим образом и компилятор не вызван в одном из стандартных режимов, то _POSIX_C_SOURCE определяется со значением 200809 (200112 с glibc версий от 2.4 до 2.9 или 199506 с glibc версии ниже 2.4).

Несколько макросов дополняют друг друга, поэтому можно, к примеру, воспользоваться следующей командой cc для явного выбора тех же установок макросов, которые предоставляются по умолчанию:

```
$ cc -D_POSIX_SOURCE -D_POSIX_C_SOURCE=199506 \
     -D_BSD_SOURCE -D_SVID_SOURCE prog.c
```

Дополнительную информацию, уточняющую значения, присваиваемые каждому макросу проверки возможностей, можно найти в заголовочном файле <features.h> и на странице руководства feature_test_macros(7).

_POSIX_C_SOURCE, _XOPEN_SOURCE и POSIX.1/SUS

В POSIX.1-2001/SUSv3 указаны только макросы проверки возможностей `_POSIX_C_SOURCE` и `_XOPEN_SOURCE` с требованием, чтобы в соответствующих приложениях они были определены со значениями `200112` и `600`. Определение `_POSIX_C_SOURCE` со значением `200112` обеспечивает соответствие базовой спецификации POSIX.1-2001 (то есть *соответствие POSIX*, исключая XSI-расширение). Определение `_XOPEN_SOURCE` со значением `600` обеспечивает соответствие спецификации SUSv3 (то есть *соответствие XSI* — базовой спецификации плюс XSI-расширению). То же самое относится к POSIX.1-2008/SUSv4 с требованием, чтобы два макроса были определены со значениями `200809` и `700`.

В SUSv3 указывается, что установка для `_XOPEN_SOURCE` значения `600` должна предоставлять все свойства, включаемые, если `_POSIX_C_SOURCE` присвоено значение `200112`. Таким образом, для соответствия SUSv3 (то есть XSI) приложению необходимо определить только `_XOPEN_SOURCE`. В SUSv4 делается аналогичное требование: установка для `_XOPEN_SOURCE` значения `700` должна предоставлять все свойства, включаемые, если `_POSIX_C_SOURCE` присвоено значение `200809`.

Макросы проверки возможностей в прототипах функций и в исходном коде примеров

На страницах руководства дается описание, какой макрос или макросы проверки возможностей должны быть заданы, чтобы из заголовочного файла было видно конкретное определение константы или объявление функции.

Все примеры исходного кода в этой книге написаны таким образом, чтобы их можно было скомпилировать, используя либо настройки по умолчанию компилятора GNU C, либо следующие ключи:

```
$ cc -std=c99 -D_XOPEN_SOURCE=600
```

Для прототипа каждой функции перечислены все макросы проверки возможностей, которые должны быть указаны, если программа компилируется с настройками по умолчанию или выше приведенными ключами компилятора `cc`. На страницах руководства даны более точные описания макроса или макросов проверки возможностей, требуемых для предоставления объявления каждой функции.

3.6.2. Типы системных данных

При использовании стандартных типов языка C вам предоставляются различные типы данных реализации, например идентификаторы процессов, идентификаторы пользователей и смещения в файлах. Конечно, для объявления переменных, хранящих подобную информацию, можно было бы использовать основные типы языка C, например `int` и `long`, но это сокращает возможность портирования между системами UNIX по следующим причинам.

- Размеры этих основных типов от реализации к реализации UNIX отличаются друг от друга (например, `long` в одной системе может занимать 4 байта, а в другой — 8 байт). Иногда отличия могут прослеживаться даже в разных средах компиляции одной и той же реализации. Кроме того, в разных реализациях для представления одной и той же информации могут использоваться различные типы. Например, в одной системе идентификатор процесса может быть типа `int`, а в другой — типа `long`.
- Даже в одной и той же реализации UNIX типы, используемые для представления информации, могут в разных выпусках отличаться друг от друга. Наглядными примерами в Linux могут послужить идентификаторы пользователей и групп. В Linux 2.2

и более ранних версиях эти значения были представлены в 16 разрядах. В Linux 2.4 более поздних версиях они представлены в виде 32-разрядных значений.

Чтобы избежать подобных проблем портирования, в SUSv3 указываются различные стандартные типы системных данных, а к реализации предъявляются требования по надлежащему определению и использованию этих типов.

Каждый из этих типов определен с помощью имеющегося в языке C спецификатора `typedef`. Например, тип данных `pid_t` предназначен для представления идентификаторов процессов и в Linux/x86-32 определяется следующим образом:

```
typedef int pid_t;
```

У большинства стандартных типов системных данных имена оканчиваются на `_t`. Многие из них объявлены в заголовочном файле `<sys/types.h>`, хотя некоторые объявлены в других заголовочных файлах.

Приложение должно использовать эти определения типов, чтобы портируемым образом объявить используемые им переменные. Например, следующее объявление позволит приложению правильно представить идентификаторы процессов в любой совместимой с SUSv3 системе:

```
pid_t mypid;
```

В табл. 3.1 перечислены типы системных данных, которые будут встречаться в данной книге. Для отдельных типов в этой таблице SUSv3 требует, чтобы они были реализованы в качестве арифметических типов. Это означает, что при реализации в качестве базового типа может быть выбран либо целочисленный тип, либо тип с плавающей точкой (вещественный или комплексный).

Таблица 3.1. Отдельные типы системных данных

Тип данных	Требование к типу в SUSv3	Описание
blkcnt_t	Целое число со знаком	Количество блоков файла (см. раздел 15.1)
blksize_t	Целое число со знаком	Размер блока файла (см. раздел 15.1)
cc_t	Целое число без знака	Специальный символ терминала (см. раздел 58.4)
clock_t	Целое число или вещественное число с плавающей точкой	Системное время в тиках часов (см. раздел 10.7)
clockid_t	Арифметический тип	Идентификатор часов для определенных в POSIX.1b функций часов и таймера (см. раздел 23.6)
comp_t	В SUSv3 отсутствует	Сжатые тики часов (см. раздел 28.1)
dev_t	Арифметический тип	Номер устройства, состоящий из старшего и младшего номеров (см. раздел 15.1)
DIR	Требования к типу отсутствуют	Поток каталога (см. раздел 18.8)
fd_set	Структурный тип	Дескриптор файла, установленный для select() (см. подраздел 58.2.1)
fsblkcnt_t	Целое число без знака	Количество блоков в файловой системе (см. раздел 14.11)

Продолжение ⇗

Таблица 3.1 *(продолжение)*

Тип данных	Требование к типу в SUSv3	Описание
fsfilcnt_t	Целое число без знака	Количество файлов (см. раздел 14.11)
gid_t	Целое число	Числовой идентификатор группы (см. раздел 8.3)
id_t	Целое число	Базовый тип для хранения идентификаторов; достаточно большой, по крайней мере для pid_t, uid_t и gid_t
in_addr_t	32-разрядное целое число без знака	IPv4 адрес (см. раздел 55.4)
in_port_t	16-разрядное целое число без знака	Номер порта IP (см. раздел 55.4)
ino_t	Целое число без знака	Номер индексного дескриптора файла (см. раздел 15.1)
key_t	Арифметический тип	Ключ IPC в System V
mode_t	Целое число	Тип файла и полномочия доступа к нему (см. раздел 15.1)
mqd_t	Требования к типу отсутствуют, но не должен быть типом массива	Дескриптор очереди сообщений POSIX
msglen_t	Целое число без знака	Количество байтов, разрешенное в очереди сообщений в System V
msgqnum_t	Целое число без знака	Количество сообщений в очереди сообщений в System V
nfds_t	Целое число без знака	Количество дескрипторов файлов для poll() (см. подраздел 59.2.2)
nlink_t	Целое число	Количество жестких ссылок на файл (см. раздел 15.1)
off_t	Целое число со знаком	Смещение в файле или размер файла (см. разделы 4.7 и 15.1)
pid_t	Целое число со знаком	Идентификатор процесса, группы процессов или сессии (см. разделы 6.2, 34.2 и 34.3)
ptrdiff_t	Целое число со знаком	Разница между двумя значениями указателей в виде целого числа со знаком
rlim_t	Целое число без знака	Ограничение ресурса (см. раздел 36.2)
sa_family_t	Целое число без знака	Семейство адресов сокета (см. раздел 52.4)
shmatt_t	Целое число без знака	Количество прикрепленных процессов для совместно используемого сегмента памяти System V
sig_atomic_t	Целое число	Тип данных, который может быть доступен атомарно (см. раздел 21.1.3)
siginfo_t	Структурный тип	Информация об источнике сигнала (см. раздел 21.4)

Тип данных	Требование к типу в SUSv3	Описание
sigset_t	Целое число или структурный тип	Набор сигналов (см. раздел 20.9)
size_t	Целое число без знака	Размер объекта в байтах
socklen_t	Целочисленный тип, состоящий как минимум из 32 разрядов	Размер адресной структуры сокета в байтах (см. раздел 52.3)
speed_t	Целое число без знака	Скорость строки терминала (см. раздел 58.7)
ssize_t	Целое число со знаком	Количество байтов или (при отрицательном значении) признак ошибки
stack_t	Структурный тип	Описание дополнительного стека сигналов (см. раздел 21.3)
suseconds_t	Целое число со знаком в разрешенном диапазоне [-1, 1 000 000]	Интервал времени в микросекундах (см. раздел 10.1)
tcflag_t	Целое число без знака	Маска флагового разряда режима терминала (см. раздел 58.2)
time_t	Целое число или вещественное число с плавающей точкой	Календарное время в секундах от начала отсчета времени (см. раздел 10.1)
timer_t	Арифметический тип	Идентификатор таймера для функций временных интервалов POSIX.1b (см. раздел 23.6)
uid_t	Целое число	Числовой идентификатор пользователя (см. раздел 8.1)

При рассмотрении типов данных из табл. 3.1 в последующих главах я буду часто говорить, что некий тип «является целочисленным типом (указанным в SUSv3)». Это означает, что SUSv3 требует, чтобы тип был определен в качестве целого числа, но не требует обязательного использования конкретного присущего системе целочисленного типа (например, `short`, `int` или `long`). (Зачастую не будет говориться, какой именно присущий системе тип данных фактически применяется для представления в Linux каждого типа системных данных, поскольку портируемое приложение должно быть написано так, чтобы в нем не ставился вопрос о том, какой тип данных используется.)

Вывод значений типов системных данных

При выводе значений одного из типов системных данных, показанных в табл. 3.1 (например, `pid_t` и `uid_t`), нужно проследить, чтобы в вызов функции `printf()` не была включена зависимость представления данных. Она может возникнуть из-за того, что имеющиеся в языке C правила расширения аргументов приводят к преобразованию значений типа `short` в `int`, но оставляют значения типа `int` и `long` в неизменном виде. Иными словами, в зависимости от определения типа системных данных вызову `printf()` передается либо `int`, либо `long`. Но, поскольку функция `printf()` не может определять типы в ходе выполнения программы, вызывающий код должен предоставить эту информацию в явном виде, используя спецификатор формата `%d` или `%ld`. Проблема в том, что простое включение в программу одного из этих спецификаторов внутри вызова `printf()` создает зависимость от реализации. Обычно применяется подход, при котором используется спецификатор `%ld`, с неизменным приведением соответствующего значения к типу `long`:

```
pid_t mypid;
mypid = getpid();     /* Возвращает идентификатор вызывающего процесса */
printf("My PID is %ld\n", (long) mypid);
```

Из указанного выше подхода следует сделать одно исключение. Поскольку в некоторых средах компиляции тип данных `off_t` имеет размерность `long long`, мы приводим `off_t`-значения к этому типу и в соответствии с описанием из раздела 5.10 используем спецификатор `%lld`.

> В стандарте C99 для printf() определен модификатор длины z, показывающий, что Результат следующего целочисленного преобразования соответствует типу size_t или ssize_t. Следовательно, вместо использования %ld и приведения к этим типам можно указать %zd для ssize_t и аналогично %zu для size_t. Хотя этот спецификатор доступен в glibc, нам нужно избегать его применения, поскольку он доступен не во всех реализациях UNIX.
>
> В стандарте C99 также определен модификатор длины j, который указывает на то, что соответствующий аргумент имеет тип intmax_t (или uintmax_t) — целочисленный тип, гарантированно достаточно большой для представления целого значения любого типа. По сути, использование приведения к типу (intmax_t) и добавление спецификатора %jd должно заменить приведение к типу (long) и задание спецификатора %ld, а также стать лучшим способом вывода числовых значений типов системных данных. Первый подход справляется и со значениями long long, и с любыми расширенными целочисленными типами, такими как int128_t. Но и в данном случае нам следует избегать применения этой методики, поскольку она доступна не во всех реализациях UNIX.

3.6.3. Прочие вопросы, связанные с портированием

В этом разделе рассматриваются некоторые другие вопросы портирования, с которыми можно столкнуться при написании системных программ.

Инициализация и использование структур

В каждой реализации UNIX указывается диапазон стандартных структур, используемых в различных системных вызовах и библиотечных функциях. Рассмотрим в качестве примера структуру `sembuf`, которая применяется для представления операции с семафором, выполняемой системным вызовом `semop()`:

```
struct sembuf {
  unsigned short sem_num;     /* Номер семафора */
  short sem_op;               /* Выполняемая операция */
  short sem_flg;              /* Флаги операции */
};
```

Хотя в SUSv3 определены такие структуры, как `sembuf`, важно уяснить следующее.

- Обычно порядок определения полей внутри таких структур не определен.
- В некоторых случаях в такие структуры могут включаться дополнительные поля, имеющие отношение к конкретной реализации.

Таким образом, при использовании следующего инициализатора структуры не удастся обеспечить портируемость:

```
struct sembuf s = { 3, -1, SEM_UNDO };
```

Хотя этот инициализатор будет работать в Linux, он не станет работать в других реализациях, где поля в структуре `sembuf` определены в ином порядке. Чтобы инициализиро-

вать такие структуры портируемым образом, следует воспользоваться явно указанными инструкциями присваивания:

```
struct sembuf s;
s.sem_num = 3;
s.sem_op = -1;
s.sem_flg = SEM_UNDO;
```

Если применяется C99, то для написания эквивалентной инициализации можно воспользоваться новым синтаксисом:

```
struct sembuf s = { .sem_num = 3, .sem_op = -1, .sem_flg = SEM_UNDO };
```

Порядок следования элементов стандартных структур также придется учитывать, если нужно записать содержимое стандартной структуры в файл. Чтобы обеспечить в данном случае портируемость, мы не можем просто выполнить двоичную запись в структуру. Вместо этого поля структуры должны быть записаны по отдельности (возможно, в текстовом формате) в указанном порядке.

Использование макросов, которых может не быть во всех реализациях

В некоторых случаях макрос может быть не определен во всех реализациях UNIX. Например, широкое распространение получил макрос `WCOREDUMP()` (проверяет, создается ли дочерним процессом файл дампа ядра), но его определение в SUSv3 отсутствует. Следовательно, этот макрос может быть не представлен в некоторых реализациях UNIX. Чтобы для обеспечения портируемости преодолеть подобные обстоятельства, можно воспользоваться директивой препроцессора языка C `#ifdef`:

```
#ifdef WCOREDUMP
    /* Использовать макрос WCOREDUMP() */
#endif
```

Отличия в требуемых заголовочных файлах в разных реализациях

В зависимости от реализации UNIX будут различаться списки необходимых прототипу заголовочных файлов с различными системными вызовами и библиотечными функциями. В данной книге показываются требования применительно к Linux и обращается внимание на любые отклонения от SUSv3.

В некоторых функциях, кратко рассматриваемых в книге, показан конкретный заголовочный файл, сопровождаемый комментарием `/* For portability */` (`/* Из соображений портируемости */`). Это свидетельствует о том, что данный заголовочный файл для Linux или согласно SUSv3 не требуется, но, поскольку некоторым другим (особенно старым) реализациям он может понадобиться, нам приходится включать его в портируемые программы.

> Для многих определяемых POSIX.1-1990 функций требуется, чтобы заголовочный файл <sys/types.h> был включен ранее любого другого заголовочного файла, связанного с функцией. Но данное требование стало излишним, поскольку большинство современных реализаций UNIX не требуют от приложений включения этого заголовочного файла. Поэтому из SUSv1 это требование было удалено. И тем не менее при написании портируемых программ будет все же разумнее поставить этот заголовочный файл на первое место. (Но из наших примеров программ этот заголовочный файл исключен, поскольку для Linux он не требуется, и мы можем сократить длину примеров на одну строку.)

3.7. Резюме

Системные вызовы позволяют процессам запрашивать сервисы из ядра. Даже для самых простых системных вызовов по сравнению с вызовом функции из пользовательского пространства характерно существенное потребление ресурсов, поскольку для выполнения системного вызова система должна временно переключиться в режим ядра, а ядро должно проверить аргументы системного вызова и осуществить портирование данных между пользовательской памятью и памятью ядра.

Стандартная библиотека языка C предоставляет множество библиотечных функций, выполняющих широкий диапазон задач. Одним библиотечным функциям для выполнения их работы требуются системные вызовы, другие же выполняют свои задачи исключительно в пользовательском пространстве. В Linux в качестве реализации стандартной библиотеки языка C обычно применяется glibc.

Большинство системных вызовов и библиотечных функций возвращают признак, показывающий, каким был вызов — успешным или неудачным. Надо всегда проверять этот признак.

В данной главе были введены некоторые функции, реализованные нами для использования в примерах книги. Задачи, выполняемые этими функциями, включают диагностику ошибок и анализ аргументов командной строки.

В главе рассмотрены правила и подходы, которыми можно воспользоваться для написания портируемых системных программ, запускаемых на любой соответствующей стандарту системе.

При компиляции приложения можно задавать различные макросы проверки возможностей. Они управляют определениями, которые предоставляются заголовочными файлами. Их использование пригодится для обеспечения гарантий соответствия программы формальному или определяемому реализацией стандарту (или стандартам).

Портируемость системных программ можно улучшить, используя типы системных данных, которые определены в различных стандартах и могут отличаться от типов, присущих языку C. В SUSv3 указывается широкий диапазон типов системных данных, которые должны поддерживаться реализациями и использоваться приложениями.

3.8. Упражнение

3.1. Когда для перезапуска системы используется характерный для Linux системный вызов `reboot()`, в качестве второго аргумента `magic2` необходимо указать одно из магических чисел (например, `LINUX_REBOOT_MAGIC2`). Какой смысл несут эти числа? (Подсказка: обратите внимание на шестнадцатеричное представление такого числа[4].)

[4] Таким образом закодировны дни рождения Торвальдса и его дочерей: https://stackoverflow.com/questions/4808748/magic-numbers-of-the-linux-reboot-system-call.

4 Файловый ввод-вывод: универсальная модель ввода-вывода

Теперь перейдем к подробному рассмотрению API системных вызовов. Лучше всего начать с файлов, поскольку они лежат в основе всей философии UNIX. Основное внимание в этой главе будет уделено системным вызовам, предназначенным для выполнения файлового ввода-вывода.

Вы узнаете, что такое дескриптор файла, а затем мы рассмотрим системные вызовы, составляющие так называемую универсальную модель ввода-вывода. Это те самые системные вызовы, которые открывают и закрывают файл, а также считывают и записывают данные.

Особое внимание мы уделим вводу-выводу, относящимся к дисковым файлам. При этом большая часть информации из текущей главы важна для усвоения материала последующих глав, поскольку те же самые системные вызовы используются для выполнения ввода-вывода во всех типах файлов, в том числе конвейерах и терминалах.

В главе 5 рассматриваемые здесь вопросы будут расширены дополнительными сведениями, касающимися файлового ввода-вывода. Еще одна особенность файлового ввода-вывода — буферизация — настолько сложна, что заслуживает отдельной главы. Буферизация ввода-вывода в ядре и с помощью средств библиотеки stdio будет описана в главе 13.

4.1. Общее представление

Все системные вызовы для выполнения ввода-вывода совершаются в отношении открытых файлов с использованием *дескриптора файла*, представленного неотрицательным (обычно небольшим) целым числом. Дескрипторы файлов применяются для обращения ко всем типам открытых файлов, включая конвейеры, FIFO-устройства, сокеты, терминалы, аппаратные устройства и обычные файлы. Каждый процесс имеет свой собственный набор дескрипторов файлов.

Обычно от большинства программ ожидается возможность использования трех стандартных дескрипторов файлов, перечисленных в табл. 4.1. Эти три дескриптора открыты оболочкой от имени программы еще до запуска самой программы. Точнее говоря, программа наследует у оболочки копии дескрипторов файлов, а оболочка обычно работает с этими всегда открытыми тремя дескрипторами файлов. (В интерактивной оболочке эти три дескриптора обычно ссылаются на терминал, на котором запущена оболочка.) Если в командной строке указано перенаправление ввода-вывода, то оболочка перед запуском программы обеспечивает соответствующее изменение дескрипторов файлов.

Таблица 4.1. Стандартные дескрипторы файлов

Дескриптор файла	Назначение	Имя согласно POSIX	Поток stdio
0	Стандартный ввод	STDIN_FILENO	stdin
1	Стандартный вывод	STDOUT_FILENO	stdout
2	Стандартная ошибка	STDERR_FILENO	stderr

При ссылке на эти дескрипторы файлов в программе можно использовать либо номера (0, 1 или 2), либо, что предпочтительнее, стандартные имена POSIX, определенные в файле `<unistd.h>`.

> Хотя переменные stdin, stdout и stderr изначально ссылаются на стандартные ввод, вывод и ошибку процесса, с помощью библиотечной функции freopen() их можно изменить для ссылки на любой файл. В качестве части своей работы freopen() способен изменить дескриптор файла на основе вновь открытого потока. Иными словами, после вызова freopen() в отношении, к примеру, stdout, нельзя с полной уверенностью предполагать, что относящийся к нему дескриптор файла по-прежнему имеет значение 1.

Рассмотрим следующие четыре системных вызова, которые являются ключевыми для выполнения файлового ввода-вывода (языки программирования и программные пакеты обычно используют их исключительно опосредованно, через библиотеки ввода-вывода).

- `fd = open(pathname, flags, mode)` — открытие файла, идентифицированного по путевому имени — `pathname`, с возвращением дескриптора файла, который используется для обращения к открытому файлу в последующих вызовах. Если файл не существует, вызов `open()` может его создать в зависимости от установки битовой маски аргумента флагов — `flags`. В аргументе флагов также указывается, с какой целью открывается файл: для чтения, для записи или для проведения обеих операций. Аргумент `mode` (режим), определяет права доступа, которые будут накладываться на файл, если он создается этим вызовом. Если вызов `open()` не будет использоваться для создания файла, этот аргумент игнорируется и может быть опущен.
- `numread = read(fd, buffer, count)` — считывание не более указанного в `count` количества байтов из открытого файла, ссылка на который дана в `fd`, и сохранение их в буфере `buffer`. При вызове `read()` возвращается количество фактически считанных байтов. Если данные не могут быть считаны (то есть встретился конец файла), `read()` возвращает `0`.
- `numwritten = write(fd, buffer, count)` — запись из буфера байтов, количество которых указано в `count`, в открытый файл, ссылка на который дана в `fd`. При вызове `write()` возвращается количество фактически записанных байтов, которое может быть меньше значения, указанного в `count`.
- `status = close(fd)` — вызывается после завершения ввода-вывода с целью высвобождения дескриптора файла `fd` и связанных с ним ресурсов ядра.

Перед тем как подробно разбирать эти системные вызовы, посмотрим на небольшую демонстрацию их использования в листинге 4.1. Эта программа является простой версией команды `cp(1)`. Она копирует содержимое существующего файла, чье имя указано в первом аргументе командной строки, в новый файл с именем, указанным во втором аргументе командной строки.

Программой, показанной в листинге 4.1, можно воспользоваться следующим образом:

```
$ ./copy oldfile newfile
```

Листинг 4.1. Использование системных вызовов ввода-вывода

fileio/copy.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#ifndef BUF_SIZE        /* Позволяет "cc -D" перекрыть определение */
#define BUF_SIZE 1024
#endif

int
main(int argc, char *argv[])
{
  int inputFd, outputFd, openFlags;
  mode_t filePerms;
  ssize_t numRead;
  char buf[BUF_SIZE];

  if (argc != 3 || strcmp(argv[1], "--help") == 0)
    usageErr("%s old-file new-file\n", argv[0]);
  /* Открытие файлов ввода и вывода */
  inputFd = open(argv[1], O_RDONLY);
  if (inputFd == -1)
    errExit("opening file %s", argv[1]);
  openFlags = O_CREAT | O_WRONLY | O_TRUNC;
  filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
              S_IROTH | S_IWOTH;      /* rw-rw-rw- */
  outputFd = open(argv[2], openFlags, filePerms);
  if (outputFd == -1)
    errExit("opening file %s", argv[2]);

  /* Перемещение данных до достижения конца файла ввода или возникновения ошибки */
  while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
    if (write(outputFd, buf, numRead) != numRead)
        fatal("couldn't write whole buffer");
  if (numRead == -1)
    errExit("read");
  if (close(inputFd) == -1)
    errExit("close input");
  if (close(outputFd) == -1)
    errExit("close output");

  exit(EXIT_SUCCESS);
}
```

fileio/copy.c

4.2. Универсальность ввода-вывода

Одна из отличительных особенностей модели ввода-вывода UNIX состоит в *универсальности ввода-вывода*. Это означает, что одни и те же четыре системных вызова — `open()`, `read()`, `write()` и `close()` — применяются для выполнения ввода-вывода во всех типах файлов, включая устройства, например терминалы. Следовательно, если программа написана с использованием лишь этих системных вызовов, она будет работать с любым типом файла. Например, следующие примеры показывают вполне допустимое использование программы, чей код приведен в листинге 4.1:

```
$ ./copy test test.old          Копирование обычного файла
$ ./copy a.txt /dev/tty         Копирование обычного файла в этот терминал
$ ./copy /dev/tty b.txt         Копирование ввода с этого терминала в обычный файл
$ ./copy /dev/pts/16 /dev/tty   Копирование ввода с другого терминала
```

Универсальность ввода-вывода достигается обеспечением того, что в каждой файловой системе и в каждом драйвере устройства реализуется один и тот же набор системных вызовов ввода-вывода. Поскольку детали реализации конкретной файловой системы или устройства обрабатываются внутри ядра, при написании прикладных программ мы можем вообще игнорировать факторы, относящиеся к устройству. Когда требуется получить доступ к конкретным свойствам файловой системы или устройства, в программе можно использовать всеобъемлющий системный вызов `ioctl()` (см. раздел 4.8). Он предоставляет интерфейс для доступа к свойствам, которые выходят за пределы универсальной модели ввода-вывода.

4.3. Открытие файла: open()

Системный вызов `open()` либо открывает существующий файл, либо создает и открывает новый файл.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Возвращает дескриптор при успешном завершении или `-1` при ошибке

Чтобы файл открылся, он должен пройти идентификацию по аргументу `pathname`. Если в этом аргументе находится символьная ссылка, она разыменовывается. В случае успеха `open()` возвращает дескриптор файла, который используется для ссылки на файл в последующих системных вызовах. В случае ошибки `open()` возвращает `-1`, а для `errno` устанавливается соответствующее значение.

Аргумент `flags` является битовой маской, указывающей *режим доступа* к файлу с использованием одной из констант, перечисленных в табл. 4.2.

> В ранних версиях UNIX вместо имен, приведенных в табл. 4.2, использовались числа 0, 1 и 2. В более современных реализациях UNIX эти константы определяются с указанными в таблице значениями. Из нее видно, что O_RWDW (10 в двоичном представлении) не совпадает с результатом операции O_RDONLY | O_WRONLY (0 | 1 = 1); последняя комбинация прав доступа является логической ошибкой.

Когда `open()` применяется для создания нового файла, аргумент битовой маски режима (`mode`) указывает на права доступа, которые должны быть присвоены файлу. (Используемый тип данных `mode_t` является целочисленным типом, определенным в SUSv3.) Если при вызове `open()` не указывается флаг `O_CREAT`, то аргумент `mode` может быть опущен.

Таблица 4.2. Режимы доступа к файлам

Режим доступа	Описание
O_RDONLY	Открытие файла только для чтения
O_WRONLY	Открытие файла только для записи
O_RDWR	Открытие файла как для чтения, так и для записи

Подробное описание прав доступа дается в разделе 15.4. Позже будет показано, что права доступа, фактически присваиваемые новому файлу, зависят не только от аргумента `mode`, но и от значения `umask` процесса (см. подраздел 15.4.6) и от (дополнительно имеющегося) списка контроля доступа по умолчанию (access control list) (см. раздел 17.6) родительского каталога. А пока просто отметим для себя, что аргумент `mode` может быть указан в виде числа (обычно восьмеричного) или, что более предпочтительно, путем применения операции логического ИЛИ (`|`) к нескольким константам битовой маски, перечисленным в табл. 15.4.

В листинге 4.2 показаны примеры использования `open()`. В некоторых из них указываются дополнительные биты флагов, которые вскоре будут рассмотрены.

Листинг 4.2. Примеры использования open()

```
/* Открытие существующего файла для чтения */
fd = open("startup", O_RDONLY);

if (fd == -1)
    errExit("open");
/* Открытие нового или существующего файла для чтения и записи с усечением до нуля
   байтов; предоставление владельцу исключительных прав доступа на чтение и запись */

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
/* Открытие нового или существующего файла для записи; записываемые данные
   должны всегда добавляться в конец файла */
fd = open("w.log", O_WRONLY | O_CREAT | O_APPEND,
                   S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Номер дескриптора файла, возвращаемый системным вызовом open()

В SUSv3 определяется, что при успешном завершении системного вызова `open()` гарантируется, что процессу будет выделен наименьший неиспользуемый дескриптор файл. Этим можно воспользоваться, чтобы быть уверенными, что файл открыт с конкретным дескриптором файла. Например, следующая последовательность дает уверенность в том, что файл открывается с использованием дескриптора стандартного ввода (нулевой файловый дескриптор).

```
if (close(STDIN_FILENO) == -1)      /* Закрытие нулевого файлового дескриптора */
  errExit("close");
fd = open(pathname, O_RDONLY);
if (fd == -1)
  errExit("open");
```

Поскольку дескриптор файла `0` не используется, `open()` гарантирует открытие файла с этим дескриптором. В разделе 5.5 показывается применение для получения аналогичного результата вызовов `dup2()` и `fcntl()`, но с более гибким управлением дескриптором файла. В этом разделе также приводится пример того, как можно извлечь пользу от управления файловым дескриптором, с которым открывается файл.

4.3.1. Аргумент flags системного вызова open()

В некоторых примерах вызова `open()`, показанных в листинге 4.2, во флаги, кроме режима доступа к файлу, включены дополнительные биты (`O_CREAT`, `O_TRUNC` и `O_APPEND`). Рассмотрим аргумент `flags` более подробно. В табл. 4.3 приведен полный набор констант, любая комбинация которых с помощью побитового ИЛИ (`|`), может быть передана в аргументе

flags. В последнем столбце показано, какие из этих констант были включены в стандарт SUSv3 или SUSv4.

Таблица 4.3. Значения для аргументов флагов системного вызова open()

Флаг	Назначение	SUS?
O_RDONLY	Открытие только для чтения	v3
O_WRONLY	Открытие только для записи	v3
O_RDWR	Открытие для чтения и записи	v3
O_CLOEXEC	Установка флага закрытия при выполнении (close-on-exec) (начиная с версии Linux 2.6.23)	v4
O_CREAT	Создание файла, если он еще не существует	v3
O_DIRECTORY	Отказ, если аргумент pathname указывает не на каталог	v4
O_EXCL	С флагом O_CREAT: исключительное создание файла	v3
O_LARGEFILE	Используется в 32-разрядных системах для открытия больших файлов	
O_NOCTTY	Pathname запрещено становиться управляющим терминалом данного процесса	v3
O_NOFOLLOW	Запрет на разыменование символьных ссылок	v4
O_TRUNC	Усечение существующего файла до нулевой длины	v3
O_APPEND	Записи добавляются исключительно в конец файла	v3
O_ASYNC	Генерация сигнала, когда возможен ввод/вывод	
O_DIRECT	Операции ввода-вывода осуществляются без использования кэша	
O_DSYNC	Синхронизированный ввод-вывод с обеспечением целостности данных (начиная с версии Linux 2.6.33)	v3
O_NOATIME	Запрет на обновление времени последнего доступа к файлу при чтении с помощью системного вызова read() (начиная с версии Linux 2.6.8)	
O_NONBLOCK	Открытие в неблокируемом режиме	v3
O_SYNC	Ведение записи в файл в синхронном режиме	v3

Константы в табл. 4.3 разделяются на следующие группы.

- *Флаги режима доступа к файлу*. Это рассмотренные ранее флаги O_RDONLY, O_WRONLY и O_RDWR. Их можно извлечь с помощью операции F_GETFL функции fcntl() (см. раздел 5.3).
- *Флаги создания файла*. Это флаги, показанные во второй части табл. 4.3. Они управляют различными особенностями поведения системного вызова open(), а также вариантами для последующих операций ввода-вывода. Эти флаги не могут быть извлечены или изменены.
- *Флаги состояния открытия файла*. Это все остальные флаги, показанные в табл. 4.3. Они могут быть извлечены и изменены с помощью операций F_GETFL и F_SETFL функции fcntl() (см. раздел 5.3). Эти флаги иногда называют просто *флагами состояния файла*.

Начиная с версии ядра 2.6.22, для получения информации о дескрипторах файлов любого имеющегося в системе процесса могут быть прочитаны файлы в каталоге /proc/PID/fdinfo, которые есть только в Linux. В этом каталоге находится по одному файлу для каждого дескриптора открытого процессом файла, с именем, совпадающим с номером дескриптора. В поле pos этого файла показано текущее смещение в файле (см. раздел 4.7). В поле flags находится восьмеричное число, показывающее флаги режима доступа к файлу и флаги состояния открытого файла. (Чтобы декодировать эти числа, нужно посмотреть на числовые значения флагов в заголовочных файлах библиотеки языка C.)

Рассмотрим константы флагов подробнее.

- `O_APPEND` — записи добавляются исключительно в конец файла. Значение этого флага рассматривается в разделе 5.1.
- `O_ASYNC` — генерирование сигнала при появлении возможности ввода-вывода с использованием файлового дескриптора, возвращенного системным вызовом `open()`. Это свойство называется *вводом-выводом под управлением сигналов*. Оно доступно только для файлов определенного типа, таких как терминалы, FIFO-устройства и сокеты. (Флаг `O_ASYNC` не определен в SUSv3, но в большинстве реализаций UNIX он или его синоним `FASYNC` присутствует.) В Linux указание флага `O_ASYNC` при вызове `open()` не имеет никакого эффекта. Чтобы включить ввод/вывод с сигнальным управлением, нужно установить этот флаг, указывая в `fcntl()` операцию `F_SETFL` (см. раздел 5.3). (Некоторые другие реализации UNIX ведут себя аналогичным образом.) Дополнительные сведения о флаге `O_ASYNC` можно найти в разделе 59.3.
- `O_CLOEXEC` (с выходом Linux 2.6.23) — установка флага закрытия при выполнении — флага close-on-exec (`FD_CLOEXEC`) для нового дескриптора файла. Флаг `FD_CLOEXEC` рассматривается в разделе 27.4. Использование флага `O_CLOEXEC` позволяет программе не выполнять дополнительные операции `F_GETFD` и `F_SETFD` при вызове `fcntl()` для установки флага close-on-exec. Кроме того, в многопоточных программах необходимо избегать состояния гонки, возможное при использовании данной технологии. Например, такое состояние может возникать, когда один поток открывает дескриптор файла, а затем пытается пометить его флагом close-on-exec, и в то же самое время другой поток выполняет системный вызов `fork()`, а затем `exec()` из какой-нибудь другой программы. (Предположим, что второй поток справляется и с `fork()`, и с `exec()` в период между тем, как первый поток открывает файловый дескриптор и использует `fcntl()` для установки флага close-on-exec.) Такое состязание может привести к тому, что открытые дескрипторы файлов могут непреднамеренно быть переданы небезопасным программам. (Состояние гонки подробнее рассматривается в разделе 5.1.)
- `O_CREAT` — создание нового, пустого файла, если такого файла еще не существует. Этот флаг срабатывает, даже если файл открывается только для чтения. Если указывается `O_CREAT`, то при вызове `open()` нужно также обязательно предоставлять аргумент `mode`. В противном случае права доступа к новому файлу будут установлены по какому-либо произвольному значению, взятому из стека.
- `O_DIRECT` — разрешение файловому вводу-выводу обходить буферный кэш. Это свойство рассматривается в разделе 13.6. Чтобы сделать определение этой константы доступным из `<fcntl.h>`, должен быть задан макрос проверки возможностей `_GNU_SOURCE`.
- `O_DIRECTORY` — возвращение ошибки (в этом случае `errno` присваивается значение `ENOTDIR`), если путевое имя не является каталогом. Этот флаг представляет собой расширение, разработанное главным образом для реализации `opendir()` (см. раздел 18.8). Чтобы сделать определение этой константы доступным из `<fcntl.h>`, должен быть задан макрос проверки возможностей `_GNU_SOURCE`.

- O_DSYNC (с выходом Linux 2.6.33) — выполнение записи в файл в соответствии с требованиями соблюдения целостности данных при синхронизированном вводе-выводе. Обратите внимание на буферизацию ввода-вывода на уровне ядра, рассматриваемую в разделе 13.3.
- O_EXCL — используется в сочетании с флагом O_CREAT как указание, что файл, если он уже существует, не должен быть открыт. Вместо этого системный вызов open() не выполняется, а errno присваивается значение EEXIST. Иными словами, флаг позволяет вызывающему коду убедиться в том, что это и есть процесс, создающий файл. Проверка существования и создание файла выполняются в атомарном режиме. Понятие атомарности рассматривается в разделе 5.1. Когда в качестве флагов указаны и O_CREAT, и O_EXCL, системный вызов open() не выполняется (с ошибкой EEXIST), если путевое имя является символьной ссылкой. Такое поведение в SUSv3 требуется, чтобы привилегированные приложения могли создавать файл в определенном месте и при этом исключалась возможность создания файла в другом месте с использованием символьной ссылки (например, в системном каталоге), которая негативно скажется на безопасности.
- O_LARGEFILE — открытие файла в режиме поддержки больших файлов. Этот флаг применяется в 32-разрядных системах для работы с большими файлами. Хотя флаг O_LARGEFILE в SUSv3 не указан, его можно найти в некоторых других реализациях UNIX. В 64-разрядных реализациях Linux, таких как Alpha и IA-64, этот флаг работать не будет. Дополнительные сведения о нем даются в разделе 5.10.
- O_NOATIME (с выходом Linux 2.6.8) — отказ от обновления времени последнего обращения к файлу (поле st_atime рассматривается в разделе 15.1) при чтении из файла. Чтобы можно было воспользоваться этим флагом, действующий идентификатор пользователя вызывающего процесса должен соответствовать владельцу файла или же процесс должен быть привилегированным (CAP_FOWNER). В противном случае системный вызов open() не будет выполнен и будет выдана ошибка EPERM. (В действительности, как указывается в разделе 9.5, для непривилегированного процесса речь идет о пользовательском идентификаторе файловой системы, а не о его действующем ID пользователя. Именно он должен совпадать с идентификатором пользователя файла при открытии этого файла с флагом O_NOATIME.) Флаг относится к нестандартным расширениям Linux. Для предоставления его определения из <fcntl.h> следует задать макрос проверки возможностей _GNU_SOURCE. Флаг O_NOATIME предназначен для использования программами индексации и создания резервных копий. Его применение может существенно сократить объем активного использования диска, поскольку не потребуются многочисленные перемещения вперед и назад по диску для чтения содержимого файла, а также обновления времени последнего обращения к файлу в индексном дескрипторе (см. раздел 14.4). Функциональные возможности, похожие на обеспечиваемые флагом O_NOATIME, доступны при использовании флагов MS_NOATIME и FS_NOATIME_FL (см. раздел 15.5) во время системного вызова mount() (см. подраздел 14.8.1).
- O_NOCTTY — предотвращение превращения открываемого файла в управляющий терминал, если он является терминальным устройством. Управляющие терминалы рассматриваются в разделе 34.4. Если открываемый файл не является терминалом, флаг не работает.
- O_NOFOLLOW — обычно системный вызов open() разыменовывает символьную ссылку. Но, если задан флаг O_NOFOLLOW и аргумент pathname является символьной ссылкой, вызов open() не выполняется (в errno заносится значение ELOOP). Этот флаг особенно пригодится в привилегированных программах, чтобы обеспечить отказ от разыменования символьной ссылки при системном вызове open(). Для предоставления определения этого флага из <fcntl.h> следует добавить макрос проверки возможностей _GNU_SOURCE.

- O_NONBLOCK — открытие файла в неблокируемом режиме (см. раздел 5.9).
- O_SYNC — открытие файла для синхронизированного ввода-вывода. Обратите внимание на буферизацию ввода-вывода на уровне ядра, рассматриваемую в разделе 13.3.
- O_TRUNC — усечение файла до нулевой длины с удалением любых существующих данных, если файл уже существует и является обычным. В Linux усечение происходит, когда файл открывается для чтения или для записи (в обоих случаях нужны права доступа к файлу для записи). В SUSv3 сочетание флагов O_RDONLY и O_TRUNC не оговорено техническими условиями, но большинство других реализаций UNIX ведут себя так же, как и Linux.

4.3.2. Ошибки, возвращаемые из системного вызова open()

В случае возникновения ошибки при попытке открытия файла системный вызов open() возвращает -1, а в errno идентифицируется причина ошибки. Далее перечислены возможные ошибки, которые могут произойти (вдобавок к тем, что уже были упомянуты при описании только что рассмотренного аргумента flags).

- EACCES — права доступа к файлу не позволяют вызывающему процессу открыть файл в режиме, указанном флагами. Из-за прав доступа к каталогу доступ к файлу невозможен или файл не существует и не может быть создан.
- EISDIR — указанный файл является каталогом, а вызывающий процесс пытается открыть его для записи. Это запрещено. (С другой стороны, есть случаи, когда может быть полезно открыть каталог для чтения. Пример будет рассмотрен в разделе 18.11.)
- EMFILE — достигнуто ограничение ресурса процесса на количество файловых дескрипторов (RLIMIT_NOFILE, рассматривается в разделе 36.3).
- ENFILE — достигнуто ограничение на количество открытых файлов, накладываемое на всю систему.
- ENOENT — заданный файл не существует, и ключ O_CREAT не указан; или O_CREAT был указан, и один из каталогов в путевом имени не существует или является символьной ссылкой, ведущей на несуществующее путевое имя (битой ссылкой).
- EROFS — указанный файл находится в файловой системе, предназначенной только для чтения, а вызывающий процесс пытается открыть его для записи.
- ETXTBSY — заданный файл является исполняемым (программой), и в данный момент выполняется. Изменение исполняемого файла, связанного с выполняемой программой (то есть его открытие для записи), запрещено. (Чтобы изменить исполняемый файл, сначала следует завершить программы.)

При дальнейшем описании других системных вызовов или библиотечных функций мы не будем перечислять возможные ошибки, которые могут произойти при подобных обстоятельствах. (Этот перечень можно найти на соответствующих страницах руководства для каждого системного вызова или библиотечной функции.) Во-первых, это связано с тем, что open() — первый системный вызов, который мы подробно рассматриваем, и из списка выше можно увидеть, что системный вызов или библиотечная функция может потерпеть неудачу по любой из множества причин. Во-вторых, конкретные причины, по которым вызов open() может закончиться неудачей, сами по себе составляют весьма интересный список, иллюстрируя множество факторов и проверок, которые нужно учитывать при обращении к файлу. (Приведенный выше список неполон: дополнительные причины отказа open() можно найти на странице руководства open(2).)

4.3.3. Системный вызов creat()

В ранних реализациях UNIX у `open()` было только два аргумента, и этот вызов нельзя было использовать для создания нового файла. Вместо него для создания и открытия нового файла использовался системный вызов `creat()`.

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Возвращает дескриптор файла или `-1` при ошибке

Системный вызов `creat()` создает и открывает новый файл с заданным путевым именем или, если файл уже существует, открывает файл и усекает его до нулевой длины. В качестве результата своей работы `creat()` возвращает дескриптор файла, который может быть использован в последующих системных вызовах. Вызов `creat()` эквивалентен такому вызову `open()`:

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Поскольку аргумент `flags` системного вызова `open()` предоставляет больше контроля над тем, как открывается файл (например, вместо `O_WRONLY` можно указать `O_RDWR`), системный вызов `creat()` теперь считается устаревшим, хотя в старых программах он еще встречается.

4.4. Чтение из файла: read()

Системный вызов `read()` позволяет считывать данные из открытого файла, на который ссылается дескриптор `fd`.

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Возвращает количество считанных байтов, `0` при EOF или `-1` при ошибке

Аргумент `count` определяет максимальное количество считываемых байтов (тип данных `size_t` — беззнаковый целочисленный). Аргумент `buffer` предоставляет адрес буфера памяти, в который должны быть помещены входные данные. Этот буфер должен иметь длину в байтах не менее той, что задана в аргументе `count`.

> Системные вызовы не выделяют память под буферы, которые используются для возвращения информации вызывающему процессу. Вместо этого следует передать указатель на ранее выделенный буфер памяти подходящего размера. Этим вызовы отличаются от ряда библиотечных функций, которые выделяют буферы в памяти с целью возвращения информации вызывающему процессу.

При успешном вызове `read()` возвращается количество фактически считанных байтов или `0`, если встретился символ конца файла. При ошибке обычно возвращается `-1`. Тип данных `ssize_t` относится к целочисленному типу со знаком. Этот тип используется для хранения количества байтов или значения `-1`, которое служит признаком ошибки.

При вызове `read()` количество считанных байтов может быть меньше запрашиваемого. Возможная причина для обычных файлов — близость считываемой области к концу файла.

При использовании вызова `read()` в отношении других типов файлов, например конвейеров, FIFO-устройств, сокетов или терминалов, также могут складываться различные обстоятельства, при которых количество считанных байтов оказывается меньше запрашиваемого. Например, изначально применение `read()` в отношении терминала приводит к считыванию символов только до следующего встреченного символа новой строки (`\n`). Эти случаи будут рассматриваться при изучении других типов файлов далее в книге.

При использовании `read()` для ввода последовательности символов из, скажем, терминала, можно предполагать, что сработает следующий код:

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
  errExit("read");
printf("The input data was: %s\n", buffer);
```

Этот фрагмент кода выведет весьма странные данные, поскольку в них, скорее всего, будут включены символы, дополняющие фактически введенную строку. Дело в том, что вызов `read()` не добавляет завершающий нулевой байт в конце строки, которая задается для вывода функции `printf()`. Нетрудно догадаться, что именно так и должно быть, поскольку `read()` может использоваться для чтения любой последовательности байтов из файла. В некоторых случаях входные данные могут быть текстом, но бывает, что это двоичные целые числа или структуры языка C в двоичном виде. Невозможно «объяснить» вызову `read()` разницу между ними, поэтому он не в состоянии выполнять соглашение языка C о завершении строки символов нулевым байтом. Если в конце буфера входных данных требуется наличие завершающего нулевого байта, его нужно вставлять явным образом:

```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
  errExit("read");
buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

Поскольку для завершающего нулевого байта требуется байт памяти, размер буфера должен быть как минимум на один байт больше максимальной предполагаемой считываемой строки.

4.5. Запись в файл: write()

Системный вызов `write()` записывает данные в открытый файл.

```
#include <unistd.h>

ssize_t write(int fd, const void *buffer, size_t count);
```

Возвращает количество записанных байтов или `-1` при ошибке

Аргументы для `write()` аналогичны тем, что использовались для `read()`: `buffer` представляет собой адрес записываемых данных, `count` является количеством записываемых из буфера данных, а `fd` содержит дескриптор файла, который ссылается на тот файл, куда будут записываться данные.

В случае успеха вызов `write()` возвращает количество фактически записанных данных, которое может быть меньше значения аргумента `count`. Для дискового файла возможными причинами такой частичной записи может оказаться переполнение диска или достижение ограничения ресурса процесса на размеры файла. (Речь идет об ограничении `RLIMIT_FSIZE`, которое рассматривается в разделе 36.3.)

При выполнении ввода-вывода в отношении дискового файла успешный выход из `write()` не гарантирует перемещения данных на диск, поскольку ядро занимается буферизацией дискового ввода-вывода, чтобы сократить объем работы с диском и ускорить выполнение системного вызова `write()`. Более подробно этот вопрос рассматривается в главе 13.

4.6. Закрытие файла: close()

Системный вызов `close()` закрывает открытый дескриптор файла, высвобождая его для последующего повторного использования процессом. Когда процесс прекращает работу, все его открытые дескрипторы файлов автоматически закрываются.

```
#include <unistd.h>

int close(int fd);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Обычно предпочтительнее явно закрывать ненужные дескрипторы файлов. Тогда код, с учетом последующих изменений, будет проще для чтения и надежнее. Более того, дескрипторы файлов являются расходуемым ресурсом, поэтому сбой при закрытии дескриптора файла может вылиться в исчерпание процессом ограничения дескрипторов. Это, в частности, играет важную роль при написании программ, рассчитанных на долговременную работу и обращающихся к большому количеству файлов, например при создании оболочек или сетевых серверов.

Как и любые другие системные вызовы, `close()` должен сопровождаться проверкой кода на ошибки:

```
if (close(fd) == -1)
  errExit("close");
```

Такой код отлавливает ошибки вроде попыток закрытия неоткрытого дескриптора файла или закрытия одного и того же дескриптора файла дважды. Он также отлавливает сбойные ситуации, диагностируемые конкретной файловой системой в ходе операции закрытия.

> Сетевая файловая система — NFS (Network File System) — предоставляет пример такой специфичной для нее ошибки. Когда в NFS происходит сбой завершения транзакции, означающий, что данные не достигли удаленного диска, эта ошибка доходит до приложения в виде сбоя системного вызова close().

4.7. Изменение файлового смещения: lseek()

Для каждого открытого файла в ядре записывается *файловое смещение*, которое иногда также называют *смещением чтения-записи* или *указателем*. Оно обозначает место в файле, откуда будет стартовать работа следующего системного вызова `read()` или `write()`. Файловое смещение выражается в виде обычной байтовой позиции относительно начала файла. Первый байт файла расположен со смещением 0.

При открытии файла смещение устанавливается на его начало, а затем автоматически корректируется каждым последующим вызовом `read()` или `write()`, чтобы указывать на следующий байт файла непосредственно после считанного или записанного байта (или байтов). Таким образом, успешно проведенные вызовы `read()` и `write()` идут по файлу последовательно.

Системный вызов `lseek()` устанавливает файловое смещение открытого файла, на который указывает дескриптор `fd`, в соответствии со значениями, заданными в аргументах `offset` и `whence`.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Возвращает новое файловое смещение
при успешном завершении или `-1` при ошибке

Аргумент `offset` определяет значение смещения в байтах. (Тип данных `off_t` — целочисленный тип со знаком, определенный в SUSv3.) Аргшумент `whence` указывает на отправную точку, от которой отсчитывается смещение, и может иметь следующие значения:

- `SEEK_SET` — файловое смещение устанавливается в байтах на расстоянии `offset` от начала файла;
- `SEEK_CUR` — смещение устанавливается в байтах на расстоянии `offset` относительно текущего файлового смещения;
- `SEEK_END` — файловое смещение устанавливается на размер файла плюс `offset`. Иными словами, `offset` рассчитывается относительно следующего байта после последнего байта файла.

Порядок интерпретации аргумента `whence` показан на рис. 4.1.

В ранних реализациях UNIX вместо констант SEEK_*, перечисленных выше, использовались целые числа 0, 1 и 2. В старых версиях BSD для этих значений применялись другие имена: L_SET, L_INCR и L_XTND.

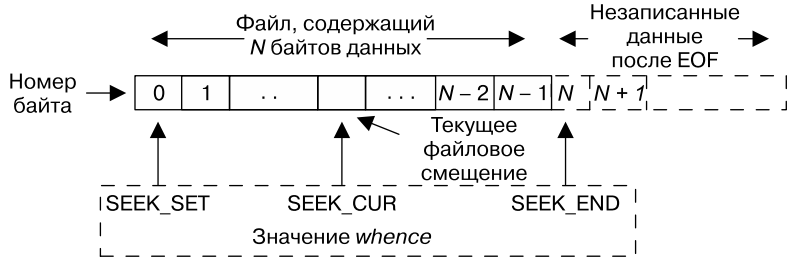


Рис. 4.1. Интерпретация аргумента whence системного вызова lseek()

Если аргумент `whence` содержит значение `SEEK_CUR` или `SEEK_END`, то у аргумента `offset` может быть положительное или отрицательное значение. Для `SEEK_SET` значение `offset` должно быть неотрицательным.

При успешном выполнении `lseek()` возвращается значение нового файлового смещения. Следующий вызов извлекает текущее расположение файлового смещения, не изменяя его значения:

```
curr = lseek(fd, 0, SEEK_CUR);
```

В некоторых реализациях UNIX (но не в Linux) имеется нестандартная функция tell(fd), которая служит той же цели, что и описанный системный вызов lseek().

Рассмотрим некоторые другие примеры вызовов `lseek()`, а также комментарии, объясняющие, куда передвигается файловое смещение:

```
lseek(fd, 0, SEEK_SET);       /* Начало файла */
lseek(fd, 0, SEEK_END);       /* Следующий байт после конца файла */
lseek(fd, -1, SEEK_END);      /* Последний байт файла */
lseek(fd, -10, SEEK_CUR);     /* Десять байтов до текущего размещения */
lseek(fd, 10000, SEEK_END);   /* 10 000 и 1 байт после
                                 последнего байта файла */
```

Вызов `lseek()` просто устанавливает значение для записи ядра, содержащей файловое смещение и связанной с дескриптором файла. Никакого физического доступа к устройству при этом не происходит.

Некоторые дополнительные подробности взаимоотношений между файловыми смещениями, дескрипторами файлов и открытыми файлами рассматриваются в разделе 5.4.

Не ко всем типам файлов можно применять системный вызов `lseek()`. Запрещено применение `lseek()` к конвейеру, FIFO-устройству, сокету или терминалу — вызов аварийно завершится с установленным для `errno` значением `ESPIPE`. С другой стороны, `lseek()` можно применять к тем устройствам, в отношении которых есть смысл это делать, например, при наличии возможности установки на конкретное место на дисковом или ленточном устройстве.

Буква l в названии lseek() появилась из-за того, что как для аргумента offset, так и для возвращаемого значения первоначально определялся тип long. В ранних реализациях UNIX предоставлялся системный вызов seek(), в котором для этих значений определялся тип int.

Файловые дыры

Что происходит, когда программа перемещает указатель, переходя при этом за конец файла, а затем выполняет ввод-вывод? При вызове `read()` возвращается `0`, показывающий, что достигнут конец файла. А вот записывать байты можно в произвольное место после окончания файла.

Пространство между предыдущим концом файла и только что записанными байтами называется *файловой дырой*. С точки зрения программиста, байты в дыре имеются, и чтение из дыры возвращает буфер данных, содержащий 0 (нулевые байты).

Файловые дыры не занимают места на диске. Файловая система не выделяет для дыры дисковые блоки до тех пор, пока в нее не будут записаны данные. Основное преимущество файловых дыр заключается в том, что для слабозаполненного файла потребляется меньше дискового пространства, чем понадобилось бы, если бы для нулевых байтов действительно нужно было выделять дисковые блоки. Файлы дампов ядра (см. раздел 22.1) — яркие примеры файлов с большими дырами.

Утверждение о том, что файловые дыры не потребляют дисковое пространство, требует уточнения. На большинстве файловых систем файловое пространство выделяется поблочно (см. раздел 14.3). Размер блока зависит от типа файловой системы, но обычно составляет 1024, 2048 или 4096 байт. Если край дыры попадает в блок, а не на границу блока, тогда для хранения байтов в другой части блока выделяется весь блок, и та часть, которая относится к дыре, заполняется нулевыми байтами.

Большинство нативных для UNIX файловых систем поддерживают концепцию файловых дыр, в отличие от многих «неродных» файловых систем (например, VFAT от Microsoft). В файловой системе, не поддерживающей дыры, в файл записывается явно указанное количество нулевых байтов.

Наличие дыр означает, что номинальный размер файла может быть больше, чем занимаемый им объем дискового пространства (иногда существенно больше). Запись байтов в середину дыры сократит объем свободного дискового пространства, поскольку для заполнения дыры ядро выделит блоки, даже притом, что размер файла не изменится. Подобное случается редко, но это все равно следует иметь в виду.

В SUSv3 определена функция posix_fallocate(fd, offset, len). Она гарантирует выделение дискового пространства для байтового диапазона, указанного аргументами offset и len для дискового файла, ссылка на который дается в дескрипторе fd. Это позволяет приложению получить гарантию, что при последующем вызове write() в отношении данного файла не будет сбоя, связанного с исчерпанием дискового пространства (который в противном случае может произойти при заполнении дыры в файле или потреблении дискового пространства каким-нибудь другим приложением). Исторически, реализация этой функции в glibc достигает нужного результата, записывая в каждый блок указанного диапазона нули. Начиная с версии 2.6.23, в Linux предоставляется системный вызов fallocate(). Он предлагает более эффективный способ обеспечения выделения необходимого пространства, и реализация posix_fallocate() в glibc использует этот системный вызов при его доступности.

В разделе 14.4 описывается способ представления дыр в файле, а в разделе 15.1 рассматривается системный вызов `stat()`, который способен сообщить о текущем размере файла, а также о количестве блоков, фактически выделенных файлу.

Пример программы

В листинге 4.3 показывается использование вызова `lseek()` в сочетании с `read()` и `write()`. Первый аргумент, передаваемый в командной строке для запуска этой программы, является именем открываемого файла. В остальных аргументах указываются операции ввода-вывода, выполняемые в отношении файла. Название каждой из этих операций состоит из буквы, за которой следует связанное с ней значение (без разделительного пробела):

- `soffset` — установка байтового смещения `offset` с начала файла;
- `rlength` — чтение `length` байтов из файла, начиная с текущего файлового смещения, и вывод их в текстовой форме;
- `Rlength` — чтение `length` байтов из файла, начиная с текущего файлового смещения и вывод их в виде шестнадцатеричных чисел;
- `wstr` — запись строки символов, указанной в `str`, начиная с позиции текущего файлового смещения.

Листинг 4.3. Демонстрация работы read(), write() и lseek()

fileio/seek_io.c

```
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <ctype.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
  size_t len;
  off_t offset;
  int fd, ap, j;
  char *buf;
  ssize_t numRead, numWritten;

  if (argc < 3 || strcmp(argv[1], "--help") == 0)
    usageErr("%s file {r<length>|R<length>|w<string>|s<offset>}...\n",
             argv[0]);

  fd = open(argv[1], O_RDWR | O_CREAT,
             S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
             S_IROTH | S_IWOTH);                    /* rw-rw-rw- */
  if (fd == -1)
    errExit("open");

  for (ap = 2; ap < argc; ap++) {
    switch (argv[ap][0]) {
    case 'r':  /* Вывод байтов с позиции текущего смещения в виде текста */
    case 'R':  /* Вывод байтов с позиции текущего смещения в виде hex-чисел */
      len = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
      buf = malloc(len);
      if (buf == NULL)
        errExit("malloc");

      numRead = read(fd, buf, len);
      if (numRead == -1)
        errExit("read");

      if (numRead == 0) {
        printf("%s: end-of-file\n", argv[ap]);
      } else {
        printf("%s: ", argv[ap]);
        for (j = 0; j < numRead; j++) {
          if (argv[ap][0] == 'r')
            printf("%c", isprint((unsigned char) buf[j]) ?
                                    buf[j] : '?');
          else
            printf("%02x ", (unsigned int) buf[j]);
          }
          printf("\n");
        }

        free(buf);
        break;

      case 'w':   /* Запись строки, начиная с позиции текущего смещения */
        numWritten = write(fd, &argv[ap][1], strlen(&argv[ap][1]));
        if (numWritten == -1)
          errExit("write");
        printf("%s: wrote %ld bytes\n", argv[ap], (long) numWritten);
        break;
```

```
      case 's':   /* Изменение файлового смещения */
        offset = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
        if (lseek(fd, offset, SEEK_SET) == -1)
          errExit("lseek");
        printf("%s: seek succeeded\n", argv[ap]);
        break;

      default:
        cmdLineErr("Argument must start with [rRws]: %s\n", argv[ap]);
      }
  }

  exit(EXIT_SUCCESS);
}
```

fileio/seek_io.c

Использование программы, приведенной в листинге 4.3, показано в следующих сессиях командной оболочки, с демонстрацией того, что произойдет при попытке чтения байтов из файловой дыры:

```
$ touch tfile                       Создание нового, пустого файла[5]
$ ./seek_io tfile s100000 wabc      Установка смещения 100000, запись "abc"
s100000: seek succeeded
wabc: wrote 3 bytes
$ ls -l tfile                       Проверка размера файла
-rw-r--r--    1 mtk    users   100003 Feb 10 10:35 tfile
$ ./seek_io tfile s10000 R5         Установка смещения 10000, чтение пяти байт из дыры
s10000: seek succeeded
R5: 00 00 00 00 00                  В байтах дыры содержится 0
```

4.8. Операции, не вписывающиеся в модель универсального ввода-вывода: ioctl()

Системный вызов `ioctl()` — механизм общего назначения для выполнения операций в отношении файлов и устройств, выходящих за пределы универсальной модели ввода-вывода, рассмотренной ранее в данной главе.

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, ... /* argp */);
```

Возвращаемое при успешном завершении значение зависит от *request* или при ошибке равно –1

Аргумент `fd` содержит дескриптор открываемого файла, представленного устройством или файлом, в отношении которого выполняется управляющая операция (указана в аргументе `request`). Как показывает стандартная для языка C запись в виде многоточия (`...`), третий аргумент для `ioctl()`, обозначенный как `argp`, может быть любого типа. Аргумент

[5] Только если файла с таким именем еще не было в текущем каталоге. Иначе эта команда лишь обновит время последнего обращения к файлу. — *Примеч. пер.*

request позволяет ioctl() определить, какого типа значение следует ожидать в argp. Обычно argp представляет собой указатель либо на целое число, либо на структуру. В некоторых случаях этот аргумент не применяется.

Использование ioctl() будет показано в следующих главах (к примеру, в разделе 15.5).

Единственная спецификация, имеющаяся в SUSv3 для ioctl(), регламентирует операции по управлению STREAMS-устройствами. (Среда STREAMS относится к особенностям System V, не поддерживаемым основной ветвью ядра Linux, хотя было разработано несколько реализаций в виде дополнений.) Ни одна из других рассматриваемых в книге операций ioctl() в SUSv3 не регламентирована. Но вызов ioctl() был частью системы UNIX с самых ранних версий, вследствие чего несколько операций ioctl() предоставляются во многих других реализациях UNIX. По мере рассмотрения каждой операции ioctl() будут обсуждаться и вопросы портируемости.

4.9. Резюме

Чтобы выполнить ввод/вывод в отношении обычного файла, сначала нужно получить его дескриптор, воспользовавшись системным вызовом open(). Затем ввод/вывод выполняется с помощью системных вызовов read() и write(). После завершения всех операций ввода-вывода следует высвободить дескриптор файла и связанные с ним ресурсы, воспользовавшись системным вызовом close(). Эти системные вызовы могут применяться для выполнения ввода-вывода в отношении всех типов файлов.

Поскольку для всех типов файлов и драйверов устройств реализован один и тот же интерфейс ввода-вывода, позволяющий получить универсальный ввод/вывод, то программа, как правило, может быть использована с любым типом файла, без использования кода, специфичного для типа файла.

Для каждого открытого файла ядро хранит файловое смещение, определяющее место, с которого будут осуществляться следующие чтение или запись. Файловое смещение косвенным образом обновляется при чтении и записи. Используя вызов lseek(), можно явным образом установить позицию файлового смещения в любое место файла или даже за его конец. Запись данных в позицию, находящуюся дальше предыдущего конца файла, приводит к созданию дыры в файле. Чтение из файловой дыры возвращает байты, содержащие нули.

Системный вызов ioctl() предлагает для устройства и файла разнообразные операции, которые не вписываются в стандартную модель файлового ввода-вывода.

4.10. Упражнения

4.1. Команда tee считывает свой стандартный ввод, пока ей не встретится символ конца файла, записывает копию своего ввода на стандартное устройство вывода и в файл, указанный в аргументе ее командной строки. (Пример использования этой команды будет показан при рассмотрении FIFO-устройств в разделе 44.7.) Реализуйте tee, используя системные вызовы ввода-вывода. По умолчанию tee перезаписывает любой существующий файл с заданным именем. Укажите ключ командной строки -a (tee -a *file*), который заставит tee добавлять текст к концу уже существующего файла.

4.2. Напишите программу, похожую на cp, которая при использовании для копирования обычного файла, содержащего дыры (последовательности нулевых байтов), будет также создавать соответствующие дыры в целевом файле.

5 Файловый ввод-вывод: дополнительные сведения

В этой главе мы продолжим рассматривать файловый ввод-вывод. Возвращаясь к системному вызову `open()`, мы познакомимся с концепцией *атомарности*. Она подразумевает, что действия в рамках системного вызова выполняются в виде единого непрерываемого шага — это неотъемлемое требование для корректной работы многих системных вызовов.

Будет представлен еще один многоцелевой системный вызов, имеющий отношение к файлам, — `fcntl()`. Мы рассмотрим один из примеров его использования: извлечение и установку флагов состояния открытого файла.

Затем будет описана структура данных ядра, которая применяется для представления файловых дескрипторов и открытых файлов. Понимая взаимоотношения между этими структурами, вы сможете разобраться в некоторых тонкостях файлового ввода-вывода, рассматриваемых в последующих главах. На основе этой модели будет объяснен порядок создания дубликатов дескрипторов файлов.

Затем будут перечислены некоторые системные вызовы, предоставляющие расширенные функциональные возможности чтения и записи. Они могут позволить нам выполнять ввод/вывод в конкретном месте файла без изменения файлового смещения и перемещать данные между несколькими буферами в программе.

Кроме того, мы затронем тему концепции неблокируемого ввода-вывода, а также рассмотрим некоторые расширения, предоставляемые для поддержки ввода-вывода в очень больших файлах.

Поскольку многими системными программами используются временные файлы, будут также перечислены некоторые библиотечные функции, позволяющие создавать и использовать временные файлы с произвольно создаваемыми уникальными именами.

5.1. Атомарность и состояние гонки

С понятием атомарности при рассмотрении операций системных вызовов придется сталкиваться довольно часто. Все системные вызовы выполняются атомарно. Это означает, что ядро гарантирует завершение всех этапов системного вызова в рамках одной операции, которая не прерывается другим процессом или потоком.

Для завершения некоторых операций атомарность играет весьма важную роль. В частности, она позволяет избежать *состояния гонки* (которое иногда называют *состязательной ситуацией*). Состязательной называют ситуацию, при которой на результат, выдаваемый двумя процессами (или потоками), работающими на совместно используемых ресурсах, влияет непредсказуемость относительного порядка получения процессами доступа к центральному процессору (или процессорам).

Далее мы рассмотрим две ситуации, развивающиеся на фоне файлового ввода-вывода, при которых возникает состояние гонки. Вы увидите, как эти состязания устраняются путем использования флагов системного вызова `open()`, гарантирующего атомарность соответствующих файловых операций.

Мы вернемся к теме состояния гонки, когда приступим к рассмотрению системного вызова `sigsuspend()` в разделе 22.9 и системного вызова `fork()` в разделе 24.4.

Эксклюзивное создание файла

В подразделе 4.3.1 отмечалось, что указание флага `O_EXCL` в сочетании с флагом `O_CREAT` заставляет `open()` возвращать ошибку, если файл уже существует. Тем самым процессу гарантируется, что именно он является создателем файла. Проводимая заранее проверка существования файла и создание файла выполняются атомарно. Чтобы понять, насколько это важно, рассмотрим код, показанный в листинге 5.1. Мы могли бы им воспользоваться при отсутствии флага `O_EXCL`. (В этом коде выводится идентификатор процесса, возвращаемый системным вызовом `getpid()`, позволяющий отличить данные на выходе двух различных запусков этой программы.)

Листинг 5.1. Код, не подходящий для эксклюзивного открытия файла

Из файла **fileio/bad_exclusive_open.c**

```
fd = open(argv[1], O_WRONLY); /* Открытие 1: проверка существования файла */
if (fd != -1) {               /* Открытие прошло успешно */
    printf("[PID %ld] File \"%s\" already exists\n",
            (long) getpid(), argv[1]);
    close(fd);
  } else {
    if (errno != ENOENT) {       /* Сбой по неожиданной причине */
      errExit("open");
    } else {
      /* ОТРЕЗОК ВРЕМЕНИ НА СБОЙ */
      fd = open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
      if (fd == -1)
        errExit("open");
       printf("[PID %ld] Created file \"%s\" exclusively\n",
               (long) getpid(), argv[1]);   /* МОЖЕТ БЫТЬ ЛОЖЬЮ! */
    }
  }
```

Из файла **fileio/bad_exclusive_open.c**

Кроме пространного использования двух вызовов `open()`, код в листинге 5.1 содержит ошибку. Представим себе, что один из наших процессов первым вызвал `open()`. Файл еще не существовал, но до того, как состоялся второй вызов `open()`, какой-то другой процесс создал его. Это могло произойти, если диспетчер ядра решил, что отрезок времени, выделенный процессу, истек, и передал управление, как показано на рис. 5.1, другому процессу, или, если два процесса были запущены одновременно в многопроцессорной системе. На рис. 5.1 изображен случай, когда оба таких процесса выполняют код, показанный в листинге 5.1. В данном сценарии процесс A придет к неверному заключению, что файл создан именно им, поскольку второй вызов `open()` будет успешен независимо от того, существовал файл или нет.

Хотя шанс на заблуждение процесса относительно того, что именно он является создателем файла, относительно мал, сама возможность такого события делает этот код ненадежным. Тот факт, что исход этих операций зависит от порядка диспетчеризации двух процессов, означает, что возникло состояние гонки.

Чтобы показать несомненную проблемность кода, можно заменить закомментированную строку `ОТРЕЗОК ВРЕМЕНИ НА СБОЙ` в листинге 5.1 фрагментом кода, создающим искусственную задержку между проверкой существования файла и созданием файла:

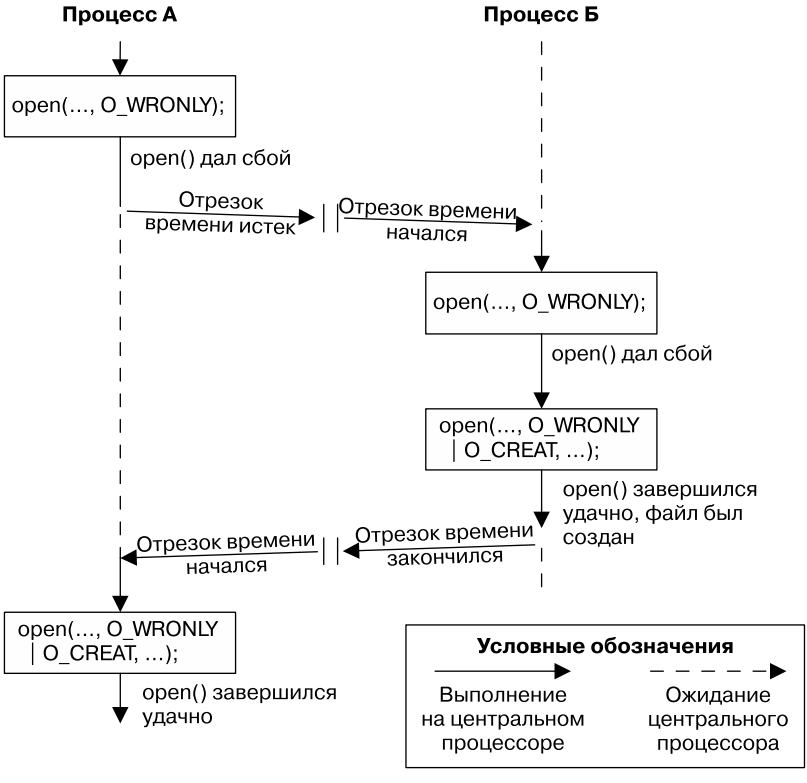


Рис. 5.1. Неудачная попытка эксклюзивного создания файла

```
printf("[PID %ld] File \"%s\" doesn't exist yet\n", (long) getpid(), argv[1]);
if (argc > 2) {             /* Задержка между проверкой и созданием */
  sleep(5);                 /* Приостановка выполнения на 5 секунд */
  printf("[PID %ld] Done sleeping\n", (long) getpid());
}
```

Библиотечная функция sleep() приостанавливает выполнение процесса на указанное количество секунд. Эта функция рассматривается в разделе 23.4.

Если одновременно запустить два экземпляра программы, показанной в листинге 5.1, станет видно, что они обе утверждают, что создали файл эксклюзивно:

```
$ ./bad_exclusive_open tfile sleep &
[PID 3317] File "tfile" doesn't exist yet
[1] 3317
$ ./bad_exclusive_open tfile
[PID 3318] File "tfile" doesn't exist yet
[PID 3318] Created file "tfile" exclusively
$ [PID 3317] Done sleeping
[PID 3317] Created file "tfile" exclusively                    Ложь
```

В предпоследней строке показанного экранного вывода видно, как смешались символ приглашения оболочки ко вводу ($) и вывод из первого экземляра тестовой программы.

Оба процесса утверждают о создании файла, потому что код первого процесса был прерван между проверкой на существование файла и созданием файла. Применение одного вызова `open()` с указанием флагов `O_CREAT` и `O_EXCL` предотвращает подобную ситуацию, гарантируя, что этапы проверки и создания выполняются как единая атомарная (то есть непрерывная) операция.

Добавление данных к файлу

Второй пример необходимости атомарности касается добавления данных к одному и тому же файлу сразу несколькими процессами (например, к глобальному журнальному файлу). Для этого можно было бы рассмотреть возможность использования каждым из записывающих процессов следующего фрагмента кода:

```
if (lseek(fd, 0, SEEK_END) == -1)
  errExit("lseek");
if (write(fd, buf, len) != len)
  fatal("Partial/failed write");
```

Но в этом коде, как и в предыдущем примере, есть точно такой же недостаток. Если первый выполняющий код процесс будет прерван между вызовами `lseek()` и `write()` вторым процессом, делающим то же самое, тогда оба процесса установят свои файловые смещения перед записью на одно и то же место, и, когда первому процессу диспетчер снова выделит процессорное время, он перепишет данные, уже записанные вторым процессом. Здесь опять возникает состояние гонки, поскольку результаты зависят от порядка диспетчеризации двух процессов.

Избежать возникновения данной проблемы можно при условии, что смещение на следующий байт за конец файла и операция записи будут происходить атомарно. Именно это гарантирует открытие файла с флагом `O_APPEND`.

> В некоторых файловых системах (например, в NFS) флаг O_APPEND не поддерживается. В таком случае ядро возвращается к показанной выше неатомарной последовательности вызовов с возможностью описанного выше повреждения файла.

5.2. Операции управления файлом: fcntl()

Системный вызов `fcntl()` может выполнять операции управления, используя дескриптор открытого файла.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

Значение, возвращаемое при успешном завершении, зависит от значения *cmd* или равно –1 при сбое

Аргумент `cmd` может указывать на широкий диапазон операций. Одни из них будут рассмотрены в следующих разделах, а до других мы доберемся лишь в последующих главах.

Многоточие показывает, что третий аргумент `fcntl()` может быть различных типов или же может быть опущен. Ядро использует значение аргумента `cmd` для определения типа данных (если таковой будет), который следует ожидать для этого аргумента.

5.3. Флаги состояния открытого файла

Один из примеров использования `fcntl()` — извлечение или изменение флагов режима доступа и состояния открытого файла. (Это значения, установленные аргументом `flags`, указанным в вызове `open()`.) Чтобы извлечь эти установки, для `cmd` указывается значение `F_GETFL`:

```
int flags, accessMode;

flags = fcntl(fd, F_GETFL);       /* Третий аргумент не требуется */
if (flags == -1)
  errExit("fcntl");
```

После этого фрагмента кода можно проверить, был ли файл открыт для синхронизированной записи:

```
if (flags & O_SYNC)
  printf("записи синхронизированы \n");
```

> В SUSv3 требуется, чтобы открытому файлу соответствовали лишь те флаги, которые были указаны при системном вызове open() или последующих операциях F_SETFL вызова fcntl(). В Linux есть единственное отклонение от этого требования: если приложение было скомпилировано с использованием одного из подходов, рассматриваемых в разделе 5.10 для открытия больших файлов, то среди флагов, извлекаемых операцией F_GETFL всегда будет установлен O_LARGEFILE.

Проверка режима доступа к файлу происходит немного сложнее, поскольку константы `O_RDONLY` (`0`), `O_WRONLY` (`1`) и `O_RDWR` (`2`) не соответствуют отдельным разрядам флагов состояния открытого файла. По этой причине на значение флагов накладывается маска с помощью константы `O_ACCMODE`, а затем проводится проверка на равенство одной из констант:

```
accessMode = flags & O_ACCMODE;
if (accessMode == O_WRONLY || accessMode == O_RDWR)
  printf("file is writable\n");
```

Команду `F_SETFL` системного вызова `fcntl()` можно использовать для изменения некоторых флагов состояния открытого файла. К ним относятся `O_APPEND`, `O_NONBLOCK`, `O_NOATIME`, `O_ASYNC` и `O_DIRECT`. Попытки изменить другие флаги игнорируются. (В некоторых других реализациях UNIX системному вызову `fcntl()` разрешается изменять и другие флаги, например `O_SYNC`.)

Возможность использования вызова `fcntl()` для изменения флагов состояния открытого файла может особенно пригодиться в следующих случаях.

- Файл был открыт не вызывающей программой, поэтому она не может управлять флагами, использованными в вызове `open()` (например, файл мог быть представлен одним из стандартных дескрипторов, открытых еще до запуска программы).
- Файловый дескриптор был получен не из `open()`, а из другого системного вызова. Примерами таких системных вызовов могут служить `pipe()`, который создает конвейер и возвращает два файловых дескриптора, ссылающихся на оба конца конвейера, и `socket()`, который создает сокет и возвращает дескриптор файла, ссылающийся на сокет.

Чтобы изменить флаги состояния открытого файла, сначала с помощью вызова `fcntl()` извлекаются копии существующих флагов, затем изменяются нужные разряды

и, наконец, делается еще один вызов `fcntl()` для обновления флагов. Таким образом, чтобы включить флаг `O_APPEND`, можно написать следующий код:

```
int flags;

flags = fcntl(fd, F_GETFL);
if (flags == -1)
  errExit("fcntl");
flags |= O_APPEND;
if (fcntl(fd, F_SETFL, flags) == -1)
  errExit("fcntl");
```

5.4. Связь файловых дескрипторов с открытыми файлами

К этому моменту у вас могло создаться впечатление, что между файловым дескриптором и открытым файлом существует соотношение «один к одному». Но это не так. Есть весьма полезная возможность иметь сразу несколько дескрипторов, ссылающихся на один и тот же открытый файл. Эти файловые дескрипторы могут быть открыты в одном и том же или в разных процессах.

Чтобы разобраться в происходящем, нужно изучить три структуры данных, обслуживаемые ядром:

- таблицу дескрипторов файлов для каждого процесса;
- общесистемную таблицу дескрипторов открытых файлов;
- таблицу индексных дескрипторов файловой системы.

Для каждого процесса ядро поддерживает таблицу *дескрипторов открытых файлов*. Каждая запись в этой таблице содержит информацию об одном файловом дескрипторе, включая:

- набор флагов, управляющих работой файлового дескриптора (такой флаг всего один — флаг закрытия при выполнении — close-on-exec, и он будет рассмотрен в разделе 27.4);
- ссылку на дескриптор открытого файла.

Ядро обслуживает общесистемную таблицу всех дескрипторов открытых файлов. (Она иногда называется *таблицей открытых файлов*, а записи в ней — *дескрипторами открытых файлов*.) В дескрипторе открытого файла хранится вся информация, относящаяся к открытому файлу, включая:

- текущее файловое смещение (обновляемое системными вызовами `read()` и `write()` или явно изменяемое с помощью системного вызова `lseek()`);
- флаги состояния при открытии файла (то есть аргумент `flags` системного вызова `open()`);
- режим доступа к файлу (только для чтения, только для записи или для чтения и записи, согласно установкам для системного вызова `open()`);
- установки, относящиеся к вводу-выводу, управляемому сигналами (см. раздел 59.3);
- ссылку на индексный дескриптор для этого файла.

У каждой файловой системы есть таблица *индексных дескрипторов* для всех размещенных в ней файлов. Структура индексных дескрипторов и в целом файловых систем более подробно рассматривается в главе 14. А сейчас следует отметить, что индексный дескриптор для каждого файла включает такую информацию:

- тип файла (например, обычный файл, сокет или FIFO-устройство) и права доступа;
- указатель на список блокировок, удерживаемых на этом файле;
- разные свойства файла, включая его размер и метки времени, связанные с различными типами файловых операций.

Здесь мы не учитываем разницу между представлением индексного дескриптора на диске и в памяти. В индексном дескрипторе на диске записываются постоянные атрибуты, такие как его тип, права доступа и отметки времени. Когда происходит доступ к файлу, создается копия индексного дескриптора, хранящаяся в памяти, и в эту версию индексного дескриптора записывается количество файловых дескрипторов, ссылающихся на индексный дескриптор, и главные и второстепенные идентификаторы устройства, из которого был скопирован индексный дескриптор. В индексный дескриптор, хранящийся в памяти, также записываются различные недолговечные атрибуты, связанные с файлом при его открытии, например блокировки файлов.

Связь между дескрипторами файлов, дескрипцией открытых файлов и индексными дескрипторами показана на рис. 5.2. На этой схеме у двух процессов имеется несколько дескрипторов открытых файлов.

В процессе А два дескриптора — 1 и 20 — ссылаются на один и тот же дескриптор открытого файла (с пометкой 23). Такая ситуация может возникать в результате вызова `dup()`, `dup2()` или `fcntl()` (см. раздел 5.5).

Дескриптор 2 процесса А и дескриптор 2 процесса Б ссылаются на один и тот же файловый дескриптор (73). Этот сценарий может сложиться после вызова `fork()` (то есть процесс А является родительским по отношению к процессу Б или наоборот) либо при условии, что один процесс передал открытый дескриптор другому процессу, используя доменный сокет UNIX (см. подраздел 57.13.3).

И наконец, можно увидеть, что дескриптор 0 процесса А и дескриптор 3 процесса Б ссылаются на различные дескрипторы открытых файлов, но эти дескрипции ссылаются на одну и ту же запись в таблице индексных дескрипторов (1976), то есть на один и тот же файл. Дело в том, что каждый процесс независимо вызвал `open()` для одного и того же файла. Похожая ситуация может возникнуть, если один и тот же процесс дважды откроет один и тот же файл.

В результате можно прийти к следующим заключениям.

- Два различных файловых дескриптора, ссылающихся на одну и ту же дескрипцию открытого файла, совместно используют значение файлового смещения. Поэтому, если файловое смещение изменяется в связи с работой с одним файловым дескриптором (в результате вызовов `read()`, `write()` или `lseek()`), это изменение прослеживается через другой файловый дескриптор. Это применимо как к случаю, когда оба файловых дескриптора принадлежат одному и тому же процессу, так и к случаю, когда они принадлежат разным процессам.
- Аналогичные правила видимости применяются и к извлечению и изменению флагов состояния открытых файлов (например, `O_APPEND`, `O_NONBLOCK` и `O_ASYNC`) при использовании в системном вызове `fcntl()` операций `F_GETFL` и `F_SETFL`.
- В отличие от этого, флаги файлового дескриптора (то есть флаг закрытия при исполнении — close-on-exec) находятся в исключительном владении процесса и файлового дескриптора. Изменение этих флагов не влияет на другие файловые дескрипторы в одном и том же или в разных процессах.

5.5. Дублирование дескрипторов файлов

Использование синтаксиса перенаправления ввода-вывода (присущего Bourne shell) `2>&1` информирует оболочку о необходимости перенаправления стандартной ошибки (файловый дескриптор 2) в то же место, в которое выдается стандартный вывод (дескриптор файла 1). Таким образом, следующая команда станет (поскольку оболочка вычисляет направление ввода-вывода слева направо) отправлять и стандартный вывод, и стандартную ошибку в файл `results.log`:

```
$ ./myscript > results.log 2>&1
```

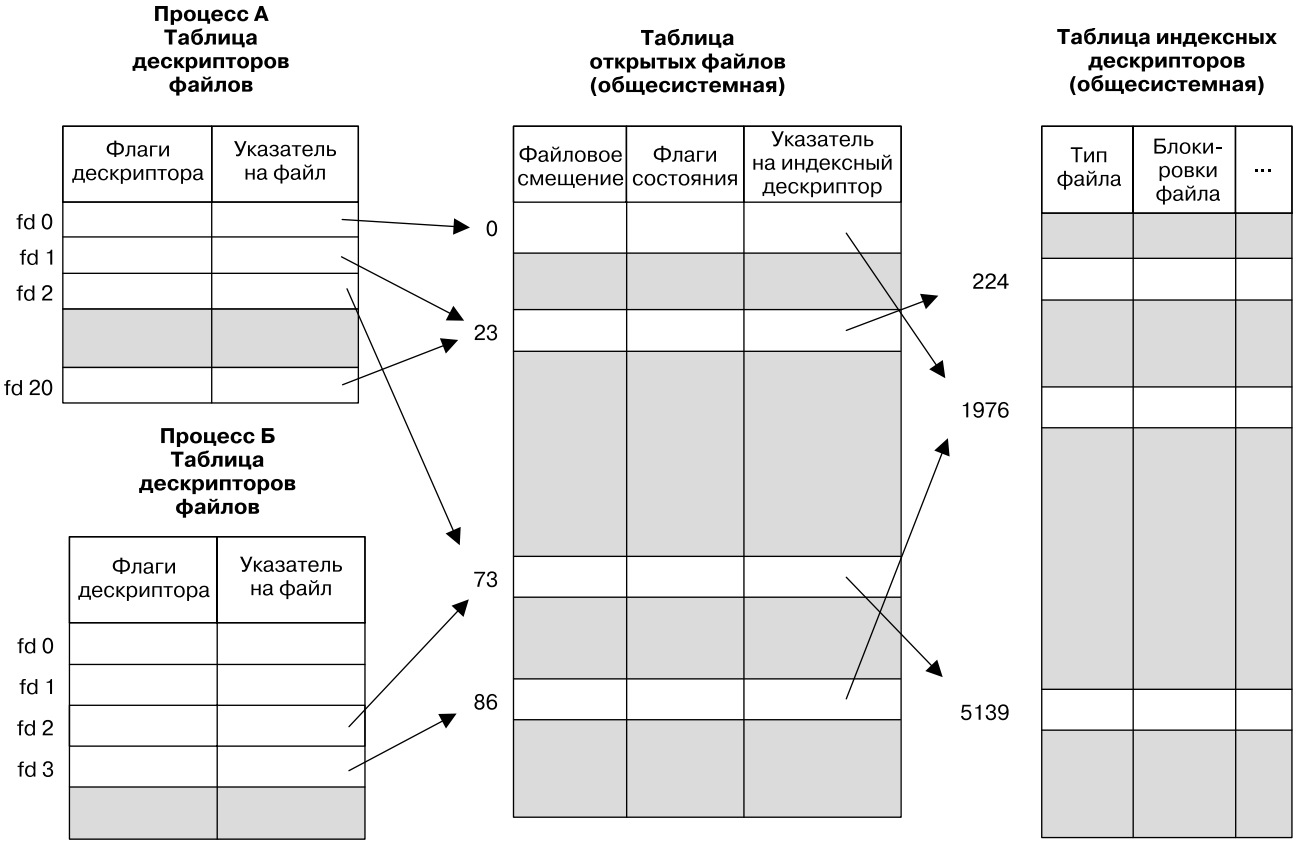


Рис. 5.2. Связь между дескрипторами файлов, дескрипцией открытых файлов и индексными дескрипторами

Оболочка перенаправляет стандартную ошибку, создавая дескриптор файла 2 дубликата дескриптора файла 1, так что он ссылается на ту же дескрипцию открытого файла, что и файловый дескриптор 1 (точно так же, как дескрипторы 1 и 20 процесса A ссылаются на одну и ту же дескрипцию открытого файла на рис. 5.2). Этого эффекта можно достичь, используя системные вызовы `dup()` и `dup2()`.

Заметьте, что для оболочки недостаточно просто дважды открыть файл `results.log`: один раз с дескриптором 1 и один раз с дескриптором 2. Одна из причин состоит в том, что два файловых дескриптора не смогут совместно использовать указатель файлового смещения и это приведет к перезаписи вывода друг друга. Другая причина заключается в том, что файл может не быть дисковым. Рассмотрим следующую команду, отправляющую стандартную ошибку по тому же конвейеру, что и стандартный вывод:

```
$ ./myscript 2>&1 | less
```

Вызов `dup()` на основании аргумента `oldfd` открывает файловый дескриптор, возвращая новый дескриптор, ссылающийся на ту же самую дескрипцию открытого файла. Новый дескриптор гарантированно будет наименьшим неиспользованным файловым дескриптором.

```
#include <unistd.h>

int dup(int oldfd);
```

При успешном завершении возвращает новый файловый дескриптор, а при ошибке выдает `-1`

Предположим, что осуществляется следующий вызов:

```
newfd = dup(1);
```

Если предположить, что сложилась обычная ситуация, при которой оболочка открыла от имени программы файловые дескрипторы 0, 1 и 2, и не используются никакие другие дескрипторы, `dup()` откроет дубликат дескриптора 1, используя файловый дескриптор 3.

Если нужно, чтобы дубликатом стал дескриптор 2, можно воспользоваться следующей технологией:

```
close(2);              /* Высвобождение файлового дескриптора 2 */
newfd = dup(1);        /* Повторное использование файлового дескриптора 2 */
```

Этот код работает, только если был открыт дескриптор 0. Чтобы упростить показанный выше код и обеспечить неизменное получение нужного нам файлового дескриптора, можно воспользоваться системным вызовом `dup2()`.

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

При успешном завершении возвращает новый файловый дескриптор, а при ошибке выдает `-1`

Системный вызов `dup2()` создает дубликат файлового дескриптора, заданного в аргументе `oldfd`, используя номер дескриптора, предоставленный в аргументе `newfd`. Если файловый дескриптор, указанный в `newfd`, уже открыт, `dup2()` сначала закрывает его.

(Любые ошибки, происходящие при этом закрытии, просто игнорируются. Закрытие и повторное использование `newfd` выполняются атомарно, что исключает возможность повторного применения `newfd` между двумя шагами обработчика сигнала или параллельного потока, который выделяет файловый дескриптор.)

Предыдущие вызовы `close()` и `dup()` можно упростить, сведя их к следующему вызову:

```
dup2(1, 2);
```

Успешно завершенный вызов `dup2()` возвращает номер продублированного дескриптора (то есть значение, переданное в аргументе `newfd`).

Если аргумент `oldfd` не является допустимым файловым дескриптором, `dup2()` дает сбой с указанием на ошибку `EBADF`, и дескриптор, заданный в `newfd`, не закрывается. Если аргумент `oldfd` содержит допустимый файловый дескриптор и в аргументах `oldfd` и `newfd` хранится одно и то же значение, то `dup2()` не совершает никаких действий — дескриптор, указанный в `newfd`, не закрывается и `dup2()` возвращает в качестве результата своей работы значение аргумента `newfd`.

Еще один интерфейс, предоставляющий дополнительную гибкость для дублирования файловых дескрипторов, предусматривает использование операции `F_DUPFD` системного вызова `fcntl()`:

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

Этот вызов создает дубликат дескриптора, указанного в `oldfd`, путем использования наименьшего неиспользуемого дескриптора файла, который больше или равен номеру, заданному в `startfd`. Применяется, когда нужно обеспечить попадание нового дескриптора (*`newfd`*) в конкретный диапазон значений. Вызовы `dup()` и `dup2()` всегда могут быть записаны как вызовы `close()` и `fcntl()`, хотя они лаконичнее. (Следует также заметить, что некоторые коды ошибок в `errno`, возвращаемые `dup2()` и `fcntl()`, отличаются друг от друга — подробности см. на страницах руководств этих вызовов.)

На рис. 5.2 можно увидеть, что продублированные файловые дескрипторы совместно используют одно и то же значение файлового смещения и одни и те же флаги состояния в своих совместно используемых дескрипциях открытых файлов. Но новый файловый дескриптор имеет собственный набор флагов файлового дескриптора, и его флаг закрытия при выполнении — close-on-exec (`FD_CLOEXEC`) — всегда сброшен. Следующий рассматриваемый интерфейс позволяет получить явный контроль над флагом закрытия при выполнении нового файлового дескриптора.

Системный вызов `dup3()` выполняет ту же задачу, что и `dup2()`, но к нему добавляется новый аргумент, `flags`, который является битовой маской, изменяющей поведение системного вызова.

```
#define _GNU_SOURCE
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

При успешном завершении возвращает новый файловый дескриптор, а при ошибке выдает `-1`

В настоящее время `dup3()` поддерживает один флаг — `O_CLOEXEC`, заставляющий ядро установить флаг закрытия при выполнении (`FD_CLOEXEC`) для нового файлового дескриптора. Польза от применения этого флага такая же, как от флага `O_CLOEXEC` системного вызова `open()`, рассмотренного в разделе 4.3.1.

Системный вызов dup3() появился в Linux 2.6.27 и характерен только для Linux.

Начиная с версии Linux 2.6.24, в этой ОС также поддерживается дополнительная операция системного вызова fcntl(), предназначенная для дублирования файловых дескрипторов: F_DUPFD_CLOEXEC. Этот флаг делает то же самое, что и F_DUPFD, но дополнительно он устанавливает для нового файлового дескриптора флаг закрытия при выполнении (FD_CLOEXEC). Польза от этой операции обусловлена теми же причинами, что и применение флага O_CLOEXEC для системного вызова open(). Операция F_DUPFD_CLOEXEC не определена в SUSv3, но поддерживается в SUSv4.

5.6. Файловый ввод-вывод по указанному смещению: pread() и pwrite()

Системные вызовы pread() и pwrite() работают практически так же, как read() и write(), за исключением того, что файловый ввод-вывод осуществляется с места, указанного значением offset, а не с текущего файлового смещения. Эти вызовы не изменяют файлового смещения.

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
                Возвращает количество считанных байтов, 0 при EOF или -1 при ошибке
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
                            Возвращает количество записанных байтов или -1 при ошибке
```

Вызов pread() эквивалентен *атомарному* выполнению следующих вызовов:

```
off_t orig;

orig = lseek(fd, 0, SEEK_CUR);   /* Сохранение текущего смещения */
lseek(fd, offset, SEEK_SET);
s = read(fd, buf, len);
lseek(fd, orig, SEEK_SET);       /* Восстановление исходного файлового смещения */
```

Как для pread(), так и для pwrite() файл, ссылка на который дается в аргументе fd, должен быть пригодным для изменения смещения (то есть представлен файловым дескриптором, в отношении которого допустимо вызвать lseek()).

В частности, такие системные вызовы могут пригодиться в многопоточных приложениях. В главе 29 будет показано, что все потоки в процессе совместно используют одну и ту же таблицу файловых дескрипторов. Это означает, что файловое смещение для каждого открытого файла является для всех потоков глобальным. Используя pread() или pwrite(), несколько потоков могут одновременно осуществлять ввод-вывод в отношении одного и того же файлового дескриптора, без влияния тех изменений, которые производят в отношении файлового смещения другие потоки. Если попытаться воспользоваться вместо этого lseek() плюс read() (или write()), то мы создадим состояние гонки, подобной одной из тех, описание которых давалось при рассмотрении флага O_APPEND в разделе 5.1. (Системные вызовы pread() и pwrite() могут также пригодиться для устранения состояния гонки в приложениях, когда у нескольких процессов имеются файловые дескрипторы, ссылающиеся на одну и ту же дескрипцию открытого файла.)

При условии многократного выполнения вызовов lseek() с последующим файловым вводом-выводом системные вызовы pread() и pwrite() могут также предложить в некоторых случаях преимущества в производительности. Дело в том, что отдельный системный вызов pread() (или pwrite()) приводит к меньшим издержкам, чем два системных вызова: lseek() и read() (или write()). Но издержки, связанные с системными вызовами, обычно незначительны по сравнению со временем фактического выполнения ввода-вывода.

5.7. Ввод-вывод по принципу фрагментации-дефрагментации: readv() и writev()

Системные вызовы `readv()` и `writev()` выполняют фрагментированный ввод/вывод (scatter-gather I/O).

```
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

Возвращает количество считанных байтов, `0` при EOF или `-1` при ошибке

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Возвращает количество записанных байтов или `-1` при ошибке

За один системный вызов обрабатываются несколько таких буферов данных. Набор передаваемых буферов определяется массивом `iov`. Количество элементов в `iov` указывается в `iovcnt`. Каждый элемент в `iov` является структурой с такой формой:

```
struct iovec {
  void  *iov_base;  /* Начальный адрес буфера */
  size_t iov_len;   /* Количество байтов для передачи в буфер или из него */
};
```

Согласно спецификации SUSv3, допускается устанавливать ограничение по количеству элементов в iov. Реализация может уведомить о своем ограничении, определив значение IOV_MAX в заголовочном файле <limits.h> или в ходе выполнения через возвращаемое значение вызова sysconf(_SC_IOV_MAX). (Вызов sysconf() рассматривается в разделе 11.2.) В спецификации SUSv3 требуется, чтобы это ограничение было не меньше 16. В Linux для IOV_MAX определено значение 1024, что соответствует ограничениям ядра на размер этого вектора (задается константой ядра UIO_MAXIOV).

При этом функции оболочки из библиотеки glibc для readv() и writev() незаметно выполняют дополнительные действия. Если системный вызов дает сбой по причине слишком большого значения iovcnt, функция-оболочка временно выделяет один буфер, чьего объема достаточно для хранения всех элементов, описанных iov, и выполняет вызов read() или write() (см. далее тему о возможной реализации writev() с использованием write()).

На рис. 5.3 показан пример взаимосвязанности аргументов `iov` и `iovcnt`, а также буферов, на которые они ссылаются.

Фрагментированный ввод

Системный вызов `readv()` выполняет фрагментированный ввод: он считывает непрерывную последовательность байтов из файла, ссылка на который дается в файловом

дескрипторе `fd`, и помещает («фрагментирует») эти байты в буферы, указанные аргументом `iov`. Каждый из буферов, начиная с того, что определен элементом `iov[0]`, полностью заполняется, прежде чем `readv()` переходит к следующему буферу.

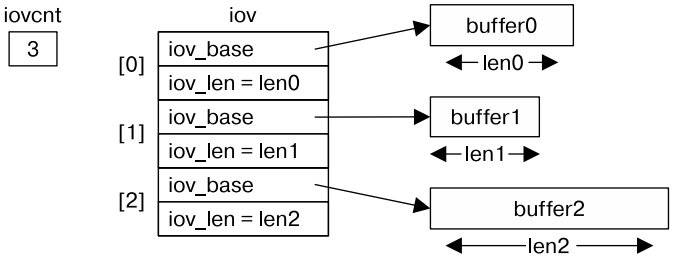


Рис. 5.3. Пример массива iov и связанных с ним буферов

Важным свойством `readv()` является выполнение всей работы в атомарном режиме, то есть с позиции вызывающего процесса ядро совершает единое портирование данных между файлом, на который указывает `fd`, и пользовательской памятью. Это означает, к примеру, что при чтении из файла можно быть уверенными, что диапазон считываемых байтов непрерывен, даже если другой процесс (или поток), совместно используя то же файловое смещение, предпринимает попытку манипулировать смещением в то время, когда выполняется системный вызов `readv()`.

При успешном завершении `readv()` возвращает количество считанных байтов или `0`, если встречен конец файла. Вызывающий процесс должен проверить это количество, чтобы убедиться, что были считаны все запрошенные байты. Если было доступно недостаточное количество байтов, то заполненными могут оказаться не все буферы — последние буферы могут быть заполнены лишь частично.

Пример использования вызова `readv()` показан в листинге 5.2.

Будем придерживаться следующего соглашения: если название файла состоит из префикса t_ и имени функции(...) (например, t_readv.c в листинге 5.2), это значит, что программа главным образом демонстрирует работу одного системного вызова или библиотечной функции.

Листинг 5.2. Выполнение фрагментированного ввода с помощью readv()

fileio/t_readv.c

```
#include <sys/stat.h>
#include <sys/uio.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
  int fd;
  struct iovec iov[3];
  struct stat myStruct;       /* Первый буфер */
  int x;                      /* Второй буфер */
#define STR_SIZE 100
  char str[STR_SIZE];         /* Третий буфер */
  ssize_t numRead, totRequired;
```

```
  if (argc != 2 || strcmp(argv[1], "--help") == 0)
    usageErr("%s file\n", argv[0]);
  fd = open(argv[1], O_RDONLY);
  if (fd == -1)
    errExit("open");
  totRequired = 0;
  iov[0].iov_base = &myStruct;
  iov[0].iov_len = sizeof(struct stat);
  totRequired += iov[0].iov_len;
  iov[1].iov_base = &x;
  iov[1].iov_len = sizeof(x);
  totRequired += iov[1].iov_len;
  iov[2].iov_base = str;
  iov[2].iov_len = STR_SIZE;
  totRequired += iov[2].iov_len;

  numRead = readv(fd, iov, 3);
  if (numRead == -1)
    errExit("readv");
  if (numRead < totRequired)
    printf("Read fewer bytes than requested\n");

  printf("total bytes requested: %ld; bytes read: %ld\n",
         (long) totRequired, (long) numRead);
  exit(EXIT_SUCCESS);
}
```

fileio/t_readv.c

Дефрагментированный вывод

Системный вызов `writev()` выполняет *дефрагментированный вывод*. Он объединяет («дефрагментирует») данные из всех буферов, указанных в аргументе `iov`, и записывает их в виде непрерывной последовательности байтов в файл, ссылка на который находится в файловом дескрипторе `fd`. Дефрагментация буферов происходит в порядке следования элементов массива, начиная с буфера, определяемого элементом `iov[0]`.

Как и `readv()`, системный вызов `writev()` выполняется атомарно, все данные передаются в рамках одной операции из пользовательской памяти в файл, на который ссылается аргумент `fd`. Таким образом, при записи в обычный файл можно быть уверенными, что все запрошенные данные записываются в него непрерывно, не перемежаясь с записями других процессов (или потоков).

Как и в случае с `write()`, возможна частичная запись. Поэтому нужно проверять значение, возвращаемое `writev()`, чтобы увидеть, все ли запрошенные байты были записаны.

Главными преимуществами `readv()` и `writev()` являются удобство и скорость. Например, вызов `writev()` можно заменить:

- кодом, выделяющим один большой буфер и копирующим в него записываемые данные из других мест в адресном пространстве процесса, а затем вызывающим `write()` для вывода данных из буфера;
- либо серией вызовов `write()`, выводящих данные из отдельных буферов.

Первый из вариантов, будучи семантическим эквивалентом использования `writev()`, неудобен (и неэффективен), так как требуется выделять буферы и копировать данные в пользовательском пространстве. Второй вариант не является семантическим эквивалентом одному вызову `writev()`, так как вызовы `write()` не выполняются атомарно. Более того, выполнение одного системного вызова `writev()` обходится дешевле выполнения нескольких вызовов `write()` (вспомним раздел 3.1).

Выполнение фрагментированного ввода-вывода по указанному смещению

В Linux 2.6.30 также были добавлены два новых системных вызова, сочетающих в себе функциональные возможности фрагментированного ввода-вывода с возможностью выполнения ввода-вывода по указанному смещению: `preadv()` и `pwritev()`. Это нестандартные системные вызовы, которые также доступны в современных BSD-системах.

```
#define _BSD_SOURCE
#include <sys/uio.h>

ssize_t preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset);
```

Возвращает количество считанных байтов, `0` при EOF или `-1` при ошибке

```
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset);
```

Возвращает количество записанных байтов или `-1` при ошибке

Системные вызовы `preadv()` и `pwritev()` выполняют ту же задачу, что и `readv()` и `writev()`, но осуществляют ввод/вывод в отношении того места в файле, которое указано смещением (наподобие `pread()` и `pwrite()`). Эти системные вызовы не меняют смещение файла.

Эти системные вызовы окажутся полезными для приложений (например, многопоточных), где нужно сочетать преимущества фрагментированного ввода-вывода с возможностью выполнения ввода-вывода в месте, не зависящем от текущего файлового смещения.

5.8. Усечение файла: truncate() и ftruncate()

Системные вызовы `truncate()` и `ftruncate()` устанавливают для файла размер, соответствующий значению, указанному в аргументе `length`.

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

Оба возвращают `0` при успешном завершении или `-1` при ошибке

Если длина файла больше значения, указанного в аргументе `length`, избыточные данные утрачиваются. Если текущее значение длины файла меньше значения аргумента `length`, файл наращивается за счет добавления последовательности нулевых байтов, или дыры.

Эти два системных вызова отличаются друг от друга способом указания файла. При использовании `truncate()` файл, который должен быть доступен и открыт для записи, указывается в строке путевого имени — `pathname`. Если `pathname` является символьной ссылкой, она разыменовывается. Системный вызов `ftruncate()` получает дескриптор того файла, который был открыт для записи. Файловое смещение для файла не изменяется.

Если значение аргумента `length` для `ftruncate()` превышает текущий размер файла, в спецификации SUSv3 разрешается проявлять один из двух вариантов поведения: либо файл расширяется (как в Linux), либо системный вызов возвращает ошибку. XSI-совместимые системы должны принять первую линию поведения. В SUSv3 требуется, чтобы `truncate()` всегда расширял файл, если значение `length` превышает его текущий размер.

Уникальность системного вызова truncate() состоит в том, что это единственный системный вызов, способный изменять содержимое файла, не получая для него предварительно дескриптор посредством вызова open() (или какими-то другими способами).

5.9. Неблокирующий ввод-вывод

Указание флага O_NONBLOCK при открытии файла служит двум целям.

- Если файл не может быть открыт немедленно, вызов open() вместо блокирования возвращает ошибку. Одним из случаев, при котором open() может проводить блокировку, является использование этого системного вызова в отношении FIFO-устройств (см. раздел 44.7).
- После успешного завершения open() дальнейшие операции ввода-вывода также являются неблокирующими. Если системный вызов ввода-вывода не может завершиться немедленно, то либо выполняется частичное портирование данных, либо системный вызов дает сбой с выдачей одной из ошибок: EAGAIN или EWOULDBLOCK. Какая из ошибок будет возвращена, зависит от системного вызова. В Linux, как и во многих реализациях UNIX, эти две константы ошибок синонимичны.

Неблокирующий режим может использоваться с устройствами (например, с терминалами и псевдотерминалами), конвейерами, FIFO-устройствами и сокетами. (Поскольку при использовании open() дескрипторы для конвейеров и сокетов не получаются, этот флаг должен устанавливаться при использовании операции F_SETFL системного вызова fcntl(), рассматриваемого в разделе 5.3.)

Для обычных файлов указание флага O_NONBLOCK, как правило, не требуется, поскольку буферный кэш ядра гарантирует, что ввод-вывод в отношении обычных файлов, как описывается в разделе 13.1, не блокируется. Но O_NONBLOCK оказывает влияние на обычные файлы, когда используется обязательная блокировка файлов (см. раздел 51.4).

Неблокирующий ввод-вывод будет также рассматриваться в разделе 44.9 и в главе 59.

Исторически реализации, берущие начало из System V, предоставляли флаг O_NDELAY, имеющий сходную с O_NONBLOCK семантику. Основное отличие состояло в том, что неблокирующий системный вызов write() в System V возвращал 0, если write () не мог быть завершен, а неблокирующий вызов read() возвращал 0, если ввод был недоступен. Это поведение создавало проблемы для read(), поскольку было неотличимо от условий, при которых встречался конец файла. Поэтому в первом стандарте POSIX.1 был введен флаг O_NONBLOCK. В некоторых реализациях UNIX по-прежнему предоставляется флаг O_NDELAY со старой семантикой. В Linux определена константа O_NDELAY, но она является синонимом O_NONBLOCK.

5.10. Ввод-вывод, осуществляемый в отношении больших файлов

Тип данных off_t, используемый для хранения файлового смещения, обычно реализуется как длинное целое число со знаком. (Тип данных со знаком нужен потому, что при ошибке возвращается -1.) На 32-разрядных архитектурах (таких как x86-32) это будет ограничивать размер файлов $2^{31} - 1$ байтами (то есть 2 Гбайт).

Но емкость дисковых накопителей давным-давно преодолела это ограничение, и перед 32-разрядными реализациями Unix встала необходимость работать с файлами, превышающими этот размер. Поскольку проблема затрагивала многие реализации, группа

поставщиков UNIX приняла решение объединить свои усилия на саммите, посвященном работе с большими файлами — Large File Summit (LFS), с целью улучшения спецификации SUSv2. Планировалось добавить дополнительные функциональные возможности, необходимые для доступа к большим файлам. Усовершенствования, предложенные в рамках LFS, мы рассмотрим в текущем разделе. (Полная LFS-спецификация, работа над которой завершилась в 1996 году, находится по адресу http://opengroup.org/platform/lfs.html.)

В Linux LFS-поддержка для 32-разрядных систем была предоставлена, начиная с версии ядра 2.4 (для чего также требуется версия glibc 2.2 или выше). Кроме того, соответствующая файловая система должна поддерживать большие файлы. Эта поддержка предоставляется большинством свойственных для Linux файловых систем, в отличие от некоторых несвойственных систем (характерными примерами могут послужить разработанные Microsoft файловые системы VFAT и NFSv2, накладывающие жесткие ограничения 2 Гбайт на файл, независимо от того, используются LFS-расширения или нет).

> Поскольку для длинных целых чисел на 64-разрядных архитектурах (например, x86-64, Alpha, IA-64) используются 64 бита, на такие архитектуры вообще не влияют ограничения, для преодоления которых были разработаны LFS-усовершенствования. И все же особенности реализации некоторых свойственных Linux файловых систем предполагают, что теоретический максимальный размер может быть меньше $2^{63} - 1$ даже в 64-разрядных системах. В большинстве случаев эти ограничения существенно выше, чем современные объемы дисков, поэтому они не накладывают значимых ограничений на размеры файлов.

Создавать приложения, требующие применения функциональных возможностей LFS, можно одним из двух способов.

- Воспользоваться альтернативным API, поддерживающим большие файлы. Он был разработан LFS в качестве «переходного расширения» для спецификации Single UNIX Specification. В результате наличие этого интерфейса не требуется в системах, соответствующих SUSv2 или SUSv3, но многие подобные системы его предоставляют. На данный момент этот подход уже устарел.
- Определить макрос `_FILE_OFFSET_BITS` со значением `64` при компиляции своей программы. Это наиболее предпочтительный подход, поскольку он позволяет соответствующим приложениям получать функциональные возможности LFS без внесения каких-либо изменений в исходный код.

Переходный API LFS

Чтобы воспользоваться переходным API LFS, нужно при компилировании своей программы определить макрос проверки возможностей `_LARGEFILE64_SOURCE` либо в командной строке, либо внутри исходного файла перед включением любых заголовочных файлов. Этот API предоставляет функции, способные работать с 64-разрядными размерами файлов и файловыми смещениями. У этих функций такие же имена, как и у их 32-разрядных двойников, но с добавлением к именам функций суффикса 64. К числу таких функций относятся `fopen64()`, `open64()`, `lseek64()`, `truncate64()`, `stat64()`, `mmap64()` и `setrlimit64()`. (Некоторые из их 32-разрядных двойников уже рассматривались, другие же будут описаны в этой главе чуть позже.)

Чтобы обратиться к большому файлу, нужно просто воспользоваться 64-разрядной версией функции. Например, чтобы открыть большой файл, можно написать следующий код:

```
fd = open64(name, O_CREAT | O_RDWR, mode);
if (fd == -1)
  errExit("open");
```

Вызов open64() эквивалентен указанию флага O_LARGEFILE при вызове open(). Попытки открыть файл, размер которого превышает 2 Гбайт, с помощью open() без этого флага приведут к возвращению ошибки.

Вдобавок к вышеупомянутым функциям переходный API добавляет несколько типов данных, в числе которых:

- `struct stat64`: аналог структуры `stat` (см. раздел 15.1), позволяющий работать с большими файлами;
- `off64_t`: 64-разрядный тип для представления файловых смещений.

Как показано в листинге 5.3, тип данных `off64_t` используется (кроме всего прочего) с функцией `lseek64()`. Демонстрируемая там программа получает два аргумента командной строки: имя открываемого файла и целочисленное значение, указывающее файловое смещение. Программа открывает указанный файл, переходит по заданному файловому смещению, а затем записывает строку. В следующей сессии командной оболочки показано использование программы для перехода в файле по очень большому файловому смещению (больше 10 Гбайт) с дальнейшей записью нескольких байтов:

```
$ ./large_file x 10111222333
$ ls -l x               Проверка размера получившегося в результате файла
-rw-------    1 mtk      users    10111222337 Mar  4 13:34 x
```

Листинг 5.3. Обращение к большим файлам

fileio/large_file.c

```
#define _LARGEFILE64_SOURCE
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
  int fd;
  off64_t off;

  if (argc != 3 || strcmp(argv[1], "--help") == 0)
    usageErr("%s pathname offset\n", argv[0]);
  fd = open64(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  if (fd == -1)
    errExit("open64");
  off = atoll(argv[2]);
  if (lseek64(fd, off, SEEK_SET) == -1)
    errExit("lseek64");

  if (write(fd, "test", 4) == -1)
    errExit("write");
  exit(EXIT_SUCCESS);
}
```

fileio/large_file.c

Макрос _FILE_OFFSET_BITS

Для получения функциональных возможностей LFS рекомендуется определить макрос `_FILE_OFFSET_BITS` со значением `64` при компиляции программы. Один из способов предусматривает использование ключа командной строки при запуске компилятора языка C:

```
$ cc -D_FILE_OFFSET_BITS=64 prog.c
```

Альтернативой может послужить определение этого макроса в исходном файле на языке C перед включением любых заголовочных файлов:

```
#define _FILE_OFFSET_BITS 64
```

Это определение автоматически переводит использование всех соответствующих 32-разрядных функций и типов данных на применение их 64-разрядных двойников. Так, например, вызов `open()` фактически превращается в вызов `open64()`, а тип данных `off_t` определяется в виде 64-разрядного длинного целого числа. Иными словами, мы можем перекомпилировать существующую программу для работы с большими файлами, не внося при этом никаких изменений в исходный код.

Добавление макроса проверки возможностей `_FILE_OFFSET_BITS` явно проще применения переходного API LFS, но этот подход зависит от чистоты написания приложений (например, от правильного использования `off_t` для объявления переменных, хранящих файловые смещения, вместо применения свойственного языку C целочисленного типа).

Наличие макроса `_FILE_OFFSET_BITS` в LFS-спецификации не требуется, он лишь упоминается в ней как дополнительный метод указания размера типа данных `off_t`. Для получения этих же функциональных возможностей в некоторых реализациях UNIX используются другие макросы проверки возможностей.

При попытке обращения к большому файлу с использованием 32-разрядных функций (то есть из программы, скомпилированной без установки для _FILE_OFFSET_BITS значения 64) можно столкнуться с ошибкой EOVERFLOW. Например, она может быть выдана при попытке использовать 32-разрядную версию функции stat() (см. раздел 15.1) для извлечения информации о файле, размер которого превышает 2 Гбайт.

Передача значений off_t вызовам printf()

Надо отметить, что LFS-расширения не решают для нас одну проблему: как выбрать способ передачи значений `off_t` вызовам `printf()`. В подразделе 3.6.2 было отмечено, что портируемый метод, который выводит значения одного из предопределенных типов системных данных (например, `pid_t` или `uid_t`), заключается в приведении значения к типу `long` и использовании для `printf()` спецификатора `%ld`. Но если применяются LFS-расширения, для типа данных `off_t` этого зачастую недостаточно, поскольку он может быть определен как тип, который длиннее `long`, обычно как `long long`. Поэтому для вывода значения типа `off_t` оно приводится к `long long`, а для `printf()` задается спецификатор `%lld`:

```
#define _FILE_OFFSET_BITS 64

off_t offset;         /* Должен быть 64 бита, а это размер 'long long' */

/* Некоторый код, присваивающий значение 'offset' */
printf("offset=%lld\n", (long long) offset);
```

Подобные замечания применимы и к родственному типу данных `blkcnt_t`, используемому в структуре `stat` (рассматриваемой в разделе 15.1).

Если аргументы функции, имеющие тип off_t или stat, передаются между отдельно откомпилированными модулями, необходимо обеспечить использование в обоих модулях одинаковых размеров для этих типов (то есть оба должны быть скомпилированы либо с установкой для _FILE_OFFSET_BITS значения 64, либо без этих установок).

5.11. Каталог /dev/fd

Каждому процессу ядро предоставляет специальный виртуальный каталог `/dev/fd`. Он содержит имена файлов вида `/dev/fd/`*n*, где *n* является номером, соответствующим одному из дескрипторов файла, открытого для этого процесса. К примеру, `/dev/fd/0` является для процесса стандартным вводом. (Свойство каталога `/dev/fd` в SUSv3 не указано, но некоторые другие реализации UNIX его предоставляют.)

В некоторых системах (но не в Linux) открытие одного из файлов в каталоге `/dev/fd` эквивалентно дублированию соответствующего файлового дескриптора. Таким образом, следующие инструкции эквивалентны друг другу:

```
fd = open("/dev/fd/1", O_WRONLY);
fd = dup(1);                          /* Дублирование стандартного вывода */
```

Аргумент `flags` вызова `open()` интерпретируется, поэтому следует позаботиться об указании точно такого же режима доступа, который был использован исходным дескриптором. Указывать другие флаги, такие как `O_CREAT`, в данном контексте не имеет смысла (они просто игнорируются).

В Linux открытие одного из файлов в `/dev/fd` эквивалентно повторному открытию исходного файла. Иначе говоря, новый файловый дескриптор связан с новым описанием открытого файла (и, следовательно, имеет различные флаги состояния файла и смещение файла).

> Фактически /dev/fd является символьной ссылкой на характерный для Linux каталог /proc/self/fd. Он представляет собой частный случай свойственных для Linux каталогов /proc/PID/fd, в каждом из которых хранятся символьные ссылки, соответствующие всем файлам, содержащимся процессом в открытом состоянии.

В программах файлы в каталоге `/dev/fd` редко используются. Наиболее часто они применяются в оболочке. Многие из доступных пользователю команд принимают в качестве аргументов имена файлов, и иногда удобно соединить эти команды с помощью конвейера, чтобы использовать стандартный ввод или вывод в качестве такого аргумента. Для этой цели некоторые программы (например, diff, ed, tar и comm) задействуют аргумент, состоящий из одиночного дефиса (`-`), означающего: «в качестве файла, имя которого должно быть указано в аргументах, использовать стандартный ввод или вывод (что больше соответствует)». Так, для сравнения списка файлов из `ls` с ранее созданным списком файлов можно набрать такую команду:

```
$ ls | diff - oldfilelist
```

У этого подхода имеются различные недостатки. Во-первых, он требует определенной интерпретации символа дефиса в части каждой программы, и многие программы подобной интерпретации не осуществляют; они написаны для работы только с аргументами в виде имен файлов, и у них нет средств указания стандартного ввода или вывода в качестве файлов, с которыми им нужно работать. Во-вторых, некоторые программы вместо этого интерпретируют одиночный дефис в качестве разделителя, обозначающего конец ключей командной строки.

Использование `/dev/fd` устраняет эти трудности, позволяя указывать стандартный ввод, вывод и ошибку в виде аргументов, обозначающих имя файла для любой программы, которой они требуются. Поэтому предыдущую команду оболочки можно переписать в таком виде:

```
$ ls | diff /dev/fd/0 oldfilelist
```

Для удобства в качестве символьных ссылок на /dev/fd/0, /dev/fd/1 и /dev/fd/2 соответственно предоставляются имена /dev/stdin, /dev/stdout и /dev/stderr.

5.12. Создание временных файлов

Некоторые программы нуждаются в создании временных файлов, используемых только при выполнении программы, а при завершении программы такие файлы должны быть удалены. Например, временные файлы создаются в ходе компиляции многими компиляторами. Для этих целей в GNU-библиотеке языка C предоставляется несколько библиотечных функций. Здесь будут рассмотрены две такие функции: mkstemp() и tmpfile().

Функция mkstemp() создает уникальные имена файлов на основе шаблона, предоставляемого вызывающим процессом, и открывает файл, возвращая файловый дескриптор, который может быть использован с системными вызовами ввода-вывода.

```
#include <stdlib.h>

int mkstemp(char *template);
```

При успешном завершении возвращает новый файловый дескриптор, а при ошибке выдает -1

Аргумент template принимает форму путевого имени, последними шестью символами которого должны быть XXXXXX. Эти шесть символов заменяются строкой, придающей имени уникальность, и измененная строка возвращается через аргумент template. Поскольку template изменен, он должен указываться как массив символов, а не как строковая константа.

Функция mkstemp() создает файл с правами на чтение и запись для владельца файла (и без прав для других пользователей) и открывает его с флагом O_EXCL, гарантируя вызывающему процессу эксклюзивный доступ к файлу.

Обычно временный файл отсоединяется (удаляется) вскоре после своего открытия, с помощью системного вызова unlink() (см. раздел 18.3). Функцией mkstemp() можно воспользоваться следующим образом:

```
int fd;
char template[] = "/tmp/somestringXXXXXX";

fd = mkstemp(template);
if (fd == -1)
  errExit("mkstemp");
printf("Generated filename was: %s\n", template);
unlink(template);   /* Имя тут же исчезает, но файл удаляется только после close() */

/* Использование системных вызовов ввода-вывода – read(), write() и т. д. */
if (close(fd) == -1)
    errExit("close");
```

Для создания уникальных имен файлов могут также применяться функции tmpnam(), tempnam() и mktemp(). Но их использования следует избегать, поскольку они могут создавать в приложении бреши в системе безопасности. Дополнительные подробности об этих функциях можно найти на страницах руководства.

Функция `tmpfile()` создает временный файл с уникальным именем, открытый для чтения и записи. (Файл открыт с флагом `O_EXCL`, чтобы защититься от маловероятной возможности, что файл с таким же именем уже был создан другим процессом.)

```
#include <stdio.h>

FILE *tmpfile(void);
```

Возвращает указатель на файл при успешном завершении или `NULL` при ошибке

При удачном завершении `tmpfile()` возвращает файловый поток, который может использоваться функциями библиотеки stdio. Временный файл при закрытии автоматически удаляется. Для этого `tmpfile()` совершает внутренний вызов `unlink()` для немедленного удаления имени файла после его открытия.

5.13. Резюме

В этой главе была описана концепция атомарности, играющая важную роль для правильного функционирования некоторых системных вызовов. В частности, флаг `O_EXCL` системного вызова `open()` позволяет вызывающему процессу гарантировать, что тот является создателем файла, а флаг `O_APPEND` системного вызова `open()` гарантирует, что несколько процессов, добавляющих данные в один и тот же файл, не смогут переписать вывод друг друга.

Системный вызов `fcntl()` может выполнять над файлом разнообразные операции, включая изменение флагов состояния файла и дублирование файловых дескрипторов. Последнюю операцию можно также выполнить с помощью системных вызовов `dup()` и `dup2()`.

Была рассмотрена взаимосвязь между файловыми дескрипторами, дескрипциями открытых файлов и файловыми индексными дескрипторами и отмечено, что с каждым из этих трех объектов связана различная информация. Продублированные файловые дескрипторы ссылаются на одну и ту же дескрипцию открытого файла и поэтому совместно используют флаги состояния открытого файла и файловое смещение.

Были описаны некоторые системные вызовы, расширяющие функциональные возможности обычных системных вызовов `read()` и `write()`. Системные вызовы `pread()` и `pwrite()` выполняют ввод/вывод в указанном месте файла, не изменяя при этом файлового смещения. Системные вызовы `readv()` и `writev()` выполняют фрагментированный ввод/вывод. Вызовы `preadv()` и `pwritev()` сочетают в себе функциональные возможности фрагментированного ввода-вывода с возможностью выполнять ввод/вывод в указанном месте файла.

Системные вызовы `truncate()` и `ftruncate()` могут использоваться для уменьшения размера файла, для избавления от избыточных байтов или для наращивания размера путем добавления файловых дыр, заполненных нулевыми байтами.

Была также кратко рассмотрена концепция неблокирующего ввода-вывода, к которой мы еще вернемся в последующих главах.

LFS-спецификация определяет набор расширений, позволяющих процессам, запущенным на 32-разрядных системах, выполнять операции над файлами, размер которых слишком велик, чтобы быть представленным в 32 битах.

Пронумерованные файлы в виртуальном каталоге `/dev/fd` позволяют процессу обращаться к его собственным открытым файлам по номерам файловых дескрипторов, что, в частности, может пригодиться в командах оболочки.

Функции `mkstemp()` и `tmpfile()` позволяют приложению создавать временные файлы.

5.14. Упражнения

5.1. Если у вас есть доступ к 32-разрядной системе Linux, измените программу в листинге 5.3 под использование стандартных системных вызовов файлового ввода-вывода (`open()` и `lseek()`) и под тип данных `off_t`. Откомпилируйте программу с установленным для макроса `_FILE_OFFSET_BITS` значением `64` и протестируйте ее, показав, что она может успешно создавать большие файлы.

5.2. Напишите программу, открывающую существующий файл для записи с флагом `O_APPEND`, а затем переведите файловое смещение в начало файла перед записью каких-либо данных. Куда в файле будут помещены добавляемые данные? Почему?

5.3. Это упражнение демонстрирует необходимость атомарности, гарантированной при открытии файла с флагом `O_APPEND`. Напишите программу, получающую до трех аргументов командной строки:

```
$ atomic_append filename num-bytes [x]
```

Эта программа должна открыть файл с именем, указанным в аргументе *`filename`* (создав его при необходимости), и дополнить его количеством байтов, заданным в аргументе *`num-bytes`*, используя вызов `write()` для побайтовой записи. По умолчанию программа должна открыть файл с флагом `O_APPEND`, но, если есть третий аргумент командной строки (*`x`*), флаг `O_APPEND` должен быть опущен. При этом, вместо того чтобы добавлять байты, программа должна выполнять перед каждым вызовом `write()` вызов `lseek(fd, 0, SEEK_END)`. Запустите одновременно два экземпляра этой программы без аргумента *`x`* для записи одного миллиона байтов в один и тот же файл:

```
$ atomic_append f1 1000000 & atomic_append f1 1000000
```

Повторите те же действия, ведя запись в другой файл, но на этот раз с указанием аргумента *`x`*:

```
$ atomic_append f2 1000000 x & atomic_append f2 1000000 x
```

Выведите на экран размеры файлов `f1` и `f2`, воспользовавшись командой `ls -l`, и объясните разницу между ними.

5.4. Реализуйте функции `dup()` и `dup2()`, используя функцию `fcntl()` и, там где это необходимо, функцию `close()`. (Тот факт, что `dup2()` и `fcntl()` в некоторых случаях возникновения ошибок возвращают различные значения `errno`, можно проигнорировать.) Для `dup2()` не забудьте учесть особый случай, когда `oldfd` равен `newfd`. В этом случае нужно проверить допустимость значения `oldfd`, что можно сделать, к примеру, проверкой успешности выполнения вызова `fcntl(oldfd, F_GETFL)`. Если значение `oldfd` недопустимо, функция должна возвратить `-1`, а значение `errno` должно быть установлено в `EBADF`.

5.5. Напишите программу для проверки совместного использования файловыми дескрипторами значения файлового смещения и флагов состояния открытого файла.

5.6. Объясните, каким должно быть содержимое выходного файла после каждого вызова `write()` в следующем коде и почему:

```
fd1 = open(file, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
fd2 = dup(fd1);
fd3 = open(file, O_RDWR);
write(fd1, "Hello,", 6);
write(fd2, " world", 6);
lseek(fd2, 0, SEEK_SET);
write(fd1, "HELLO,", 6);
write(fd3, "Gidday", 6);
```

5.7. Реализуйте функции `readv()` и `writev()`, используя системные вызовы `read()`, `write()` и подходящие функции из пакета malloc (см. подраздел 7.1.2).

6 Процессы

В этой главе будет рассмотрена структура процесса, при этом особое внимание мы уделим структуре и содержимому виртуальной памяти процесса. Будут также изучены некоторые атрибуты процесса. В следующих главах мы рассмотрим другие атрибуты процесса (например, идентификаторы процесса в главе 9 и приоритеты процесса и его диспетчеризацию в главе 35). В главах 24–27 описываются особенности создания процесса, методы прекращения его работы и методы создания процессов для выполнения новых программ.

6.1. Процессы и программы

Процесс является экземпляром выполняемой программы. В данном разделе мы подробно разберем это определение и вы узнаете разницу между программой и процессом.

Программа представляет собой файл, содержащий различную информацию о том, как сконструировать процесс в ходе выполнения. В эту информацию включается следующее.

- *Идентификационный признак двоичного формата*. Каждый программный файл включает в себя метаинформацию с описанием формата исполняемого файла. Это позволяет ядру интерпретировать всю остальную содержащуюся в файле информацию. Изначально для исполняемых файлов UNIX было предусмотрено два широко используемых формата: исходный формат a.out (assembler output — вывод на языеке ассемблера) и появившийся позже более сложный общий формат объектных файлов — COFF (Common Object File Format). В настоящее время в большинстве реализаций UNIX (включая Linux) применяется формат исполняемых и компонуемых файлов — Executable and Linking Format (ELF), предоставляющий множество преимуществ по сравнению со старыми форматами.
- *Машинный код*. В нем закодирован алгоритм программы.
- *Адрес входа в программу*. В нем указывается место той инструкции, с которой должно начаться выполнение программы.
- *Данные*. В программном файле содержатся значения, используемые для инициализации переменных, а также применяемые программой символьные константы (например, строки).
- *Таблицы имен и переадресации*. В них дается описание расположений и имен функций и переменных внутри программы. Эти таблицы предназначены для различных целей, включая отладку и разрешение имен в ходе выполнения программы (динамическое связывание).
- *Информация о совместно используемых библиотеках и динамической компоновке*. В программный файл включаются поля, где перечисляются совместно используемые библиотеки, которые программе потребуются в ходе выполнения, а также путевое имя динамического компоновщика, который должен применяться для загрузки этих библиотек.
- *Другая информация*. В программном файле есть и другая информация, описывающая способ построения процесса.

Одна программа может использоваться для построения множества процессов, или же, если наоборот, во множестве процессов может быть запущена одна и та же программа.

Определение процесса, которое было дано в начале этого раздела, можно переформулировать следующим образом. Процесс является абстрактной сущностью, которая установлена ядром и которой для выполнения программы выделяются системные ресурсы.

С позиции ядра процесс состоит из памяти пользовательского пространства, содержащей код программы и переменных, используемых этим кодом, а также из ряда структур данных ядра, хранящих информацию о состоянии процесса. Информация, записанная в структурах данных ядра, включает в себя различные идентификаторы, связанные с процессом, таблицы виртуальной памяти, таблицу дескрипторов открытых файлов, сведения, относящиеся к доставке и обработке сигналов, использованию и ограничениям ресурсов процесса, сведения о текущем рабочем каталоге, а также множество других данных.

6.2. Идентификатор процесса и идентификатор родительского процесса

У каждого процесса есть *идентификатор* (process ID — PID), положительное целое число, уникальным образом идентифицирующее процесс в системе. Идентификаторы процессов используются и возвращаются различными системными вызовами. Например, системный вызов `kill()` (см. раздел 20.5) позволяет отправить сигнал процессу с указанным идентификатором. PID также используется при необходимости создания идентификатора, который будет уникальным для процесса. Характерный пример — применение идентификатора процесса как части уникального для процесса имени файла.

Идентификатор вызывающего процесса возвращается системным вызовом `getpid()`.

```
#include <unistd.h>

pid_t getpid(void);
```

Всегда успешно возвращает идентификатор вызывающего процесса

Тип данных `pid_t`, используемый для значения, возвращаемого `getpid()`, является целочисленным типом. Он определен в спецификации SUSv3 для хранения идентификаторов процессов.

За исключением нескольких системных процессов, таких как `init` (чей PID равен 1), между программой и идентификатором процесса, созданным для ее выполнения, нет никакой фиксированной связи.

Ядро Linux ограничивает количество идентификаторов процессов числом, меньшим или равным 32 767. При создании нового процесса ему присваивается следующий по порядку PID. Всякий раз при достижении ограничения в 32 767 идентификаторов ядро перезапускает свой счетчик идентификаторов процессов, чтобы они назначались, начиная с наименьших целочисленных значений.

> По достижении числа 32 767 счетчик идентификаторов процессов переустанавливается на значение 300, а не на 1. Так происходит потому, что многие идентификаторы процессов с меньшими номерами находятся в постоянном использовании системными процессами и демонами, и время на поиск неиспользуемого PID в этом диапазоне будет потрачено впустую.

В Linux 2.4 и более ранних версиях ограничение идентификаторов процессов в 32 767 единиц определено в константе ядра PID_MAX. Начиная с Linux 2.6, ситуация изменилась. Хотя исходный верхний порог для идентификаторов процессов остался прежним — 32 767, его можно изменить, задав значение в характерном для Linux файле /proc/sys/kernel/pid_max (которое на единицу больше, чем максимально возможное количество идентификаторов процессов). На 32-разрядной платформе максимальным значением для этого файла является 32 768, но на 64-разрядной платформе оно может быть установлено в любое значение вплоть до 2^{22} (приблизительно 4 миллиона), позволяя справиться с очень большим количеством процессов.

У каждого процесса имеется родительский процесс, то есть тот процесс, который его создал. Определить идентификатор своего родительского процесса вызывающий процесс может с помощью системного вызова `getppid()`.

```
#include <unistd.h>

pid_t getppid(void);
```

Всегда успешно возвращает идентификатор родительского процесса для того процесса, который его вызвал

По сути, имеющийся у каждого процесса атрибут идентификатора родительского процесса представляет древовидную связь всех процессов в системе. Родитель каждого процесса имеет собственного родителя и т. д., возвращаясь в конечном итоге к процессу 1, `init`, предку всех процессов. (Это «родовое дерево» может быть просмотрено с помощью команды `pstree(1)`.)

Если дочерний процесс становится «сиротой» из-за завершения работы «породившего» его родительского процесса, то он оказывается приемышем у процесса `init` и последующий за этим вызов `getppid()`, сделанный из дочернего процесса, возвратит результат `1` (см. раздел 26.2).

Родитель любого процесса может быть найден при просмотре поля `PPid`, предоставляемого характерным для Linux файлом `/proc/`*`PID`*`/status`.

6.3. Структура памяти процесса

Память, выделяемая каждому процессу, состоит из нескольких частей, которые обычно называют *сегментами*. К числу таких сегментов относятся следующие.

- *Текстовый сегмент* — содержит машинный код, который принадлежат программе, запущенной процессом. Текстовый сегмент создается только для чтения, чтобы процесс не мог случайно изменить свои собственные инструкции из-за неверного значения указателя. Поскольку многие процессы могут выполнять одну и ту же программу, текстовый сегмент создается с возможностью совместного использования. Таким образом, единственная копия кода программы может быть отображена на виртуальное адресное пространство всех процессов.
- *Сегмент инициализированных данных* — хранит глобальные и статические переменные, инициализированные явным образом. Значения этих переменных считываются из исполняемого файла при загрузке программы в память.
- *Сегмент неинициализированных данных* — содержит глобальные и статические переменные, не инициализированные явным образом. Перед запуском программы система

инициализирует всю память в этом сегменте значением `0`. По историческим причинам этот сегмент часто называют bss. Его имя произошло из старого ассемблерного мнемонического термина block started by symbol («блок, начинающийся с символа»). Основная причина помещения прошедших инициализацию глобальных и статических переменных в отдельный от неинициализированных переменных сегмент заключается в том, что, когда программа сохраняется на диске, нет никакого смысла выделять пространство под неинициализированные данные. Вместо этого исполняемой программе просто нужно записать местоположение и размер, требуемый для сегмента неинициализированных данных, и это пространство выделяется загрузчиком программы в ходе ее выполнения.

- Динамически увеличивающийся и уменьшающийся *сегмент стека* — содержит стековые фреймы. Для каждой вызванной на данный момент функции выделяется один стековый фрейм. Во фрейме хранятся локальные переменные функции (так называемые автоматические переменные), аргументы и возвращаемое значение. Более подробно стековые фреймы рассматриваются в разделе 6.5.
- *Динамическая память*, или *куча*, — область, из которой память (для переменных) может динамически выделяться в ходе выполнения программы. Верхний конец кучи называют *крайней точкой программы (program break)*.

Не такими популярными, но более наглядными маркировками для сегментов инициализированных и неинициализированных данных являются *сегмент данных, инициализированных пользователем* (user-initialized data segment), и *сегмент данных с нулевой инициализацией* (zero-initialized data segment).

Команда `size(1)` выводит размеры текстового сегмента, сегментов инициализированных и неинициализированных (bss) данных двоичной исполняемой программы.

> Термин «сегмент», который употребляется в основном тексте, не нужно путать с аппаратной сегментацией, используемой в некоторой аппаратной архитектуре, например в x86-32. В нашем случае сегменты представляют собой логические разделения виртуальной памяти процесса в системах UNIX. Иногда вместо сегмента употребляется термин «раздел» (section), поскольку он более соответствует терминологии, используемой в настоящее время повсеместно согласно ELF-спецификации для форматов исполняемого файла.
>
> В этой книге часто встречаются места, где говорится, что библиотечная функция возвращает указатель на статически выделяемую память. Под этим понимается, что память выделена либо под сегмент инициализированных данных, либо под сегмент неинициализированных данных. (В некоторых случаях библиотечные функции могут вместо этого выполнять однократное динамическое выделение памяти в куче, но эта деталь реализации не имеет отношения к рассматриваемому здесь смысловому значению слова «указатель».) О случаях, когда библиотечная функция возвращает информацию посредством статически выделяемой памяти, важно знать, поскольку эта память существует независимо от привлечения функции и может быть переписана последующими вызовами той же самой функции (или же, в некоторых случаях, путем последующих вызовов родственных функций). При использовании статической памяти функция становится нереентерабельной (не допускается повторный вызов функции до завершения ее работы). Дополнительные сведения о реентерабельности даются в подразделе 21.1.2 и разделе 31.1.

В листинге 6.1 продемонстрированы различные типы переменных в коде на языке C, а также комментарии, показывающие, в каких сегментах каждая переменная размещается. Эти комментарии предполагают применение неоптимизирующего компилятора и такого двоичного интерфейса приложения, в котором все аргументы передаются в стек. На практике оптимизирующий компилятор может поместить часто используемые переменные в регистры или провести оптимизацию, вообще исключая существование переменной. Кро-

ме того, некоторые ABI требуют, чтобы аргументы функций и результаты их выполнения передавались через регистры, а не через стек. Как бы то ни было, этот пример предназначен для демонстрации отображения переменных кода на языке C на сегменты процесса.

Листинг 6.1. Размещение переменных программы в сегментах памяти процесса

proc/mem_segments.c

```
#include <stdio.h>
#include <stdlib.h>

char globBuf[65536];            /* Сегмент неинициализированных данных */
int primes[] = { 2, 3, 5, 7 };  /* Сегмент инициализированных данных */

static int
square(int x)           /* Размещается в фрейме для square() */
{
  int result;           /* Размещается в фрейме для square() */
  result = x * x;
  return result;        /* Возвращаемое значение передается через регистр */
}

static void
doCalc(int val)         /* Размещается в фрейме для doCalc() */
{
  printf("The square of %d is %d\n", val, square(val));
  if (val < 1000) {
    int t;                      /* Размещается в фрейме для doCalc() */
    t = val * val * val;
    printf("The cube of %d is %d\n", val, t);
  }
}

int
main(int argc, char *argv[])    /* Размещается в фрейме для main() */
{
  static int key = 9973;        /* Сегмент инициализированных данных */
  static char mbuf[10240000];   /* Сегмент неинициализированных данных */
  char *p;                      /* Размещается в фрейме для main() */
  p = malloc(1024);             /* Указывает на память в сегменте кучи */
  doCalc(key);
  exit(EXIT_SUCCESS);
}
```

proc/mem_segments.c

> Двоичный интерфейс приложений — Application Binary Interface (ABI) представляет собой набор правил, регулирующих порядок обмена информацией между двоичной исполняемой программой в ходе ее выполнения и каким-либо сервисом (например, ядром или библиотекой). Помимо всего прочего, ABI определяет, какие регистры и места в стеке используются для обмена этой информацией и какой смысл придается обмениваемым значениям. Программа, единожды скомпилированная в соответствии с требованием некоторого ABI, должна запускаться в любой системе, предоставляющей точно такой же ABI. Это отличается от стандартизированного API (например, SUSv3), гарантирующего портируемость только для приложений, скомпилированных из исходного кода.

Хотя это и не описано в SUSv3, среда программы на языке C во многих реализациях UNIX (включая Linux) предоставляет три глобальных идентификатора: `etext`, `edata`

и end. Они могут использоваться из программы для получения адресов следующего байта соответственно за концом текста программы, за концом сегмента инициализированных данных и за концом сегмента неинициализированных данных. Чтобы воспользоваться этими идентификаторами, их нужно явным образом объявить:

```
extern char etext, edata, end;
        /* К примеру, &etext сообщает адрес первого байта после окончания
           текста программы/начала инициализированных данных */
```

На рис. 6.1 показано расположение различных сегментов памяти в архитектуре x86-32. Пространство с пометкой argv, охватывающее верхнюю часть этой схемы, содержит аргументы командной строки программы (которые в C доступны через аргумент argv функции main()) и список переменных среды процесса (который вскоре будет рассмотрен). Шестнадцатеричные адреса, приведенные в схеме, могут варьироваться в зависимости от конфигурации ядра и ключей компоновки программы. Области, закрашенные серым цветом, представляют собой недопустимые диапазоны в виртуальном адресном пространстве процесса, то есть области, для которых не созданы таблицы страниц (см. далее раздел, посвященный управлению виртуальной памятью).

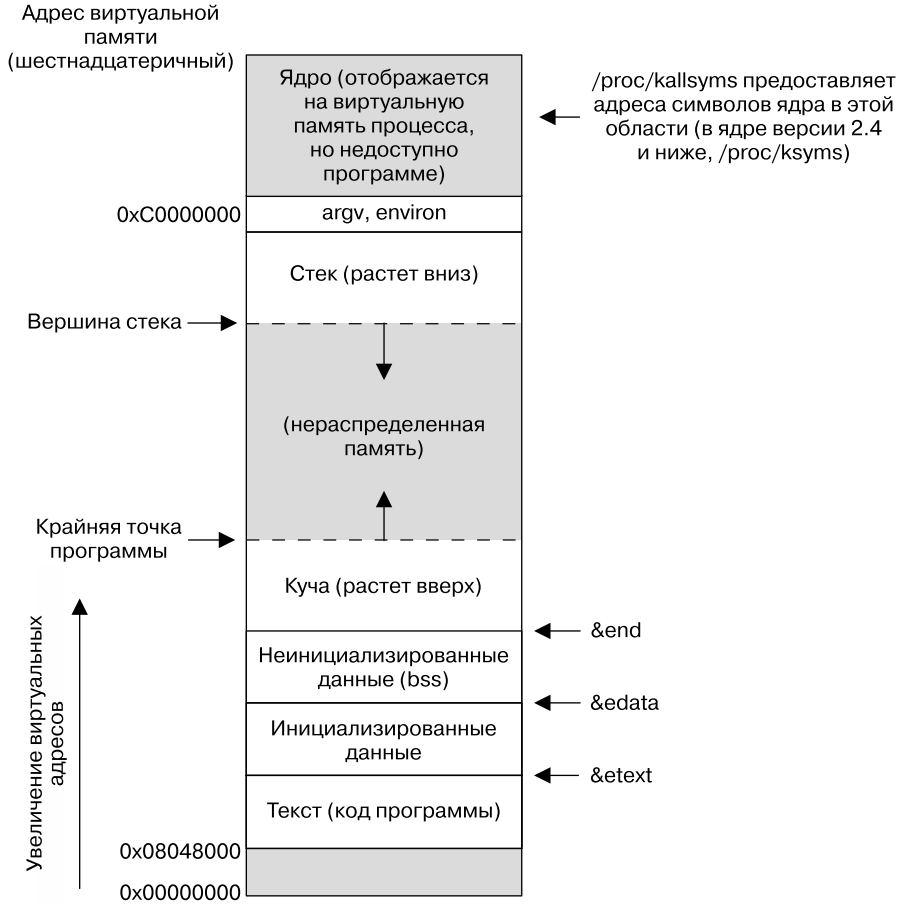


Рис. 6.1. Типичная структура памяти процесса в Linux/x86-32

6.4. Управление виртуальной памятью

В предыдущем разделе при рассмотрении структуры памяти процесса умалчивался тот факт, что речь шла о структуре в *виртуальной памяти*. Хотя к описанию виртуальной памяти лучше было бы приступить чуть позже, при изучении таких тем, как системный вызов `fork()`, совместно используемая память и отображаемые файлы, некоторые особенности придется рассмотреть прямо сейчас.

Следуя в ногу с большинством современных ядер, Linux использует подход, известный как *управление виртуальной памятью*. Цель этой технологии заключается в создании условий для эффективного использования как центрального процессора, так и оперативной (физической) памяти путем применения свойства локальности ссылок, присущего многим программам. Локальность в большинстве программ проявляется в двух видах.

- *Пространственная локальность*, которая характеризуется присущей программам тенденцией ссылаться на адреса памяти, близкие к тем, к которым недавно обращались (из-за последовательного характера обработки инструкций и иногда последовательного характера обработки структур данных).
- *Локальность по отношению ко времени* — характеризуется свойственной программам тенденцией обращаться к тем же адресам памяти в ближайшем будущем, к которым обращение уже было в недавнем прошлом (из-за использования циклов).

В результате локальности ссылок появляется возможность выполнять программу, располагая в оперативной памяти лишь часть ее адресного пространства.

Структура виртуальной памяти подразумевает разбиение памяти, используемой каждой программой, на небольшие блоки фиксированного размера, называемые *страницами*. Соответственно, оперативная память делится на блоки *страничных кадров (фреймов)* одинакового размера. В любой отдельно взятый момент времени в страничных кадрах физической памяти требуется наличие только некоторых страниц программы. Эти страницы формируют так называемый *резидентный набор*. Копии неиспользуемых страниц программы размещаются в *области подкачки* — зарезервированной области дискового пространства, применяемой для дополнения оперативной памяти компьютера, — и загружаются в оперативную память лишь по мере надобности. Когда процесс ссылается на страницу, которой нет в оперативной памяти, происходит ошибка отсутствия страницы, в результате чего ядро приостанавливает выполнение процесса, пока страница загружается с диска в память.

> В системах x86-32 размер страницы составляет 4096 байт. В некоторых других реализациях Linux используются страницы больших размеров. Например, в Alpha — страницы размером 8192 байт, а в IA-64 — изменяемый размер страниц, обычно с исходным объемом 16 384 байт. Программа может определить размер страницы виртуальной памяти системы с помощью вызова sysconf(_SC_PAGESIZE), рассматриваемого в разделе 11.2.

Для поддержки этой организации ядро ведет для каждого процесса *таблицу страниц* (рис. 6.2). В ней дается описание размещения каждой страницы в *виртуальном адресном пространстве* процесса (набора всех страниц виртуальной памяти, доступных процессу). В каждой записи таблицы страниц указывается либо расположение виртуальной страницы в памяти, либо то место, которое она в данный момент занимает на диске.

Записи в таблице страниц нужны не всем адресным диапазонам виртуального адресного пространства процесса. Обычно большие диапазоны потенциального виртуального пространства не используются, поэтому нет необходимости вести соответствующие записи в таблице страниц. Если процесс пытается получить доступ к адресу, для которого не имеется соответствующей записи в таблице страниц, он получает сигнал `SIGSEGV`.

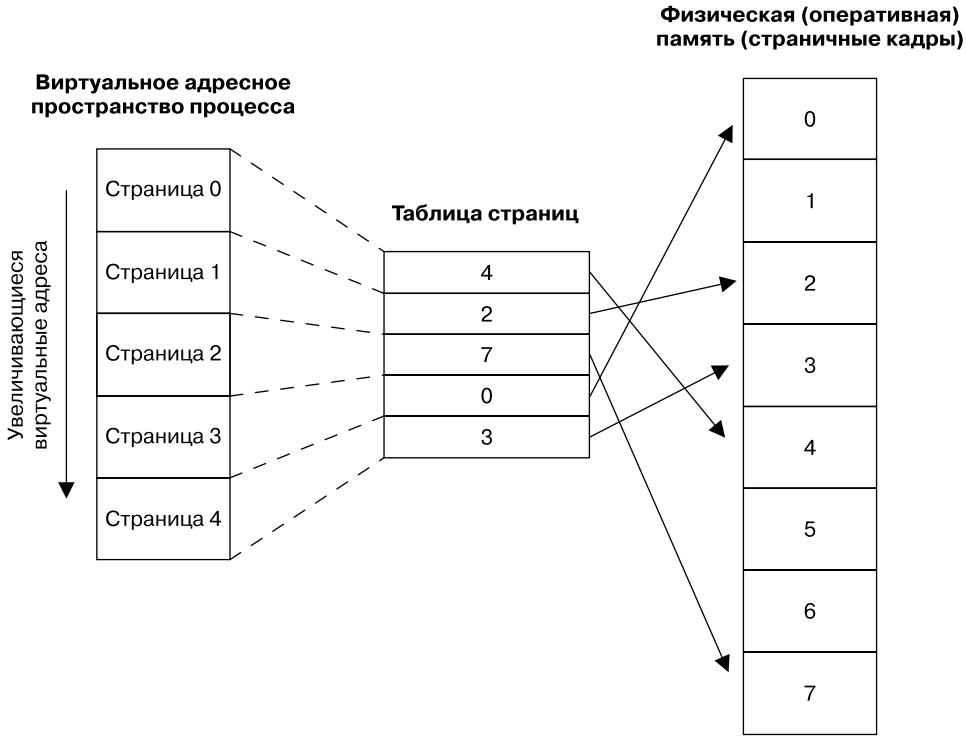


Рис. 6.2. Общий вид виртуальной памяти

Диапазон допустимых для процесса виртуальных адресов за время его жизненного цикла может измениться по мере того, как ядро будет выделять для процесса и высвобождать страницы (и записи в таблице страниц). Это может происходить при следующих обстоятельствах:

- когда стек разрастается вниз, выходя за ранее обозначенные ограничения;
- когда память выделяется в куче или высвобождается в ней путем подъема крайней точки программы с использованием вызовов `brk()`, `sbrk()` или семейства функций `malloc` (см. главу 7);
- когда области совместно используемой памяти (System V) прикрепляются с помощью вызова `shmat()` и открепляются вызовом `shmdt()`;
- когда отображение памяти создается с применением вызова `mmap()` и убирается с помощью `munmap()` (см. главу 45).

Реализация виртуальной памяти требует аппаратной поддержки в виде блока управления страничной памятью — Paged Memory Management Unit (PMMU). Блок PMMU переводит каждую ссылку на адрес виртуальной памяти в соответствующий адрес физической памяти и извещает ядро об ошибке отсутствия страницы, когда конкретный адрес виртуальной памяти ссылается на страницу, отсутствующую в оперативной памяти.

Управление виртуальной памятью отделяет виртуальное адресное пространство процесса от физического адресного пространства оперативной памяти. Это дает множество преимуществ.

- Процессы изолированы друг от друга и от ядра, поэтому один процесс не может прочитать или изменить память другого процесса или ядра. Это достигается за счет за-

писей в таблице страниц для каждого процесса, указывающих на различные наборы физических страниц в оперативной памяти (или в области подкачки).

- При необходимости два или несколько процессов могут задействовать память совместно. Ядро дает такую возможность благодаря наличию записей в таблице страниц в различных процессах, ссылающихся на одни и те же страницы оперативной памяти. Совместное использование памяти происходит при двух наиболее распространенных обстоятельствах.
  - Несколько процессов, выполняющих одну и ту же программу, могут совместно использовать одну и ту же (предназначенную только для чтения) копию программного кода. Эта разновидность совместного применения памяти осуществляется неявно, когда несколько программ выполняют один и тот же программный файл (или загружают одну и ту же совместно используемую библиотеку).
  - Для явного запроса областей совместно используемой памяти с другими процессами процессы могут задействовать системные вызовы `shmget()` и `mmap()`. Это делается в целях обмена данными между процессами.
- Упрощается реализация схем защиты памяти, то есть записи в таблице страниц могут быть помечены, чтобы показать, что содержимое соответствующей страницы защищено от всего, кроме чтения, записи, выполнения или некоторого сочетания допустимых действий. Когда страницы оперативной памяти совместно применяются несколькими процессами, можно указать, что у памяти есть защита от каждого процесса. Например, у одного процесса может быть доступ только к чтению страницы, а у другого — как к чтению, так и к записи.
- Программистам и таким инструментам, как компилятор и компоновщик, не нужно знать о физическом размещении программы в оперативной памяти.
- Программа загружается и запускается быстрее, поскольку в памяти требуется разместить только ее часть. Кроме того, объем памяти для среды выполнения процесса (то есть виртуальный размер) может превышать емкость оперативной памяти.

И еще одно последнее преимущество, получаемое за счет управления виртуальной памятью, заключается в том, что факт задействования каждым процессом меньшего объема оперативной памяти позволяет одновременно содержать в ней большее количество процессов. Как правило, это приводит к более эффективному использованию центрального процессора, поскольку увеличивает вероятность того, что в любой момент времени найдется хотя бы один процесс, который может быть выполнен центральным процессором.

6.5. Стек и стековые фреймы

По мере вызова функций и возврата из них стек расширяется и сжимается. В Linux на архитектуре x86-32 (и в большинстве других реализаций Linux и UNIX) стек располагается в верхней части памяти и растет вниз (по направлению к куче). Текущая вершина стека отслеживается в специально предназначенном для этого регистре — *указателе стека*. Как только вызывается функция, стеку выделяется еще один фрейм, который удаляется, как только происходит возврат из функции.

> Хотя стек растет вниз, мы все равно называем растущий край стека вершиной, поскольку, абстрактно говоря, он таковым и является. Фактическое направление роста относится к подробностям аппаратной реализации. В одной из реализаций Linux, HP PA-RISC, используется стек, растущий вверх.
>
> С точки зрения виртуальной памяти, при выделении стекового фрейма сегмент стека увеличивается в размере, но в большинстве реализаций его размер после высвобождения фреймов не уменьшается (память просто повторно используется при выделении новых стековых

фреймов). Когда говорится о расширении и сжатии сегмента стека, речь идет о логической перспективе добавляемых в стек и удаляемых из него фреймов.

Иногда применяется выражение *«пользовательский стек»* — это позволяет отличить рассматриваемый здесь стек от *стека ядра*. Стек ядра — поддерживаемая в памяти ядра область, выделяемая каждому процессу, которая используется в качестве стека для выполнения функций, вызываемых внутри системного вызова в ходе его работы. (Ядро не может применять для этой цели пользовательский стек, поскольку тот размещается в незащищенной пользовательской памяти.)

Каждый фрейм пользовательского стека содержит следующую информацию.

- *Аргументы функции и локальные переменные*. В языке C они упоминаются как *автоматические* переменные, поскольку при вызове функции создаются в автоматическом режиме. Исчезают они также автоматически, когда происходит возврат из функции (поскольку исчезает фрейм стека). В этом основное семантическое отличие автоматических переменных от статических (глобальных): последние существуют постоянно, независимо от выполнения функций.
- *Информация, связанная с вызовом*. Каждая функция задействует некоторые регистры центрального процессора, например счетчик команд, указывающий на следующий исполняемый машинный код. Когда одна функция вызывает другую, копия этих регистров сохраняется в стековом фрейме вызываемой функции, чтобы при возврате из функции можно было восстановить значения соответствующих регистров для вызывающей функции.

Поскольку функции способны вызывать друг друга, в стеке может быть несколько фреймов. (Если функция рекурсивно вызывает саму себя, то для этой функции в стеке будет несколько фреймов.) Если вспомнить листинг 6.1, то в ходе выполнения функции `square()` в стеке будут содержаться фреймы, показанные на рис. 6.3.

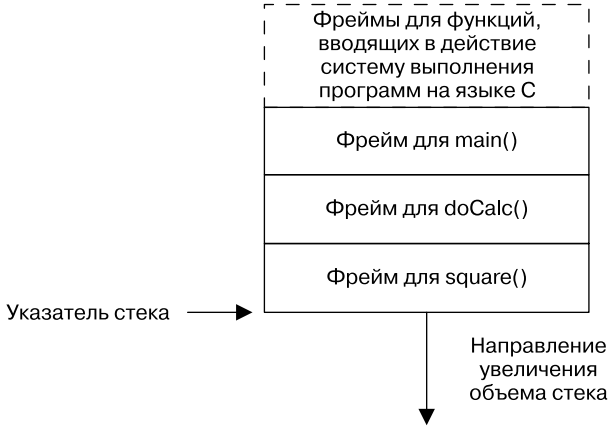


Рис. 6.3. Пример стека процесса

6.6. Аргументы командной строки (argc, argv)

У каждой программы на языке C должна быть функция по имени `main()`, с которой и начинается выполнение программы. Когда программа выполняется, к аргументам командной строки (отдельным словам, анализируемым оболочкой) открывается доступ через

два аргумента функции `main()`. Первый аргумент, `int argc`, показывает, сколько есть аргументов командной строки. Второй аргумент, `char *argv[]`, является массивом указателей на аргументы командной строки, каждый из которых представляет символьную строку, завершающуюся нулевым байтом. Первая из этих строк, `argv[0]`, является (по традиции) именем самой программы. Список указателей в `argv` завершается указателем со значением `NULL` (то есть `argv[argc]` имеет значение `NULL`).

Поскольку в `argv[0]` содержится имя, под которым программа была вызвана, это можно использовать для выполнения полезного приема. На одну и ту же программу можно создать несколько ссылок (то есть имен для нее), а затем заставить программу заглянуть в `argv[0]` и выполнить различные действия в зависимости от имени, используемого для ее вызова. Пример такой технологии предоставляется командами `gzip(1)`, `gunzip(1)` и `zcat(1)`: в некоторых дистрибутивах это ссылки на один и тот же исполняемый файл. (Применяя эту технологию, нужно быть готовым обработать вызов пользователя программы по ссылке с именем, не входящим в перечень ожидаемых имен.)

На рис. 6.4 продемонстрирован пример структур данных, связанных с `argc` и `argv`, при выполнении программы, приведенной в листинге 6.2. На этой схеме завершающие нулевые байты в конце каждой строки показаны с использованием принятой в языке C записи `\0`.

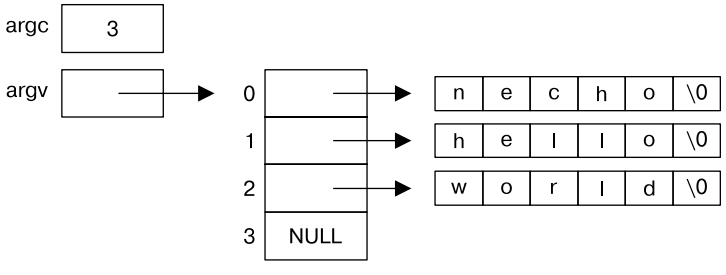


Рис. 6.4. Значения argc и argv для команды necho hello world

Программа в листинге 6.2 повторяет на экране аргументы своей командной строки, выводя каждый из них с новой строки и ставя перед ними строку, показывающую, какой именно по счету элемент `argv` выводится на экран.

Листинг 6.2. Повторение на экране аргументов командной строки

proc/necho.c

```
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    int j;
    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);
    exit(EXIT_SUCCESS);
}
```

proc/necho.c

Поскольку список `argv` завершается значением `NULL`, для построчного вывода только аргументов командной строки тело программы в листинге 6.2 можно записать по-другому:

```
char **p;

for (p = argv; *p != NULL; p++)
    puts(*p);
```

Одно из ограничений механизма, использующего argc и argv, заключается в том, что эти переменные доступны только как аргументы функции main(). Чтобы сделать аргументы командной строки переносимыми и доступными в других функциях, нужно либо передать argv таким функциям в качестве аргумента, либо определить глобальную переменную, указывающую на argv.

Существует два непереносимых способа получения доступа ко всей этой информации или к ее части из любого места программы.

- Аргументы командной строки любого процесса могут быть считаны через характерный для Linux файл /proc/*PID*/cmdline, в котором каждый аргумент завершается нулевым байтом. (Программа может получить доступ к аргументам своей собственной командной строки через файл /proc/self/cmdline.)
- GNU-библиотека C предоставляет две глобальные переменные, который могут применяться в любом месте программы с целью получения имени, использовавшегося для ее запуска (то есть первого аргумента командной строки). Первая из этих переменных, program_invocation_name, предоставляет полное путевое имя, использованное для запуска программы. Вторая переменная, program_invocation_short_name, обеспечивает версию этого имени без указания каких-либо каталогов (то есть базовую часть путевого имени). Объявления этих двух переменных могут быть получены из <errno.h> путем определения макроса _GNU_SOURCE.

Как показано на рис. 6.1, массивы argv и environ, а также строки, на которые они изначально указывают, находятся в одной непрерывной области памяти непосредственно над стеком процесса. (Массив environ, содержащий список переменных среды, будет рассмотрен в следующем разделе.) Предусмотрено верхнее ограничение общего количества байтов, сохраняемых в этой области. Согласно SUSv3 этот лимит можно определить через константу ARG_MAX (определенная в <limits.h>) или вызов sysconf(_SC_ARG_MAX). (Описание sysconf() дается в разделе 11.2.) В SUSv3 требуется, чтобы значение ARG_MAX было не меньше значения _POSIX_ARG_MAX, равного 4096 байт. Во многих реализациях UNIX допускается существенно более высокое ограничение. В SUSv3 не указано, входят ли в ограничение, определяемое ARG_MAX, служебные байты, характерные для реализации (завершающие нулевые байты, выравнивающие байты и массивы указателей argv и environ).

> В Linux исторически сложилось так, что под ARG_MAX выделяется 32 страницы (то есть 131 072 байта в Linux/x86-32), включая пространство для служебных байтов. Начиная с версии ядра 2.6.23, ограничением на общее пространство, используемым для argv и environ, можно управлять через ограничение ресурса RLIMIT_STACK, и для argv и environ допускается гораздо большее значение. Это ограничение вычисляется как одна четвертая нежесткого ограничения ресурса RLIMIT_STACK, имевшего место на время вызова execve(). Дополнительные подробности можно найти на странице руководства execve(2).

Многие программы (включая примеры, приведенные в этой книге) проводят анализ ключей командной строки (то есть аргументов, начинающихся с дефиса), используя для этого библиотечную функцию getopt().

6.7. Список переменных среды

У каждого процесса есть связанный с ним строковый массив, который называется *списком переменных среды (окружения)* или просто *средой (окружением)*. Каждая из его строк является определением вида *имя=значение*. Таким образом, среда представляет собой набор пар «имя — значение», которые могут применяться для хранения произвольной информации.

Когда создается новый процесс, он наследует копию среды своего родителя. Это простая, но часто используемая форма межпроцессного взаимодействия (IPC) — среда предоставляет способ переноса информации от родительского процесса его дочернему процессу или процессам. Поскольку дочерний процесс при создании получает копию среды своего родительского процесса, этот перенос информации является односторонним и однократным. После создания дочернего процесса любой процесс может изменить свою собственную среду, и эти изменения будут незаметны для других процессов.

Переменные среды часто используются оболочкой. Помещая значения в свою собственную среду, оболочка может обеспечить передачу этих значений процессам, создаваемым ею для выполнения пользовательских команд. Например, для переменной среды `SHELL` в качестве значения определяется путевое имя самой программы оболочки. Многие программы интерпретируют данную переменную в качестве имени оболочки, и она может быть задействована для вызова оболочки изнутри программы.

Некоторые библиотечные функции позволяют изменять свое поведение путем установки переменных среды. Это дает возможность пользователю управлять поведением приложения без изменения кода приложения или его перекомпоновки с использованием соответствующей библиотеки.

В большинстве оболочек значение может быть добавлено к среде с помощью команды:

```
export: $ SHELL=/bin/bash      Создание переменной оболочки
$ export SHELL                 Помещение переменной в среду процесса оболочки
```

В оболочках bash и Korn можно воспользоваться следующей сокращенной записью:

```
$ export SHELL=/bin/bash
```

В оболочке C shell применяется альтернативная команда `setenv`:

```
% setenv SHELL /bin/bash
```

Показанные выше команды навсегда добавляют значение к среде оболочки, после чего эта среда наследуется всеми создаваемыми оболочкой дочерними процессами. Переменная среды может быть в любой момент удалена командой `unset` (`unsetenv` в оболочке C shell).

В оболочке Bourne shell и ее потомках (например, bash и Korn) для добавления значений к среде, применяемой для выполнения одной программы без влияния на родительскую оболочку (и последующие команды), может использоваться такой синтаксис:

```
$ NAME=value program              /* ИМЯ=значение программа */
```

Определение будет добавлено к среде только того дочернего процесса, который выполняет указанную программу. Если нужно, то перед именем программы можно добавить сразу несколько присваиваний (отделенных друг от друга пробелами).

> Команда env запускает программу, используя измененную копию списка переменных среды оболочки. Список переменных среды может быть изменен как с добавлением, так и с удалением определений из списка, копируемого из оболочки. Более подробное описание можно найти на странице руководства env(1).

Текущий список переменных среды выводится на экран командой `printenv`. Вот как выглядит пример выводимой ею информации:

```
$ printenv
LOGNAME=mtk
SHELL=/bin/bash
HOME=/home/mtk
```

```
PATH=/usr/local/bin:/usr/bin:/bin:.
TERM=xterm
```

Назначение большинства перечисленных выше переменных среды будет рассмотрено в последующих главах (эти сведения также можно найти на странице руководства `environ(7)`).

Из показанного выше примера вывода видно, что список переменных среды неотсортирован. По сути, это не создает никакой проблемы, так как обычно нужно обращаться к отдельно взятым переменным среды, а не к какой-то упорядоченной их последовательности.

Список переменных среды любого процесса можно изучить, обратившись к характерному для Linux `/proc/`*`PID`*`/environ`, в котором каждая пара *`ИМЯ=значение`* заканчивается нулевым байтом.

Обращение к среде из программы

Из программы на языке C список переменных среды может быть доступен с помощью глобальной переменной `char **environ`. (Она определяется кодом инициализации среды выполнения программ на языке C, и ей присваивается значение, указывающее на местоположение списка переменных среды.) Как и `argv`, переменная `environ` указывает на список указателей на строки, заканчивающиеся нулевыми байтами, а сам этот список заканчивается значением `NULL`. Структура данных списка переменных среды в том же порядке, как их вывела выше команда `printenv`, показана на рис. 6.5.

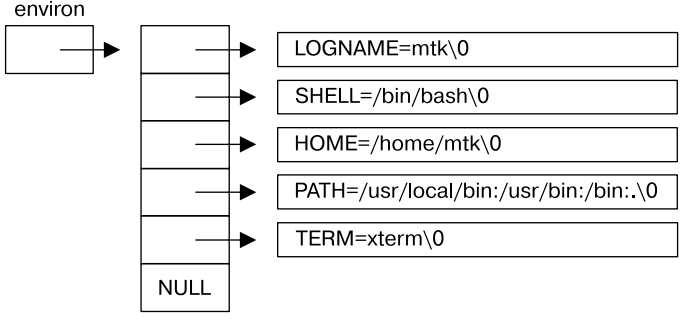


Рис. 6.5. Пример структуры данных в отношении списка переменных среды для процесса

Программа, показанная в листинге 6.3, обращается к `environ`, чтобы вывести список всех переменных, имеющихся в среде процесса.

Листинг 6.3. Вывод на экран переменных среды процесса

proc/display_env.c

```
#include "tlpi_hdr.h"

extern char **environ;
int
main(int argc, char *argv[])
{
    char **ep;
    for (ep = environ; *ep != NULL; ep++)
        puts(*ep);
    exit(EXIT_SUCCESS);
}
```

proc/display_env.c

Вывод программы совпадает с выводом команды `printenv`. Цикл в этой программе основан на использовании указателей для последовательного перебора содержимого массива `environ`. Даже если бы можно было рассматривать `environ` именно в качестве массива (как это делалось при использовании `argv` в листинге 6.2), это было бы менее естественно, поскольку элементы в списке переменных среды не располагаются в определенном порядке и нет переменной (соответствующей переменной `argc`), указывающей на размер списка переменных среды. (По той же причине элементы массива `environ` на рис. 6.5 не пронумерованы.)

Альтернативный метод обращения к списку переменных среды заключается в объявлении для функции `main()` третьего аргумента:

```
int main(int argc, char *argv[], char *envp[])
```

Этот аргумент может рассматриваться в том же качестве, что и `environ`, с той лишь разницей, что его область видимости является локальной для функции `main()`. Хотя это свойство широко реализовано в системах UNIX, его использования следует избегать, поскольку, вдобавок к ограничениям по области видимости, оно не указано в спецификации SUSv3.

Отдельные значения из среды процесса извлекаются с помощью функции `getenv()`.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Возвращает указатель на строку (значение) или `NULL`, если такой переменной не существует

Получив имя переменной среды, функция `getenv()` возвращает указатель на соответствующее строковое значение. Если в ранее рассмотренном примере среды в качестве аргумента `name` указать `SHELL`, будет возвращена строка `/bin/bash`. Если переменной среды с таким именем не существует, `getenv()` возвращает `NULL`.

При использовании `getenv()` нужно учитывать следующие условия обеспечения портируемости.

- В SUSv3 требуется, чтобы приложение не изменяло строку, возвращенную `getenv()`. Дело в том, что в большинстве реализаций она является фактически частью среды (то есть строка, предоставляющая в паре *`ИМЯ=значение`* ту часть, которая является значением). Если нужно изменить значение переменной среды, можно воспользоваться одной из рассматриваемых далее функций: либо `setenv()`, либо `putenv()`.
- В SUSv3 разрешается реализация функции `getenv()` для возвращения ее результата с использованием статически выделенного буфера, который может быть перезаписан последующими вызовами `getenv()`, `setenv()`, `putenv()` или `unsetenv()`. Хотя в glibc-реализации `getenv()` статический буфер таким образом не применяется, портируемая программа, которой нужно сохранить строку, возвращенную вызовом `getenv()`, прежде чем вызвать одну из этих функций, должна скопировать эту строку в другое место.

Изменение среды

Иногда процессу есть смысл изменить свою среду. Одна из причин такого изменения состоит в том, что факт изменения среды будет виден дочерним процессам, создаваемым им впоследствии.

Возможно также, что нужно определить переменную, которая станет видна новой, исполняемой с помощью функции `exec()` программе, которая будет загружена в память

этого процесса. В этом смысле среда является формой не только межпроцессного, но и межпрограммного взаимодействия. (Более подробно этот метод будет рассмотрен в главе 27, где объясняется порядок разрешения программе с помощью функций `exec()` заменять саму себя новой программой в том же процессе.)

Функция `putenv()` добавляет новую переменную к среде вызывающего ее процесса или изменяет значение существующей переменной.

```
#include <stdlib.h>

int putenv(char *string);
```

Возвращает `0` при успешном завершении или ненулевое значение при ошибке

Аргумент `string` является указателем на строку вида *`ИМЯ=значение`*. После вызова `putenv()` эта строка становится частью среды. Иначе говоря, строка, на которую указывает `string`, не будет скопирована в среду, а, наоборот, один из элементов среды будет указывать на то же самое место, что и `string`. Следовательно, если в дальнейшем изменить байты, на которые указывает `string`, такое изменение повлияет на среду процесса. Поэтому `string` не должна быть автоматически создаваемой переменной (то есть символьным массивом, размещаемым в стеке), поскольку эта область памяти может быть перезаписана после возвращения из функции, в которой определена переменная.

Обратите внимание на то, что `putenv()` возвращает при ошибке не `-1`, а ненулевое значение.

В glibc-реализации функции `putenv()` предоставляется нестандартное расширение. Если в строке, на которую указывает аргумент `string`, нет знака равенства (`=`), то переменная среды, идентифицируемая аргументом `string`, удаляется из списка переменных среды.

Функция `setenv()` является альтернативой `putenv()`, предназначенной для добавления переменной к среде.

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `setenv()` создает новую переменную среды путем выделения буфера памяти для строки вида *`ИМЯ=значение`* и копирует в этот буфер строки, указываемые аргументами `name` и `value`. Заметьте, что мы ни в коем случае не должны ставить знак равенства после `name` или в начале `value`, поскольку `setenv()` дописывает этот символ при добавлении нового определения к среде.

Функция `setenv()` не изменяет среду, если переменная, идентифицируемая аргументом `name`, уже существует, а аргумент `overwrite` имеет значение `0`. Если у `overwrite` ненулевое значение, среда всегда изменяется.

Тот факт, что `setenv()` копирует свои аргументы, означает, что в отличие от `putenv()` мы можем впоследствии изменить содержимое строк, на которые указывают аргументы `name` и `value`, не оказывая влияния на среду. Это также означает, что использование автоматически создаваемых переменных в качестве аргументов `setenv()` не создает никаких проблем.

Функция `unsetenv()` удаляет из среды переменную, идентифицируемую аргументом `name`.

```
#include <stdlib.h>

int unsetenv(const char *name);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Как и для `setenv()`, аргумент `name` не должен включать в себя знак равенства.

И `setenv()` и `unsetenv()` берут начало из BSD и не так популярны, как `putenv()`. Хотя в исходном стандарте POSIX.1 или в SUSv2 они не упоминались, их включили в SUSv3.

В версиях glibc, предшествующих 2.2.2, функция unsetenv() имела прототип, возвращающий void. Именно такой прототип unsetenv() был в исходной реализации BSD, и некоторые реализации UNIX до сих пор следуют этому BSD-прототипу.

Временами требуется удалить целиком всю среду, а затем выстроить ее заново с заданными значениями. Это, к примеру, может понадобиться для безопасного выполнения программ с установлением идентификатором пользователя (set-user-ID) (см. раздел 38.8). Среду можно удалить, присвоив переменной `environ` значение `NULL`:

```
environ = NULL;
```

Именно такое действие и предпринимается в библиотечной функции `clearenv()`.

```
#define _BSD_SOURCE           /* Или: #define _SVID_SOURCE */
#include <stdlib.h>

int clearenv(void)
```

Возвращает `0` при успешном завершении или ненулевое значение при ошибке

В некоторых обстоятельствах использование функций `setenv()` и `clearenv()` может привести к утечкам памяти в программе. Как уже говорилось, функция `setenv()` выделяет буфер памяти, который затем составляет часть среды. Когда вызывается функция `clearenv()`, она не высвобождает данный буфер (это невозможно, поскольку ей ничего не известно о его существовании). Программа, неоднократно использующая эти две функции, будет постоянно допускать утечку памяти. С практической точки зрения это вряд ли станет проблемой, поскольку обычно программа вызывает `clearenv()` только один раз в начале своего выполнения, чтобы удалить из среды все записи, унаследованные от своего предка (то есть от программы, вызвавшей `exec()` для ее запуска).

Функция clearenv() предоставляется во многих реализациях UNIX, но в SUSv3 она не определена. В SUSv3 определено, что, если приложение напрямую изменяет среду, как это делается функцией clearenv(), то поведение функций setenv(), unsetenv() и getenv() становится неопределенным. (Обоснование следующее: если запретить соответствующему приложению непосредственно изменять среду, то реализация ядра сможет полностью контролировать структуры данных, которые применяются ею для создания переменных среды.) Единственный способ очистки среды, разрешенный в SUSv3 приложению, заключается в получении списка всех

переменных среды (путем извлечения их имен из environ), с последующим использованием функции unsetenv() для поименного удаления каждой переменной.

Пример программы

Использование всех ранее рассмотренных в этом разделе функций продемонстрировано в листинге 6.4. После начальной очистки среды программа добавляет любые определения среды, предоставленные в виде аргументов командной строки. Затем она добавляет определение для переменной GREET, если таковой еще не имеется в среде, удаляет определение для переменной BYE и, наконец, выводит на экран текущий список переменных среды. Вот как выглядит вывод этой программы:

```
$ ./modify_env "GREET=Guten Tag" SHELL=/bin/bash BYE=Ciao
GREET=Guten Tag
SHELL=/bin/bash
$ ./modify_env SHELL=/bin/sh BYE=byebye
SHELL=/bin/sh
GREET=Hello world
```

Если присвоить переменной environ значение NULL (как это делается при вызове clearenv() в листинге 6.4), то мы вправе ожидать, что следующий цикл (в том виде, в котором он используется в программе) даст сбой, поскольку запись *environ будет некорректна:

```
for (ep = environ; *ep != NULL; ep++)
    puts(*ep);
```

Но если функции setenv() и putenv() определят, что environ имеет значение NULL, они создадут новый список переменных среды и установят для environ значение, указывающее на этот список. Это приведет к тому, что описанный выше цикл станет работать правильно.

Листинг 6.4. Изменение среды процесса

proc/modify_env.c

```
#define _GNU_SOURCE   /* Для получения различных объявлений из <stdlib.h> */
#include <stdlib.h>
#include "tlpi_hdr.h"

extern char **environ;

int
main(int argc, char *argv[])
{
    int j;
    char **ep;

    clearenv();       /* Удаление всей среды */

    for (j = 1; j < argc; j++)
        if (putenv(argv[j]) != 0)
            errExit("putenv: %s", argv[j]);
    if (setenv("GREET", "Hello world", 0) == -1)
        errExit("setenv");
    unsetenv("BYE");
    for (ep = environ; *ep != NULL; ep++)
        puts(*ep);
    exit(EXIT_SUCCESS);
}
```

proc/modify_env.c

6.8. Выполнение нелокального перехода: setjmp() и longjmp()

Библиотечные функции `setjmp()` и `longjmp()` используются для нелокального перехода. Термин «нелокальный» обозначает, что цель перехода находится где-то за пределами той функции, которая выполняется в данный момент.

Как и многие другие языки программирования, язык C включает в себя инструкцию `goto`. Злоупотребление ею затрудняет чтение программы и ее сопровождение. Однако временами это весьма полезная инструкция в плане упрощения программы, ускорения ее работы или достижения обоих результатов.

Одно из ограничений имеющейся в языке C инструкции `goto` заключается в том, что она не позволяет осуществить переход из текущей функции в другую функцию. Но такая функциональная возможность временами может оказаться весьма полезной.

Рассмотрим следующий довольно распространенный сценарий, относящийся к обработке ошибки. В ходе глубоко вложенного вызова функции произошла ошибка, которая должна быть обработана за счет отказа от выполнения текущей задачи, возвращения сквозь несколько вызовов функций и последующего выполнения обработки в какой-нибудь функции более высокого уровня (возможно, даже `main()`). Добиться этого можно, если каждая функция станет возвращать код завершения, который будет проверяться и соответствующим образом обрабатываться вызывавшим ее кодом. Это вполне допустимый и во многих случаях желательный метод обработки такого рода развития событий. Но программирование давалось бы проще, если бы можно было перейти из середины вложенного вызова функции назад к одной из вызвавших ее функций (к непосредственно вызвавшей ее функции или к той функции, что вызвала эту функцию, и т. д.). Именно такая возможность и предоставляется функциями `setjmp()` и `longjmp()`.

> Ограничение, согласно которому goto не может использоваться для перехода между функциями, накладывается в языке C из-за того, что все функции в C находятся на одном и том же уровне области видимости (то есть стандарт языка C не предусматривает вложенных объявлений функций, хотя gcc допускает такую возможность в качестве расширения). Следовательно, если взять две функции, X и Y, то у компилятора не будет возможности узнать, может ли стековый фрейм для функции X быть в стеке ко времени вызова функции Y, а стало быть, возможен ли переход из функции Y в функцию X.
>
> В таких языках, как Pascal, где объявления функций могут быть вложенными и переход из вложенной функции на уровень выше разрешен, статическая область видимости функции позволяет компилятору определить информацию о динамической области видимости этой функции. Таким образом, если функция Y лексически вложена в функцию X, то компилятор знает, что стековый фрейм для X должен уже быть в стеке ко времени вызова функции Y, и может сгенерировать код для перехода из функции Y в какое-либо место внутри функции X.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
                                    Возвращает 0 при первом вызове, ненулевое значение
                                                при возвращении через longjmp()
void longjmp(jmp_buf env, int val);
```

Вызов `setjmp()` устанавливает цель для последующего перехода, выполняемого функцией `longjmp()`. Этой целью является та самая точка в программе, откуда вызывается функция `setjmp()`. С точки зрения программирования после `longjmp()` это выглядит абсолютно так же, как будто мы только что вернулись из вызова `setjmp()` во второй раз. Способ, позволяющий отличить второе «возвращение» от первого, основан на целочисленном значении, возвращаемом функцией `setjmp()`. При первом вызове `setjmp()` возвращает `0`, а при последующем, фиктивном возвращении предоставляется то значение, которое указано в аргументе `val` при вызове функции `longjmp()`. Путем использования для аргумента `val` различных значений можно отличать друг от друга переходы к одной и той же цели из различных мест программы.

Если бесконтрольно указать для функции `longjmp()` аргумент `val`, равный нулю, то это вызовет фиктивное возвращение из `setjmp()`, которое будет выглядеть, как будто это первое возвращение. По этой причине, если для `val` указано значение `0`, `longjmp()` фактически применяет значение `1`.

Используемый обеими функциями аргумент `env` предоставляет связующий элемент, позволяющий осуществить переход. При вызове `setjmp()` в `env` сохраняется различная информация о среде текущего процесса. Это позволяет выполнить вызов `longjmp()`, которому для осуществления фиктивного возвращения нужно указать ту же переменную `env`. Поскольку вызовы `setjmp()` и `longjmp()` — различные функции (в противном случае мы могли бы обойтись и простой инструкцией `goto`), `env` объявляется глобально или, что менее распространено, передается в качестве аргумента функции.

На момент вызова `setjmp()` в `env` наряду с другой информацией хранятся копии регистра *счетчика команд* (он указывает на тот машинный код, который выполняется в данный момент) и регистра *указателя стека* (где отмечается вершина стека). Эта информация позволяет осуществить последующий вызов `longjmp()`, чтобы выполнить два основных действия.

- Удалить из стека стековые фреймы для всех промежуточных функций между функцией, вызвавшей `longjmp()`, и функцией, которая перед этим вызвала `setjmp()`. Эту процедуру иногда называют «раскруткой стека», и она выполняется путем сброса регистра указателя стека и присвоением ему значения, сохраненного в аргументе `env`.
- Установить такое значение регистра, чтобы выполнение программы продолжалось с места предварительного вызова `setjmp()`. Это действие также выполняется с использованием значения, сохраненного в `env`.

Пример программы

Применение функций `setjmp()` и `longjmp()` показано в листинге 6.5. Эта программа с помощью предварительного вызова `setjmp()` устанавливает цель перехода. Дальнейшее использование инструкции `switch` (на основе значения, возвращаемого `setjmp()`) позволяет различить первоначальный возврат из функции `setjmp()` и возврат после `longjmp()`. Если возвращаемое значение равно `0`, значит, только что был сделан первоначальный вызов `setjmp()` — мы вызываем функцию `f1()`, которая либо сразу же вызывает `longjmp()`, либо переходит к вызову `f2()`, в зависимости от значения `argc` (то есть количества аргументов командной строки). Если управление перешло в `f2()`, в ней тут же происходит вызов `longjmp()`. Вызов функции `longjmp()` из любой функции возвращает нас назад, к тому месту, из которого была вызвана `setjmp()`. В двух вызовах `longjmp()` используются разные аргументы `val`, поэтому инструкция `switch` в `main()` может определить функцию, из которой произошел переход, и вывести на экран соответствующее сообщение.

При запуске программы из листинга 6.5 без каких-либо аргументов командной строки мы увидим следующее:

```
$ ./longjmp
Вызов f1() после предварительного вызова setjmp()
Мы вернулись назад из f1()
```

Указание аргумента командной строки приводит к переходу, осуществляемому из `f2()`:

```
$ ./longjmp x
Вызов f1() после предварительного вызова setjmp()
Мы вернулись назад из f2()
```

Листинг 6.5. Демонстрирует использование вызовов setjmp() и longjmp()

proc/longjmp.c

```
#include <setjmp.h>
#include "tlpi_hdr.h"

static jmp_buf env;
static void
f2(void)
{
    longjmp(env, 2);
}

static void
f1(int argc)
{
    if (argc == 1)
        longjmp(env, 1);
    f2();
}

int
main(int argc, char *argv[])
{
    switch (setjmp(env)) {
    case 0:           /* Это возвращение после предварительного вызова setjmp() */
        printf("Calling f1() after initial setjmp()\n");
        f1(argc);     /* Данная программа никогда не выполнит
                         break из следующей строки,... */
        break;        /* ... но хороший тон обязывает нас его написать.*/

    case 1:
        printf("We jumped back from f1()\n");
        break;

    case 2:
        printf("We jumped back from f2()\n");
        break;
    }

    exit(EXIT_SUCCESS);
}
```

proc/longjmp.c

Ограничения, накладываемые на использование setjmp()

В SUSv3 и C99 указывается, что вызов `setjmp()` может присутствовать только в следующих контекстах:

- в качестве цельного управляющего выражения инструкции выбора или итерации (`if`, `switch`, `while` и т. д.);

- в качестве операнда унарного оператора ! (НЕ), где получающееся в результате выражение является цельным управляющим выражением инструкции выбора или итерации;
- в качестве части операции сравнения (==, !=, < и т. д.), где другой операнд является выражением целочисленной константы и получающееся в результате выражение является цельным управляющим выражением инструкции выбора или итерации;
- в качестве обособленного вызова функции, не встроенного в какое-либо более сложное выражение.

Обратите внимание, что в приведенном выше списке отсутствует инструкция присваивания языка С. Инструкция, выраженная в следующей форме, не соответствует стандарту:

```
s = setjmp(env);                    /* НЕВЕРНО! */
```

Эти ограничения наложены из-за того, что реализация `setjmp()` в качестве обычной функции не может гарантировать достаточный объем информации, позволяющий сохранять значения всех регистров и промежуточных мест в стеке, используемых в охватывающем выражении, чтобы затем они могли быть правильно восстановлены после вызова `longjmp()`. Поэтому вызов `setjmp()` допускается только внутри достаточно простых выражений, не требующих промежуточных мест хранения данных.

Неверное применение longjmp()

Если буфер *env* объявлен глобальным для всех функций (что обычно и делается), это допускает выполнение следующей последовательности.

1. Вызов функции `x()`, использующей `setjmp()` для установки цели перехода в глобальной переменной `env`.
2. Возвращение из функции `x()`.
3. Вызов функции `y()`, выполняющей `longjmp()` с использованием *env*.

Это серьезная ошибка. Нельзя выполнять переход с помощью функции `longjmp()` в функцию, из которой уже произошел возврат. Что `longjmp()` пытается сделать со стеком? Она пытается раскрутить назад к фрейму, которого уже нет, что приводит в результате к хаосу. Если повезет, наша программа просто даст сбой. Но в зависимости от состояния стека могут возникать бесконечные циклы вызова-возврата, и программа начнет вести себя так, будто она на самом деле вернула управление из функции, которая на тот момент не выполнялась. (В многопоточной программе подобное неверное действие заключается в вызове `longjmp()` в другом потоке, отличающемся от того, где был осуществлен вызов `setjmp()`.)

> В SUSv3 говорится, что, если longjmp() вызывается из вложенного обработчика сигнала (то есть из обработчика, который вызывается при выполнении другого обработчика сигнала), поведение программы становится неопределенным.

Проблемы, связанные с оптимизирующими компиляторами

Оптимизирующие компиляторы могут переопределить порядок следования инструкций в программе и сохранить конкретные переменные в регистрах центрального процессора, а не в оперативной памяти. Обычно такая оптимизация зависит от потока управления ходом выполнения программы, отражающего лексическую структуру программы. Поскольку операции перехода, выполняемые с помощью вызовов функций `setjmp()` и `longjmp()`, создаются и происходят в ходе выполнения программы, оптимизатор компилятора не может взять их в расчет в процессе своей работы. Более того, семантика некоторых реализаций двоичных интерфейсов приложений (ABI) требует, чтобы функция `longjmp()`

восстанавливала копии регистров центрального процессора, сохраненные ранее при вызове функции `setjmp()`.

Это говорит о том, что в результате вызова `longjmp()` в оптимизированных переменных могут оказаться неверные значения. Убедиться в этом можно, изучив поведение программы, представленной в листинге 6.6.

Листинг 6.6. Демонстрация взаимного влияния оптимизации при компиляции и функции longjmp()

proc/setjmp_vars.c

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

static jmp_buf env;
static void
doJump(int nvar, int rvar, int vvar)
{
    printf("Inside doJump(): nvar=%d rvar=%d vvar=%d\n", nvar, rvar, vvar);
    longjmp(env, 1);
}

int
main(int argc, char *argv[])
{
    int nvar;
    register int rvar;          /* По возможности выделяется в регистре */
    volatile int vvar;          /* Смотрите текст */

    nvar = 111;
    rvar = 222;
    vvar = 333;

    if (setjmp(env) == 0) {     /* Код, выполняемый после setjmp() */
        nvar = 777;
        rvar = 888;
        vvar = 999;
        doJump(nvar, rvar, vvar);
    } else {                    /* Код, выполняемый после longjmp() */
        printf("After longjmp(): nvar=%d rvar=%d vvar=%d\n", nvar, rvar, vvar);
    }

    exit(EXIT_SUCCESS);
}
```

proc/setjmp_vars.c

При компиляции без оптимизации программы, представленной в листинге 6.6, мы увидим на выходе вполне ожидаемую информацию:

```
$ cc -o setjmp_vars setjmp_vars.c
$ ./setjmp_vars
Inside doJump(): nvar=777 rvar=888 vvar=999
After longjmp(): nvar=777 rvar=888 vvar=999
```

Но при компиляции с оптимизацией будут получены такие неожиданные результаты:

```
$ cc -O -o setjmp_vars setjmp_vars.c
$ ./setjmp_vars
Inside doJump(): nvar=777 rvar=888 vvar=999
After longjmp(): nvar=111 rvar=222 vvar=999
```

Здесь видно, что после вызова `longjmp()` переменные `nvar` и `rvar` были переопределены, получив значения, имевшиеся у них ко времени вызова функции `setjmp()`. Это произошло потому, что вследствие вызова `longjmp()` реорганизация оптимизатором кода привела к путанице. Эта проблема может коснуться любых локальных переменных, являющихся кандидатами на оптимизацию. Как правило, она касается переменных-указателей и переменных любого простого типа: `char`, `int`, `float` и `long`.

Подобной реорганизации кода можно избежать, объявив переменные изменяемыми — `volatile`, что даст указание оптимизатору не оптимизировать их. В предыдущем выводе информации из программы было показано, что переменная `vvar`, объявленная `volatile`, была правильно обработана даже при компиляции с оптимизацией.

Поскольку различные компиляторы используют различные приемы оптимизации, в портируемых программах в тех функциях, которые вызывают `setjmp()`, ключевое слово `volatile` должно указываться со всеми локальными переменными вышеупомянутых типов.

Если компилятору GNU C задать ключ `-Wextra` (extra warnings — «дополнительные предупреждения»), то в отношении программы `setjmp_vars.c` он выдаст следующие полезные предупреждения:

```
$ cc -Wall -Wextra -O -o setjmp_vars setjmp_vars.c
setjmp_vars.c: In function 'main':
setjmp_vars.c:17: warning: variable 'nvar' might be clobbered
 by 'longjmp' or 'vfork'
(Переменная nvar может быть «затерта» функцией longjmp или vfork.)
setjmp_vars.c:18: warning: variable 'rvar' might be clobbered
 by 'longjmp' or 'vfork'
(Переменная rvar может быть «затерта» функцией longjmp или vfork.)
```

> Поучительно будет взглянуть на ассемблерный выход, создаваемый при компиляции программы setjmp_vars.c как с оптимизацией, так и без нее. Команда cc -S создает файл с расширением .s, где содержится сгенерированный для программы ассемблерный код.

Использовать ли функции setjmp() и longjmp()

Выше говорилось, что инструкции переходов `goto` могут создавать трудности при чтении программы. В свою очередь, нелокальные переходы могут на порядок затруднить чтение, поскольку способны передавать управление между двумя любыми функциями программы. Таким образом, использование функций `setjmp()` и `longjmp()` должно стать редким исключением. Лучше потратить дополнительные усилия в проектировании и написании кода, чтобы получить программу, в которой удастся обойтись без этих функций, и в результате она станет легче читаемой и, возможно, более портируемой. Мы еще будем рассматривать варианты этих функций (`sigsetjmp()` и `siglongjmp()`, описание которых дается в подразделе 21.2.1) при изучении сигналов, поскольку их иногда полезно применять при написании обработчиков сигналов.

6.9. Резюме

У каждого процесса есть свой уникальный идентификатор, и он содержит запись идентификатора своего родительского процесса.

Виртуальная память процесса логически разделена на несколько сегментов: текстовый, сегмент данных (инициализированных и неинициализированных), стека и кучи.

Стек состоит из последовательности фреймов, при этом новый фрейм добавляется при вызове функции и удаляется при возвращении из этой функции. Каждый фрейм

содержит локальные переменные, аргументы функций и информацию, связанную с вызовом для отдельно взятого вызова функции.

Аргументы командной строки, предоставляемые при запуске программы, становятся доступны через аргументы `argc` и `argv` функции `main()`. По соглашению в `argv[0]` содержится имя, использованное для вызова программы.

Каждый процесс получает копию списка переменных среды своего родительского процесса, представляющего собой набор из пар «имя-значение». Доступ процесса к переменным в его списке переменных среды и возможность их изменения предоставляется через глобальную переменную `environ` и посредством различных библиотечных функций.

Функции `setjmp()` и `longjmp()` предлагают способ выполнения нелокальных переходов из одной функции в другую (с раскруткой стека). Чтобы избежать проблем с оптимизацией в ходе компиляции, при использовании этих функций может понадобиться объявлять переменные с модификатором `volatile`. Нелокальные переходы могут отрицательно сказаться на читаемости программы и затруднить ее сопровождение, поэтому по возможности их нужно избегать.

Дополнительная информация

Подробное описание системы управления виртуальной памятью можно найти в изданиях [Tanenbaum, 2007] и [Vahalia, 1996]. Алгоритмы управления памятью, используемые в ядре Linux, и соответствующий им код подробно рассмотрены в книге [Gorman, 2004].

6.10. Упражнения

6.1. Скомпилируйте программу из листинга 6.1 (`mem_segments.c`) и выведите на экран ее размер, воспользовавшись командой `ls -l`. Хотя программа содержит массив (`mbuf`), размер которого приблизительно составляет 10 Мбайт, размер исполняемого файла существенно меньше. Почему?

6.2. Напишите программу, чтобы посмотреть, что случится, если попытаться осуществить переход с помощью функции `longjmp()` в функцию, возвращение из которой уже произошло.

6.3. Реализуйте функции `setenv()` и `unsetenv()`, используя функции `getenv()`, `putenv()` и там, где это необходимо, код, который изменяет массив `environ` напрямую. Ваша версия функции `unsetenv()` должна проверять наличие нескольких определений переменной среды и удалять все определения (точно так же, как это делает glibc-версия функции `unsetenv()`).

7 Выделение памяти

Почти все системные программы должны обладать возможностью выделения дополнительной памяти для динамических структур данных. Например, такая память нужна для работы связанных списков и двоичных деревьев, чей размер зависит от информации, доступной только в ходе выполнения программы. В этой главе рассматриваются функции, используемые для выделения памяти в куче или стеке.

7.1. Выделение памяти в куче

Процесс может выделить память, увеличив размер кучи (сегмента непрерывной виртуальной памяти переменного размера), который начинается сразу же после сегмента неинициализированных данных процесса и увеличивается/уменьшается по мере выделения/высвобождения памяти (см. рис. 6.1). Текущее ограничение кучи называется *крайней точкой программы* (program break).

Для выделения памяти в программах на языке C обычно используется семейство функций `malloc`, которое мы вскоре рассмотрим. Но сначала разберем функции `brk()` и `sbrk()`, на применении которых основана работа функций `malloc`.

7.1.1. Установка крайней точки программы: brk() и sbrk()

Изменение размеров кучи (то есть выделение и высвобождение памяти) сводится лишь к тому, чтобы всего лишь объяснить ядру, где располагается крайняя точка программы (program break). Изначально крайняя точка программы находится непосредственно сразу же за окончанием сегмента неинициализированных данных (то есть там же, где на рис. 6.1 стоит метка `&end`). После того как эта точка будет сдвинута вверх, программа сможет получать доступ к любому адресу во вновь выделенной области, но страницы физической памяти пока выделяться не будут. Ядро автоматически выделит новые физические страницы при первой же попытке процесса обратиться к адресам этих страниц.

Традиционно для манипуляций с крайней точкой программы система UNIX предоставляла два системных вызова, и они оба доступны в Linux: `brk()` и `sbrk()`. Хотя в программах эти системные вызовы напрямую используются довольно редко, в их работе стоит разобраться, чтобы выяснить порядок выделения памяти.

```
#include <unistd.h>

int brk(void *end_data_segment);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

```
void *sbrk(intptr_t increment);
```

Возвращает предыдущую крайнюю точку программы при успешном завершении или `(void *) -1` при ошибке

Системный вызов `brk()` устанавливает крайнюю точку программы на место, указанное окончанием сегмента данных — `end_data_segment`. Поскольку виртуальная память выделяется постранично, `end_data_segment` фактически округляется до границы следующей страницы.

Попытки установить крайнюю точку программы ниже ее первоначального значения, (то есть ниже метки `&end`), скорее всего, приведут к неожиданному поведению, например к сбою сегментирования (сигнал `SIGSEGV` рассматривается в разделе 20.2) при обращении к данным в уже не существующих частях сегментов инициализированных или неинициализированных данных. Точный верхний предел возможной установки крайней точки программы зависит от нескольких факторов, в числе которых: ограничение ресурсов процесса для размера сегмента данных (`RLIMIT_DATA`, рассматриваемое в разделе 36.3), а также расположение отображений памяти, сегментов совместно используемой памяти и совместно используемых библиотек.

Вызов `sbrk()` приводит к изменению положения точки программы путем добавления к ней приращения `increment`. (В Linux функция `sbrk()` является библиотечной и реализована в виде надстройки над функцией `brk()`.) Используемый для описания приращения `increment` тип `intptr_t` является целочисленным типом данных. В случае успеха функция `sbrk()` возвращает предыдущий адрес крайней точки программы. Иными словами, если мы подняли крайнюю точку программы, то возвращаемым значением будет указатель на начало только что выделенного блока памяти.

Вызов `sbrk(0)` возвращает текущее значение установки крайней точки программы без ее изменения. Этот вызов может пригодиться, если нужно отследить размер кучи, возможно, чтобы изучить поведение пакета средств выделения памяти.

> В SUSv2 имеются описания brk() и sbrk() (с пометкой LEGACY, то есть устаревшие). Из SUSv3 эти описания удалены.

7.1.2. Выделение памяти в куче: malloc() и free()

Обычно в программах на языке C для выделения памяти в куче и ее высвобождения используется семейство функций `malloc`. Эти функции по сравнению с `brk()` и `sbrk()` предоставляют несколько преимуществ. В частности, они:

- стандартизированы в качестве части языка C;
- проще в использовании в программах, выполняемых в нескольких потоках;
- предоставляют простой интерфейс, позволяющий выделять память небольшими блоками;
- позволяют произвольно высвобождать блоки памяти, сохраняемые в списке свободных блоков и заново возвращаемые в оборот при последующих вызовах выделения памяти.

Функция `malloc()` выделяет из кучи `size` байтов и возвращает указатель на начало только что выделенного блока памяти. Выделенная память не инициализируется.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Возвращает при успешном завершении указатель на выделенную память или `NULL` при ошибке

Поскольку функция `malloc()` возвращает тип `void *`, ее можно присваивать любому типу указателя языка C. Блок памяти, возвращенный `malloc()`, всегда выравнивается по байтовой границе, обеспечивающей эффективное обращение к данным любого типа языка C.

На практике это означает, что выделение на большинстве архитектур происходит по 8- или 16-байтовой границе.

> В SUSv3 определяется, что вызов malloc(0) может возвращать либо NULL, либо указатель на небольшой фрагмент памяти, который может (и должен быть) высвобожден с помощью функции free(). В Linux вызов malloc(0) придерживается второго варианта поведения.

Если память не может быть выделена (например, по причине достижения того предела, до которого может быть поднята крайняя точка программы), функция `malloc()` возвращает `NULL` и устанавливает для `errno` значение, указывающее на характер ошибки. Хотя сбой при выделении памяти случается редко, все вызовы `malloc()`, и родственных функций, которые будут рассмотрены далее, должны проверяться на отсутствие этой ошибки.

Функция `free()` высвобождает блок памяти, указанный в ее аргументе `ptr`, который должен быть адресом, ранее возвращенным функцией `malloc()` или одной из других функций выделения памяти в куче, которые будут рассмотрены далее в этой главе.

```
#include <stdlib.h>

void free(void *ptr);
```

Фактически функция `free()` не сдвигает вниз крайнюю точку программы, а вместо этого добавляет блок памяти к списку свободных блоков, которые будут снова использованы при дальнейших вызовах функции `malloc()`. Это делается по следующим причинам.

- Высвобождаемый блок памяти обычно находится где-нибудь в середине кучи, а не в ее конце, поэтому сдвинуть вниз крайнюю точку программы не представляется возможным.
- Это помогает свести к минимуму количество системных вызовов `sbrk()`, используемых программой. (Как уже отмечалось в разделе 3.1, системные вызовы приводят к небольшим, но все же существенным издержкам.)
- Во многих случаях сдвиг вниз крайней точки программы не поможет программам, выделяющим большие объемы памяти, поскольку они обычно имеют склонность удерживать выделенную память или многократно высвобождать и заново выделять память, а не высвобождать всю ее целиком, и после этого продолжать свое выполнение в течение длительного периода времени.

Если аргумент, предоставляемый функции `free()`, является `NULL`-указателем, то при ее вызове ничего не происходит. (Иными словами, предоставление функции `free()` `NULL`-указателя не будет ошибкой.)

Какое-либо использование аргумента `ptr` после вызова `free()`, например повторная передача значения этого аргумента функции `free()`, является ошибкой, которая может привести к непредсказуемым результатам.

Пример программы

Программа в листинге 7.1 может использоваться для иллюстрации того, как вызов функции `free()` влияет на крайнюю точку программы. Эта программа выделяет несколько блоков памяти, а затем высвобождает некоторые из них или все блоки, в зависимости от применения необязательных аргументов командной строки.

Первые два аргумента командной строки указывают количество и размер выделяемых блоков. Третий аргумент командной строки указывает шаг цикла, используемый при высвобождении блоков памяти. Если здесь указать `1` (это значение используется по

умолчанию, если аргумент опущен), программа высвобождает все блоки памяти. Если указать 2, высвобождается каждый второй выделенный блок и т. д. Четвертый и пятый аргументы командной строки указывают диапазон блоков, намеченных к высвобождению. Если эти аргументы опущены, высвобождаются все выделенные блоки (с шагом, заданным в третьем аргументе командной строки).

Листинг 7.1. Демонстрация происходящего с крайней точкой программы при высвобождении памяти
memalloc/free_and_sbrk.c

```
#define _BSD_SOURCE
#include "tlpi_hdr.h"
#define MAX_ALLOCS 1000000

int
main(int argc, char *argv[])
{
    char *ptr[MAX_ALLOCS];
    int freeStep, freeMin, freeMax, blockSize, numAllocs, j;

    printf("\n");
    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-allocs block-size [step [min [max]]]\n", argv[0]);
    numAllocs = getInt(argv[1], GN_GT_0, "num-allocs");
    if (numAllocs > MAX_ALLOCS)
        cmdLineErr("num-allocs > %d\n", MAX_ALLOCS);
    blockSize = getInt(argv[2], GN_GT_0 | GN_ANY_BASE, "block-size");
    freeStep = (argc > 3) ? getInt(argv[3], GN_GT_0, "step") : 1;
    freeMin =  (argc > 4) ? getInt(argv[4], GN_GT_0, "min") : 1;
    freeMax =  (argc > 5) ? getInt(argv[5], GN_GT_0, "max") : numAllocs;

    if (freeMax > numAllocs)
        cmdLineErr("free-max > num-allocs\n");
    printf("Initial program break:         %10p\n", sbrk(0));
    printf("Allocating %d*%d bytes\n", numAllocs, blockSize
    for (j = 0; j < numAllocs; j++) {
        ptr[j] = malloc(blockSize);
        if (ptr[j] == NULL)
            errExit("malloc");
    }
    printf("Program break is now:          %10p\n", sbrk(0));
    printf("Freeing blocks from %d to %d in steps of %d\n",
                freeMin, freeMax, freeStep);
    for (j = freeMin - 1; j < freeMax; j += freeStep)
        free(ptr[j]);
    printf("After free(), program break is: %10p\n", sbrk(0));

    exit(EXIT_SUCCESS);
}
```

memalloc/free_and_sbrk.c

Запуск программы из листинга 7.1 со следующей командной строкой приведет к выделению 1000 блоков памяти, а затем к высвобождению каждого второго блока:

```
$ ./free_and_sbrk 1000 10240 2
```

Информация, выведенная на экран, показывает, что после высвобождения этих блоков крайняя точка программы осталась на том же уровне, который был достигнут после выделения всех блоков памяти:

```
Initial program break:                          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:                           0x8a13000
Freeing blocks from 1 to 1000 in steps of 2
After free(), program break is:                 0x8a13000
```

В следующей командной строке указывается, что должны быть высвобождены все выделенные блоки, кроме последнего. В данном случае крайняя точка программы также остается на своей отметке «наивысшего поднятия уровня».

```
$ ./free_and_sbrk 1000 10240 1 1 999
Initial program break:                          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:                           0x8a13000
Freeing blocks from 1 to 999 in steps of 1
After free(), program break is:                 0x8a13000
```

Если же будет высвобожден весь набор блоков в верхней части кучи, мы увидим, что крайняя точка программы снизится по сравнению со своим пиковым значением, показывая, что функция `free()` использовала системный вызов `sbrk()`, чтобы снизить положение крайней точки программы. Здесь высвобождаются последние 500 блоков выделенной памяти:

```
$ ./free_and_sbrk 1000 10240 1 500 1000
Initial program break:                          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:                           0x8a13000
Freeing blocks from 500 to 1000 in steps of 1
After free(), program break is:                 0x852b000
```

В этом случае функция `free()` (из библиотеки glibc) способна распознать, что высвобождается целая область на вершине кучи, поэтому при высвобождении блоков она объединяет соседние высвобождаемые блоки в один большой блок. (Такое объединение выполняется с целью избавления от большого количества мелких фрагментов в списке свободных блоков, каждый из которых может быть слишком мал для удовлетворения запросов последующих вызовов функции `malloc()`.)

> Функция free() из библиотеки glibc осуществляет вызов sbrk() для снижения уровня крайней точки программы, только когда высвобождаемый блок на вершине кучи «достаточно» большой. Здесь «достаточность» определяется параметрами, которые управляют операциями пакета функции из семейства malloc (обычно это 128 Кбайт). Тем самым снижается количество необходимых вызовов sbrk() (то есть количество системных вызовов brk()).

Использовать или не использовать функцию free()

Когда процесс завершается, вся его память возвращается системе, включая память кучи, выделенную функциями из семейства `malloc`. В программах, которые выделяют память и продолжают ее использовать до завершения своего выполнения, зачастую вызовы `free()` не применяются, поскольку они полагаются именно на это автоматическое высвобождение памяти. Такое поведение может принести особые преимущества в программах, выделяющих большое количество блоков памяти, поскольку добавление множества вызовов функции `free()` может дорого обойтись с точки зрения затрат времени центрального процессора, а также, возможно, усложнить сам код программы.

Хотя для многих программ вполне допустимо надеяться на автоматическое высвобождение памяти при завершении процесса, есть две причины, по которым желательно проводить явное высвобождение всей выделенной памяти.

- Явный вызов функции `free()` может повысить читаемость и упростить сопровождение программы при необходимости ее доработок.
- Если для поиска в программе утечек памяти используется отладочная библиотека пакета `malloc` (рассматриваемая ниже), то любая память, которая не была высвобождена явным образом, будет показана как утечка памяти. Это может усложнить задачу выявления реальных утечек памяти.

7.1.3. Реализация функций malloc() и free()

Хотя функциями `malloc()` и `free()` предоставляется интерфейс выделения памяти, который гораздо легче использовать, чем результат работы функций `brk()` и `sbrk()`, все же при его применении можно допустить ряд ошибок программирования. Разобраться в глубинных причинах таких ошибок и в способах их обхода поможет понимание внутреннего устройства функций `malloc()` и `free()`.

Реализация функции `malloc()` достаточно проста. Сначала она сканирует список ранее высвобожденных функцией `free()` блоков памяти, чтобы найти тот блок, размер которого больше или равен предъявленным требованиям. (В зависимости от конкретной реализации для этого сканирования могут применяться различные стратегии, например *первый же подходящий блок* или же *наиболее подходящий*.) Если блок в точности подходит по размеру, он возвращается вызывавшему функцию коду. Если он больше по размеру, то он разбивается, и вызывавшему функцию коду возвращается блок подходящего размера, а свободный блок меньшего размера остается в списке свободных блоков.

Если в списке не найдется ни одного достаточно большого блока, функция `malloc()` вызывает `sbrk()` для выделения большего количества памяти. Чтобы уменьшить количество вызовов `sbrk()`, вместо выделения именно того количества байтов, которое требуется, функция `malloc()` повышает уровень крайней точки программы, добавляя большее количество блоков памяти (сразу несколько единиц, соответствующих размеру виртуальной страницы памяти) и помещая избыточную память в список свободных блоков.

Если посмотреть на реализацию функции `free()`, то там все организовано еще интереснее. Как `free()`, когда она помещает блок памяти в список свободных блоков, узнает, какого размера этот блок? Это делается благодаря особому приему. Когда функция `malloc()` выделяет блок, она выделяет дополнительные байты для хранения целочисленного значения, содержащего размер блока. Это значение находится в начале блока. Адрес, возвращаемый вызывавшему функцию коду, указывает на то место, которое следует сразу же за значением длины (рис. 7.1).

Рис. 7.1. Блок памяти, возвращенный функцией malloc()

Когда блок помещается в двухсвязный список свободных блоков (имеющий двойную связь), функция `free()` использует для добавления блока к списку байты самого блока (рис. 7.2).

Рис. 7.2. Блок в списке свободных блоков

По мере высвобождения и нового выделения памяти все блоки в списке свободных блоков станут перемежаться с выделенными, используемыми блоками памяти (рис. 7.3).

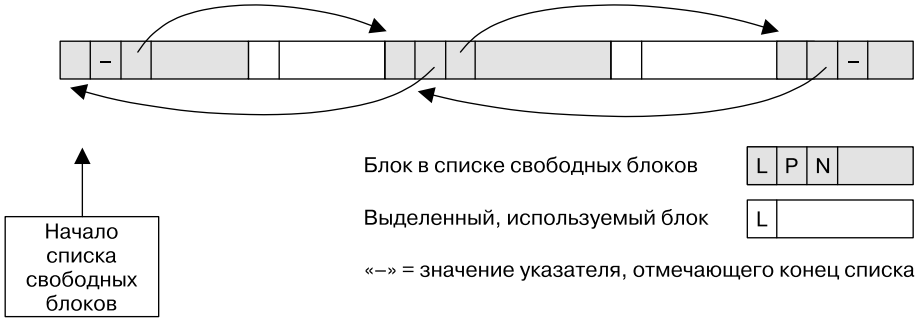


Рис. 7.3. Куча, содержащая выделенные блоки и блоки, входящие в список свободных блоков

Теперь рассмотрим то обстоятельство, что язык C позволяет создавать указатели на любое место в куче и изменять место, на которое они ссылаются, включая указатели на *длину*, на *предыдущий свободный блок* и на *следующий свободный блок*, обслуживаемые функциями `free()` и `malloc()`. Добавим это к предыдущему описанию, и у нас получится весьма огнеопасная смесь с точки зрения возможностей допущения скрытых ошибок программирования. Например, если из-за неверно установленного указателя мы случайно увеличим одно из значений длины, предшествующее выделенному блоку памяти, а после этого высвободим этот блок, то функция `free()` запишет в список свободных блоков неверный размер блока памяти. В последующем функция `malloc()` может заново выделить его, и обстоятельства сложатся так, что у программы будут указатели на два блока выделенной памяти, рассматриваемые как отдельные блоки, а на самом деле они будут перекрываться. Можно нарисовать в воображении и другие картины того, что может пойти не так.

Чтобы избежать подобных ошибок, нужно соблюдать следующие правила.

- После выделения блока памяти нужно предостеречься от манипуляции байтами за пределами диапазона этого блока. Такие манипуляции могут, к примеру, произойти в результате неверных арифметических действий в отношении указателей или допущения ошибки смещения на единицу в циклах, обновляющих содержимое блока.
- Высвобождение одного и того же блока выделенной памяти более одного раза является ошибкой. При использовании в Linux библиотеки glibc при этом скорее всего возникнет ошибка сегментации (сигнал `SIGSEGV`). Это хорошо, поскольку мы получаем предупреждение о допущенной ошибке программирования. Но в более общем смысле высвобождение одной и той же памяти дважды ведет к непредсказуемому поведению программы.
- Никогда не следует вызывать функцию `free()` со значением указателя, которое не было получено путем вызова одной из функций из пакета `malloc`.

- Если создается программа, рассчитанная на долгосрочное выполнение (например, оболочка или сетевой процесс, выполняемый в фоновом режиме) и многократно выделяющая память для различных целей, нужно обеспечить высвобождение всей памяти по окончании ее использования. Если этого не сделать, куча будет неуклонно расти до тех пор, пока не будет достигнут предел доступной виртуальной памяти, и тогда дальнейшие попытки выделения памяти начнут давать сбой. Такие обстоятельства называются *утечкой памяти*.

Средства и библиотеки для отладки выделения памяти

Несоблюдение вышеизложенных правил может привести к ошибкам, которые трудно обнаружить. Задачу обнаружения подобных ошибок можно существенно облегчить, если использовать средства отладки выделения памяти. Они предоставляются библиотекой glibc или одной из библиотек отладки выделения памяти, разработанных для этой цели.

Среди всех средств отладки, предоставляемых библиотекой glibc, можно выделить следующие.

- Функции `mtrace()` и `muntrace()`, позволяющие программе включать и выключать отслеживание вызовов выделения памяти. Функции используются в сочетании с переменной среды `MALLOC_TRACE` — она должна быть определена для хранения имени файла, в который будет записываться трассировочная информация. После того как функция `mtrace()` будет вызвана, она проверит факт определения этого файла и возможность его открытия для записи. При положительном результате этой проверки все вызовы функций из пакета `malloc` будут отслеживаться и записываться в файл. Поскольку содержимое файла будет неудобочитаемым, для его анализа и создания читаемого результата предоставляется сценарий, который также называется `mtrace`. Из соображений безопасности вызовы `mtrace()` игнорируются программами с установленными идентификаторами пользователя и/или группы (set-group-ID).
- Функции `mcheck()` и `mprobe()` позволяют программе проверять корректность блоков выделенной памяти, например отлавливать такие ошибки, как попытки записи в те места, которые находятся за пределами блока выделенной памяти. Эти функции предоставляют возможности, которые несколько накладываются на возможности рассматриваемых далее библиотек отладки `malloc`. Программы, задействующие эти функции, должны быть скомпонованы с библиотекой mcheck с использованием ключа `cc -lmcheck`.
- Переменная среды `MALLOC_CHECK_` (обратите внимание на последний символ подчеркивания) служит тем же целям, что и функции `mcheck()` и `mprobe()`. (Примечательная разница между этими двумя технологиями состоит в том, что использование `MALLOC_CHECK_` не требует внесения изменений в код и перекомпиляции программы.) Устанавливая для этой переменной различные целочисленные значения, мы можем управлять тем, как программа реагирует на ошибки выделения памяти. Возможными значениями для установки являются:
  - `0` — означает игнорирование ошибок;
  - `1` — устанавливает вывод диагностируемых ошибок на устройство стандартной ошибки — stderr;
  - `2` — означает вызов функции `abort()` для прекращения выполнения программы.

  Использование `MALLOC_CHECK_` не позволяет обнаружить абсолютно все ошибки выделения и высвобождения памяти. С ее помощью можно найти только самые характерные из них. Тем не менее эта технология является быстродействующей и простой в использовании, а также имеет низкий уровень издержек в ходе выполнения программы по сравнению с применением библиотек отладки `malloc`. Из соображений

безопасности установка значения для MALLOC_CHECK_ программами с полномочиями setuid и setgid игнорируется.

Дополнительные сведения обо всех вышеперечисленных возможностях можно найти в руководстве по glibc.

Библиотека отладки malloc предлагает такой же API, как и стандартный пакет malloc, но выполняет дополнительную работу по отлавливанию ошибок, допущенных при выделении памяти. Чтобы воспользоваться такой библиотекой, приложение следует скомпоновать вместе с ней, а не с пакетом malloc в стандартной библиотеке C. Поскольку использование таких библиотек обычно приводит к замедлению операций в ходе выполнения программы, увеличению потребления памяти или же и тому и другому вместе, их следует использовать только для отладки. После этого, при создании эксплуатационной версии приложения, нужно вернуться к компоновке со стандартным пакетом malloc. К таким библиотекам относятся Electric Fence (http://www.perens.com/FreeSoftware/), dmalloc (http://dmalloc.com/), Valgrind (http://valgrind.org/) и Insure++ (http://www.parasoft.com/).

> Библиотеки Valgrind и Insure++ способны обнаруживать многие другие виды ошибок, кроме тех, что связаны с выделением памяти в куче. Подробности можно найти на сайтах с их описаниями.

Управление пакетом malloc и отслеживание его работы

В руководстве по библиотеке glibc дается описание нестандартных функций, которые могут использоваться для отслеживания и управления выделением памяти теми функциями, что входят в пакет malloc. Среди них можно отметить следующие.

- Функция mallopt() изменяет различные параметры, которые управляют алгоритмом, используемым функцией malloc(). Например, один из таких параметров определяет минимальный объем высвобождаемого пространства, которое должно быть в конце списка свободных блоков, перед тем как используется sbrk() для сжатия кучи. Еще один параметр указывает верхний предел для размера блоков, выделяемых из кучи. Блоки, превышающие этот предел, выделяются с использованием системного вызова mmap() (см. раздел 45.7).
- Функция mallinfo() возвращает структуру, содержащую различные статистические данные о выделении памяти с помощью malloc().

Версии mallopt() и mallinfo() предоставляются многими реализациями UNIX. Но интерфейсы, предоставляемые этими функциями, у всех реализаций различаются, поэтому они не портируются.

7.1.4. Другие методы выделения памяти в куче

Наряду с malloc() библиотека языка C предоставляет ряд других функций для выделения памяти в куче. Они будут описаны в этом разделе.

Выделение памяти с помощью функций calloc() и realloc()

Функция calloc() выделяет память для массива одинаковых элементов.

```
#include <stdlib.h>

void *calloc(size_t numitems, size_t size);
```

Возвращает указатель на выделенную память при успешном завершении или NULL при ошибке

Аргумент `numitems` указывает количество выделяемых элементов, а аргумент `size` определяет их размер. После выделения блока памяти соответствующего размера `calloc()` возвращает указатель на начало блока (или `NULL`, если память не может быть выделена). В отличие от `malloc()`, функция `calloc()` инициализирует выделенную память нулевым значением.

Рассмотрим пример использования функции `calloc()`:

```
struct myStruct { /* Определение нескольких полей */ };
struct myStruct *p;

p = calloc(1000, sizeof(struct myStruct));
if (p == NULL)
    errExit("calloc");
```

Функция `realloc()` используется для изменения размера (обычно увеличения) блока памяти, ранее выделенного одной из функций из пакета `malloc`.

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Возвращает указатель на выделенную память при успешном завершении или `NULL` при ошибке

Аргумент `ptr` является указателем на блок памяти, чей размер изменяется. Аргумент `size` определяет желаемый новый размер блока.

В случае успеха функция `realloc()` возвращает указатель на местонахождение блока, размер которого был изменен. Оно может отличаться от его местонахождения до вызова этой функции. При ошибке функция `realloc()` возвращает `NULL` и оставляет блок, на который указывает аргумент `ptr`, нетронутым (это требование прописано в SUSv3).

Когда функция `realloc()` увеличивает размер блока выделенной памяти, дополнительно выделенные байты не инициализируются.

Память, выделенная с использованием функции `calloc()` или `realloc()`, должна быть высвобождена с помощью функции `free()`.

> Вызов realloc(ptr, 0) эквивалентен вызову free(ptr), за которым следует вызов malloc(0). Если для аргумента ptr указано значение NULL, то вызов функции realloc() становится эквивалентом вызова malloc(size).

При обычном применении, когда увеличивается размер блока памяти, функция `realloc()` предпринимает попытку срастить его с тем блоком памяти, который следует непосредственно за ним в списке свободных блоков, если таковой имеется и у него достаточно большой размер. Если блок находится в конце кучи, то функция `realloc()` расширяет кучу. Если блок памяти находится в середине кучи и сразу за ним недостаточно свободного пространства, то функция `realloc()` выделяет новый блок памяти и копирует все существующие данные из старого блока в новый. Последний случай довольно распространен и требует дополнительных ресурсов центрального процессора. В общем, рекомендуется использовать `realloc()` как можно реже.

Поскольку функция `realloc()` может изменить местоположение блока памяти, для последующих ссылок на этот блок нужно использовать возвращенное ею значение указателя. Функцию `realloc()` можно применять для перераспределения блока, на который указывает переменная `ptr`, следующим образом:

```
nptr = realloc(ptr, newsize);
if (nptr == NULL) {
    /* Обработка ошибки */
} else {                /* Выполнение realloc() завершилось успешно */
    ptr = nptr;
}
```

В этом примере мы не стали присваивать возвращенное функцией `realloc()` значение непосредственно `ptr`. Если функция даст сбой, то для `ptr` установится значение `NULL`, что сделает существующий блок недоступным.

Поскольку `realloc()` может переместить блок памяти, любые указатели, ссылающиеся на места внутри блока перед вызовом функции `realloc()`, могут утратить свою актуальность после вызова. Гарантированно будет актуальным только смещение относительно начала блока, остальные способы указать элемент не работают.

Выделение выровненной памяти: memalign() и posix_memalign()

Функции `memalign()` и `posix_memalign()` предназначены для выделения памяти, начиная с адреса, который будет кратен некоторой степени двойки, что может весьма пригодиться для отдельных приложений (см., к примеру, листинг 13.1).

```
#include <malloc.h>

void *memalign(size_t boundary, size_t size);
```

Возвращает указатель на выделенную память при успешном завершении или `NULL` при ошибке

Функция `memalign()` выделяет `size` байтов, начиная с адреса, выровненного по границе, кратной степени числа два. В результате выполнения функция возвращает адрес выделенной памяти.

Функция `memalign()` присутствует не во всех реализациях UNIX. Большинство других реализаций UNIX, предоставляющих `memalign()`, для получения объявления функции требуют включения вместо `<malloc.h>` заголовочного файла `<stdlib.h>`.

В SUSv3 функция `memalign()` не указана, но вместо нее есть точно такая же функция под названием `posix_memalign()`. Эта функция была недавно создана комитетом по стандартизации и появилась лишь в нескольких реализациях UNIX.

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Возвращает `0` при успешном завершении или номер ошибки в виде положительного числа при ошибке

Функция `posix_memalign()` отличается от `memalign()` двумя деталями:

- адрес выделенной памяти возвращается в `memptr`;
- память выравнивается по значению степени числа два, которое кратно значению `sizeof(void *)` (4 или 8 байт в большинстве аппаратных архитектур).

Обратите внимание также на необычное возвращаемое этой функцией значение. Вместо того чтобы при ошибке возвратить `-1`, она возвращает номер ошибки (то есть положительное целое число того типа, который обычно возвращается в `errno`).

Если значение `sizeof(void *)` равно 4, то так с помощью функции `posix_memalign()` можно выделить 65 536 байт памяти, выровненных по 4096-байтовой границе:

```
int s;
void *memptr;

s = posix_memalign(&memptr, 1024 * sizeof(void *), 65536);
if (s != 0)
    /* Обработка ошибки */
```

Блоки памяти, выделенные с использованием `memalign()` или `posix_memalign()`, должны высвобождаться с помощью функции `free()`.

> В некоторых реализациях UNIX невозможно вызвать функцию free() в отношении блока памяти, выделенного с помощью memalign(), поскольку в реализации memalign() для выделения блока памяти используется функция malloc(), а затем возвращается указатель на адрес с соответствующем выравниванием в этом блоке. Реализация функции memalign() в библиотеке glibc от этого ограничения не страдает.

7.2. Выделение памяти в стеке: alloca()

Как и функции в пакете `malloc`, функция `alloca()` выделяет память в динамическом режиме. Но вместо получения памяти в куче `alloca()` получает память из стека путем увеличения размера стекового фрейма. Это возможно потому, что вызываемая функция относится к тем, чей фрейм стека по определению находится на его вершине. Поэтому для расширения, которое возможно простым изменением значения указателя стека, доступно пространство выше фрейма.

```
#include <alloca.h>

void *alloca(size_t size);
```

Возвращает указатель на выделенный блок памяти

В аргументе `size` указывается количество байтов, выделяемое в стеке.

Нам не нужно, а на самом деле мы не должны вызывать функцию `free()` для высвобождения памяти, выделенной с помощью функции `alloca()`. Точно так же невозможно использовать функцию `realloc()` для изменения блока памяти, выделенного с помощью `alloca()`.

Хотя функция `alloca()` не является частью SUSv3, она предоставляется большинством реализаций UNIX, и поэтому ее можно считать достаточно портируемой.

> В старых версиях glibc и в некоторых других реализациях UNIX (главным образом производных от BSD) для получения объявления функции alloca() требуется включение вместо <alloca.h> заголовочного файла <stdlib.h>.

Если в результате вызова `alloca()` произойдет переполнение стека, поведение программы станет непредсказуемым. В частности, мы не получим в качестве возвращаемого

значения NULL и не будем проинформированы о возникновении ошибки. (На самом деле при таких обстоятельствах мы можем получить сигнал SIGSEGV. Подробную информацию вы получите в разделе 21.3.)

Учтите, что alloca() нельзя использовать внутри списка аргументов функции, как в следующем примере:

```
func(x, alloca(size), z);           /* НЕВЕРНО! */
```

Дело в том, что пространство стека, выделенное функцией alloca(), появится в середине пространства для аргументов функции (которые помещаются в фиксированные места внутри стекового фрейма). Вместо этого нужно воспользоваться следующим кодом:

```
void *y;

y = alloca(size);
func(x, y, z);
```

Применение функции alloca() для выделения памяти по сравнению с использованием malloc() имеет ряд преимуществ. Одно из них состоит в том, что выделение блоков памяти с alloca() происходит быстрее, чем с malloc(), поскольку первая реализуется компилятором как встроенный код, который устанавливает указатель стека напрямую. Кроме того, функция alloca() не требует поддержки списка свободных блоков.

Еще одно преимущество alloca() состоит в том, что выделяемая этой функцией память автоматически высвобождается при удалении стекового фрейма, то есть когда происходит возвращение из функции, вызвавшей alloca(). Это случается потому, что код, выполняемый в ходе возвращения из функции, переустанавливает значение регистра указателя стека на конец предыдущего фрейма (то есть, учитывая, что стек растет вниз, на адрес, находящийся непосредственно над началом текущего фрейма). Поскольку нам не нужно ничего делать для обеспечения того, чтобы высвобождаемая память очищалась от всех путей возвращения из функции, программирование некоторых функций существенно упрощается.

Функция alloca() может особенно пригодиться при использовании функции longjmp() (см. раздел 6.8) или siglongjmp() (см. подраздел 21.2.1) для выполнения нелокального перехода из обработчика сигнала. В этом случае очень трудно или даже невозможно избежать утечки памяти, если она выделяется для той функции, над которой осуществляется переход, с помощью функции malloc(). Для сравнения, функция alloca() позволяет полностью избежать подобной проблемы, поскольку, как только стек будет отмотан этими вызовами назад, выделенная память автоматически высвободится.

7.3. Резюме

С использованием семейства функций malloc процесс может выделять и высвобождать память в куче в динамическом режиме. При рассмотрении реализации этих функций было показано, что в программах, неправильно обращающихся с блоками выделенной памяти, могут происходить различные неприятности. Кроме того, было отмечено, что для содействия обнаружению источника подобных ошибок доступно несколько отладочных средств.

Функция alloca() выделяет память в стеке. Эта память автоматически высвобождается при возвращении из функции, вызвавшей alloca().

7.4. Упражнения

7.1. Измените программу из листинга 7.1 (`free_and_sbrk.c`) так, чтобы она выводила текущее значение крайней точки программы после каждого выполнения функции `malloc()`. Запустите программу, указав небольшой размер выделяемого блока. Тем самым будет продемонстрировано, что функция `malloc()` не использует `sbrk()` для изменения положения крайней точки программы при каждом вызове, а вместо этого периодически выделяет более крупные фрагменты памяти, из которых возвращает вызывающему коду небольшие фрагменты.

7.2. (Повышенной сложности.) Реализуйте функции `malloc()` и `free()`.

8 Пользователи и группы

У каждого пользователя имеется уникальное имя для входа в систему и связанный с ним числовой идентификатор пользователя (UID). Пользователи могут состоять в одной или нескольких группах. У каждой группы также есть уникальное имя и идентификатор группы (GID).

Основное предназначение пользовательских и групповых идентификаторов — определение принадлежности различных системных ресурсов и управление правами, предоставляемыми процессам по доступу к этим ресурсам. Например, каждый файл принадлежит конкретному пользователю и группе, а у каждого процесса есть несколько пользовательских и групповых идентификаторов, определяющих владельцев процесса и набор прав для доступа к файлу (подробности изложены в главе 9).

В этой главе мы рассмотрим системные файлы, используемые для определения пользователей и групп в системе, а затем библиотечные функции для извлечения информации из этих файлов. В завершение мы разберем работу функции `crypt()`, используемой для шифрования и аутентификации паролей входа в систему.

8.1. Файл паролей: /etc/passwd

В *системном файле паролей*, `/etc/passwd`, содержится по одной строке для каждой имеющейся в системе учетной записи пользователя. Каждая строка состоит из семи полей, отделенных друг от друга двоеточиями (`:`):

```
mtk:x:1000:100:Michael Kerrisk:/home/mtk:/bin/bash
```

Рассмотрим эти поля по порядку следования.

- *Имя для входа в систему*. Это уникальное имя, которое пользователь должен вводить при входе в систему. Зачастую его также называют именем пользователя. Имя для входа в систему можно рассматривать как легко читаемый (символьный) идентификатор, соответствующий числовому идентификатору пользователя (который вскоре будет рассмотрен). Это имя (вместо числового UID) выводят на экран при запросе принадлежности файла такие программы, как `ls(1)`, например при вводе команды `ls -l`.
- *Зашифрованный пароль*. В этом поле содержится 13-символьный зашифрованный пароль (более подробно мы рассмотрим его в разделе 8.5). Если в поле пароля содержится любая другая строка, в частности строка с другим количеством символов, значит, вход с этой учетной записью недопустим, поскольку такая строка не может представлять действующий зашифрованный пароль. При этом следует учесть, что, если включен режим теневых паролей (что обычно и бывает), данное поле игнорируется. В этом случае поле пароля в `/etc/passwd` содержит букву x (хотя на ее месте может быть любая непустая символьная строка), а зашифрованный пароль хранится в теневом файле (см. раздел 8.2). Если поле пароля в `/etc/passwd` пустое, значит, для регистрации под этой учетной записью пароль не нужен (это правило действует даже при наличии теневых паролей).

Здесь будет считаться, что пароли зашифрованы с помощью исторически сложившейся и по-прежнему широко используемой в UNIX схемы шифрования паролей под названием Data Encryption Standard (DES). Схему DES можно заменить другими схемами, например MD5, которая создает из данных на входе 128-битный профиль сообщения (разновидность хеша). В файле паролей (или теневом файле паролей) это значение сохраняется в виде 34-символьной строки.

- *Идентификатор пользователя (UID).* Это числовой идентификатор данного пользователя. Если поле хранит значение `0`, то пользователь с данной учетной записью — привилегированный (суперпользователь). Как правило, имеется только одна такая учетная запись, у которой в качестве имени для входа в систему используется слово `root`. В Linux 2.2 и более ранних версиях идентификаторы пользователей хранились в виде 16-битных значений, позволяющих иметь UID в диапазоне от 0 до 65 535. В Linux 2.4 и более поздних версиях идентификаторы хранятся с использованием 32 бит, позволяя задействовать значительно более широкий диапазон.

  Возможно (но редко встречается) наличие в файле паролей более одной записи с одним и тем же UID, что позволяет иметь для этого идентификатора сразу несколько имен для входа в систему. При этом нескольким пользователям разрешено иметь доступ к одним и тем же ресурсам (например, файлам), используя различные пароли. С различными наборами идентификаторов групп могут быть связаны различные имена для входа в систему.

- *Идентификатор группы (GID).* Это числовой идентификатор первой из групп, в которую входит пользователь. Дальнейшая принадлежность к группам этого пользователя определена в системном файле групп.
- *Комментарий.* Это поле содержит текст, описывающий пользователя. Такой текст выводится различными программами, например `finger(1)`.
- *Домашний каталог.* Исходный каталог, в который пользователь попадает после входа в систему. Содержимое этого поля становится значением переменной среды `HOME`.
- *Оболочка входа в систему.* Это программа, которой передается управление после входа пользователя в систему. Обычно это одна из оболочек, например bash, но может быть и любая другая программа. Если это поле остается пустым, то в качестве исходной применяется оболочка `/bin/sh`, Bourne shell. Содержимое поля становится значением переменной среды `SHELL`.

В автономной системе вся информация, касающаяся паролей, находится в файле `/etc/passwd`. Но если для хранения паролей в сетевой среде используется такая система, как Network Information System (NIS) или Lightweight Directory Access Protocol (LDAP), часть этой информации или же вся она целиком находится в удаленной системе. Поскольку программы, обращающиеся за информацией о паролях, используют рассматриваемые далее функции (`getpwnam()`, `getpwuid()` и т. д.), приложениям безразлично, что именно применяется: NIS или LDAP. То же самое можно сказать и о теневых файлах паролей и групп, рассматриваемых в следующих разделах.

8.2. Теневой файл паролей: /etc/shadow

Исторически сложилось так, что в системах UNIX вся информация о пользователях, включая зашифрованные пароли, хранится в файле `/etc/passwd`. В связи с этим возникают проблемы безопасности. Поскольку различным непривилегированным системным утилитам требуется доступ для чтения к другой информации, содержащейся в файле паролей, ее нужно делать доступной для чтения для всех пользователей. Тем самым открывается

лазейка для программ по взлому паролей, пытающихся их расшифровать с помощью длинных списков наиболее вероятных вариантов (например, стандартных записей из словарей или имен людей), чтобы определить, соответствуют ли они зашифрованному паролю пользователя. *Теневой файл паролей*, `/etc/shadow`, был разработан как средство противостояния таким атакам. Замысел заключается в том, что вся неконфиденциальная информация о пользователе находится в открытом, доступном для чтения файле паролей, а зашифрованные пароли хранятся в теневом файле паролей, доступном для чтения только программам с особыми привилегиями.

Вдобавок к имени для входа в систему, обеспечивающему совпадение с соответствующей записью в файле паролей, и зашифрованному паролю теневой файл паролей также содержит ряд других полей, связанных с обеспечением мер безопасности. Дополнительные подробности, касающиеся этих полей, можно найти на странице руководства `shadow(5)`. Нас же главным образом интересует поле зашифрованного пароля, которое более подробно мы рассмотрим при изучении библиотечной функции `crypt()` в разделе 8.5.

Теневые пароли в SUSv3 не определены. Кроме того, они предоставляются не всеми реализациями UNIX.

8.3. Файл групп: /etc/group

Пользователей для различных административных целей, в частности для управления доступом к файлам и другим системным ресурсам, полезно свести в *группы*.

Набор групп, к которым принадлежит пользователь, определен в виде сочетания поля идентификатора группы в записи пользователя в файле паролей и групп, под которыми этот пользователь перечисляется в файле групп. Это странное разбиение информации на два файла сложилось исторически. В ранних реализациях UNIX можно было одновременно входить только в одну группу. Исходная группа, в которую входил пользователь при входе в систему, определялась полем GID файла паролей и могла быть в нем изменена после использования команды `newgrp(1)`. Эта команда требовала от пользователя предоставить пароль группы (если вход в группу был защищен паролем). В 4.2BSD было введено понятие одновременной принадлежности к нескольким группам, позже ставшее стандартом в POSIX.1-1990. Согласно этой схеме в файле групп имелся список принадлежности каждого пользователя к дополнительным группам. (Команда `groups(1)` выводит либо те группы, в которые входит данный процесс оболочки, либо, если были переданы (одно или несколько) имена пользователей, — те группы, в которые входят эти пользователи.)

Файл групп `/etc/group` содержит по одной строке для каждой группы в системе. Каждая строка, как показано в следующем примере, состоит из четырех полей, отделенных друг от друга двоеточиями:

```
users:x:100:
jambit:x:106:claus,felli,frank,harti,markus,martin,mtk,paul
```

Рассмотрим эти поля в порядке следования.

- *Имя группы*. Как и имя для входа в систему в файле паролей, имя группы можно рассматривать как легко читаемый символьный идентификатор, соответствующий числовому идентификатору группы.
- *Зашифрованный пароль*. Это поле содержит необязательный пароль группы. С появлением возможности принадлежать сразу нескольким группам в наши дни в системах UNIX пароли групп используются крайне редко. Тем не менее в это поле можно поместить пароль группы (привилегированный пользователь может сделать это с помощью команды `gpasswd`). Если пользователь не входит в группу, `newgrp(1)` запрашивает этот

пароль перед запуском новой оболочки. Если включены теневые пароли, это поле игнорируется (в этом случае по соглашению в нем содержится только буква x, но вместо нее может указываться любая строка, включая пустую), а зашифрованный пароль в действительности хранится в теневом файле групп, `/etc/gshadow`, доступ к которому могут получить только привилегированные пользователи или программы. Пароли групп шифруются точно таким же образом, что и пароли пользователей (см. раздел 8.5).

- *Идентификатор группы (GID).* Это числовой идентификатор для данной группы. Как правило, есть группа, имеющая в качестве идентификатора число 0, — это группа с названием `root` (так же как и запись в `/etc/passwd` с пользовательским идентификатором со значением `0`). В Linux 2.2 и более ранних версиях идентификаторы групп хранились в виде 16-битных значений, позволяющих иметь ID в диапазоне от 0 до 65 535. В Linux 2.4 и более поздних версиях идентификаторы хранятся с использованием 32 бит.
- *Список пользователей.* Это список, элементы которого отделены друг от друга запятыми. Он содержит имена пользователей, входящих в данную группу. (Список состоит из имен пользователей, а не из пользовательских идентификаторов, поскольку, как уже упоминалось, UID в файле паролей не обладают обязательной уникальностью.)

Следующая запись в файле паролей означает, что пользователь `avr` входит в группы `users`, `staff` и `teach`:

```
avr:x:1001:100:Anthony Robins:/home/avr:/bin/bash
```

А в файле групп будут такие записи:

```
users:x:100:
staff:x:101:mtk,avr,martinl
teach:x:104:avr,rlb,alc
```

В четвертом поле записи в файле паролей содержится идентификатор группы `100`, указывающий на то, что пользователь входит в группу `users`. Вхождение в остальные группы показывается за счет однократного присутствия `avr` в каждой соответствующей записи файла групп.

8.4. Извлечение информации о пользователях и группах

В этом разделе мы рассмотрим библиотечные функции, позволяющие извлекать отдельные записи из файлов паролей, групп и их теневых аналогов, а также сканировать все записи в каждом из этих файлов.

Извлечение записей из файла паролей

Извлечение записей из файла паролей проводится с помощью функций `getpwnam()` и `getpwuid()`.

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

Обе функции при успешном завершении возвращают указатель, при ошибке — `NULL`. Описание для случая «запись не найдена» дается в тексте подраздела

При предоставлении имени в качестве аргумента name функция getpwnam() возвращает указатель на структуру следующего типа, содержащую соответствующую информацию из записи в файле паролей:

```
struct passwd {
    char *pw_name;      /* Имя для входа в систему (имя пользователя) */
    char *pw_passwd;    /* Зашифрованный пароль */
    uid_t pw_uid;       /* Идентификатор пользователя */
    gid_t pw_gid;       /* Идентификатор группы */
    char *pw_gecos;     /* Комментарий (информация о пользователе) */
    char *pw_dir;       /* Исходный рабочий (домашний) каталог */
    char *pw_shell;     /* Оболочка входа в систему */
};
```

Поля pw_gecos и pw_passwd структуры passwd в SUSv3 не определены, но доступны во всех реализациях UNIX. Поле pw_passwd содержит актуальную информацию только при выключенном режиме использования теневых паролей. (С точки зрения программирования наипростейший способ выявить включение режима использования теневых паролей состоит в вызове функции getspnam() (вскоре рассмотрим) сразу же после успешного выполнения функции getpwnam(), чтобы увидеть, сможет ли она возвратить запись теневого пароля для того же имени пользователя.) В некоторых других реализациях в этой структуре предоставляются дополнительные нестандартные поля.

Поле pw_gecos происходит из ранних реализаций UNIX, где в нем содержалась информация для связи с машиной, на которой запущена операционная система General Electric Comprehensive Operating System (GECOS). Хотя эта цель его применения давно устарела, имя поля осталось прежним, а само оно предназначено для записи информации о пользователе.

Функция getpwuid() возвращает точно такую же информацию, что и функция getpwnam(), но ведет поиск по числовому идентификатору пользователя, предоставленному в аргументе uid. Обе функции возвращают указатель на статически выделенную структуру. Эта структура перезаписывается при каждом вызове любой из этих функций (или рассматриваемой далее функции getpwent()).

Поскольку функции getpwnam() и getpwuid() возвращают указатель на статически выделенную структуру, они являются нереентерабельными. На самом деле ситуация складывается еще сложнее, поскольку возвращаемая структура passwd содержит указатели на другую информацию (например, поле pw_name), которая также является статически выделенной. (Реентерабельность объясняется в подразделе 21.1.2.) Такие же утверждения справедливы для функций getgrnam() и getgrgid(), которые мы вскоре рассмотрим.

В SUSv3 указывается эквивалентный набор реентерабельных функций — getpwnam_r(), getpwuid_r(), getgrnam_r() и getgrgid_r(), включающих в качестве аргументов как структуру passwd (или group), так и область буфера для хранения других структур, на которые указывают поля структуры passwd (group). Количество байтов, требуемое для этого дополнительного буфера, может быть получено с помощью вызова sysconf(_SC_GETPW_R_SIZE_MAX) (или sysconf(_SC_GETGR_R_SIZE_MAX) для функций, имеющих отношение к группам). Дополнительные сведения об этих функциях можно найти на страницах руководства.

В соответствии с положениями SUSv3, если нужная запись passwd не может быть найдена, функции getpwnam() и getpwuid() должны возвратить значение NULL, оставив значение errno в неизменном виде. Таким образом, можно различить ошибку и случаи «запись не найдена», используя следующий код:

```
struct passwd *pwd;

errno = 0;
pwd = getpwnam(name);
if (pwd == NULL) {
    if (errno == 0)
        /* Запись не найдена */;
    else
        /* Ошибка */;
 }
```

Однако некоторые реализации UNIX не соответствуют [требованиям] SUSv3 по этому вопросу. Если нужная запись `passwd` не найдена, эти функции возвращают значение `NULL` и устанавливают для `errno` ненулевое значение, например `ENOENT` или `ESRCH`. До выхода версии 2.7 библиотека glibc выдавала в таком случае ошибку `ENOENT`, но, начиная с версии 2.7, она стала отвечать требованиям SUSv3. Эти расхождения в реализациях возникли отчасти из-за того, что в POSIX.1-1990 данным функциям не требовалось устанавливать для `errno` значения при ошибке и позволялось устанавливать значение в случае «запись не найдена». В результате при использовании этих функций стало совершенно невозможно портируемым образом отличить ошибку от ситуации «запись не найдена».

Извлечение записей из файла групп

Записи из файла групп извлекаются с помощью функций `getgrnam()` и `getgrgid()`.

```
#include <grp.h>

struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

Обе функции при успешном завершении возвращают указатель, при ошибке — `NULL`. Описание для случая «запись не найдена» дается в тексте подраздела

Функция `getgrnam()` осуществляет поиск информации о группе по имени группы, а функция `getgrgid()` — по идентификатору группы. Обе функции возвращают указатель на структуру следующего типа:

```
struct group {
    char  *gr_name;     /* Имя группы */
    char  *gr_passwd;   /* Зашифрованный пароль (в режиме без теневых паролей) */
    gid_t  gr_gid;      /* Идентификатор группы */
    char **gr_mem;      /* Массив указателей на имена участников группы,
                           перечисленных в /etc/group, завершающийся значением NULL */
};
```

Поле gr_passwd структуры group в SUSv3 не указано, но доступно в большинстве реализаций UNIX.

Как и в случае рассмотренных выше соответствующих функций работы с записями в файле паролей, эта структура перезаписывается при каждом вызове одной из этих функций.

Если функции не могут найти запись, соответствующую группе, они демонстрируют такие же варианты поведения, которые были рассмотрены для функций `getpwnam()` и `getpwuid()`.

Пример программы

Один из примеров наиболее частого применения рассмотренных в этом разделе функций — преобразование символьных имен пользователя и группы в их числовые идентификаторы и наоборот. В листинге 8.1 показано это преобразование в виде четырех функций: `userNameFromId()`, `userIdFromName()`, `groupNameFromId()` и `groupIdFromName()`. Для удобства вызывающего функции `userIdFromName()` и `groupIdFromName()` также позволяют аргументу `name` быть числовой строкой в чистом виде. В этом случае строка преобразуется непосредственно в число и возвращается вызвавшему функцию коду. Эти функции будут использоваться в некоторых примерах программ, которые мы рассмотрим далее в книге.

Листинг 8.1. Функции для преобразования идентификаторов пользователей и групп в имена пользователей и групп и наоборот

users_groups/ugid_functions.c

```
#include <pwd.h>
#include <grp.h>
#include <ctype.h>
#include "ugid_functions.h"   /* Объявление определяемых здесь функций */

char *  /* Возвращает имя, соответствующее 'uid', или NULL при ошибке */
userNameFromId(uid_t uid)
{
    struct passwd *pwd;

    pwd = getpwuid(uid);
    return (pwd == NULL) ? NULL : pwd->pw_name;
}

uid_t  /* Возвращает идентификатор пользователя,
          соответствующего 'name', или –1 при ошибке */
userIdFromName(const char *name)
{
    struct passwd *pwd;
    uid_t u;
    char *endptr;
    if (name == NULL || *name == '\0')  /* Возвращает ошибку, если передан NULL*/
        return -1;                      /* или пустая строка */

    u = strtol(name, &endptr, 10);      /* Для удобства вызывающего */
    if (*endptr == '\0')                /* разрешение числовой строки */
        return u;
    pwd = getpwnam(name);
    if (pwd == NULL)
        return -1;

    return pwd->pw_uid;
}

char *  /* Возвращает имя, соответствующее 'gid', или NULL при ошибке */
groupNameFromId(gid_t gid)
{
    struct group *grp;

    grp = getgrgid(gid);
    return (grp == NULL) ? NULL : grp->gr_name;
}
```

```
gid_t     /* Возвращает идентификатор группы, */
          /* соответствующего 'name',или -1 при ошибке */
groupIdFromName(const char *name)
{
    struct group *grp;
    gid_t g;
    char *endptr;

    if (name == NULL || *name == '\0')  /* Возвращает ошибку, если передан NULL*/
        return -1;                      /* или пустая строка */

    g = strtol(name, &endptr, 10);      /* Для удобства вызывающего */
    if (*endptr == '\0')                /* разрешение числовой строки */
        return g;

    grp = getgrnam(name);
    if (grp == NULL)
        return -1;

    return grp->gr_gid;
}
```

users_groups/ugid_functions.c

Сканирование всех записей в файлах паролей и групп

Функции `setpwent()`, `getpwent()` и `endpwent()` используются для выполнения последовательного сканирования записей в файле паролей.

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Возвращает указатель при успешном завершении или `NULL` в случае конца потока или при ошибке

```
void setpwent(void);
void endpwent(void);
```

Функция `getpwent()` поочередно возвращает записи из файла паролей, выдавая `NULL`, когда записей уже больше нет (или при возникновении ошибки). При первом вызове функция автоматически открывает файл паролей. Когда работа с файлом завершена, для его закрытия вызывается функция `endpwent()`.

С помощью следующего кода можно пройти через весь файл паролей, выводя на экран имена для входа в систему и идентификаторы пользователей:

```
struct passwd *pwd;

while ((pwd = getpwent()) != NULL)
    printf("%-8s %5ld\n", pwd->pw_name, (long) pwd->pw_uid);

endpwent();
```

Вызов функции `endpwent()` необходим для того, чтобы при любом последующем вызове `getpwent()` (возможно, в другой части нашей программы или в какой-нибудь вызываемой нами библиотечной функции) файл паролей открывался заново и чтение выполнялось с начала файла. С другой стороны, если в файле пройдена только часть пути, для перезапуска чтения с начала файла можно воспользоваться функцией `setpwent()`.

Функции getgrent(), setgrent() и endgrent() выполняют аналогичные задачи для файла групп. Здесь не будет приводиться их описание, поскольку они аналогичны уже рассмотренным функциям для файла паролей. Соответствующие подробности, относящиеся к этим функциям, можно найти на страницах руководства.

Извлечение записей из теневого файла паролей

Следующие функции используются для извлечения отдельных записей из теневого файла паролей и сканирования всех записей в этом файле.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);
```

Возвращает при успешном завершении указатель или NULL, если запись не найдена либо произошла ошибка

```
struct spwd *getspent(void);
```

Возвращает указатель при успешном завершении или NULL в случае конца потока либо при ошибке

```
void setspent(void);
void endspent(void);
```

Мы не станем рассматривать эти функции во всех подробностях, поскольку их работа похожа на работу соответствующих функций, относящихся к файлу паролей. (Эти функции не указаны в SUSv3 и представлены не во всех реализациях UNIX.)

Функции getspnam() и getspent() возвращают указатели на структуру типа spwd. Она имеет следующую форму:

```
struct spwd {
    char *sp_namp;        /* Имя для входа в систему (имя пользователя) */
    char *sp_pwdp;        /* Зашифрованный пароль */

    /* Остальные поля поддерживают «устаревание пароля», дополнительное средство,
       заставляющее пользователей регулярно менять свои пароли, чтобы, даже если
       злоумышленник сумел получить пароль, тот со временем стал для него
       бесполезным. */

    long sp_lstchg;     /* Время последнего изменения пароля (количество
                           дней, прошедших с 1 января 1970 года) */
    long sp_min;        /* Минимальное количество дней между сменами пароля */
    long sp_max;        /* Максимальное количество дней до требуемой смены пароля */
    long sp_warn;       /* Количество дней, за которое пользователь
                           заранее получает предупреждение о скором
                           истечении срока действия пароля */
    long sp_inact;      /* Количество дней после истечения срока действия пароля
                           до признания учетной записи неактивнойи заблокированной */
    long sp_expire;     /* Дата, когда истекает срок действия учетной
                           записи (количество дней, прошедших с 1 января 1970 года) */
    unsigned long sp_flag;   /* Зарезервировано для будущего использования */
};
```

Применение функции getspnam() будет показано в листинге 8.2.

8.5. Шифрование пароля и аутентификация пользователя

Некоторые приложения требуют, чтобы пользователи прошли аутентификацию. Обычно требуется ввести имя пользователя (имя для входа в систему) и пароль. Приложение для этих целей может работать с собственной базой данных пользовательских имен и паролей. Но иногда необходимо или удобно позволять пользователям вводить их стандартные имена пользователей и пароли, определенные в файлах `/etc/passwd` и `/etc/shadow`. (В остальной части раздела будет считаться, что в системе включен режим использования теневых паролей и что эти зашифрованные пароли хранятся в файле `/etc/shadow`.) Наглядными примерами таких программ могут послужить сетевые приложения, предоставляющие какие-либо формы для входа в удаленную систему, например ssh и ftp. Они должны проверить допустимость имени пользователя и пароля точно так же, как это делают программы стандартного входа в систему.

Из соображений безопасности системы UNIX шифруют пароли, используя алгоритм *одностороннего шифрования*. Он гарантирует невозможность воссоздания исходного пароля из его зашифрованной формы. Поэтому единственный способ проверить верность проверяемого пароля — его шифрование с использованием того же метода, что позволит увидеть, соответствует ли зашифрованный результат значению, сохраненному в файле `/etc/shadow`. Алгоритм шифрования заключен в функции `crypt()`.

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

Возвращает указатель на статично выделенную строку, содержащую при успешном завершении зашифрованный пароль, или `NULL` при ошибке

Работа функции `crypt()` предусматривает получение ключа `key` (то есть пароля) длиной до восьми символов и применение к нему разновидности алгоритма Data Encryption Standard (DES). Аргумент `salt` является строкой из двух символов, чье значение используется для внесения помех в алгоритм (его изменения), то есть для применения технологии, затрудняющей взлом зашифрованного пароля. Функция возвращает указатель на статически выделенную 13-символьную строку, являющуюся зашифрованным паролем.

> Подробности, касающиеся алгоритма DES, можно найти по адресу http://www.itl.nist.gov/fipspubs/fip46-2.htm. Как уже ранее упоминалось, вместо DES могут использоваться другие алгоритмы. Например, применение алгоритма MD5 приводит к созданию 34-символьной строки, начинающейся с символа доллара ($), который позволяет функции crypt() отличать пароли, зашифрованные с помощью DES, от паролей, зашифрованных с помощью MD5.
>
> При рассмотрении вопроса шифрования паролей здесь употребляется слово «шифрование», что не совсем верно отражает действительность. Если выражаться точнее, то DES использует заданную строку пароля в качестве ключа шифрования для зашифровки фиксированной строки битов, а MD5 представляет собой сложный тип функции хеширования. Результат в обоих случаях получается один и тот же: не поддающееся расшифровке и необратимое преобразование входного пароля.

И аргумент `salt`, и шифруемый пароль состоят из символов, выбранных из 64-символьного набора [a-zA-Z0-9/.]. Таким образом, аргумент *`salt`* («соль»), состоящий из двух символов, может стать причиной изменения алгоритма шифрования любым из 64 × 64 = 4096 возможных способов. Это означает, что вместо предварительного шифрования целого словаря и проверки зашифрованного пароля на совпадение со всеми словами в словаре взломщику придется проверять пароль на соответствие 4096 зашифрованным версиям словарей.

Зашифрованный пароль, возвращенный функцией `crypt()`, содержит в двух первых символах копию исходного значения «соли». Это означает, что при шифровании потенциально подходящего пароля можно получить соответствующее значение «соли» из значения зашифрованного пароля, уже хранящегося в файле `/etc/shadow`. (Такие программы, как `passwd(1)`, при шифровании нового пароля создают произвольное значение «соли».) Фактически функция `crypt()` игнорирует любые символы в строке «соли», кроме первых двух. Поэтому можно указать в качестве аргумента *`salt`* сам зашифрованный пароль.

Если нужно воспользоваться функцией `crypt()` в Linux, следует откомпилировать программы с ключом `-lcrypt`, чтобы они были скомпонованы с библиотекой crypt.

Пример программы

В листинге 8.2 показано, как функция `crypt()` применяется для аутентификации пользователя. Программа в этом листинге сначала считывает имя пользователя, а затем извлекает соответствующую парольную запись и (если таковая существует) теневую запись в файле паролей. Если парольная запись не будет найдена или же если у программы нет полномочий на чтение из теневого файла паролей (для этого требуются полномочия привилегированного пользователя или принадлежность к группе `shadow`), то программа выводит на экран сообщение об ошибке, а затем осуществляет выход. Затем программа считывает пароль пользователя с помощью функции `getpass()`.

```
#define _BSD_SOURCE
#include <unistd.h>

char *getpass(const char *prompt);
```

Возвращает при успешном завершении указатель на статически размещаемую строку ввода пароля или `NULL` при ошибке

Функция `getpass()` сначала отключает отображение на экране и всю обработку специальных символов управления терминалом (таких как символ прерывания, обычно это Ctrl+C). (Способы изменения этих настроек терминала рассматриваются в главе 58.) Затем на экран выводится строка с приглашением на ввод и считывается введенная строка, а в качестве результата выполнения функции возвращается строка ввода с завершающим нулевым байтом и удаленным следующим за ней символом новой строки. (Эта строка размещается статически и поэтому будет перезаписана при следующем вызове `getpass()`.) Перед возвращением `getpass()` восстанавливает настройки терминала до их исходного состояния.

Прочитав пароль с помощью функции `getpass()`, программа из листинга 8.2 проверяет его. При этом функция `crypt()` используется для его шифрования и проверки того, что получившаяся строка в точности совпадает зашифрованному паролю, записанному в теневом файле паролей. Если пароль совпадает, идентификатор пользователя выводится на экран, как в следующем примере:

```
$ su                    Для чтения теневого файла паролей нужны привилегии
Password:
# ./check_password
Username: mtk
Password:                Набирается пароль, который не отображается на экране
Successfully authenticated: UID=1000
```

Программа в листинге 8.2 определяет размер массива символов, содержащего имя пользователя. Для этого применяется значение, возвращенное выражением sysconf(_SC_LOGIN_NAME_MAX), которое выдает максимальный размер имени пользователя в главной системе. Использование sysconf() объясняется в разделе 11.2.

Листинг 8.2. Аутентификация пользователя с применением теневого файла паролей
users_groups/check_password.c

```
#define _BSD_SOURCE     /* Получение объявления getpass() из <unistd.h> */
#define _XOPEN_SOURCE   /* Получение объявления crypt() из <unistd.h> */
#include <unistd.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *username, *password, *encrypted, *p;
    struct passwd *pwd;
    struct spwd *spwd;
    Boolean authOk;
    size_t len;
    long lnmax;

    lnmax = sysconf(_SC_LOGIN_NAME_MAX);
    if (lnmax == -1)                    /* Если предел не определен, */
        lnmax = 256;                    /* выбираем наугад */

    username = malloc(lnmax);
    if (username == NULL)
        errExit("malloc");
    printf("Username: ");
    fflush(stdout);
    if (fgets(username, lnmax, stdin) == NULL)
        exit(EXIT_FAILURE);             /* Выход при встрече EOF */

    len = strlen(username);
    if (username[len - 1] == '\n')
        username[len - 1] = '\0';       /* Удаление завершающего '\n' */

    pwd = getpwnam(username);
    if (pwd == NULL)
        fatal("couldn't get password record");
    spwd = getspnam(username);
    if (spwd == NULL && errno == EACCES)
        fatal("no permission to read shadow password file");

    if (spwd != NULL)                    /* Если есть запись теневого пароля */
        pwd->pw_passwd = spwd->sp_pwdp;  /* Использование теневого пароля */
```

```
    password = getpass("Password: ");

  /* Шифрование пароля с немедленным уничтожением незашифрованной версии */

    encrypted = crypt(password, pwd->pw_passwd);
    for (p = password; *p != '\0'; )
        *p++ = '\0';

    if (encrypted == NULL)
        errExit("crypt");

    authOk = strcmp(encrypted, pwd->pw_passwd) == 0;
    if (!authOk) {
        printf("Incorrect password\n");
        exit(EXIT_FAILURE);
    }

    printf("Successfully authenticated: UID=%ld\n", (long) pwd->pw_uid);

  /* Здесь совершаем то, ради чего аутентифицировались... */

    exit(EXIT_SUCCESS);
}
```

users_groups/check_password.c

В листинге 8.2 проиллюстрирован важный момент, касающийся решения вопросов безопасности. Программы, читающие пароль, должны немедленно его зашифровать и стереть незашифрованную версию из памяти. Тем самым будет минимизирована возможность аварийного завершения программы с образованием файла дампа ядра, который может быть прочитан для обнаружения пароля.

Существуют и другие пути раскрытия незашифрованного пароля. Например, пароль может быть прочитан из своп-файла привилегированной программой, если виртуальная страница памяти, содержащая пароль, сбрасывается на диск. Кроме того, в попытке обнаружения пароля процесс с достаточным уровнем привилегий может прочитать /dev/mem (виртуальное устройство, представляющее физическую память компьютера в виде последовательного потока байтов).

Функция getpass() фигурировала в SUSv2 с пометкой LEGACY (устаревшая), где отмечалось, что ее название вводит в заблуждение и она предоставляет функциональные возможности, которые в любом случае можно легко реализовать. Из SUSv3 спецификация getpass() была удалена. Тем не менее она встречается во многих реализациях UNIX.

8.6. Резюме

У каждого пользователя есть уникальное имя для входа в систему и связанный с ним числовой идентификатор. Пользователи могут принадлежать одной или нескольким группам, у каждой из которых также есть уникальное имя и связанный с ним числовой ID. Основная цель этих идентификаторов — доказательство факта принадлежности различных системных ресурсов (например, файлов) к группам и полномочий на доступ к ним.

Имя пользователя и идентификатор определяются в файле `/etc/passwd`, который содержит и другую информацию о пользователе. Принадлежность пользователя к той

или иной группе определяется полями в файлах `/etc/passwd` и `/etc/group`. Еще один файл, `/etc/shadow`, может быть прочитан только привилегированными процессами. Он применяется для отделения конфиденциальной парольной информации от пользовательских сведений, находящихся в открытом доступе в файле `/etc/passwd`. Для извлечения информации из каждого из этих файлов предоставляются различные библиотечные функции.

Функция `crypt()`, которая может пригодиться для программ, нуждающихся в аутентификации пользователя, шифрует пароль точно так же, как и стандартная программа входа в систему.

8.7. Упражнения

8.1. При выполнении следующего кода обнаруживается, что он дважды выводит одно и то же имя пользователя, даже если у двух пользователей разные идентификаторы. Почему так происходит?

```
printf("%s %s\n", getpwuid(uid1)->pw_name,
                  getpwuid(uid2)->pw_name);
```

8.2. Реализуйте функцию `getpwnam()`, используя функции `setpwent()`, `getpwent()` и `endpwent()`.

9 Идентификаторы процессов

У каждого процесса есть набор связанных с ним числовых идентификаторов пользователей (UID) и идентификаторов групп (GID). Иногда их называют идентификаторами процесса. В число этих идентификаторов входят:

- реальный (real) ID пользователя и группы;
- действующий (effective) ID пользователя и группы;
- сохраненный установленный ID пользователя (saved set-user-ID) и сохраненный установленный ID группы (saved set-group-ID);
- характерный для Linux пользовательский и групповой ID файловой системы;
- дополнительные идентификаторы групп.

В этой главе будут подробно рассмотрены назначения этих идентификаторов процессов, а также системные вызовы и библиотечные функции, которые могут использоваться для их извлечения и изменения. Будут также рассмотрены понятия привилегированных и непривилегированных процессов и применение механизмов установленных идентификаторов пользователей и установленных идентификаторов групп, позволяющих создавать программы, выполняемые с полномочиями конкретного пользователя или группы.

9.1. Реальный идентификатор пользователя и реальный идентификатор группы

Реальные идентификаторы пользователя и группы идентифицируют пользователя и группу, которым принадлежит процесс. При входе в систему оболочка получает свои реальные ID пользователя и группы из третьего и четвертого полей записи в файле `/etc/passwd` (см. раздел 8.1). При создании нового процесса (например, когда оболочка выполняет программу) он наследует эти идентификаторы у своего родительского процесса.

9.2. Действующий идентификатор пользователя и действующий идентификатор группы

В большинстве реализаций UNIX (Linux, как объясняется в разделе 9.5, в этом плане от них немного отличается) действующие UID и GID в совокупности с дополнительными идентификаторами групп используются для определения полномочий, которыми наделен процесс, при его попытке выполнения различных операций (в частности, системных вызовов). Например, эти идентификаторы определяют полномочия, которыми процесс наделен при доступе к таким ресурсам, как файлы и объекты межпроцессного взаимодействия (IPC) в System V. У таких объектов, в частности, есть собственные связанные с ними пользовательские и групповые идентификаторы, определяющие их принадлеж-

ность. В разделе 20.5 будет показано, что действующий UID также проверяется ядром для определения того, может ли один процесс отправить сигнал другому.

Процесс, чей действующий идентификатор пользователя имеет значение `0` (он принадлежит пользователю с именем `root`), имеет все полномочия суперпользователя. Такой процесс называют *привилегированным*. Некоторые системные вызовы могут быть выполнены только привилегированными процессами.

> В главе 39 мы рассмотрим реализацию Linux-возможностей — схему разделения полномочий, которыми наделяется привилегированный пользователь, на ряд отдельных составляющих, которые могут независимо друг от друга включаться и отключаться.

Обычно действующие идентификаторы пользователя и группы имеют точно такие же значения, что и у соответствующих реальных ID, но есть два способа, позволяющие действующим идентификаторам принимать другие значения. Один из способов связан с использованием системных вызовов (рассматриваются в разделе 9.7). Второй способ связан с выполнением программ с установленным идентификатором пользователя и установленным идентификатором группы.

9.3. Программы с установленным идентификатором пользователя и установленным идентификатором группы

Программа с установленным идентификатором пользователя позволяет процессу получить полномочия, которые он обычно не получает, путем установки действующего ID пользователя на то же значение, которое имеется у идентификатора пользователя (владельца) исполняемого файла. Программа с установленным ID группы выполняет аналогичную задачу для принадлежащего процессу действующего идентификатора группы. (Выражения *«программа с установленным идентификатором пользователя»* и *«программа с установленным идентификатором группы»* иногда сокращают до видов *«set-UID-программа»* и *«set-GID-программа»*.)

Как и любой другой файл, файл исполняемой программы имеет связанный с ним идентификатор пользователя и идентификатор группы, которые определяют принадлежность файла. Кроме того, у исполняемого файла имеется два специальных бита полномочий: бит установленного идентификатора пользователя (set-user-ID) и бит установленного идентификатора группы (set-group-ID). (В действительности эти два бита полномочий есть у каждого файла, но нас здесь интересует их использование применительно к исполняемым файлам.) Эти биты полномочий устанавливаются командой `chmod`. Непривилегированный пользователь может устанавливать эти биты для тех файлов, которыми он владеет. Привилегированный пользователь (`CAP_FOWNER`) может устанавливать эти биты для любого файла. Рассмотрим пример:

```
$ su
Password:
# ls -l prog
-rwxr-xr-x    1 root     root       302585 Jun 26 15:05 prog
# chmod u+s prog                  Установка бита полномочий set-user-ID
# chmod g+s prog                  Установка бита полномочий set-group-ID
```

Как показано в этом примере, у программы могут быть установлены оба этих бита, хотя такое встречается нечасто. Когда для вывода списка полномочий программы, имеющей

установленный бит set-user-ID или set-group-ID, используется команда `ls -l`, в нем буква x, которая обычно применяется для демонстрации установки полномочия на выполнение, заменяется буквой s:

```
# ls -l prog
-rwsr-sr-x    1 root     root       302585 Jun 26 15:05 prog
```

Когда set-user-ID-программа запускается (то есть загружается в память процесса с помощью команды `exec()`), ядро устанавливает для действующего пользовательского ID точно такое же значение, что и у пользовательского ID исполняемого файла. Запуск программы с полномочиями setgid имеет такой же эффект относительно действующего группового идентификатора процесса. Изменение действующего пользовательского или группового ID таким способом дает процессу (а иными словами, пользователю, для которого выполняется программа) полномочия, которые он не имел бы при других обстоятельствах. Например, если исполняемый файл принадлежит пользователю по имени `root` (привилегированному пользователю) и имеет установленный бит set-user-ID, то процесс при запуске программы обретает полномочия суперпользователя.

Программы с полномочиями setuid и setgid могут также использоваться с целью смены действующих идентификаторов процесса на какие-либо другие, отличные от `root`. Например, чтобы предоставить доступ к защищенному файлу (или к другому системному ресурсу), может быть достаточно создать специально предназначенный для этого ID пользователя (группы) с полномочиями, требуемыми для доступа к файлу, и создать программу с полномочиями setuid (setgid), изменяющую действующий пользовательский (групповой) ID на этот идентификатор. Это даст программе полномочия по доступу к файлу без предоставления ей всех полномочий привилегированного пользователя.

Иногда мы будем использовать выражение set-user-ID-root, чтобы отличать set-user-ID-программу, владельцем которой является `root`, от программы, которой владеет другой пользователь и которая просто дает процессу полномочия, предоставляемые этому пользователю.

> Теперь мы станем употреблять слово «привилегированный» в двух разных смыслах. Первый мы определили ранее: это процесс с действующим идентификатором пользователя со значением 0, у которого имеются все полномочия, присущие пользователю по имени root. Но, когда речь заходит о set-user-ID-программе, владельцем которой является другой, не root-пользователь, то мы называем процесс наделенным полномочиями, соответствующими идентификатору пользователя set-user-ID-программы. Какой именно смысл вкладывается в понятие «привилегированный», в каждом случае будет понятно из контекста.
>
> По причинам, объясняемым в разделе 38.3, биты полномочий set-user-ID и set-group-ID не оказывают никакого влияния на используемые в Linux сценарии оболочки.

В качестве примеров часто используемых в Linux set-user-ID-программ можно привести `passwd(1)`, изменяющую пользовательский пароль, `mount(8)` и `umount(8)`, которые занимаются монтированием и размонтированием файловых систем, и `su(1)`, которая позволяет пользователю запускать оболочку под различными UID. В качестве примера программы с полномочиями setgid можно привести `wall(1)`, которая записывает сообщение на все терминалы, владельцами которых является группа `tty` (обычно она является владельцем каждого терминала).

В разделе 8.5 уже отмечалось, что программа из листинга 8.2 должна быть запущена под учетной записью `root`, чтобы получить доступ к файлу `/etc/shadow`. Эту программу можно сделать запускаемой любым пользователем, назначив ее set-user-ID-root-программой:

```
$ su
Password:
# chown root check_password   Закрепление владения этой программой за root
# chmod u+s check_password    С установленным битом set-user-ID
# ls -l check_password
-rwsr-xr-x    1 root   users    18150 Oct 28 10:49 check_password
# exit
$ whoami                      Это непривилегированный пользователь
mtk
$ ./check_password            Но теперь мы можем получить доступ к файлу
Username: avr                 теневых паролей, используя эту программу
Password:
Successfully authenticated: UID=1001
```

Технология set-user-ID/set-group-ID является полезным и эффективным средством, но при недостаточно тщательно спроектированных приложениях может создать бреши в системе безопасности. Практические наработки, которых следует придерживаться при написании программ с полномочиями setuid и setgid, перечисляются в главе 38.

9.4. Сохраненный set-user-ID и сохраненный set-group-ID

Сохраненный установленный идентификатор пользователя (set-user-ID) и сохраненный установленный идентификатор группы (set-group-ID) предназначены для применения с программами с полномочиями setuid и setgid. При выполнении программы наряду со многими другими происходят и следующие действия.

1. Если у исполняемого файла установлен бит полномочий set-user-ID (set-group-ID), то действующий пользовательский (групповой) ID процесса становится таким же, что и у владельца исполняемого файла. Если у исполняемого файла не установлен бит полномочий set-user-ID (set-group-ID), то действующий пользовательский (групповой) ID процесса не изменяется.
2. Значения для сохраненного set-user-ID и сохраненного set-group-ID копируются из соответствующих действующих идентификаторов. Это копирование осуществляется независимо от того, был ли у выполняемого на данный момент файла установлен бит set-user-ID или бит set-group-ID.

Рассмотрим пример того, что происходит в ходе вышеизложенных действий. Предположим, что процесс, чьи пользовательские идентификаторы — реальный, действительный и сохраненный set-user-ID — равны `1000`, выполняет set-user-ID-программу, владельцем которой является `root` (UID равен `0`). После выполнения пользовательские идентификаторы процесса будут изменены следующим образом:

```
real=1000 effective=0 saved=0   (реальный=1000 действующий=0 сохраненный=0)
```

Различные системные вызовы позволяют set-user-ID-программе переключать ее действующий пользовательский идентификатор между значениями реального UID и сохраненного set-user-ID. Аналогичные системные вызовы позволяют программе с полномочиями setgid изменять ее действующий GID. Таким образом, программа может временно сбросить и восстановить любые полномочия, связанные с пользовательским (групповым) идентификатором исполняемого файла. (Иными словами, она может перемещаться между состояниями потенциальной привилегированности

и фактической работы с полномочиями.) При более подробном рассмотрении вопроса в разделе 38.2 выяснится, что требования безопасного программирования гласят: программа должна работать под непривилегированным (реальным) ID до тех пор, пока ей на самом деле не понадобятся права привилегированного (то есть сохраненного установленного) ID.

Иногда в качестве синонимов сохраненного установленного идентификатора пользователя и сохраненного установленного идентификатора группы употребляются выражения «сохраненный идентификатор пользователя» и «сохраненный идентификатор группы».

Сохраненные установленные идентификаторы являются нововведениями, появившимися в System V и принятыми в POSIX. В выпусках BSD, предшествующих 4.4, они не предоставлялись. В исходном стандарте POSIX.1 поддержка этих идентификаторов была необязательной, но в более поздних стандартах (начиная с FIPS 151-1 в 1988 году) стала обязательной.

9.5. Пользовательские и групповые ID файловой системы

В Linux для определения полномочий при выполнении операций, связанных с файловой системой (открытие файла, изменение его собственника и модификация полномочий), применяются не действующие пользовательские и групповые ID, а пользовательские и групповые ID файловой системы. Они используются в этом качестве наряду с дополнительными групповыми идентификаторами. (Действующие идентификаторы по-прежнему, как и в других реализациях UNIX, используются для других, ранее рассмотренных целей.)

Обычно пользовательские и групповые идентификаторы файловой системы имеют те же значения, что и соответствующие действующие идентификаторы (и, таким образом, нередко совпадают с соответствующими реальными идентификаторами). Более того, когда изменяется действующий пользовательский или групповой ID (либо посредством системного вызова, либо из-за выполнения программы с полномочиями setuid или setgid), изменяется, получая такое же значение, и соответствующий идентификатор файловой системы. Поскольку идентификаторы файловой системы следуют таким образом за действующими идентификаторами, это означает, что Linux при проверке привилегий и полномочий фактически ведет себя точно так же, как любая другая реализация UNIX. Лишь когда используются два характерных для Linux системных вызова — `setfsuid()` и `setfsgid()`, поведение Linux отличается от поведения других реализаций UNIX, и ID файловой системы отличаются от соответствующих действующих идентификаторов.

Зачем в Linux предоставляются идентификаторы файловой системы и при каких обстоятельствах нам понадобятся разные значения для действующих идентификаторов и идентификаторов файловой системы? Причины главным образом имеют исторические корни. Идентификаторы файловой системы впервые появились в Linux 1.2. В этой версии ядра один процесс мог отправлять сигнал другому, лишь если действующий идентификатор пользователя отправителя совпадал с реальным или действующим идентификатором пользователя целевого процесса. Это повлияло на некоторые программы, например на программу сервера Linux NFS (Network File System — сетевая файловая система), которой нужна была возможность доступа к файлам, как будто у нее есть действующие идентификаторы соответствующих клиентских процессов. Но, если бы NFS-сервер изменял свой действующий идентификатор пользователя, он стал бы уязвим от сигналов непривилегированных пользовательских процессов. Для

предотвращения этой возможности были придуманы отдельные пользовательские и групповые ID файловой системы. Оставляя неизмененными свои действующие идентификаторы, но изменяя идентификаторы файловой системы, NFS-сервер может выдавать себя за другого пользователя с целью обращения к файлам без уязвимости от сигналов пользовательских процессов.

Начиная с версии ядра 2.0, в Linux приняты установленные SUSv3 правила относительно разрешений на отправку сигналов. Эти правила не касаются действующего ID пользователя целевого процесса (см. раздел 20.5). Таким образом, наличие идентификатора файловой системы утратило свою актуальность (теперь процесс может удовлетворить возникающие потребности, изменив значение действующего пользовательского ID на ID привилегированного пользователя (и обратно) путем разумного применения системных вызовов, рассматриваемых далее в этой главе). Но эта возможность по-прежнему предусмотрена для сохранения совместимости с существующим программным обеспечением.

Поскольку идентификаторы файловой системы теперь уже считаются некой экзотикой и обычно значения совпадают с соответствующими действующими идентификаторами, во всем остальном тексте книги описания различных проверок полномочий по доступу к файлам, а также установок прав на владение новыми файлами будут даваться в понятиях действующих пользовательских ID процесса. Хотя в Linux по-прежнему для этих целей реально используются принадлежащие процессу идентификаторы файловой системы, на практике их наличие редко вносит в действия какую-либо существенную разницу.

9.6. Дополнительные групповые идентификаторы

Дополнительные групповые идентификаторы представляют собой набор дополнительных групп, которым принадлежит процесс. Новый процесс наследует эти идентификаторы от своего родительского процесса. Оболочка входа в систему получает свои дополнительные идентификаторы групп из файла групп системы. Как уже ранее отмечалось, эти идентификаторы используются в совокупности с действующими идентификаторами и идентификаторами файловой системы для определения полномочий по доступу к файлам, IPC-объектам System V и другим системным ресурсам.

9.7. Извлечение и модификация идентификаторов процессов

В Linux для извлечения и изменения различных пользовательских и групповых идентификаторов, рассматриваемых в данной главе, предоставляется ряд системных вызовов и библиотечных функций. В SUSv3 определяется только часть этих API. Из оставшихся некоторые широко доступны в иных реализациях UNIX, а другие характерны только для Linux. По мере рассмотрения каждого интерфейса мы также будем обращать внимание на вопросы портируемости. Ближе к концу главы в табл. 9.1 мы перечислим операции всех интерфейсов, используемых для изменения идентификаторов процессов.

В качестве альтернативы применения системных вызовов, описываемых на следующих страницах, идентификаторы любого процесса могут быть определены путем анализа строк `Uid`, `Gid` и `Groups`, предоставляемых Linux-файлом `/proc/PID/status`. В строках `Uid` и `Gid` перечисляются идентификаторы в следующем порядке: реальный, действующий, сохраненный установленный и идентификатор файловой системы.

В следующих разделах будет использоваться традиционное определение привилегированного процесса как одного из процессов, чей действительный идентификатор пользователя имеет значение `0`. Но, как описывается в главе 39, в Linux понятие полномочий привилегированного пользователя разбивается на отдельные составляющие. К рассмотрению нашего вопроса относительно всех системных вызовов, применяемых для изменения пользовательских и групповых идентификаторов процесса, имеют отношение две характеристики.

- `CAP_SETUID` позволяет процессу произвольно менять свои пользовательские идентификаторы.
- `CAP_SETGID` позволяет процессу произвольно изменять свои групповые идентификаторы.

9.7.1. Извлечение и изменение реальных, действующих и сохраненных установленных идентификаторов

В следующих абзацах мы рассмотрим системные вызовы, извлекающие и изменяющие реальные, действующие и сохраненные установленные идентификаторы. Существует несколько системных вызовов, выполняющих эти задачи, и в некоторых случаях их функциональные возможности перекрываются, отражая тот факт, что различные системные вызовы произошли от разных реализаций UNIX.

Извлечение реальных и действующих идентификаторов

Системные вызовы `getuid()` и `getgid()` возвращают соответственно реальный пользовательский идентификатор и реальный идентификатор группы вызывающего процесса. Системные вызовы `geteuid()` и `getegid()` выполняют соответствующие задачи для действующих идентификаторов. Эти системные вызовы всегда завершаются успешно.

```
#include <unistd.h>

uid_t getuid(void);
```

Возвращает реальный идентификатор пользователя вызывающего процесса

```
uid_t geteuid(void);
```

Возвращает действительный идентификатор пользователя вызывающего процесса

```
gid_t getgid(void);
```

Возвращает реальный идентификатор группы вызывающего процесса

```
gid_t getegid(void);
```

Возвращает действующий идентификатор группы вызывающего процесса

Изменение действующих идентификаторов

Системный вызов `setuid()` изменяет действующий идентификатор пользователя, и, возможно, реальный ID пользователя и сохраненный установленный ID пользователя вызывающего процесса, присваивая значение, заданное его аргументом `uid`. Системный вызов `setgid()` выполняет аналогичную задачу для соответствующих идентификаторов группы.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Оба возвращают 0 при успешном завершении и –1 — при ошибке

Правила, согласно которым процесс может вносить изменения в свои полномочия с помощью `setuid()` и `setgid()`, зависят от того, привилегированный ли он (то есть имеет ли он действующий пользовательский идентификатор, равный `0`). К системному вызову `setuid()` применяются следующие правила.

1. Когда вызов `setuid()` осуществляется непривилегированным процессом, изменяется только действующий пользовательский идентификатор процесса. Кроме того, он может быть изменен только на то же самое значение, которое имеется либо у реального идентификатора пользователя, либо у сохраненного установленного идентификатора пользователя. (Попытки нарушить это ограничение приводят к выдаче ошибки `EPERM`.) Это означает, что для непривилегированных пользователей данный вызов полезен лишь при выполнении set-user-ID-программы, поскольку при выполнении обычной программы у процесса обнаруживаются одинаковые по значению реальный, действующий и сохраненный установленный пользовательские идентификаторы. В некоторых реализациях, уходящих корнями в BSD, вызовы `setuid()` или `setgid()` непривилегированным процессом имеют иную семантику, отличающуюся от применяемой другими реализациями UNIX. В BSD вызовы изменяют реальный, действующий и сохраненный установленный идентификаторы на значение текущего реального или действующего идентификатора.
2. Когда привилегированный процесс выполняет `setuid()` с ненулевым аргументом, все идентификаторы — реальный, действующий и сохраненный установленный пользовательский ID — получают значение, указанное в аргументе `uid`. Последствия необратимы, поскольку, как только идентификатор у привилегированного процесса таким образом изменится, процесс утратит все полномочия и не сможет впоследствии воспользоваться `setuid()`, чтобы снова переключить идентификаторы на нуль. Если такой исход нежелателен, то вместо `setuid()` нужно воспользоваться либо `seteuid()`, либо `setreuid()` — системными вызовами, которые вскоре будут рассмотрены.

Правила, регулирующие изменения, которые могут быть внесены с помощью `setgid()` в идентификаторы группы, аналогичны рассмотренным, но с заменой `setuid()` на `setgid()`, а группы на пользователя. С этими изменениями правило 1 применимо без оговорок. В правиле 2, поскольку изменение группового идентификатора не вызывает потери полномочий (которые определяются значением действующего пользовательского идентификатора, UID), привилегированные программы могут задействовать `setgid()` для свободного изменения групповых идентификаторов на любые желаемые значения.

Следующий вызов является предпочтительным способом для set-user-ID-root-программы, чей действующий UID в этот момент равен `0`, безвозвратно сбросить все полномочия (путем установки как действующего, так и сохраненного установленного пользовательского идентификатора на то же значение, которое имеется у реального UID):

```
if (setuid(getuid()) == -1)
    errExit("setuid");
```

Set-user-ID-программа, принадлежащая пользователю, отличному от `root`, может применять `setuid()` для переключения действующего UID между значениями реального UID и сохраненного установленного UID по соображениям безопасности,

рассмотренным в разделе 9.4. Но для этой цели предпочтительнее обратиться к системному вызову `seteuid()`, поскольку он действует точно так же, независимо от того, принадлежит пользователю по имени `root` set-user-ID-программа или нет.

Процесс может воспользоваться системным вызовом `seteuid()` для изменения своего действующего пользовательского идентификатора (на значение, указанное в аргументе `euid`) и системным вызовом `setegid()` для изменения его действующего группового идентификатора (на значение, указанное в аргументе `egid`).

```
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Оба возвращают при успешном завершении `0`, а при ошибке — `-1`

Изменения, которые процесс может вносить в свои действующие идентификаторы с использованием `seteuid()` и `setegid()`, регулируются следующими правилами.

1. Непривилегированный процесс может изменять действующий идентификатор, присваивая ему только то значение, которое соответствует реальному или сохраненному установленному идентификатору. (Иными словами, для непривилегированного процесса функции `seteuid()` и `setegid()` произведут тот же эффект, что и функции `setuid()` и `setgid()` соответственно, за исключением ранее упомянутых вопросов портируемости на BSD-системы.)
2. Привилегированный процесс может изменять действующий идентификатор, присваивая ему любое значение. Если привилегированный процесс применяет `seteuid()` для изменения своего действующего пользовательского идентификатора на ненулевое значение, то он перестает быть привилегированным (но в состоянии вернуть себе полномочия в силу предыдущего правила).

Использование `seteuid()` является предпочтительным методом для программ с полномочиями setuid и setgid с целью временного сброса и последующего восстановления полномочий. Рассмотрим пример.

```
euid = geteuid();               /* Сохранение исходного действующего UID
                                   (совпадает с установленным UID) */
if (seteuid(getuid()) == -1)    /* Сброс полномочий */
    errExit("seteuid");
if (seteuid(euid) == -1)        /* Возвращение полномочий */
    errExit("seteuid");
```

Изначально происходящие из BSD, функции `seteuid()` и `setegid()` теперь определены в SUSv3 и встречаются во многих реализациях UNIX.

В старых версиях библиотеки GNU C (glibc 2.0 и более ранние) выражение seteuid(euid) было реализовано в виде setreuid(–1, euid). В современных версиях glibc функция seteuid(euid) реализована в виде setresuid(–1, euid, –1). (Функции setreuid(), setresuid() и их аналоги по работе с групповыми идентификаторами будут вскоре рассмотрены.) Обе реализации позволяют нам указать в качестве euid такое же значение, которое на данный момент имеется у действующего идентификатора пользователя (то есть не требовать изменения). Но в SUSv3 такое поведение для seteuid() не определено, и получить его в некоторых реализациях UNIX невозможно. Как правило, различие в поведении разных реализаций не проявляется, так как в обычных условиях действующий идентификатор пользователя имеет то же самое значение, которое имеется либо у реального идентификатора пользо-

вателя, либо у сохраненного установленного идентификатора пользователя. (Единственный способ, позволяющий сделать в Linux действующий ID пользователя отличным как от реального ID пользователя, так и от сохраненного установленного ID пользователя, предусматривает применение нестандартного системного вызова setresuid().)

Во всех версиях glibc (включая современные) setegid(egid) реализуется в виде setregid(–1, egid). Как и в случае использования seteuid(), это означает, что мы можем указать для egid такое же значение, которое на данный момент имеется у действующего идентификатора группы, хотя такое поведение не указано в SUSv3. Это также означает, что setegid() изменяет сохраненный установленный ID группы, если для действующего ID группы установлено значение, отличное от имеющегося на данный момент у реального ID группы. (То же самое можно сказать и о более старых реализациях seteuid(), использующих setreuid().) Это поведение также не указано в SUSv3.

Изменение реальных и действующих идентификаторов

Системный вызов `setreuid()` позволяет вызывающему процессу независимо изменять значение его реального и действующего пользовательского идентификатора. Системный вызов `setregid()` выполняет аналогичную задачу для реального и действующего идентификатора группы.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Оба при успешном завершении возвращают `0` или `-1` при ошибке

Первым аргументом для каждого из этих системных вызовов является новый реальный идентификатор. Вторым аргументом является новый действующий идентификатор. Если нужно изменить только один идентификатор, для другого аргумента можно указать значение `-1`.

Первоначально появившись в BSD, теперь `setreuid()` и `setregid()` указаны в SUSv3 и доступны в большинстве реализаций UNIX.

К изменениям, возможным при использовании `setreuid()` и `setregid()`, как и других системных вызовов, рассматриваемых в этом разделе, применяются определенные правила. Они будут рассмотрены с точки зрения `setreuid()` с учетом того, что для `setregid()` они аналогичны, за исключением некоторых оговорок.

1. Непривилегированный процесс может присвоить реальному идентификатору пользователя только имеющееся на данный момент значение реального (то есть оставить его без изменений) или действующего идентификатора пользователя. Для действующего идентификатора пользователя может быть установлено только имеющееся на данный момент значение реального ID пользователя, действующего ID пользователя (то есть остается без изменений) или сохраненного установленного ID пользователя.

   В SUSv3 говорится, что возможность использования setreuid() для изменения значения реального ID пользователя на текущее значение реального, действующего или сохраненного установленного ID пользователя не определена, и подробности того, какие в точности изменения могут вноситься в значение реального ID пользователя, варьируются в зависимости от реализации. В SUSv3 дается описание несколько отличающегося поведения setregid(): непривилегированный процесс может установить для реального ID группы текущее значение сохраненного установленного ID группы или для действующего ID группы текущее

значение либо реального, либо сохраненного установленного ID группы. Подробности того, какие в точности изменения могут вноситься в значение реального идентификатора группы, также варьируются в зависимости от реализации.

2. Привилегированный процесс может вносить в идентификаторы любые изменения.
3. Как для привилегированного, так и для непривилегированного процесса сохраненный установленный идентификатор пользователя также устанавливается на то же самое значение, которое имеется у (нового) действующего идентификатора пользователя, при соблюдении одного из следующих условий:
   1) значение `ruid` не равно `-1` (то есть для реального идентификатора пользователя устанавливается в точности то же значение, которое у него уже имелось);
   2) для действующего идентификатора пользователя устанавливается значение, отличающееся от того, которое имелось у реального идентификатора пользователя до вызова.

   С другой стороны, если процесс использует `setreuid()` только для изменения действующего идентификатора пользователя на то же значение, которое имеется на данный момент у реального ID пользователя, то сохраненный установленный ID пользователя остается неизмененным, и последующий вызов `setreuid()` (или `seteuid()`) может восстановить действующий ID пользователя, присвоив ему значение сохраненного установленного ID пользователя. (В SUSv3 не определяется влияние от применения `setreuid()` и `setregid()` на сохраненные установленные идентификаторы пользователя, но в SUSv4 указывается только что рассмотренное поведение.)

Третье правило предоставляет способ, позволяющий set-user-ID-программам лишаться своего привилегированного состояния безвозвратно, с помощью следующего вызова:

```
setreuid(getuid(), getuid());
```

Процесс с установленным идентификатором привилегированного пользователя (set-user-ID-root), которому нужно изменить как свои пользовательские, так и групповые полномочия на произвольные значения, должен вызвать сначала `setregid()`, а затем `setreuid()`. Если вызов делается в обратном порядке, вызов `setregid()` даст сбой, потому что после вызова `setregid()` программа уже не будет привилегированной. Те же замечания применимы к системным вызовам `setresuid()` и `setresgid()` (рассматриваемым ниже), если они используются для достижения аналогичной цели.

Выпуски BSD до 4.3BSD включительно не имели сохраненного установленного идентификатора пользователя и сохраненного установленного идентификатора группы (наличие которых теперь предписывается в SUSv3). Вместо этого в BSD системные вызовы setreuid() и setregid() позволяли процессу сбрасывать и восстанавливать полномочия, меняя местами значения реального и действующего идентификаторов в обе стороны. В результате возникал нежелательный побочный эффект изменения реального идентификатора пользователя с целью изменения действительного идентификатора пользователя.

Извлечение реального, действительного и сохраненного установленного идентификаторов

Во многих реализациях UNIX процесс не может напрямую извлечь (или изменить) свой сохраненный установленный идентификатор пользователя и сохраненный установленный идентификатор группы. Но в Linux предоставляются два нестандартных системных вызова — `getresuid()` и `getresgid()`. Они позволяют нам решить именно эту задачу.

```
#define _GNU_SOURCE
#include <unistd.h>

int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

Оба при успешном завершении возвращают 0 или –1 при ошибке

Системный вызов getresuid() возвращает текущие значения принадлежащих вызывающему процессу реального, действующего и сохраненного установленного идентификатора пользователя в те места, которые указываются тремя его аргументами. Системный вызов getresgid() делает то же самое для соответствующих групповых идентификаторов.

Изменение реального, действительного и сохраненного установленного идентификаторов

Системный вызов setresuid() позволяет вызывающему процессу независимым образом изменять значения всех его трех пользовательских идентификаторов. Новые значения для каждого из его пользовательских идентификаторов указываются тремя аргументами системного вызова. Аналогичные задачи для групповых идентификаторов может выполнять системный вызов setresgid().

```
#define _GNU_SOURCE
#include <unistd.h>

int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

Оба при успешном завершении возвращают 0 или –1 при ошибке

Если не нужно изменять все идентификаторы, для того из них, который не требует изменений, указывается значение –1 аргумента. Например, следующий вызов эквивалентен seteuid(x):

```
setresuid(-1, x, -1);
```

В отношении изменений, которые могут производиться с использованием setresuid(), действуют следующие правила (они распространяются и на вызов setresgid()).

1. Непривилегированный процесс может установить для любого из своих пользовательских идентификаторов — реального, действующего и сохраненного установленного — любое из значений его текущих ID: реального, действительного или сохраненного установленного ID пользователя.
2. Привилегированный процесс может вносить произвольные изменения в свой реальный идентификатор пользователя, действительный идентификатор пользователя и сохраненный установленный идентификатор пользователя.
3. Независимо от того, вносит ли вызов какие-либо изменения в другие идентификаторы, идентификатор файловой системы всегда установлен на то же самое значение, что и (возможно, уже новый) действительный ID пользователя.

Вызовы setresuid() и setresgid() делают «все или ничего» Либо успешно изменяются все запрошенные идентификаторы, либо не изменяется ни один из них. (То же самое

можно сказать и о других системных вызовах, рассмотренных в этой главе и изменяющих сразу несколько идентификаторов.)

Хотя `setresuid()` и `setresgid()` предоставляют самый очевидный API для изменения идентификаторов процесса, невозможно применять их портируемым образом в приложениях — они не определены в SUSv3 и доступны только в немногих других реализациях UNIX.

9.7.2. Извлечение и изменение идентификаторов файловой системы

Все ранее рассмотренные системные вызовы, изменяющие действующие пользовательские или групповые идентификаторы процесса, также всегда изменяют и соответствующий идентификатор файловой системы. Чтобы изменить идентификаторы файловой системы независимо от действующих идентификаторов, следует применить два характерный только для Linux системных вызова: `setfsuid()` и `setfsgid()`.

```
#include <sys/fsuid.h>

int setfsuid(uid_t fsuid);
                                    Всегда возвращает предыдущий пользовательский
                                    идентификатор файловой системы

int setfsgid(gid_t fsgid);
                                    Всегда возвращает предыдущий групповой
                                    идентификатор файловой системы
```

Системный вызов `setfsuid()` изменяет пользовательский идентификатор файловой системы процесса на значение, указанное в аргументе `fsuid`. Системный вызов `setfsgid()` изменяет групповой идентификатор файловой системы на значение, указанное в аргументе `fsgid`.

Здесь также применяются некоторые правила. Правила для `setfsgid()` аналогичны правилам для `setfsuid()` и звучат таким образом.

1. Непривилегированный процесс может установить пользовательский идентификатор файловой системы на текущее значение реального идентификатора пользователя, действующего идентификатора пользователя, идентификатора пользователя файловой системы (то есть оставить все без изменений) или сохраненного установленного идентификатора пользователя.
2. Привилегированный процесс может установить идентификатор пользователя файловой системы на любое значение.

Реализация этих вызовов слегка не доработана. Для начала следует отметить отсутствие соответствующих системных вызовов, извлекающих текущее значение идентификаторов файловой системы. Кроме того, в системных вызовах отсутствует проверка на возникновение ошибки; если непривилегированный процесс предпринимает попытку установить для своего идентификатора файловой системы неприемлемое значение, она игнорируется. Возвращаемым значением для каждого из этих системных вызовов является предыдущее значение соответствующего идентификатора файловой системы, независимо от успешности выполнения системного вызова. Таким образом, у нас есть способ определения текущих значений идентификаторов файловой системы, но только с одновременной попыткой (либо успешной, либо нет) их изменения.

Использование системных вызовов `setfsuid()` и `setfsgid()` больше не имеет в Linux никакой практической необходимости, и его следует избегать в тех приложениях, которые разрабатываются с прицелом на портирование для работы в других реализациях UNIX.

9.7.3. Извлечение и изменение дополнительных групповых идентификаторов

Системный вызов `getgroups()` записывает в массив, указанный в аргументе `grouplist`, набор групп, в которые на данный момент входит вызывающий процесс.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Возвращает при успешном завершении количество групповых идентификаторов, помещенное в *grouplist*, а при ошибке — -1

В Linux, как и в большинстве реализаций UNIX, `getgroups()` просто возвращает дополнительные групповые идентификаторы вызывающего процесса. Но SUSv3 также разрешает реализации включать в возвращаемый `grouplist` действующий групповой идентификатор вызывающего процесса.

Вызывающая программа должна выделить память под массив `grouplist` и указать его длину в аргументе `gidsetsize`. При успешном завершении `getgroups()` возвращает количество групповых идентификаторов, помещенных в `grouplist`.

Если количество групп, в который входит процесс, превышает значение, указанное в `gidsetsize`, системный вызов `getgroups()` возвращает ошибку (`EINVAL`). Во избежание этого можно задать для массива `grouplist` значение, большее на единицу (для разрешения портируемости при возможном включении действующего группового идентификатора), чем значение константы `NGROUPS_MAX` (определенной в заголовочном файле `<limits.h>`). Эта константа определяет максимальное количество дополнительных групп, в которые может входить процесс. Таким образом, `grouplist` можно объявить с помощью следующего выражения:

```
gid_t grouplist[NGROUPS_MAX + 1];
```

В ядрах Linux, предшествующих версии 2.6.4, у `NGROUPS_MAX` было значение `32`. Начиная с версии 2.6.4, значение у `NGROUPS_MAX` стало равно `65536`.

Приложение может также определить предельное значение `NGROUPS_MAX` в ходе своего выполнения следующими способами:

- вызвать `sysconf(_SC_NGROUPS_MAX)` (использование `sysconf()` рассматривается в разделе 11.2);
- считать ограничение из предназначенного только для чтения и характерного только для Linux файла `/proc/sys/kernel/ngroups_max`. Этот файл предоставляется ядрами, начиная с версии 2.6.4.

Кроме этого, приложение может выполнить вызов `getgroups()`, указав в качестве аргумента `gidsetsize` значение `0`. В этом случае `grouplist` не изменяется, но возвращаемое вызовом значение содержит количество групп, в которые входит процесс.

Значение, полученное любым из этих способов, применяемых в ходе выполнения приложения, может затем использоваться для динамического выделения памяти под массив `grouplist` с целью последующего вызова `getgroups()`.

Привилегированный процесс может изменить свой набор дополнительных групповых идентификаторов, выполнив `setgroups()` и `initgroups()`.

```
#define _BSD_SOURCE
#include <grp.h>

int setgroups(size_t gidsetsize, const gid_t *grouplist);
int initgroups(const char *user, gid_t group);
```

Оба возвращают при успешном завершении `0`, а при ошибке — `-1`

Системный вызов `setgroups()` может заменить дополнительные групповые идентификаторы вызывающего процесса набором, заданным в массиве `grouplist`. Количество групповых идентификаторов в массиве аргумента `grouplist` указывается в аргументе `gidsetsize`.

Функция `initgroups()` инициализирует дополнительные групповые идентификаторы вызывающего процесса путем сканирования файла `/etc/group` и создания списка групп, в которые входит указанный пользователь. Кроме того, к набору дополнительных групповых идентификаторов процесса добавляется групповой идентификатор, указанный в аргументе `group`.

В основном `initgroups()` используется программами, создающими сеансы входа в систему. Например `login(1)` устанавливает различные атрибуты процесса перед запуском оболочки входа пользователя в систему. Такие программы обычно получают значение, используемое для аргумента `group`, путем считывания поля идентификатора группы из пользовательской записи в файле паролей. Это создает небольшую путаницу, поскольку идентификатор группы из файла паролей на самом деле не относится к дополнительным групповым идентификаторам, но, как бы то ни было, `initgroups()` именно так обычно и применяется.

Хотя в SUSv3 системные вызовы `setgroups()` и `initgroups()` не фигурируют, они доступны во всех реализациях UNIX.

9.7.4. Сводный обзор вызовов, предназначенных для изменения идентификаторов процесса

В табл. 9.1 дается сводная информация о действиях различных системных вызовов и библиотечных функций, используемых для изменения идентификаторов и полномочий процесса.

Таблица 9.1. Сводные данные по интерфейсам, используемым для изменения идентификаторов процесса

Интерфейс	Назначение и действие в:		Портируемость
	Непривилегированном процессе	Привилегированном процессе	
setuid(u) setgid(g)	Изменение действующего ID на такое же значение, что и у текущего реального или сохраненного установленного ID	Изменение реального, действующего и сохраненного установленного ID на любое (единое) значение	Вызовы указываются в SUSv3; у вызовов, берущих происхождение от BSD, другая семантика

Интерфейс	Назначение и действие в:		Портируемость
	Непривилегированном процессе	Привилегированном процессе	
seteuid(e) setegid(e)	Изменение действующего ID на такое же значение, что и у текущего реального или сохраненного установленного ID	Изменение действующего ID на любое значение	Вызовы указываются в SUSv3
setreuid(r, e) setregid(r, e)	(Независимое) изменение реального ID на такое же значение, что и у текущего реального или действующего ID, и действующего ID на такое же значение, что у текущего реального, действующего или сохраненного установленного ID	(Независимое) изменение реального и действующего ID на любое значение	Вызовы указываются в SUSv3, но в различных реализациях работают по-разному
setresuid(r, e, s) setresgid(r, e, s)	Изменение ID файловой системы на то же значение, что и у текущего реального, действительного, сохраненного установленного ID или ID файловой системы	Изменение ID файловой системы на любое значение	Вызовы характерны только для Linux
setgroups(n, l)	Этот системный вызов не может быть сделан из непривилегированных процессов	Установка для дополнительных групповых ID любых значений	Этот системный вызов в SUSv3 не фигурирует, но доступен во всех реализациях UNIX

На рис. 9.1 представлен графический обзор той же информации, которая приводится в табл. 9.1. Отображенная на схеме информация касается вызовов, изменяющих пользовательские идентификаторы, но к изменениям групповых идентификаторов применяются точно такие же правила. Обратите внимание на следующую информацию, дополняющую сведения, изложенные в табл. 9.1.

- Имеющиеся в glibc реализации `seteuid()` и `setegid()` также позволяют устанавливать для действующего идентификатора такое же значение, какое у него и было, но эта особенность в SUSv3 не упоминается.
- Если при вызовах `setreuid()` и `setregid()` как привилегированными, так и непривилегированными процессами, до осуществления вызовов значение `r` (реального идентификатора) не равно `-1` или для `e` (действующего идентификатора) указано значение, отличное от значения реального идентификатора, то сохраненный установленный пользовательский или сохраненный установленный групповой ID также устанавливаются на то же значение, что и у нового действующего идентификатора. (В SUSv3 не указано, что `setreuid()` и `setregid()` вносят изменения в сохраненные установленные ID.)
- Когда изменяется действующий пользовательский (групповой) идентификатор, характерный для Linux пользовательский (групповой) идентификатор файловой системы изменяется, принимая то же самое значение.
- Вызовы `setresuid()` всегда изменяют пользовательский идентификатор файловой системы, присваивая ему такое же значение, что и у действующего пользовательского

ID, независимо от того, изменяется ли вызовом действующий пользовательский идентификатор. Вызовы `setresgid()` делают то же самое в отношении групповых идентификаторов файловой системы.

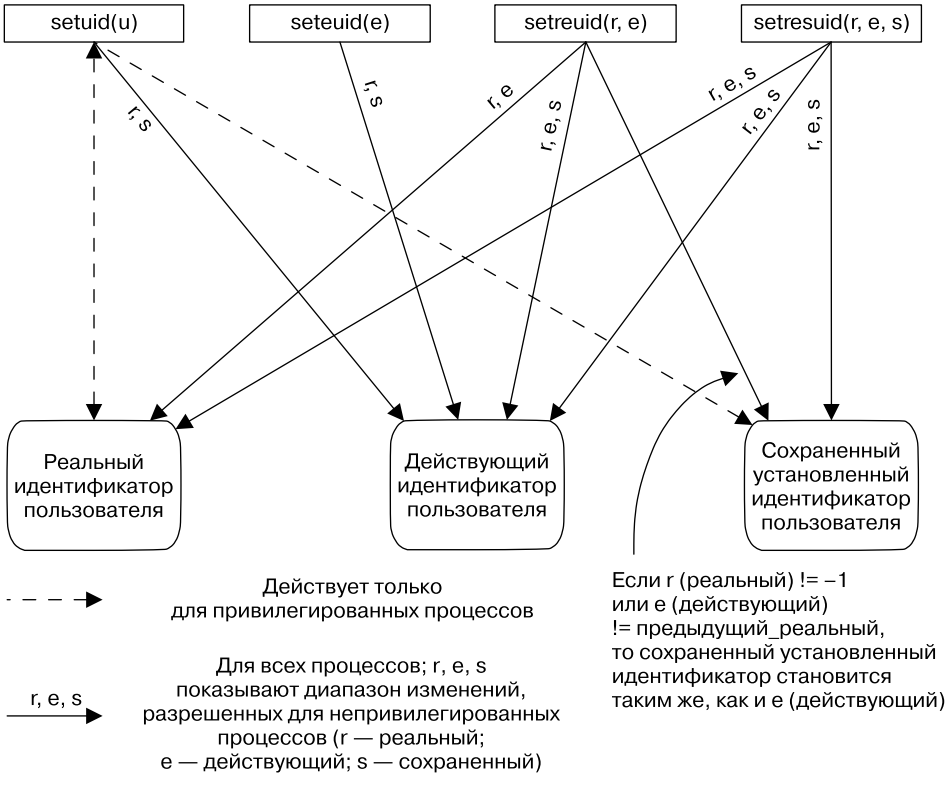


Рис. 9.1. Действия функций, изменяющих полномочия процесса, связанные с его пользовательскими идентификаторами

9.7.5. Пример: вывод на экран идентификаторов процесса

Программа, показанная в листинге 9.1, использует системные вызовы и библиотечные функции, рассмотренные на предыдущих страницах, для извлечения всех пользовательских и групповых идентификаторов процесса и вывода их на экран.

Листинг 9.1. Отображение на экране всех пользовательских и групповых идентификаторов процесса

proccred/idshow.c

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/fsuid.h>
#include <limits.h>
#include "ugid_functions.h"   /* userNameFromId() и groupNameFromId() */
#include "tlpi_hdr.h"

#define SG_SIZE (NGROUPS_MAX + 1)
```

```
int
main(int argc, char *argv[])
{
    uid_t ruid, euid, suid, fsuid;
    gid_t rgid, egid, sgid, fsgid;
    gid_t suppGroups[SG_SIZE];
    int numGroups, j;
    char *p;

    if (getresuid(&ruid, &euid, &suid) == -1)
        errExit("getresuid");
    if (getresgid(&rgid, &egid, &sgid) == -1)
        errExit("getresgid");

    /* Попытки изменения идентификаторов файловой системы для непривилегированных
       процессов всегда игнорируются, но даже при этом следующие вызовы
       возвращают текущие идентификаторы файловой системы */

    fsuid = setfsuid(0);
    fsgid = setfsgid(0);

    printf("UID: ");
    p = userNameFromId(ruid);
    printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) ruid);
    p = userNameFromId(euid);
    printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) euid);
    p = userNameFromId(suid);
    printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) suid);
    p = userNameFromId(fsuid);
    printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsuid);
    printf("\n");

    printf("GID: ");
    p = groupNameFromId(rgid);
    printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) rgid);
    p = groupNameFromId(egid);
    printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) egid);
    p = groupNameFromId(sgid);
    printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) sgid);
    p = groupNameFromId(fsgid);
    printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsgid);
    printf("\n");

    numGroups = getgroups(SG_SIZE, suppGroups);
    if (numGroups == -1)
        errExit("getgroups");

    printf("Supplementary groups (%d): ", numGroups);
    for (j = 0; j < numGroups; j++) {
        p = groupNameFromId(suppGroups[j]);
        printf("%s (%ld) ", (p == NULL) ? "???" : p, (long) suppGroups[j]);
    }
    printf("\n");

    exit(EXIT_SUCCESS);
}
```

proccred/idshow.c

9.8. Резюме

У каждого процесса имеется несколько пользовательских и групповых идентификаторов. Реальные идентификаторы определяют принадлежность процесса. В большинстве реализаций UNIX для определения полномочий процесса при доступе к таким ресурсам, как файлы, применяются действующие идентификаторы. Но в Linux для определения полномочий доступа к файлам используются идентификаторы файловой системы, а действующие идентификаторы предназначены для проверки других полномочий. (Поскольку идентификаторы файловой системы обычно имеют такие же значения, как и соответствующие действующие идентификаторы, Linux при проверке полномочий доступа к файлам ведет себя точно так же, как и другие реализации UNIX.) Права доступа процесса также определяются с помощью дополнительных групповых идентификаторов — набора групп, в которые входит данный процесс. Извлекать и изменять его пользовательские и групповые ID процессу позволяют различные системные вызовы и библиотечные функции.

Когда запускается set-user-ID-программа, действующий пользовательский идентификатор процесса устанавливается на то значение, которое имеется у владельца файла. Этот механизм позволяет пользователю присвоить идентификатор, а следовательно, и полномочия другого пользователя при запуске конкретной программы. Аналогично, программы с полномочиями setgid изменяют действующий групповой ID процесса, в котором выполняется программа. Сохраненный установленный идентификатор пользователя (saved set-user-ID) и сохраненный установленный идентификатор группы (saved set-group-ID) позволяют программам с полномочиями setuid и setgid временно сбрасывать, а затем позже восстанавливать полномочия.

Пользовательский ID, равный нулю, имеет специальное значение. Обычно его имеет только одна учетная запись с именем `root`. Процессы с действующим идентификатором пользователя, равным нулю, являются привилегированными, то есть освобождаются от многих проверок полномочий, которые обычно выполняются при осуществлении процессом различных системных вызовов (например, при произвольном изменении различных пользовательских и групповых идентификаторов процесса).

9.9. Упражнения

9.1. Предположим, что в каждом из следующих случаев исходный набор пользовательских идентификаторов процесса такой: реальный = `1000`, действующий = `0`, сохраненный = `0`, файловой системы = `0`. Какими станут пользовательские идентификаторы после следующих вызовов:
   1) `setuid(2000);`
   2) `setreuid(-1, 2000);`
   3) `seteuid(2000);`
   4) `setfsuid(2000);`
   5) `setresuid(-1, 2000, 3000)`?

9.2. Является ли привилегированным процесс со следующими идентификаторами пользователя? Обоснуйте ответ.

```
real=0 effective=1000 saved=1000 file-system=1000
```

9.3. Реализуйте функцию `initgroups()`, используя `setgroups()` и библиотечные функции, для извлечения информации из файлов паролей и групп (см. раздел 8.4). Не забудьте, что для возможности вызова `setgroups()` процесс должен быть привилегированным.

9.4. Если процесс, чьи пользовательские идентификаторы имеют одинаковое значение `X`, выполняет set-user-ID-программу, пользовательский идентификатор которой равен `Y` и имеет ненулевое значение, то полномочия процесса устанавливаются следующим образом:

```
real=X effective=Y saved=Y
```

(Мы игнорируем пользовательский идентификатор файловой системы, поскольку его значение следует за действующим идентификатором пользователя.) Запишите соответственно вызовы `setuid()`, `seteuid()`, `setreuid()` и `setresuid()`, которые будут применяться для выполнения таких операций, как:

1) приостановление и возобновление set-user-ID-идентичности (то есть переключение действующего идентификатора пользователя на значение реального пользовательского идентификатора, а затем возвращение к сохраненному установленному идентификатору пользователя);
2) безвозвратный сброс set-user-ID-идентичности (то есть гарантия того, что для действующего пользовательского идентификатора и сохраненного установленного идентификатора пользователя устанавливается значение реального идентификатора пользователя).

(Это упражнение также требует использования вызовов `getuid()` и `geteuid()` для извлечения реального и действующего идентификаторов пользователя.) Учтите, что для некоторых системных вызовов ряд этих операций не может быть выполнен.

9.5. Повторите предыдущее упражнение для процесса выполнения set-user-ID-root-программы, у которой следующий исходный набор идентификаторов процесса:

```
real=X effective=0 saved=0
```

10 Время

При выполнении программы нас могут интересовать два вида времени.

- *Реальное время*. Это время, отмеренное либо от какого-то стандартного момента (*календарное* время), либо от какого-то фиксированного момента в жизни процесса, обычно от его запуска (*затраченное* или *физическое* время). Получение календарного времени требуется в программах, которые, к примеру, ставят отметки времени на записях баз данных или на файлах. Замеры затраченного времени нужны в программах, предпринимающих периодические действия или совершающих регулярные замеры на основе данных, поступающих от внешних устройств ввода.
- *Время процесса*. Это продолжительность использования процессом центрального процессора. Замеры времени процесса нужны для проверки или оптимизации производительности программы либо алгоритма.

Большинство компьютерных архитектур предусматривают наличие встроенных аппаратных часов, позволяющих ядру замерять реальное время и время процесса. В этой главе мы рассмотрим системные вызовы, работающие с обоими видами времени, и библиотечные функции, занимающиеся преобразованием показателей времени между их легко читаемым и внутренним представлениями. Поскольку легко читаемое представление времени зависит от географического местоположения, а также от языковых и культурных традиций, перед рассмотрением этих представлений потребуется разобраться с понятиями часовых поясов и локали.

10.1. Календарное время

В зависимости от географического местоположения, внутри систем UNIX время представляется отмеренным в секундах от начала его отсчета (Epoch): от полуночи 1 января 1970 года, по всемирному координированному времени — Universal Coordinated Time (UTC, ранее называвшемуся средним временем по Гринвичу — Greenwich Mean Time, или GMT). Примерно в это время начали свое существование системы UNIX. Календарное время сохраняется в переменных типа `time_t`, который относится к целочисленным типам, указанным в SUSv3.

> В 32-разрядных системах Linux тип time_t, относящийся к целочисленным типам со знаком, позволяет представлять даты в диапазоне от 13 декабря 1901 года, 20:45:52, до 19 января 2038 года, 03:14:07. (В SUSv3 нет определения отрицательного значения типа time_t.) Таким образом, многие имеющиеся на сегодня 32-разрядные системы UNIX сталкиваются с теоретически возможной проблемой 2038 года, которую им предстоит решить до его наступления, если они в будущем будут выполнять вычисления, связанные с датами. Эту проблему существенно смягчает уверенность в том, что к 2038 году все системы UNIX станут, скорее всего, 64-разрядными или даже более высокой разрядности. Но встроенные 32-разрядные системы, век которых продлится, видимо, намного дольше, чем представлялось поначалу, все же могут столкнуться с этой проблемой. Кроме того, она останется неразрешенной для любых устаревших данных и приложений, работающих со временем в 32-разрядном формате time_t.

Системный вызов `gettimeofday()` возвращает календарное время в буфер, на который указывает значение аргумента *`tv`*.

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `tv` является указателем на структуру следующего вида:

```
struct timeval {
    time_t      tv_sec;   /* Количество секунд с 00:00:00, 1 янв 1970 UTC */
    suseconds_t tv_usec;  /* Дополнительные микросекунды (long int) */
};
```

Хотя для поля `tv_usec` предусмотрена микросекундная точность, конкретная точность возвращаемого в нем значения определяется реализацией, зависящей от архитектуры системы. (Буква «u» в `tv_usec` произошла от сходства с греческой буквой μ («мю»), используемой в метрической системе для обозначения одной миллионной доли.) В современных системах x86-32 (то есть в системах типа Pentium с регистром счетчика меток реального времени — Timestamp Counter, значение которого увеличивается на единицу с каждым тактовым циклом центрального процессора), вызов `gettimeofday()` предоставляет микросекундную точность.

Аргумент *`tz`* в вызове `gettimeofday()` является историческим артефактом. В более старых реализациях UNIX он использовался в целях извлечения для системы информации о часовом поясе (timezone). Сейчас этот аргумент уже вышел из употребления и в качестве его значения нужно всегда указывать `NULL`.

> При предоставлении аргумента tz возвращается структура timezone, в чьих полях содержатся значения, указанные в устаревшем аргументе tz предшествующего вызова settimeofday(). Структура включает два поля: tz_minuteswest и tz_dsttime. Поле tz_minuteswest показывает количество минут, которое нужно добавить в этом часовом поясе (zone) для соответствия UTC; отрицательное значение показывает коррекцию в минутах по отношению к востоку от UTC (например, для ценральноевропейского времени это на один час больше, чем UTC, и поле будет содержать значение –60). Поле tz_dsttime содержит константу, придуманную для представления режима летнего времени — day-light saving time (DST), вводимого в этом часовом поясе. Дело в том, что режим летнего времени в устаревшем аргументе tz не может быть представлен с помощью простого алгоритма. (Это поле в Linux никогда не поддерживалось.) Подробности можно найти на странице руководства gettimeofday(2).

Системный вызов `time()` возвращает количество секунд, прошедших с начала отсчета времени (то есть точно такое же значение, которое возвращает `gettimeofday()` в поле `tv_sec` своего аргумента `tv`).

```
#include <time.h>

time_t time(time_t *timep);
```

Возвращает при успешном завершении количество секунд, прошедших с начала отсчета времени, или `(time_t) -1` при ошибке

Если значение аргумента `timep` не равно `NULL`, количество секунд, прошедшее с начала отсчета времени, также помещается по адресу, который указывает `timep`.

Поскольку `time()` возвращает одно и то же значение двумя способами, и единственной возможной ошибкой, которая может произойти при использовании `time()`, является предоставление неверного адреса в аргументе `timep` (`EFAULT`), зачастую применяется такой вызов (без проверки на ошибку):

```
t = time(NULL);
```

Причина существования двух системных вызовов (time() и gettimeofday()) с практически одинаковым предназначением имеет исторические корни. В ранних реализациях UNIX предоставлялся системный вызов time(). В 4.2BSD добавился более точный системный вызов gettimeofday(). Существование time() в качестве системного вызова теперь считается избыточным; он может быть реализован в виде библиотечной функции, вызывающей gettimeofday().

10.2. Функции преобразования представлений времени

На рис. 10.1 показаны функции, используемые для преобразования между значениями типа `time_t` и другими форматами времени, включая его представления для устройств вывода информации. Эти функции ограждают нас от сложностей, привносимых в такие преобразования часовыми поясами, режимами летнего времени и тонкостями локализации. (Часовые пояса будут рассмотрены в разделе 10.3, а вопросы локали — в разделе 10.4.)

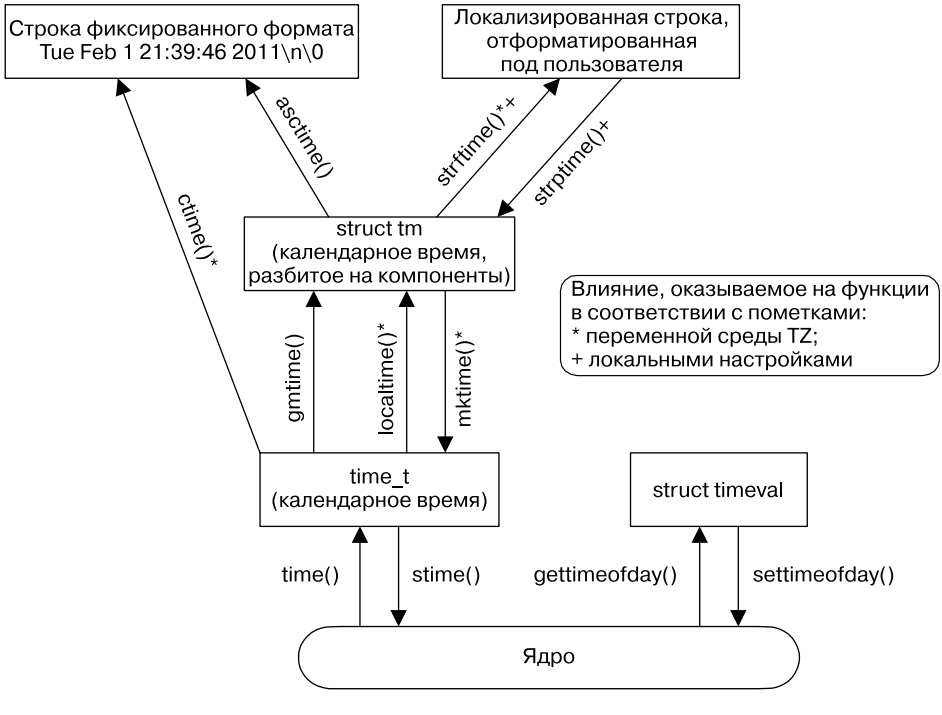


Рис. 10.1. Функции для извлечения календарного времени и работы с ним

10.2.1. Преобразование значений типа time_t к виду, подходящему для устройств вывода информации

Функция `ctime()` предоставляет простой метод преобразования значения типа `time_t` к виду, подходящему для устройств вывода информации.

```
#include <time.h>

char *ctime(const time_t *timep);
```

Возвращает при успешном завершении указатель на статически размещенную строку, которая оканчивается символом новой строки и `\0`, или `NULL` при ошибке

При предоставлении в `timep` указателя в виде значения типа `time_t` функция `ctime()` возвращает 26-байтовую строку, содержащую, как показано в следующем примере, дату и время в стандартной форме:

```
Wed Jun  8 14:22:34 2011
```

Строка включает в себя завершающие элементы: символ новой строки и нулевой байт. При осуществлении преобразования функция `ctime()` автоматически учитывает местный часовой пояс и режим летнего времени. (Порядок определения этих настроек рассматривается в разделе 10.3.) Возвращаемая строка будет статически размещенной; последующие вызовы `ctime()` станут ее перезаписывать.

В SUSv3 утверждается, что вызовы любой из функций — `ctime()`, `gmtime()`, `localtime()` или `asctime()` — могут перезаписать статически размещенную структуру значениями, возвращенными другими функциями. Иными словами, эти функции могут совместно использовать копии возвращенных массивов из символов и структуру `tm`, что и делается в некоторых версиях glibc. Если нужно работать с возвращенной информацией в ходе нескольких вызовов этих функций, следует сохранять локальные копии.

> Реентерабельная версия ctime() предоставляется в виде ctime_r(). (Реентерабельность рассматривается в подразделе 21.1.2.) Эта функция позволяет вызывающему коду задать дополнительный аргумент — указатель на предоставляемый этим кодом буфер для возвращения строки с данными времени. Другие реентерабельные версии функций, упоминаемые в данной главе, ведут себя точно так же.

10.2.2. Преобразования между time_t и разделенным календарным временем

Функции `gmtime()` и `localtime()` преобразуют значение типа `time_t` в так называемое broken-down time, разделенное календарное время (или время, разбитое на компоненты). Это время помещается в статически размещаемую структуру, чей адрес возвращается в качестве результата выполнения функции.

```
#include <time.h>

struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
```

Обе функции при успешном завершении возвращают указатель на статически размещаемую структуру разделенного календарного времени, а при ошибке — NULL

Функция gmtime() выполняет преобразование календарного времени в разделенное время, соответствующее UTC. (Буквы gm происходят от понятия Greenwich Mean Time.) Напротив, функция localtime() учитывает настройки часового пояса и режима летнего времени, чтобы возвратить разбитое на компоненты время, соответствующее местному системному времени.

Реентерабельные версии этих функций предоставляются в виде gmtime_r() и localtime_r().

Структура tm, возвращаемая этими функциями, содержит поля даты и времени, разбитые на отдельные части. Она имеет следующий вид:

```
struct tm {
    int tm_sec;         /* Секунды (0–60) */
    int tm_min;         /* Минуты (0–59) */
    int tm_hour;        /* Часы (0–23) */
    int tm_mday;        /* День месяца (1–31) */
    int tm_mon;         /* Месяц (0–11) */
    int tm_year;        /* Год с 1900 года */
    int tm_wday;        /* День недели (воскресенье = 0)*/
    int tm_yday;        /* День в году (0–365; 1 января = 0)*/
    int tm_isdst;       /* Флаг летнего времени
                           > 0: летнее время действует;
                           = 0: летнее время не действует;
                           < 0: информация о летнем времени недоступна */
};
```

Поле tm_sec может быть расширено до 60 (а не до 59), чтобы учитывать корректировочные секунды, применяемые для правки актуального для человечества календаря под астрономически точный (так называемый тропический) год.

Если определен макрос проверки возможностей _BSD_SOURCE, определяемая библиотекой glibc структура tm также включает два дополнительных поля с более подробной информацией о представленном времени. Первое из них, long int tm_gmtoff, содержит количество секунд, на которое представленное время отстоит на восток от UTC. Второе поле, const char *tm_zone, является сокращенным названием часового пояса (например, CEST для центральноевропейского летнего времени). Ни одно из этих полей в SUSv3 не упоминается, и они появляются лишь в нескольких других реализациях UNIX (в основном происходящих от BSD).

Функция mktime() преобразует местное время, разбитое на компоненты, в значение типа time_t, которое возвращается в качестве результата ее работы. Вызывающий код предоставляет разбитое на компоненты время в структуре tm, на которую указывает значение аргумента timeptr. В ходе этого преобразования поля tm_wday и tm_yday вводимой tm-структуры игнорируются.

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

Возвращает при успешном завершении количество секунд, прошедшее с начала отсчета времени и соответствующее содержимому, на которое указывает *timeptr*, или значение (time_t) -1 при ошибке

Функция mktime() может изменить структуру, на которую указывает аргумент timeptr. Как минимум, она гарантирует, что для полей tm_wday и tm_yday будут установлены значения, соответствующие значениям других вводимых полей.

Кроме того, mktime() не требует, чтобы другие поля структуры tm ограничивались рассмотренными ранее диапазонами. Для каждого поля, чье значение выходит за границы диапазона, функция mktime() скорректирует это значение таким образом, чтобы оно попало в диапазон, и сделает соответствующие корректировки других полей. Все эти настройки выполняются до того, как mktime() обновляет значения полей tm_wday и tm_yday и вычисляет возвращаемое значение времени с типом time_t.

Например, если вводимое поле tm_sec хранило значение 123, тогда по возвращении из функции значением поля станет 3, а к предыдущему значению поля tm_min будет добавлено 2. (И если это добавление приведет к переполнению tm_min, значение tm_min будет скорректировано, увеличится значение поля tm_hour и т. д.) Эти корректировки применяются даже к полям с отрицательными значениями. Например, указание -1 для tm_sec означает 59-ю секунду предыдущей минуты. Данное свойство позволяет выполнять арифметические действия в отношении даты и времени, выраженных в виде отдельных компонентов.

При выполнении преобразования функцией mktime() используется настройка часового пояса. Кроме того, в зависимости от значения вводимого поля tm_isdst, функцией могут учитываться, а могут и не учитываться настройки летнего времени.

- Если поле tm_isdst имеет значение 0, это время рассматривается как стандартное (то есть настройки летнего времени игнорируются, даже если они должны применяться к данному времени года).
- Если поле tm_isdst имеет значение больше нуля, это время рассматривается с учетом перехода на режим летнего времени (то есть ведет себя, как будто режим летнего времени введен, даже если этого не должно быть в текущее время года).
- Если поле tm_isdst имеет значение меньше нуля, предпринимается попытка определить, должен ли режим летнего времени применяться в это время года. Обычно именно такая установка нам и требуется.

Перед завершением своей работы (и независимо от исходной установки значения tm_isdst) функция mktime() устанавливает для поля tm_isdst положительное значение, если режим летнего времени применяется в это время года, или нулевое значение, если он не применяется.

10.2.3. Преобразования между разделенным календарным временем и временем в печатном виде

В этом разделе мы рассмотрим функции, выполняющие преобразование разделенного календарного времени в печатный вид и наоборот.

Преобразование разделенного календарного времени в печатный вид

Функция `asctime()`, которой в аргументе `timeptr` передается указатель на структуру, содержащую разделенное время, возвращает указатель на статически размещенную строку, хранящую время в той же форме, в которой оно возвращается функцией `ctime()`.

```
#include <time.h>

char *asctime(const struct tm *timeptr);
```

Возвращает при успешном завершении указатель на статически размещенную строку, оканчивающуюся символом новой строки и `\0`, или `NULL` при ошибке

В отличие от функции `ctime()`, установки часового пояса не влияют на работу функции `asctime()`, поскольку она выполняет преобразование разделенного времени, которое является либо уже локализованным благодаря использованию функции `localtime()`, либо временем UTC, возвращенным функцией `gmtime()`.

Как и в случае применения функции `ctime()`, у нас нет средств для управления форматом строки, создаваемой функцией `asctime()`.

Реентерабельная версия функции asctime() предоставляется в виде asctime_r().

В листинге 10.1 показывается пример использования функции `asctime()`, а также всех рассмотренных до сих пор в этой главе функций преобразования времени. Программа извлекает текущее календарное время, а затем использует различные функции преобразования времени и выдает результаты их работы. Далее приведен пример того, что будет показано при запуске этой программы в Мюнхене, Германия, где (зимой) применяется центральноевропейское время, на один час больше UTC:

```
$ date
Tue Dec 28 16:01:51 CET 2010
$ ./calendar_time
Seconds since the Epoch (1 Jan 1970): 1293548517 (about 40.991 years)
  gettimeofday() returned 1293548517 secs, 715616 microsecs
Broken down by gmtime():
  year=110 mon=11 mday=28 hour=15 min=1 sec=57 wday=2 yday=361 isdst=0
Broken down by localtime():
  year=110 mon=11 mday=28 hour=16 min=1 sec=57 wday=2 yday=361 isdst=0

asctime() formats the gmtime() value as: Tue Dec 28 15:01:57 2010
ctime() formats the time() value as:     Tue Dec 28 16:01:57 2010
mktime() of gmtime() value:    1293544917 secs
mktime() of localtime() value: 1293548517 secs   На 3600 секунд больше UTC
```

Листинг 10.1. Извлечение и преобразование значений календарного времени

time/calendar_time.c

```
#include <locale.h>
#include <time.h>
#include <sys/time.h>
#include "tlpi_hdr.h"

#define SECONDS_IN_TROPICAL_YEAR (365.24219 * 24 * 60 * 60)

int
main(int argc, char *argv[])
```

```
{
    time_t t;
    struct tm *gmp, *locp;
    struct tm gm, loc;
    struct timeval tv;
    t = time(NULL);

    printf("Seconds since the Epoch (1 Jan 1970): %ld", (long) t);
    printf(" (about %6.3f years)\n", t / SECONDS_IN_TROPICAL_YEAR);
    if (gettimeofday(&tv, NULL) == -1)
        errExit("gettimeofday");
    printf(" gettimeofday() returned %ld secs, %ld microsecs\n",
            (long) tv.tv_sec, (long) tv.tv_usec);
    gmp = gmtime(&t);
    if (gmp == NULL)
        errExit("gmtime");
    gm = *gmp;   /* Сохранение локальной копии, так как содержимое, на которое указывает
                    *gmp, может быть изменено вызовом asctime() или gmtime() */

    printf("Broken down by gmtime():\n");
    printf(" year=%d mon=%d mday=%d hour=%d min=%d sec=%d ", gm.tm_year,
           gm.tm_mon, gm.tm_mday, gm.tm_hour, gm.tm_min, gm.tm_sec);
    printf("wday=%d yday=%d isdst=%d\n", gm.tm_wday, gm.tm_yday, gm.tm_isdst);
           gm.tm_isdst);
    locp = localtime(&t);
    if (locp == NULL)
        errExit("localtime");
    loc = *locp;        /* Сохранение локальной копии */

    printf("Broken down by localtime():\n");
    printf(" year=%d mon=%d mday=%d hour=%d min=%d sec=%d ",
           loc.tm_year, loc.tm_mon, loc.tm_mday,
           loc.tm_hour, loc.tm_min, loc.tm_sec);
    printf("wday=%d yday=%d isdst=%d\n\n", loc.tm_wday, loc.tm_yday, loc.tm_isdst);

    printf("asctime() formats the gmtime() value as: %s", asctime(&gm));
    printf("ctime() formats the time() value as: %s", ctime(&t));
    printf("mktime() of gmtime() value: %ld secs\n", (long) mktime(&gm));
    printf("mktime() of localtime() value: %ld secs\n", (long) mktime(&loc));
    exit(EXIT_SUCCESS);
}
```

time/calendar_time.c

Функция `strftime()` предоставляет нам более тонкую настройку управления при преобразовании разделенного календарного времени в печатный вид.

Функция `strftime()`, которой в аргументе `timeptr` передается указатель на структуру, содержащую разделенное время, возвращает соответствующую строку, завершаемую нулевым байтом, в буфер, заданный аргументом `outst`. В этой строке хранятся и дата и время.

```
#include <time.h>

size_t strftime(char *outstr, size_t maxsize, const char *format,
                const struct tm *timeptr);
```

Возвращает при успешном завершении количество байтов, помещенных в строку, на которую указывает `outstr` (исключая завершающий нулевой байт), или `0` при ошибке

Строка, возвращенная в буфер, на который указывает `outstr`, отформатирована в соответствии со спецификаторами, заданными аргументом `format`. Аргумент `maxsize` указывает максимальное пространство, доступное в буфере, заданном аргументом `outstr`. В отличие от `ctime()` и `asctime()` функция `strftime()` не включает в окончание строки символ новой строки (кроме того, что включается в спецификацию формата, указываемую аргументом `format`).

В случае успеха функция `strftime()` возвращает количество байтов, помещенных в буфер, на который ссылается `outstr`, исключая завершающий нулевой байт. Если общая длина получившейся строки, включая завершающий нулевой байт, станет превышать количество байтов, заданное в аргументе `maxsize`, функция `strftime()` возвратит `0`, чтобы показать ошибку; в этом случае содержимое буфера, на который указывает `outstr`, станет неопределенным.

Аргумент `format`, используемый при вызове `strftime()`, представляет собой строку по типу той, что задается в функции `printf()`. Последовательности, начинающиеся с символа процента (`%`), являются спецификаторами преобразования, которые заменяются различными компонентами даты и времени в соответствии с символом, следующим за символом процента. Предусмотрен довольно обширный выбор спецификаторов преобразования, часть компонентов которого перечислена в табл. 10.1. (Полный перечень можно найти на странице руководства `strftime(3)`.) За исключением особо оговариваемых, все эти спецификаторы преобразования стандартизированы в SUSv3.

Спецификаторы `%U` и `%W` выводят номер недели в году. Номера недель, выводимые с помощью `%U`, исчисляются из расчета, что первая неделя, начиная с воскресенья, получает номер 1, а предшествующая ей неполная неделя получает номер 0. Если воскресенье приходится на первый день года, то неделя с номером 0 отсутствует и последний день года приходится на неделю под номером 53. Нумерация недель, выводимых с помощью `%W`, работает точно так же, но вместо воскресенья в расчет берется понедельник.

Зачастую в книге нам придется выводить текущее время в различных демонстрационных программах. Для этого мы предоставляем функцию `currTime()`, которая возвращает строку с текущим временем, отформатированным функцией `strftime()` при заданном аргументе `format`.

```
#include "curr_time.h"

char *currTime(const char *format);
```

Возвращает при успешном завершении указатель на статически размещенную строку или `NULL` при ошибке

Реализация функции `currTime()` показана в листинге 10.2.

Таблица 10.1. Отдельные спецификаторы преобразования для strftime()

Спецификатор	Описание	Пример
%%	Символ %	%
%a	Сокращенное название дня недели	Tue
%A	Полное название дня недели	Tuesday
%b, %h	Сокращенное название месяца	Feb
%B	Полное название месяца	February

Спецификатор	Описание	Пример
%c	Дата и время	Tue Feb 1 21:39:46 2011
%d	День месяца (две цифры, от 01 до 31)	01
%D	Дата в американском формате (то же самое, что и %m/%d/%y)	02/01/11
%e	День месяца (два символа)	_1
%F	Дата в формате ISO (то же самое, что и %Y-%m-%d)	2011-02-01
%H	Час (24-часовой формат, две цифры)	21
%I	Час (12-часовой формат, две цифры)	09
%j	День года (три цифры, от 001 до 366)	032
%m	Месяц в виде десятичного числа (две цифры, от 01 до 12)	02
%M	Минута (две цифры)	39
%p	AM/PM (до полудня/после полудня)	PM
%P	am/pm (GNU-расширение)	pm
%R	Время в 24-часовом формате (то же самое, что и %H:%M)	21:39
%S	Секунда (от 00 до 60)	46
%T	Время (то же самое, что и %H:%M:%S)	21:39:46
%u	Номер дня недели (от 1 до 7, Понедельник = 1)	2
%U	Номер недели, начинающейся с воскресенья (от 00 до 53)	05
%w	Номер дня недели (от 0 до 6, воскресенье = 0)	2
%W	Номер недели, начинающейся с понедельника (от 00 до 53)	05
%x	Дата (локализированная версия)	02/01/11
%X	Время (локализированная версия)	21:39:46
%y	Последние две цифры года	11
%Y	Год в формате четырех цифр	2011
%Z	Название часового пояса	CET

Листинг 10.2. Функция, возвращающая строку с текущим временем

time/curr_time.c

```
#include <time.h>
#include "curr_time.h"        /* Объявление определяемых здесь функций */

#define BUF_SIZE 1000
/* Возвращает строку, содержащую текущее время, отформатированное в сооответствии
   со спецификацией в 'format' (спецификаторы на странице руководства strftime(3)).
   Если 'format' имеет значение NULL, в качестве спецификатора мы используем "%c"
   (что дает дату и время, как для ctime(3), но без завершающего символа новой строки).
   При ошибке возвращается NULL. */
```

```
char *
currTime(const char *format)
{
    static char buf[BUF_SIZE];  /* Нереентерабельная */
    time_t t;
    size_t s;
    struct tm *tm;

    t = time(NULL);
    tm = localtime(&t);

    if (tm == NULL)
        return NULL;
    s = strftime(buf, BUF_SIZE, (format != NULL) ? format : "%c", tm);
    return (s == 0) ? NULL : buf;
}
```

time/curr_time.c

Преобразование из печатного вида в разделенное календарное время

Функция `strptime()` выполняет преобразование, обратное тому, которое делает функция `strftime()`. Она преобразует строку в виде даты и времени в разделенное календарное время (время, разбитое на компоненты).

```
#define _XOPEN_SOURCE
#include <time.h>

char *strptime(const char *str, const char *format, struct tm *timeptr);
```

Возвращает при успешном завершении указатель на следующий необработанный символ в *str* или `NULL` при ошибке

Функция `strptime()` использует спецификацию, заданную в аргументе `format`, для разбора строки в формате «дата плюс время», указанной в аргументе `str`. Затем она помещает результат преобразования в разделенное календарное время в структуру, на которую указывает аргумент `timeptr`.

В случае успеха `strptime()` возвращает указатель на следующий необработанный символ в `str`. (Это пригодится, если строка содержит дополнительную информацию для обработки вызывающей программой.) Если полная строка формата не может быть подобрана, функция `strptime()` возвращает значение `NULL`, чтобы показать, что возникла ошибка.

Спецификация формата, заданная функцией `strptime()`, похожа на ту, что задается `scanf(3)`. В ней содержатся следующие типы символов:

- спецификации преобразования, начинающиеся с символа процента (`%`);
- пробельные символы, соответствующие нулю или большему количеству пробелов во введенной строке;
- непробельные символы (отличающиеся от `%`), которые должны соответствовать точно таким же символам во введенной строке.

Спецификации преобразования похожи на те, которые задаются в функции `strftime()` (см. табл. 10.1). Основное отличие заключается в их более общем характере. Например,

спецификаторы %a и %A могут принять название дня недели как в полной, так и в сокращенной форме, а %d или %e могут использоваться для чтения дня месяца, если он может быть выражен одной цифрой с ведущим нулем или без него. Кроме того, регистр символов игнорируется. Например, для названия месяца одинаково подходят May и MAY. Строка %% применяется для соответствия символу процента во вводимой строке. Дополнительные сведения можно найти на странице руководства strptime(3).

Реализация strptime(), имеющаяся в библиотеке glibc, не вносит изменений в те поля структуры tm, которые не инициализированы спецификаторами из аргумента format. Это означает, что для создания одной структуры tm на основе информации из нескольких строк, из строки даты и строки времени, мы можем воспользоваться серией вызовов strptime(). Хотя в SUSv3 такое поведение допускается, оно не является обязательным, и поэтому полагаться на них в других реализациях UNIX не стоит. В портируемом приложении, прежде чем вызвать strptime(), нужно обеспечить наличие в аргументах str и format входящей информации, которая установит все поля получаемой в итоге структуры tm, или же предоставить подходящую инициализацию структуры tm. В большинстве случаев достаточно задать всей структуре нулевые значения, используя функцию memset(). Но нужно иметь в виду, что значение 0 в поле tm_mday соответствует в glibc-версии и во многих других реализациях функции преобразования времени *последнему дню предыдущего месяца*. И наконец, следует учесть, что strptime() никогда не устанавливает значение имеющегося в структуре tm для аргумента tm_isdst.

> GNU-библиотека C также предоставляет две другие функции, которые служат той же цели, что и strptime(): это getdate() (широкодоступная и указанная в SUSv3) и ее реентерабельный аналог getdate_r() (не указанный в SUSv3 и доступный только в некоторых других реализациях UNIX). Здесь эти функции не рассматриваются, потому что они для указания формата, применяемого при сканировании даты, используют внешний файл (указываемый с помощью переменной среды DATEMSK), что затрудняет их применение, а также создает бреши безопасности в set-user-ID-программах.

Использование функций strptime() и strftime() показано в программе, код которой приводится в листинге 10.3. Эта программа получает аргумент командной строки с датой и временем, преобразует их в календарное время, разбитое на компоненты, с помощью функции strptime(), а затем выводит результат обратного преобразования, выполненного функцией strftime(). Программа получает три аргумента, два из которых обязательны. Первый аргумент является строкой, содержащей дату и время. Второй аргумент — спецификация формата, используемого функцией strptime() для разбора первого аргумента. Необязательный третий аргумент — строка формата, используемого функцией strftime() для обратного преобразования. Если этот аргумент не указан, применяется строка формата по умолчанию. (Функция setlocale(), используемая в этой программе, рассматривается в разделе 10.4.) Примеры применения этой программы показаны в следующей записи сеанса работы с оболочкой:

```
$ ./strtime "9:39:46pm 1 Feb 2011" "%I:%M:%S%p %d %b %Y"
calendar time (seconds since Epoch): 1296592786
strftime() yields: 21:39:46 Tuesday, 01 February 2011 CET
```

Следующий код похож на предыдущий, но на этот раз формат для strftime() указан явным образом:

```
$ ./strtime "9:39:46pm 1 Feb 2011" "%I:%M:%S%p %d %b %Y" "%F %T"
calendar time (seconds since Epoch): 1296592786
strftime() yields: 2011-02-01 21:39:46
```

Листинг 10.3. Извлечение и преобразование данных календарного времени

time/strtime.c

```
#define _XOPEN_SOURCE
#include <time.h>
#include <locale.h>
#include "tlpi_hdr.h"

#define SBUF_SIZE 1000

int
main(int argc, char *argv[])
{
    struct tm tm;
    char sbuf[SBUF_SIZE];
    char *ofmt;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s input-date-time in-format [out-format]\n", argv[0]);
    if (setlocale(LC_ALL, "") == NULL)
        errExit("setlocale"); /* Использование настроек локали при преобразовании */

    memset(&tm, 0, sizeof(struct tm));          /* Инициализация 'tm' */
    if (strptime(argv[1], argv[2], &tm) == NULL)
        fatal("strptime");

    tm.tm_isdst = -1;   /* Не устанавливается функцией strptime(); заставляет функцию
                           mktime() определить действие режима летнего времени */

    printf("calendar time (seconds since Epoch): %ld\n", (long) mktime(&tm));

    ofmt = (argc > 3) ? argv[3] : "%H:%M:%S %A, %d %B %Y %Z";
    if (strftime(sbuf, SBUF_SIZE, ofmt, &tm) == 0)
        fatal("strftime returned 0");
    printf("strftime() yields: %s\n", sbuf);

    exit(EXIT_SUCCESS);
}
```

time/strtime.c

10.3. Часовые пояса

Разные страны (а иногда даже и разные регионы одной страны) находятся в разных часовых поясах и режимах действия летнего времени. Программы, где используется ввод и вывод времени, должны учитывать часовой пояс и режим действия летнего времени той системы, в которой они запускаются. К счастью, все эти особенности обрабатываются средствами библиотеки языка C.

Определение часовых поясов

Информация о часовом поясе характеризуется, как правило, обширностью и нестабильностью. Поэтому, вместо того, чтобы вносить ее в код программ или библиотек напрямую, система хранит эту информацию в файлах в стандартных форматах.

Эти файлы находятся в каталоге `/usr/share/zoneinfo`. Каждый файл в нем содержит информацию о часовом поясе конкретной страны или региона. Файлы названы в соответ-

ствии с тем часовым поясом, описание которого в них дается, поэтому там можно найти файлы с такими именами, как `EST` (US Eastern Standard Time — североамериканское восточное время), `CET` (Central European Time — центральноевропейское время), `UTC`, `Turkey` и `Iran`. Кроме того, для создания иерархии групп, связанных с часовыми поясами, могут использоваться подкаталоги. Например, в каталоге `Pacific` можно найти файлы `Auckland`, `Port_Moresby` и `Galapagos`. Когда мы указываем программе, какой именно часовой пояс использовать, на самом деле указывается относительное путевое имя для одного из файлов часового пояса в этом каталоге.

Местное время для системы определяется файлом часового пояса `/etc/localtime`, который часто ссылается на один из файлов в каталоге `/usr/share/zoneinfo`.

> Формат файлов часовых поясов задокументирован на странице руководства tzfile(5). Файлы часовых поясов создаются с помощью zic(8), компилятора информации о часовых поясах. С помощью команды zdump можно вывести текущее время для указанных файлов часовых поясов.

Указание часового пояса для программы

Чтобы указать часовой пояс при выполнении программы, переменной среды `TZ` присваивается значение в виде строки, содержащей символ двоеточия (`:`), за которым следует одно из названий часовых поясов, определенное в `/usr/share/zoneinfo`. Установка часового пояса автоматически влияет на функции `ctime()`, `localtime()`, `mktime()` и `strftime()`.

Для получения текущей установки часового пояса в каждой из этих функций применяется функция `tzset(3)`, которая инициализирует три глобальные переменные:

```
char *tzname[2];    /* Название часового пояса и альтернативного часового пояса
                       с учетом действия режима летнего времени */
int daylight;       /* Ненулевое значение при наличии альтернативного
                       часового пояса с учетом действия режима летнего времени */
long timezone;      /* Разница в секундах между UTC и местным [поясным] временем */
```

Функция `tzset()` сначала проверяет значение переменной среды `TZ`. Если значение для нее не установлено, часовой пояс инициализируется значением по умолчанию, определенным в файле часового пояса `/etc/localtime`. Если переменная `TZ` определена и имеет значение, которое не может соответствовать файлу часового пояса, или если оно представляет собой пустую строку, тогда используется UTC. Для переменной среды `TZDIR` (нестандартное GNU-расширение) может быть установлено имя каталога, в котором требуется вести поиск информации о часовом поясе вместо исходного каталога `/usr/share/zoneinfo`.

Эффект использования переменной `TZ` можно увидеть, запустив на выполнение программу, показанную в листинге 10.4. При первом запуске будет виден вывод, соответствующий исходному часовому поясу системы (центральноевропейского времени, CET). При втором запуске будет указан часовой пояс для Новой Зеландии, где в заданное время года действует режим летнего времени и местное время опережает CET на 12 часов.

```
$ ./show_time
ctime() of time() value is:  Tue Feb  1 10:25:56 2011
asctime() of local time is:  Tue Feb  1 10:25:56 2011
strftime() of local time is: Tuesday, 01 Feb 2011, 10:25:56 CET
$ TZ=":Pacific/Auckland" ./show_time
ctime() of time() value is:  Tue Feb  1 22:26:19 2011
asctime() of local time is:  Tue Feb  1 22:26:19 2011
strftime() of local time is: Tuesday, 01 February 2011, 22:26:19 NZDT
```

Листинг 10.4. Демонстрация эффекта часовых поясов и локалей

time/show_time.c

```
#include <time.h>
#include <locale.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 200

int
main(int argc, char *argv[])
{
    time_t t;
    struct tm *loc;
    char buf[BUF_SIZE];

    if (setlocale(LC_ALL, "") == NULL)
        errExit("setlocale");   /* Использование в преобразовании настроек локали */
    t = time(NULL);

    printf("ctime() of time() value is: %s", ctime(&t));
    loc = localtime(&t);
    if (loc == NULL)
        errExit("localtime");

    printf("asctime() of local time is: %s", asctime(loc));
    if (strftime(buf, BUF_SIZE, "%A, %d %B %Y, %H:%M:%S %Z", loc) == 0)
        fatal("strftime returned 0");
    printf("strftime() of local time is: %s\n", buf);
    exit(EXIT_SUCCESS);
}
```

time/show_time.c

В SUSv3 определяются два основных способа установки значения для переменной среды `TZ`. Как уже было рассмотрено, значение `TZ` может быть установлено в виде последовательности символов, содержащей двоеточие и строку. Эта строка идентифицирует часовой пояс в том виде, который присущ конкретной реализации, как правило, в виде путевого имени файла, содержащего описание часового пояса. (В Linux и в некоторых других реализациях UNIX в этом случае допускается не ставить двоеточие, но в SUSv3 это не указывается; из соображений портируемости двоеточие нужно ставить всегда.)

Еще один метод установки значения для `TZ` полностью указан в SUSv3. Согласно ему, переменной `TZ` присваивается срока следующего вида:

```
std offset [ dst [ offset ][ , start-date [ /time ] , end-date [ /time ]]]
```

Пробелы включены в показанную выше строку для удобства чтения, но в значении `TZ` их быть не должно. Квадратные скобки (`[]`) используются для обозначения необязательных компонентов. Компоненты `std` и `dst` — это строки, показывающие стандартный часовой пояс и часовой пояс с учетом действия режима летнего времени; например CET и CEST для центральноевропейского времени и центральноевропейского летнего времени. Смещение `offset` в каждом случае указывается положительным или отрицательным корректировочным значением, которое прибавляется к местному времени для его преобразования во время UTC. Последние четыре компонента предоставляют правило, описывающее период перехода со стандартного на летнее время.

Даты могут указываться в разных формах, одной из которых является `Mm.n.d`. Эта запись означает день *d* (0 = воскресенье, 6 = суббота) недели *n* (от 1 до 5, где 5 всегда означает последний *d* день) месяца *m* (от 1 до 12). Если время опущено, его значение в любом случае устанавливается по умолчанию на 02:00:00 (2 AM).

Определить `TZ` для Центральной Европы, где стандартное время на час опережает UTC и режим летнего времени (DST) вводится с последнего воскресенья марта до последнего воскресенья октября, а местное время опережает UTC на два часа, можно следующим образом:

```
TZ="CET-1:00:00CEST-2:00:00,M3.5.0,M10.5.0"
```

Время перехода на режим летнего времени не показано, поскольку переход осуществляется в устанавливаемое по умолчанию время 02:00:00. Разумеется, предыдущая форма менее удобочитаема, чем ее ближайший эквивалент:

```
TZ=":Europe/Berlin"
```

10.4. Локали

В мире говорят на нескольких тысячах языков, существенная часть которых постоянно используется в компьютерных системах. Кроме того, в разных странах есть разные соглашения для отображения такой информации, как числа, денежные суммы, даты и показания времени. Например, в большинстве европейских стран для отделения целой части от дробной в действительных числах используется запятая, а не точка и в большинстве стран используются форматы для записи дат, отличающиеся от формата MM/DD/YY, принятого в США. В SUSv3 *локаль* характеризуется как «подмножество переменных пользовательской среды, которые зависят от языковых и культурных норм».

В идеале все программы, созданные для работы в более чем одном месте, должны работать с локалью, чтобы отображаемая и вводимая информация была в привычном для пользователя формате и на его языке. Возникает весьма непростой вопрос *интернационализации*. В идеальном мире программа была бы создана как единое целое, а затем, в зависимости от того, где именно она запускается, она бы автоматически правильно обрабатывала ввод/вывод, то есть решала бы *задачу локализации*. Интернационализация программ — весьма затратная задача, для облегчения которой доступно множество различных средств. Библиотеки, и в частности glibc, предоставляют возможности, облегчающие локализацию.

> Термин «интернационализация» (internationalization) часто записывается в виде i18N, то есть в виде I плюс 18 букв плюс N. Кроме того, что в таком виде это слово записывается быстрее, данная запись устраняет различия в его написании, существующие в английском и американском вариантах английского языка.

Определения локали

Так же как информация о часовых поясах, сведения о локали обычно отличаются обширностью и изменчивостью. По этой причине, вместо того чтобы требовать от каждой программы и библиотеки хранения информации о локали, система хранит эти сведения в файлах в стандартных форматах.

Информации о локали содержится в иерархии каталогов, которая находится в каталоге `/usr/share/locale` (или в некоторых дистрибутивах в каталоге `/usr/lib/locale`). Каждый имеющийся в этом каталоге подкаталог хранит информацию о конкретном месте (в географическом смысле). Эти каталоги называются с использованием следующего соглашения:

```
language[_territory[.codeset]][@modifier]
```

В качестве *Language* используется двухбуквенный код языка по стандарту ISO, а в качестве *territory* — двухбуквенный код страны по стандарту ISO. Компонент *codeset* обозначает кодировку символов. Компонент *modifier* предоставляет средства, позволяющие отличить друг от друга несколько каталогов с локалями, чьи языки, территории и кодировки символов совпадают. Примером полного имени каталога с локалями может служить `de_DE.utf-8@euro`, которое соответствует следующим региональным настройкам: немецкий язык, Германия, кодировка символов UTF-8, в качестве денежного знака используется евро.

Квадратные скобки в формате наименования каталога показывают, что некоторые части названия каталога локали могут быть опущены. Зачастую название состоит просто из языка (*Language*) и страны (*territory*). Следовательно, каталог `en_US` является каталогом локали для англоговорящих Соединенных Штатов, а `fr_CH` — каталогом локали для франкоговорящего региона Швейцарии.

> CH означает Confoederatio Helvetica, латинское (и в силу этого нейтрального по языку для данной местности) название Швейцарии. Имея четыре официальных национальных языка, Швейцария в плане региональных настроек аналогична стране с несколькими часовыми поясами.

Когда в программе указывается, какую именно локаль использовать, мы, по сути, определяем название одного из подкаталогов, находящихся в каталоге `/usr/share/locale`. Если локаль, определенная в программе, не соответствует в точности названию каталога локали, библиотека языка C ведет поиск соответствия путем разбора компонентов из заданной локали в следующем порядке.

1. Кодировка символов (codeset).
2. Нормализованная кодировка символов (normalized codeset).
3. Страна (territory).
4. Модификатор (modifier).

Нормализованная кодировка символов представляет собой версию имени кодировки символов, в которой удалены все символы, не являющиеся буквами и цифрами, все буквы приведены к нижнему регистру и для получившейся строки указан префикс iso. Цель нормализации — обработка вариаций в регистре букв и пунктуации (например, в дополнительных дефисах) имен кодировок символов.

Например, если для программы локаль запрошена как `fr_CH.utf-8`, но каталога локали под таким названием не существует, то для такой локали подойдет каталог `fr_CH`, если таковой обнаружится. Если каталога с названием `fr_CH` не будет, то будет использован каталог локали `fr`. В маловероятном случае отсутствия каталога `fr` функция `setlocale()`, которая вскоре будет рассмотрена, сообщит об ошибке.

> Альтернативные способы указания локали для программы определяются в файле /usr/share/locale/locale.alias. Подробности можно найти на странице руководства locale.aliases(5).

Как показано в табл. 10.2, в каждом подкаталоге локали имеется стандартный набор файлов с указаниями норм, принятых для данной локали. Относительно сведений, приведенных в этой таблице, следует сделать несколько пояснений.

- В файле `LC_COLLATE` устанавливается набор правил, описывающих порядок следования символов в их наборе (то есть «алфавитный» порядок для набора символов). Эти правила определяют работу функций `strcoll(3)` и `strxfrm(3)`. Даже языки, основанные на латинице, не следуют одним и тем же правилам сортировки. Например, в ряде европейских языков имеются дополнительные буквы, которые иногда при сортировке

могут следовать за буквой Z. К другим особым случаям можно отнести испанскую двухбуквенную последовательность ll, которая сортируется как одна буква, следующая за буквой l, и немецкие символы умлаутов, такие как «д», которая соответствует сочетанию ae и сортируется как эти две буквы.

- Каталог `LC_MESSAGES` является одним шагом по направлению к интернационализации сообщений, выводимых программой. Расширенная интернационализация сообщений программы может быть выполнена путем использования либо каталогов сообщений (см. страницы руководства `catopen(3)` и `catgets(3)`), либо GNU API gettext (доступного по адресу http://www.gnu.org/).

В версии glibc под номером 2.2.2 введено несколько новых, нестандартных категорий локали. В LC_ADDRESS определяются правила зависящих от локали представлений почтовых адресов. В LC_IDENTIFICATION указывается информация, идентифицирующая локаль. В LC_MEASUREMENT определяется местная система мер (например, метрическая или дюймовая). В LC_NAME устанавливаются местные правила представления личных имен и титулов. В LC_PAPER определяется стандартный для данной местности размер бумаги (например, принятый в США формат Letter или формат A4, используемый в большинстве других стран). В LC_TELEPHONE задаются правила для местного представления внутренних и международных телефонных номеров, а также международного префикса страны и префикса выхода на международную телефонную сеть.

Таблица 10.2. Содержимое подкаталогов локали

Имя файла	Назначение
LC_CTYPE	Файл содержит классификацию символов (см. isalpha(3)) и правила, применяемые при преобразовании регистра
LC_COLLATE	Файл включает правила сортировки набора символов
LC_MONETARY	Файл содержит правила форматирования денежных величин (см. localeconv(3) и <locale.h>)
LC_NUMERIC	Файл содержит правила форматирования для чисел, не являющихся денежными величинами (см. localeconv(3) и <locale.h>)
LC_TIME	Файл включает правила форматирования для даты и времени
LC_MESSAGES	Каталог содержит файлы, указывающие форматы и значения, используемые для утвердительных и отрицательных ответов (да — нет)

Фактические настройки локали, определенные в системе, могут изменяться. В SUSv3 насчет этого не выдвигается никаких требований, за исключением необходимости определения стандартной настройки локали по имени POSIX (и по историческим причинам ее синонима по имени C). Эта локаль воспроизводит исторически сложившееся поведение систем UNIX. Так, она основана на наборе кодировки символов ASCII и использует английский язык для названия дней и месяцев, а также односложных ответов yes и no. Денежные и числовые компоненты в этой локалии не определяются.

Команда locale выводит информацию о текущей локали среды (в оболочке). Команда locale –a выводит списком полный набор локалей, определенных в системе.

Задание для программы локали

Для установки и запроса локали программы используется функция `setlocale()`, синтаксис которой представлен далее.

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

Возвращает указатель на (обычно статически выделенную) строку, определяющую новые или текущие местные настройки, при успехе или NULL при ошибке

Аргумент `category` выбирает, какую часть данных о локали установить или запросить, и указывается набор констант, чьи имена совпадают с категориями локали, перечисленными в табл. 10.2. Это, к примеру, означает, что можно настроить локаль так, чтобы отображалось время как в Германии, а вместе с тем задать отображение денежных сумм в долларах США. Или же, что бывает значительно чаще, мы можем использовать значение `LC_ALL`, чтобы указать, что нам нужно установить все аспекты локали.

Есть два различных метода настройки локали с помощью функции `setlocale()`. Аргумент `locale` должен быть строкой, указывающей на одну из локалей, определяемых в системе (то есть на имя одного из подкаталогов в каталоге `/usr/lib/locale`), например `de_DE` или `en_US`. Или же `locale` может быть указан в виде пустой строки, что означает необходимость получения настроек локали из переменных среды:

```
setlocale(LC_ALL, "");
```

Такой вызов нужно использовать, чтобы программа считала информацию из соответствующих переменных среды, имеющих отношение к локализации. Без такого вызова эти переменные среды никак не повлияют на программу.

При запуске программы, выполняющей вызов `setlocale(LC_ALL,"")`, мы можем управлять различными аспектами локали, используя набор переменных среды, чьи имена также соответствуют категориям, перечисленным в табл. 10.2: `LC_CTYPE`, `LC_COLLATE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME` и `LC_MESSAGES`. Для указания настроек всей локали также можно воспользоваться переменной среды `LC_ALL` или `LANG`. При установке более одной из ранее перечисленных переменных среды у `LC_ALL` имеется приоритет над всеми другими переменными вида `LC_*`, а `LANG` имеет самый низкий уровень приоритета. Следовательно, `LANG` можно применять для настроек локали, используемых по умолчанию для всех категорий, а затем воспользоваться отдельными переменными `LC_*` для настройки составляющих локали на что-либо другое, чем эти установки по умолчанию.

В результате функция `setlocale()` возвращает указатель на (обычно статически размещаемую) строку, которая идентифицирует настройки локали для конкретной категории. Если мы интересуемся только получением текущих настроек локали без внесения в них изменений, для аргумента `locale` следует установить значение `NULL`.

Настройки локализации управляют работой множества разнообразных GNU/Linux-утилит, а также многих функций в библиотеке glibc. Среди них функции `strftime()` и `strptime()` (см. подраздел 10.2.3), о чем свидетельствуют результаты, полученные от `strftime()` при выполнении программы из листинга 10.4:

```
$ LANG=de_DE ./show_time                    Немецкая локаль
ctime() of time() value is: Tue Feb 1 12:23:39 2011
asctime() of local time is: Tue Feb 1 12:23:39 2011
strftime() of local time is: Dienstag, 01 Februar 2011, 12:23:39 CET
```

Следующий код демонстрирует, что `LC_TIME` имеет преимущество перед `LANG`:

```
$ LANG=de_DE LC_TIME=it_IT ./show_time      Немецкая и итальянская локали
ctime() of time() value is: Tue Feb 1 12:24:03 2011
```

```
asctime() of local time is: Tue Feb 1 12:24:03 2011
strftime() of local time is: martedì, 01 febbraio 2011, 12:24:03 CET
```

А этот код показывает, что LC_ALL имеет преимущество перед LC_TIME:

```
$ LC_ALL=fr_FR LC_TIME=en_US ./show_time   Французская и американская (США) локали
ctime() of time() value is: Tue Feb 1 12:25:38 2011
asctime() of local time is: Tue Feb 1 12:25:38 2011
strftime() of local time is: mardi, 01 février 2011, 12:25:38 CET
```

10.5. Обновление системных часов

Теперь рассмотрим два интерфейса, обновляющих системные часы: settimeofday() и adjtime(). Прикладными программами они используются довольно редко (поскольку обычно системное время поддерживается с помощью средств вроде демона сервиса точного времени *Network Time Protocol*), и к тому же им нужно, чтобы вызывающий процесс был привилегированным (CAP_SYS_TIME).

Системный вызов settimeofday() является обратным функции gettimeofday() (рассмотренной в разделе 10.1): он присваивает календарному времени системы значение, соответствующее количеству секунд и микросекунд, заданное в структуре timeval, указатель на которую находится в аргументе tv.

```
#define _BSD_SOURCE
#include <sys/time.h>

int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

Возвращает при успешном завершении 0
или –1 при ошибке

Как и в случае с gettimeofday(), использование аргумента tz утратило актуальность, и в качестве его значения нужно указывать NULL.

Точность до микросекунд в поле v.tv_usec не означает наличие такой же точности в управлении системными часами, поскольку точность у часов может быть ниже одной микросекунды.

Хотя системный вызов settimeofday() в SUSv3 не определен, он широко доступен во многих других реализациях UNIX.

В Linux также предоставляется системный вызов stime(), предназначенный для установки системных часов. Разница между settimeofday() и stime() состоит в том, что последний вызов позволяет установить новое календарное время с точностью всего лишь в одну секунду. Как и в случае с time() и gettimeofday(), причина существования как stime(), так и settimeofday() имеет исторические корни: последний, задающий более точное значение вызов был добавлен в версии 4.2BSD.

Резкие изменения системного времени, связанные с вызовами settimeofday(), могут плохо влиять на приложения (например, на make(1), систему управления базами данных, использующую метки времени, или на файлы журналов, использующие метки времени). Поэтому при внесении незначительных изменений в установки времени (в пределах нескольких секунд) практически всегда предпочтительнее задействовать библиотечную функцию adjtime(), заставляющую системные часы постепенно выйти на требуемое значение.

```
#define _BSD_SOURCE
#include <sys/time.h>

int adjtime(struct timeval *delta, struct timeval *olddelta);
```

Возвращает при успешном завершении 0 или –1 при ошибке

Аргумент `delta` указывает на структуру `timeval`, определяющую количество секунд и микросекунд, на которое нужно изменить время. При положительном значении время добавляется к системным часам небольшими порциями каждую секунду до тех пор, пока не будет добавлено требуемое значение. При отрицательном значении `delta` ход часов замедляется в том же режиме.

Скорость изменения в Linux/x86-32 составляет одну секунду за каждые 2000 секунд (или 43,2 секунды за день).

Может получиться так, что вызов функции `adjtime()` придется на момент, когда предыдущее изменение показания часов не завершилось. В таком случае объем оставшегося неизмененного времени возвращается в `timeval`-структуру `olddelta`. Если это значение нас не интересует, для аргумента `olddelta` нужно указать `NULL`. И наоборот, если нас интересуют только сведения о текущем объеме невыполненной коррекции времени и мы не намереваемся изменять значение, в качестве аргумента `delta` можно указать `NULL`.

Несмотря на то что в SUSv3 функция `adjtime()` не указана, она доступна в большинстве реализаций UNIX.

В Linux функция adjtime() реализована в качестве надстройки над более универсальным (и сложным) характерным для Linux системным вызовом adjtimex(). Этот системный вызов используется резидентной программой (демоном) Network Time Protocol (NTP). Дополнительные сведения можно найти в исходном коде Linux, на странице руководства Linux adjtimex(2) и в спецификации NTP ([Mills, 1992]).

10.6. Программные часы (мгновения)

Точность различных связанных со временем системных вызовов, рассматриваемых в данной книге, ограничивается разрешением *программных системных часов*, которые измеряют отрезки времени в единицах, называемых *мгновениями (jiffies)*. Размер мгновения определяется внутри исходного кода ядра константой `HZ`. Эта единица измерения используется ядром при выделении процессам времени центрального процессора в соответствии с циклическим алгоритмом его планирования (см. раздел 35.1).

В Linux/x86-32 в ядрах версий до 2.4 включительно частота программных часов была 100 Гц, то есть мгновение составляло 10 миллисекунд.

Поскольку со времени первой реализации Linux скорости работы центральных процессоров существенно возросли, в ядре версии 2.6.0 частота программных часов на Linux/x86-32 была поднята до 1000 Гц. Преимущество повышенной частоты программных часов заключается в том, что таймер может работать с более высокой точностью и замеры времени могут быть намного точнее. Но выводить частоту часов на слишком высокие значения нежелательно, поскольку каждое прерывание от таймера потребляет небольшой объем времени центрального процессора, которое уже невозможно потратить на выполнение процессов.

Споры разработчиков ядра привели в конечном счете к тому, что частота программных часов стала одной из настроек ядра (`Timer frequency` в разделе `Processor type and`

features). Начиная с версии ядра 2.6.13, частоте работы часов может устанавливаться значение 100, 250 (по умолчанию) или 1000 Гц, что определяет значения мгновений, равные 10, 4 и 1 миллисекунде соответственно. С версии ядра 2.6.20 стала доступна еще одна частота: 300 Гц, представленная числом, которое делится без остатка на две самые распространенные частоты видеокадров: 25 кадров в секунду (PAL) и 30 кадров в секунду (NTSC).

10.7. Время процесса

Временем процесса называется объем времени центрального процессора, использованный процессом с момента его создания. В целях учета ядро делит время центрального процессора на следующие два компонента.

- *Пользовательское время центрального процессора*, представляющее собой время, потраченное на выполнение кода в пользовательском режиме. Иногда его называют *фактическим временем*, и оно является временем, когда программе кажется, что она имеет доступ к ЦП.
- *Системное время центрального процессора*, представляющее собой время, потраченное на выполнение кода в режиме ядра. Это время, которое ядро затрачивает на выполнение системных вызовов или других задач в интересах программы (например, на обслуживание ошибок отсутствия страниц).

Иногда время процесса называют *общим временем центрального процессора*, потребленным процессом.

При запуске программы из оболочки для получения обоих значений времени процесса, а также реального времени, требуемого для выполнения программы, можно воспользоваться командой time(1):

```
$ time ./myprog
real    0m4.84s
user    0m1.030s
sys     0m3.43s
```

Информация о времени процесса может быть извлечена системным вызовом times(), возвращающим ее в структуре, на которую указывает аргумент buf.

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Возвращает при успешном завершении количество тиков часов (sysconf(_SC_CLK_TCK)) с некоторого момента времени в прошлом, или значение (clock_t) –1 при ошибке

Эта tms-структура, на которую указывает buf, выглядит следующим образом:

```
struct tms {
    clock_t tms_utime;   /* Пользовательское время ЦП,
                            использованное вызывающим процессом */
    clock_t tms_stime;   /* Системное время ЦП, использованное вызывающим процессом */
    clock_t tms_cutime;  /* Пользовательское время ЦП, прошедшее в ожидании
                            завершения всех дочерних процессов */
    clock_t tms_cstime;  /* Системное время ЦП, прошедшее в ожидании завершения
                            всех дочерних процессов */
};
```

В первых двух полях tms-структуры возвращаются пользовательские и системные компоненты времени центрального процессора, до сих пор затраченного вызывающим процессом. Последние два поля возвращают информацию о времени ЦП, затраченном всеми завершившимися дочерними процессами, для которых родительский процесс (то есть процесс, вызвавший times()) выполнил системный вызов wait().

Тип данных clock_t, применяемый для задания типов четырех полей tms-структуры, является целочисленным типом, который используется для времени, замеренного в единицах, называемых *тиками часов*. Чтобы привести его к секундам, надо значение типа clock_t разделить на результат (значение, которое вернула sysconf(_SC_CLK_TCK)). (Функция sysconf() рассматривается в разделе 11.2.)

В большинстве аппаратных архитектур Linux sysconf(_SC_CLK_TCK) возвращает число 100. Это соответствует константе ядра USER_HZ. Но на некоторых архитектурах, таких как Alpha и IA-64, USER_HZ может быть определена со значением, отличным от 100.

В случае успешного завершения times() возвращает затраченное (реальное) время в тиках часов, прошедшее с некоторого момента в прошлом. В SUSv3 намеренно не указывается, что собой представляет этот момент, — там просто утверждается, что он не должен поменяться в течение всего времени существования вызывающего процесса. Поэтому единственный портируемый вариант использования этого возвращаемого значения — замерить затраченное время в выполнении процесса, вычислив разницу между значениями, возвращенными парой вызовов times(). Но даже при таком использовании возвращаемое значение times() не отличается надежностью, поскольку может переполнить допустимый диапазон значений типа clock_t, и тогда значение снова начнется с нуля. Иными словами, последующий вызов times() может вернуть число, меньшее, чем ранее сделанный вызов times(). Надежный способ замерить протекание затраченного времени — использовать функцию gettimeofday() (рассмотренную в разделе 10.1).

В Linux для аргумента buf можно указать значение NULL. В таком случае times() просто возвращает результат выполнения функции. Но портируемость при этом не достигается. Использование NULL в качестве значения аргумента buf в SUSv3 не определена, и многие другие реализации UNIX требуют применения для этого аргумента значения, отличного от NULL.

Простой интерфейс для извлечения времени процесса предоставляется функцией clock(). Она возвращает одно значение — замер общего (то есть пользовательского плюс системного) времени центрального процессора, использованного вызывающим процессом.

```
#include <time.h>

clock_t clock(void);
```

Возвращает при успешном завершении общее время центрального процессора, которое было использовано вызывающим процессом (измеряется в CLOCKS_PER_SEC), или значение (clock_t) -1 при ошибке

Значение, возвращенное функцией clock(), измеряется в единицах CLOCKS_PER_SEC, поэтому для получения количества используемого процессом времени ЦП в секундах результат нужно разделить на эту величину. Для CLOCKS_PER_SEC в POSIX.1 предусмотрено фиксированное значение 1 миллион, независимо от разрешения используемых программных часов (см. раздел 10.6). И все же точность clock() ограничена разрешением программных часов.

Хотя в функции clock() в качестве возвращаемого применяется тип clock_t, являющийся таким же типом данных, что используется в системном вызове times(), этими двумя интерфей-

сами задействуются разные единицы измерений. Это произошло в результате несогласованных определений типа clock_t в POSIX.1 и в стандарте языка программирования C.

Даже притом, что `CLOCKS_PER_SEC` имеет фиксированное значение (1 миллион), в SUSv3 указывается, что эта константа в несовместимых с XSI системах может быть целочисленной переменной, поэтому мы не можем портируемым образом обращаться с ней, как с константой, заданной во время компиляции (то есть мы не можем использовать ее в выражениях препроцессора `#ifdef`). Поскольку она может быть определена в качестве *длинного* целого числа (то есть `1000000L`), мы всегда приводим эту константу к `long`, чтобы ее можно было портируемо вывести на экран с помощью функции `printf()` (см. подраздел 3.6.2).

В SUSv3 утверждается, что функция `clock()` должна возвращать «время процессора, использованное процессом». Это утверждение можно по-разному интерпретировать. В некоторых реализациях UNIX время, возвращаемое функцией `clock()`, включает время центрального процессора, использованное всеми дочерними процессами, завершения которых пришлось ожидать. В Linux это не так.

Пример программы

Программа в листинге 10.5 демонстрирует использование функций, рассмотренных в данном разделе. Функция `displayProcessTimes()` выводит сообщение, предоставляемое вызывающим кодом, а затем задействует функции `clock()` и `times()` для извлечения и вывода на экран показателей времени процесса. Основная программа первый раз вызывает `displayProcessTimes()`, а затем выполняет цикл, потребляющий время центрального процессора путем многократного вызова функции `getppid()`, прежде чем снова вызвать функцию `displayProcessTimes()`, чтобы посмотреть, сколько времени ЦП было затрачено внутри цикла. При использовании этой программы для вызова `getppid()` 10 миллионов раз мы увидим следующее:

```
$ ./process_time 10000000
CLOCKS_PER_SEC=1000000  sysconf(_SC_CLK_TCK)=100

At program start:
    clock() returns: 0 clocks-per-sec (0.00 secs)
    times() yields: user CPU=0.00; system CPU: 0.00
After getppid() loop:
    clock() returns: 2960000 clocks-per-sec (2.96 secs)
    times() yields: user CPU=1.09; system CPU: 1.87
```

Листинг 10.5. Извлечение затраченного процессом времени ЦП

time/process_time.c

```
#include <sys/times.h>
#include <time.h>
#include "tlpi_hdr.h"

static void             /* Вывод сообщения, на которое указывает 'msg',
                           и показателей времени процесса */
displayProcessTimes(const char *msg)
{
    struct tms t;
    clock_t clockTime;
    static long clockTicks = 0;

    if (msg != NULL)
        printf("%s", msg);
```

```
    if (clockTicks == 0) {      /* Извлечение тиков часов в первом вызове */
        clockTicks = sysconf(_SC_CLK_TCK);
        if (clockTicks == -1)
            errExit("sysconf");
    }

    clockTime = clock();
    if (clockTime == -1)
        errExit("clock");

    printf("      clock() returns: %ld clocks-per-sec (%.2f secs)\n",
            (long) clockTime, (double) clockTime / CLOCKS_PER_SEC);

    if (times(&t) == -1)
        errExit("times");
    printf("        times() yields: user CPU=%.2f; system CPU: %.2f\n",
            (double) t.tms_utime / clockTicks,
            (double) t.tms_stime / clockTicks);
}

int
main(int argc, char *argv[])
{
    int numCalls, j;

    printf("CLOCKS_PER_SEC=%ld  sysconf(_SC_CLK_TCK)=%ld\n\n",
            (long) CLOCKS_PER_SEC, sysconf(_SC_CLK_TCK));

    displayProcessTimes("At program start:\n");

    numCalls = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-calls")
           : 100000000;
    for (j = 0; j < numCalls; j++)
        (void) getppid();

    displayProcessTimes("After getppid() loop:\n");

    exit(EXIT_SUCCESS);
}
```

time/process_time.c

10.8. Резюме

Реальное время соответствует обычному определению времени. Когда реальное время отмеряется от какого-то стандартного момента, мы называем его календарным временем; затраченным временем называется время, отмеряемое от какого-либо момента в жизни процесса (обычно от его запуска).

Время процесса представляет собой время центрального процессора, использованное процессом, и разбивается на пользовательский и системный компоненты.

Получить и установить значение системных часов (то есть календарного времени, замеренного в секундах от начала отсчета (Epoch)) позволяют различные системные вызовы, а некоторые библиотечные функции дают возможность выполнять преобразования между календарным временем и другими форматами времени, включая разделенное время (время, разбитое на компоненты) и время в виде читаемых символьных строк.

Описание этих преобразований не обходится без рассмотрения вопросов локалей и интернационализации.

Использование времени и даты, а также вывод их на экран важны для многих приложений, и функции, рассмотренные в этой главе, будут еще часто упоминаться на протяжении всей книги. Дополнительно вопросы замеров времени мы также рассмотрим в главе 23.

Дополнительные сведения

Подробности, касающиеся способов замеров времени ядром Linux, можно найти в [Love, 2010].

Подробно о часовых поясах и интернационализации вы можете прочитать в руководстве по GNU-библиотеке C (по адресу http://www.gnu.org/). Подробности относительно локалей также можно найти в документах по SUSv3.

10.9. Упражнение

10.1. Возьмем систему, где вызов `sysconf(_SC_CLK_TCK)` возвращает значение `100`. Сколько времени пройдет, пока значение типа `clock_t`, возвращенное функцией `times()`, снова превратится в `0`, при условии, что оно представлено 32-разрядным целым числом? Выполните такое же вычисление для значения `CLOCKS_PER_SEC`, возвращенного функцией `clock()`.

11 Системные ограничения и возможности

Каждая реализация UNIX накладывает ограничения (limits) на различные системные характеристики и ресурсы и предоставляет возможности (options), определенные в различных стандартах (или отказывает в их предоставлении). Например, можно определить следующее.

- Сколько файлов процесс может одновременно держать открытыми?
- Поддерживает ли система сигналы реального времени?
- Какое наибольшее значение может быть сохранено в переменной типа `int`?
- Насколько большим может быть список аргументов программы?
- Какова максимальная длина путевого имени?

Можно, конечно, жестко задать ограничения и возможности в самом приложении, но это снизит портируемость, поскольку все ограничения и возможности могут варьироваться:

- *в различных реализациях UNIX*. Хотя в отдельно взятых реализациях ограничения и возможности могут быть четко прописаны, от реализации к реализации они могут варьироваться. В качестве примера такого ограничения можно привести максимальное значение, которое может быть сохранено в `int`-переменной;
- *динамически в конкретной реализации*. К примеру, ядро может быть перенастроено с изменением ограничения. Кроме того, приложение может быть скомпилировано в одной системе, а запущено в другой, имеющей иные ограничения и возможности;
- *от одной файловой системы к другой*. Например, традиционные файловые системы System V позволяют для имени файла задействовать до 14 байт, а традиционные файловые системы BSD и большинство файловых систем, обычно используемых в Linux, допускают имена файлов длиной до 255 байт.

Поскольку ограничения и возможности системы оказывают влияние на возможности приложения, портируемое приложение нуждается в способах определения значений для ограничений и поддерживаемых возможностей. Стандарты языка программирования C и SUSv3 предоставляют приложению два основных способа получения этой информации.

- Некоторые ограничения и возможности известны в ходе компиляции. Например, максимальное значение переменной типа `int` определяется аппаратной архитектурой и деталями реализации компилятора. Эти ограничения могут быть записаны в заголовочных файлах.
- Другие ограничения и возможности могут изменяться в ходе выполнения приложения. Для таких случаев в SUSv3 определяются три функции — `sysconf()`, `pathconf()` и `fpathconf()`. Приложение может вызвать их для проверки ограничений и возможностей данной реализации UNIX.

В SUSv3 указывается диапазон ограничений, которые могут накладываться соответствующей реализацией, а также набор возможностей, каждая из которых может быть предоставлена или не предоставлена конкретной системой. В этой главе мы рассмотрим лишь некоторые из этих ограничений и возможностей, а другие будут описаны в последующих главах.

11.1. Системные ограничения

Для каждого определяемого ограничения в SUSv3 требуется, чтобы все реализации поддерживали его *минимальное значение*. В большинстве случаев такое значение определяется в виде константы в `<limits.h>` с именем, префиксом для которого служит строка `_POSIX_` и в котором (обычно) содержится строка `_MAX`. То есть имя имеет вид `_POSIX_`*`XXX`*`_MAX`.

Если приложение ограничивает себя минимальными значениями, указанными в SUSv3, оно будет портируемым для всех реализаций стандарта. Но это не дает ему права воспользоваться преимуществами реализаций, предоставляющих более высокие ограничения. Поэтому зачастую предпочтительнее определять ограничения конкретной системы через `<limits.h>`, `sysconf()` или `pathconf()`.

> Применение строки _MAX в названиях ограничений, определенных в SUSv3, может показаться странным, учитывая их описание как минимальных значений. Смысл названий проясняется, если заметить, что каждая из этих констант устанавливает верхний предел ресурсов или возможностей, и стандарты определяют, что этот верхний предел должен иметь конкретное минимальное значение.
>
> Иногда в качестве ограничения предоставляются максимальные значения, в именах которых присутствует строка _MIN. Для этих констант верно обратное утверждение: они представляют нижний предел какого-либо ресурса, и стандарты говорят, что в соответствующей реализации этот нижний предел не может быть больше определенного значения. Например, ограничение FLT_MIN (1E–37) задает наибольшее значение, которое реализация может установить для наименьшего числа с плавающей точкой из тех, что могут быть представлены, и все соответствующие стандарту реализации будут иметь возможность для представления чисел с плавающей точкой, по крайней мере таких же малых, как это.

У каждого ограничения есть свое название, которое соответствует показанному выше *названию минимального значения*, но без префикса `_POSIX_`. В файле `<limits.h>` реализаций *может быть* определена константа с таким именем, служащая признаком соответствующего ограничения для конкретной реализации. Если ограничение определено, то оно всегда будет по крайней мере того же размера, что и рассмотренное выше минимальное значение (то есть `XXX_MAX >= _POSIX_XXX_MAX`).

Указываемые в SUSv3 ограничения разбиты на три категории: *значения, не изменяемые динамически (runtime)*, *изменяемые значения путевых имен* и *значения, которые могут увеличиваться динамически*. Далее эти категории будут рассмотрены на примерах.

Значения, не изменяемые динамически (возможно, неопределенные)

Не изменяемое динамически значение является ограничением, чье значение, если оно определено в `<limits.h>`, зафиксировано для реализации. Но значение может быть неопределенным (возможно, по причине его зависимости от доступного пространства памяти), в силу чего его может не быть в файле `<limits.h>`. В таком случае (и даже если ограничение задано в `<limits.h>`) приложение для определения значения в ходе своего выполнения может воспользоваться функцией `sysconf()`.

Примером такого не изменяемого динамически ограничения может быть `MQ_PRIO_MAX`. Как отмечается в разделе 48.5.1, это ограничение приоритетов для сообщений в очереди сообщений POSIX. В SUSv3 определена константа `_POSIX_MQ_PRIO_MAX` со значением `32`, которое служит в качестве минимального значения и должно предоставляться для этого

ограничения всеми соответствующими реализациями. Иными словами, мы можем быть уверены, что все соответствующие реализации позволят использовать для сообщений приоритеты от 0 и как минимум до 31. Реализация UNIX может установить и более высокое ограничение, определив в `<limits.h>` константу `MQ_PRIO_MAX` с его значением. Например, в Linux константа `MQ_PRIO_MAX` определена со значением `32768`. Оно также может быть определено во время выполнения программы с помощью такого вызова:

```
lim = sysconf(_SC_MQ_PRIO_MAX);
```

Изменяемые значения путевых имен

Изменяемые значения путевых имен — это ограничения, относящиеся к путевым именам (файлов, каталогов, терминалов и т. д.). Каждое ограничение может быть константой для отдельно взятой реализации или может изменяться от одной файловой системы к другой. В тех случаях, когда ограничение может изменяться в зависимости от путевого имени, приложение способно определить его значение с помощью функции `pathconf()` или `fpathconf()`.

Примером изменяемого значения путевого имени может послужить ограничение `NAME_MAX`. Оно определяет максимальный размер имени файла в конкретной файловой системе. В SUSv3 предусмотрена константа `_POSIX_NAME_MAX` со значением `14` (это ограничение из старой файловой системы System V), используемым в качестве минимального значения, которое должна допускать реализация. В реализации может быть определена константа `NAME_MAX` с ограничением выше этого значения, и (или же) информация о конкретной файловой системе может быть доступна по такому вызову:

```
lim = pathconf(directory_path, _PC_NAME_MAX)
```

Аргумент `directory_path` является путевым именем для каталога интересующей нас файловой системы.

Значения, которые могут увеличиваться в ходе выполнения программы

Значение, которое может увеличиваться в ходе выполнения программы, является ограничением, имеющим для конкретной реализации фиксированное минимальное значение. Все системы, в которых запущена данная реализация, будут предоставлять по крайней мере это минимальное значение. Но конкретная система может поднять это ограничение в ходе выполнения программы, а приложение может определить конкретное поддерживаемое в системе значение с помощью функции `sysconf()`.

Примером значения, которое может увеличиваться в ходе выполнения программы, может послужить константа `NGROUPS_MAX`. Она определяет максимальное количество одновременно используемых для процесса дополнительных групповых идентификаторов (см. раздел 9.6). В SUSv3 установлено соответствующее минимальное значение: `_POSIX_NGROUPS_MAX`, равное `8`. В ходе выполнения программы приложение может извлечь ограничение с помощью вызова `sysconf(_SC_NGROUPS_MAX)`.

Отдельные ограничения, определенные в SUSv3

В табл. 11.1 приведен список некоторых установленных в SUSv3 ограничений, имеющих отношение к материалам данной книги (остальные ограничения будут описаны в последующих главах).

Таблица 11.1. Отдельные ограничения, определенные в SUSv3

Название ограничения (<limits.h>)	Минимальное значение	sysconf()/pathconf() название (<unistd.h>)	Описание
ARG_MAX	4096	_SC_ARG_MAX	Максимальное количество байтов для аргументов (argv) и для переменных среды (environ), которое может быть предоставлено exec() (см. раздел 6.7 и подраздел 27.2.3)
Не определено	Не определено	_SC_CLK_TCK	Единица измерения для times()
LOGIN_NAME_MAX	9	_SC_LOGIN_NAME_MAX	Максимальный размер имени для входа в систему (включая завершающий нулевой байт)
OPEN_MAX	20	_SC_OPEN_MAX	Максимальное количество файловых дескрипторов, которые могут быть одновременно открыты процессом. Наибольший номер дескриптора, который можно задействовать, на единицу меньше, чем это число (см. раздел 36.2)
NGROUPS_MAX	8	_SC_NGROUPS_MAX	Максимальное количество дополнительных идентификаторов групп, в которые может входить процесс (см. подраздел 9.7.3)
Не определено	1	_SC_PAGESIZE	Размер страницы виртуальной памяти (синонимом является _SC_PAGE_SIZE)
RTSIG_MAX	8	_SC_RTSIG_MAX	Максимальное количество различных сигналов реального времени (см. раздел 22.8)
SIGQUEUE_MAX	32	_SC_SIGQUEUE_MAX	Максимальное количество сигналов реального времени, поставленных в очередь (см. раздел 22.8)

Продолжение ⇙

Таблица 11.1 *(продолжение)*

Название ограничения (<limits.h>)	Минимальное значение	sysconf()/pathconf() название (<unistd.h>)	Описание
STREAM_MAX	8	_SC_STREAM_MAX	Максимальное количество потоков стандартного ввода-вывода, которые могут быть открыты одновременно
NAME_MAX	14	_PC_NAME_MAX	Максимальное количество байтов в имени файла, не включая завершающий нулевой байт
PATH_MAX	256	_PC_PATH_MAX	Максимальное количество байтов в путевом имени, включая завершающий нулевой байт
PIPE_BUF	512	_PC_PIPE_BUF	Максимальное количество байтов, которые могут быть атомарно записаны в конвейер или в FIFO (см. раздел 44.1)

В первом столбце табл. 11.1 дается название ограничения, которое может быть определено в виде константы в файле `<limits.h>` для указания ограничения в конкретной реализации. Во втором столбце приводится определенный в SUSv3 минимум для ограничения (также указан в `<limits.h>`). В большинстве случаев каждое из минимальных значений определяется в качестве константы с префиксом в виде строки `_POSIX_`. Например, константа `_POSIX_RTSIG_MAX` (определенная со значением `8`) указывает требуемый в SUSv3 минимум, соответствующий константе реализации `RTSIG_MAX`. В третьем столбце приводится имя константы, которое может быть передано в ходе выполнения программы в функции `sysconf()` или `pathconf()` с целью извлечения ограничения, свойственного конкретной реализации. Константы, начинающиеся с `_SC_`, предназначены для использования с `sysconf()`, а константы, начинающиеся с `_PC_`, предназначены для применения с `pathconf()` и `fpathconf()`.

В качестве дополнения к табл. 11.1 обратите внимание на следующую информацию.

- Функция `getdtablesize()` является устаревшей альтернативой для определения ограничения для файловых дескрипторов процесса (`OPEN_MAX`). Она была указана в SUSv2 (с пометкой `LEGACY` — «устаревшая»), но из SUSv3 была удалена.
- Функция `getpagesize()` — устаревшая альтернатива для определения размера страницы в системе (`_SC_PAGESIZE`). Эта функция была указана в SUSv2 (с пометкой `LEGACY` — «устаревшая»), но из SUSv3 была удалена.
- Константа `FOPEN_MAX`, определенная в `<stdio.h>`, является синонимом константы `STREAM_MAX`.
- В `NAME_MAX` не учитывается завершающий нулевой байт, в то время как в `PATH_MAX` он учитывается. Это противоречие исправляет ранее допущенную непоследовательность в стандарте POSIX.1, когда было непонятно, учитывается ли завершающий нулевой байт в `PATH_MAX`. Определение константы `PATH_MAX` как учитывающей завершающий нулевой байт означает, что приложения, выделяющие лишь указанное в `PATH_MAX` количество байтов для путевого имени, будут по-прежнему соответствовать стандарту.

Выявление ограничений и возможностей из оболочки: getconf

Для получения ограничений и возможностей конкретной реализации UNIX из оболочки можно использовать команду `getconf`. Основной для этой команды является такая форма:

```
$ getconf variable-name [ pathname ]
```

Аргумент `variable-name` идентифицирует требуемое ограничение и является одним из стандартных имен ограничений, указанных в SUSV3, например `ARG_MAX` или `NAME_MAX`. Когда ограничение имеет отношение к путевому имени, то в качестве второго аргумента в команде нужно указывать путевое имя (`pathname`) (см. второй пример ниже).

```
$ getconf ARG_MAX
131072
$ getconf NAME_MAX /boot
255
```

11.2. Извлечение в ходе выполнения программы значений ограничений (и возможностей) системы

Функция `sysconf()` позволяет приложению получить значения системных ограничений в ходе выполнения программы.

```
#include <unistd.h>

long sysconf(int name);
```

Возвращает значение ограничения, указанного в аргументе *name*, при успешном завершении или `-1`, если ограничение не определено или же если произошла ошибка

Аргумент `name` является одной из констант вида `_SC_*`, определенных в файле `<unistd.h>` (см. табл. 11.1). Значение ограничения возвращается в качестве результата выполнения функции.

Если ограничение не может быть определено, функция `sysconf()` выдает `-1`. Она также может возвратить `-1`, если случится ошибка. (Единственной указываемой ошибкой является `EINVAL`, что означает недопустимость имени.) Чтобы отличить неопределенное ограничение от ошибки, нужно установить для `errno` перед вызовом значение `0`. Если вызов возвратит `-1` и после вызова для `errno` будет установлено значение, значит, произошла ошибка.

> Значения ограничений, возвращенные sysconf() (а также pathconf() и fpathconf()), всегда относятся к (длинному) целочисленному типу данных (long). В пояснительном тексте для sysconf() в SUSv3 отмечается, что в качестве возможных возвращаемых значений рассматривались строки, но они были отвергнуты из-за сложности реализации и использования.

В листинге 11.1 показывается пример использования функции `sysconf()` для вывода различных ограничений системы. Запуск этой программы в одной из систем Linux 2.6.31/x86-32 приводит к выдаче следующей информации:

```
$ ./t_sysconf
_SC_ARG_MAX:        2097152
```

```
_SC_LOGIN_NAME_MAX: 256
_SC_OPEN_MAX:       1024
_SC_NGROUPS_MAX:    65536
_SC_PAGESIZE:       4096
_SC_RTSIG_MAX:      32
```

Листинг 11.1. Использование sysconf()

syslim/t_sysconf.c

```
#include "tlpi_hdr.h"

static void /* Выводит 'msg' плюс значение sysconf() для 'name' */
sysconfPrint(const char *msg, int name)
{
    long lim;

    errno = 0;
    lim = sysconf(name);
    if (lim != -1) { /* Вызов прошел успешно, ограничение определено */
        printf("%s %ld\n", msg, lim);
    } else {
        if (errno == 0)
        /* Вызов прошел успешно, ограничение не определено */
            printf("%s (indeterminate)\n", msg);
        else /* Вызов не удался */
            errExit("sysconf %s", msg);
    }
}

int
main(int argc, char *argv[])
{
    sysconfPrint("_SC_ARG_MAX:        ", _SC_ARG_MAX);
    sysconfPrint("_SC_LOGIN_NAME_MAX: ", _SC_LOGIN_NAME_MAX);
    sysconfPrint("_SC_OPEN_MAX:       ", _SC_OPEN_MAX);
    sysconfPrint("_SC_NGROUPS_MAX:    ", _SC_NGROUPS_MAX);
    sysconfPrint("_SC_PAGESIZE:       ", _SC_PAGESIZE);
    sysconfPrint("_SC_RTSIG_MAX:      ", _SC_RTSIG_MAX);
    exit(EXIT_SUCCESS);
}
```

syslim/t_sysconf.c

В SUSv3 требуется, чтобы значение, возвращенное функцией `sysconf()` для конкретного ограничения, было постоянным на всем протяжении жизненного цикла вызывающего процесса. Например, можно предполагать, что значение, возвращаемое для `_SC_PAGESIZE`, не будет изменяться, пока продолжается работа процесса.

В Linux предусмотрены некоторые (разумные) исключения для утверждения, что значения ограничения постоянны на протяжении всего существования процесса. Для изменения ограничений своих различных ресурсов процесс может воспользоваться функцией setrlimit() (см. раздел 36.2). Она влияет на значения ограничений, возвращаемые функцией sysconf(). Например, ограничение RLIMIT_NOFILE определяет количество файлов, доступных для открытия процессу (_SC_OPEN_MAX); RLIMIT_NPROC (ограничение ресурса, не указанное в SUSv3) ограничивает для каждого пользователя возможное количество процессов, создаваемых им в данном процессе (_SC_CHILD_MAX); RLIMIT_STACK определяет (начиная с версии 2.6.23) предел пространства, выделяемого для аргументов и переменных окружения командной строки процесса (_SC_ARG_MAX; подробности можно найти на странице руководства execve(2)).

11.3. Извлечение в ходе выполнения программы значений ограничений (и возможностей), связанных с файлами

В ходе своего выполнения приложение может получить значения ограничений, связанных с файлами. Для этого предназначены функции `pathconf()` и `fpathconf()`.

```
#include <unistd.h>

long pathconf(const char *pathname , int name);
long fpathconf(int fd, int name);
```

Обе функции возвращают при успешном завершении значение ограничения, указанного с помощью аргумента *name*, или `-1`, если ограничение не определено или произошла ошибка

Единственное различие между `pathconf()` и `fpathconf()` заключается в способе указания файла или каталога. Для `pathconf()` — в виде путевого имени в аргументе `pathname`, а для `fpathconf()` — через дескриптор предварительно открытого файла.

Аргумент `name` является одной из констант вида `_PC_*`, определенных в `<unistd.h>` (см. табл. 11.1). Некоторые дополнительные подробности относительно констант вида `_PC_*` также приведены в табл. 11.2.

В качестве результата выполнения функции возвращается значение ограничения. Отличить возвращение, связанное с неопределенным ограничением, от ошибки можно точно так же, как и при использовании функции `sysconf()`.

В отличие от `sysconf()`, в SUSv3 не требуется, чтобы значения, возвращаемые `pathconf()` и `fpathconf()`, оставались неизменными в течение всего жизненного цикла процесса, поскольку, к примеру, файловая система в ходе выполнения процесса может быть демонтирована или смонтирована с другими характеристиками.

Таблица 11.2. Подробности отдельных имен вида _PC_*, используемых при вызове pathconf()

Константа	Примечание
_PC_NAME_MAX	Для каталога это имя приводит к выдаче значения для файлов в каталоге. Поведение для других типов файлов не определено
_PC_PATH_MAX	Для каталога это имя приводит к выдаче максимальной длины относительного путевого имени из этого каталога. Поведение для других типов файлов не определено
_PC_PIPE_BUF	Для FIFO-устройства или конвейера это имя приводит к выдаче значения, относящегося к указанному файлу. Для каталога это значение относится к FIFO-устройству, созданному в данном каталоге. Поведение для других типов файлов не определено

В листинге 11.2 показано использование функции `fpathconf()` для извлечения различных ограничений для файла, который будем передавать с помощью перенаправления стандартного ввода. При запуске этой программы с указанием в качестве стандартного ввода каталога файловой системы ext2 будет показано следующее:

```
$ ./t_fpathconf < .
_PC_NAME_MAX: 255
_PC_PATH_MAX: 4096
_PC_PIPE_BUF: 4096
```

Листинг 11.2. Использование fpathconf()

syslim/t_fpathconf.c

```
#include "tlpi_hdr.h"

static void /* Выводит 'msg' плюс значение fpathconf(fd, name) */
fpathconfPrint(const char *msg, int fd, int name)
{
    long lim;

    errno = 0;
    lim = fpathconf(fd, name);
    if (lim != -1) { /* Вызов прошел успешно, ограничение определено */
        printf("%s %ld\n", msg, lim);
    } else {
        if (errno == 0)
        /* Вызов прошел успешно, ограничение не определено */
            printf("%s (indeterminate)\n", msg);
        else /* Вызов не удался */
            errExit("fpathconf %s", msg);
    }
}

int
main(int argc, char *argv[])
{
    fpathconfPrint("_PC_NAME_MAX: ", STDIN_FILENO, _PC_NAME_MAX);
    fpathconfPrint("_PC_PATH_MAX: ", STDIN_FILENO, _PC_PATH_MAX);
    fpathconfPrint("_PC_PIPE_BUF: ", STDIN_FILENO, _PC_PIPE_BUF);
    exit(EXIT_SUCCESS);
}
```

syslim/t_fpathconf.c

11.4. Неопределенные ограничения

Иногда может оказаться, что какое-то системное ограничение в реализации не определено с помощью константы (например, `PATH_MAX`), поэтому функция `sysconf()` или `pathconf()` информирует нас, что ограничение (например, `_PC_PATH_MAX`) не определено. В таком случае можно применить одну из следующих стратегий.

- При написании приложения, предусматривающего портируемость между несколькими реализациями UNIX, можно выбрать использование минимального значения ограничения, указанного в SUSv3. Эти значения задаются константами вида `_POSIX_*_MAX` (см. раздел 11.1). Но иногда такой подход может не сработать, поскольку ограничение имеет невероятно низкое значение, как в случае `_POSIX_PATH_MAX` и `_POSIX_OPEN_MAX`.
- В некоторых случаях более практичным может стать игнорирование проверок ограничений и выполнение вместо этого соответствующих системных вызовов или вызовов библиотечных функций. (Такие же аргументы могут применяться и в отношении некоторых указанных в SUSv3 возможностей, рассмотренных в разделе 11.5.) Если вызов терпит неудачу и `errno` показывает, что ошибка произошла по причине превышения

некоторых системных ограничений, затем можно повторить попытку, изменив при необходимости поведение приложения. Например, большинство реализаций UNIX налагает ограничение на количество сигналов реального времени, которые могут быть поставлены в очередь процесса. Как только это ограничение будет достигнуто, попытки отправить дополнительные сигналы (с использованием `sigqueue()`) будут отклоняться с выдачей ошибки `EAGAIN`. В этом случае процесс, отправляющий сигнал, может просто повторить попытку, возможно, после небольшой паузы. Подобным образом попытка открыть файл со слишком длинным именем приводит к возникновению ошибки `ENAMETOOLONG`, и приложение может справиться с данной ситуацией, повторив попытку с использованием более короткого имени.

- Можно написать свою собственную программу или функцию, чтобы либо вычислить, либо примерно оценить ограничение. Во всех случаях делается соответствующий вызов `sysconf()` или `pathconf()`, и, если ограничение не определено, функция возвращает значение, соответствующее «разумной догадке». Пусть и не совершенное, но вполне рабочее решение.
- Можно применить такое расширяемое инструментальное средство, как GNU Autoconf. Эта программа способна определить существование и установки различных системных возможностей и ограничений. Она создает заголовочные файлы на основе определяемой информации, а эти файлы могут затем включаться в программы на языке C. Дополнительные сведения о программе Autoconf можно найти по адресу http://www.gnu.org/software/autoconf/.

11.5. Системные возможности

Наряду с указанием ограничений для различных системных ресурсов в SUSv3 указываются различные возможности, которые могут поддерживаться реализацией UNIX. В их числе сигналы реального времени, совместно используемая память POSIX, управление заданиями и потоки POSIX. За некоторыми исключениями, от реализаций не требуется поддержка этих возможностей. Вместо этого в SUSv3 реализации разрешается сообщать, как в ходе компиляции, так и в ходе выполнения программы, о поддержке той или иной конкретной возможности.

Реализация может объявлять о поддержке конкретной упоминаемой в SUSv3 возможности в ходе компиляции путем определения соответствующей константы в заголовочном файле `<unistd.h>`. Каждая такая константа начинается с префикса, определяющего стандарт ее происхождения (например, `_POSIX_` или `_XOPEN_`).

Каждая константа возможности, если она определена, имеет одно из следующих значений.

- `-1` означает, что *возможность не поддерживается*. В этом случае в данной реализации не обязаны быть определены заголовочные файлы, типы данных и интерфейсы функций, связанные с возможностью. Такой вариант можно обработать с помощью условной компиляции с применением директив препроцессора `#if`.
- `0` означает, что *возможность может поддерживаться*. Приложение должно в ходе своего выполнения проверить поддержку возможности.
- Значение больше нуля говорит о том, что *возможность поддерживается*. Все заголовочные файлы, типы данных и интерфейсы функций, связанные с возможностью, определяются и ведут себя соответствующим образом. Во многих случаях в SUSv3 требуется, чтобы это положительное значение было в виде константы `200112L`, соответствующей году и номеру месяца принятия SUSv3 в качестве стандарта. (Аналогичное значение в SUSv4 имеет вид `200809L`.)

Когда константа определена со значением 0, приложение в ходе своего выполнения для проверки поддержки возможности может использовать функции sysconf() и pathconf() (или fpathconf()). Аргумент name, передаваемый этим функциям, обычно имеет такую же форму, как и соответствующая константа времени компиляции, но с префиксом, замененным на _SC_ или _PC_. Реализация должна предоставить как минимум заголовочные файлы, константы и интерфейсы функций, необходимые для проверки в ходе выполнения программы.

В SUSv3 непонятно, что именно означает неопределенная константа: то же, что и определенная константа со значением 0 («возможность может поддерживаться») или же константа со значением –1 («возможность не поддерживается»). Комитет по стандартам впоследствии решил, что этот случай должен означать то же самое, что и конкретная константа со значением –1, и в SUSv4 это четко определено.

В табл. 11.3 приводится список некоторых возможностей, указанных в SUSv3. В первом столбце для возможности дается название связанной с ней константы времени компиляции (определенной в <unistd.h>), а также соответствующие имена для аргументов функции sysconf() (_SC_*) или pathconf() (_PC_*). Обратите внимание на следующие примечания по поводу некоторых возможностей.

- Некоторые возможности указаны в SUSv3 как обязательные, то есть константа времени компиляции всегда устанавливается в значение больше нуля. В прежние времена эти возможности фактически были необязательными, но теперь это не так. Эти возможности в столбце примечания помечены символом «плюс» (+). (Будучи необязательными в SUSv3, некоторые свойства стали обязательными в SUSv4.)

  Несмотря на то что эти возможности указаны в SUSv3 как обязательные, в отдельных системах UNIX они все равно могут быть установлены в не соответствующей стандарту конфигурации. Поэтому для портируемых приложений лучше, наверное, будет проверить, поддерживается ли возможность, влияющая на работоспособность приложения, независимо от того, что стандарт требует ее обязательной поддержки.

- Для некоторых возможностей константа времени компиляции может иметь значение, отличное от -1. Иными словами, либо возможность должна поддерживаться, либо ее поддержка в ходе выполнения программы должна быть доступна для проверки. Эти возможности в столбце примечания помечены звездочкой (*).

Таблица 11.3. Отдельные возможности, определенные в SUSv3

Название возможности (константы) (имя для sysconf() или pathconf())	**Описание**	**Примечание**
_POSIX_ASYNCHRONOUS_IO (_SC_ ASYNCHRONOUS_IO)	Асинхронный ввод/вывод	
_POSIX_CHOWN_RESTRICTED (_PC_ CHOWN_RESTRICTED)	Только привилегированные процессы могут применять chown() и fchown() для изменения пользовательского и группового идентификатора файла на произвольные значения (см. подраздел 15.3.2)	*
_POSIX_JOB_CONTROL (_SC_JOB_ CONTROL)	Управление заданиями (см. раздел 34.7)	+

Название возможности (константы) (имя для sysconf() или pathconf())	Описание	Примечание
_POSIX_MESSAGE_PASSING (_SC_MESSAGE_PASSING)	Очереди сообщений POSIX (см. главу 48)	
_POSIX_PRIORITY_SCHEDULING (_SC_PRIORITY_SCHEDULING)	Диспетчеризация процессов (см. раздел 35.3)	
_POSIX_REALTIME_SIGNALS (_SC_REALTIME_SIGNALS)	Расширение сигналов реального времени (см. раздел 22.8)	
_POSIX_SAVED_IDS (не определено)	У процесса есть сохраненные установленные идентификаторы пользователей (saved set-user-ID) и сохраненные установленные идентификаторы групп (saved set-group-ID) (см. раздел 9.4)	+
_POSIX_SEMAPHORES (_SC_SEMAPHORES)	Семафоры POSIX (см. главу 49)	
_POSIX_SHARED_MEMORY_OBJECTS (_SC_SHARED_MEMORY_OBJECTS)	Объекты совместно используемой памяти POSIX (см. главу 50)	
_POSIX_THREADS (_SC_THREADS)	Потоки POSIX	
_XOPEN_UNIX (_SC_XOPEN_UNIX)	Поддержка XSI-расширения (см. подраздел 1.3.4)	

11.6. Резюме

В SUSv3 указываются ограничения, которые могут накладываться реализацией, и системные возможности, которые реализация может поддерживать.

Зачастую желательно не задавать жестко в коде программы предположения насчет системных ограничений и возможностей, поскольку от реализации к реализации, а также в отдельно взятой реализации они могут варьироваться: либо во время выполнения программы, либо между файловыми системами. В результате в SUSv3 указываются методы, при использовании которых реализация может извещать об установленных ограничениях и поддерживаемых возможностях. Для большинства ограничений в SUSv3 указываются минимальные значения, которые должны поддерживаться всеми реализациями. Кроме того, каждая реализация может известить о свойственных ей ограничениях и возможностях в ходе компиляции (через константы, определенные в заголовочных файлах `<limits.h>` или `<unistd.h>`) и (или) в ходе выполнения программы (через вызов `sysconf()`, `pathconf()` или `fpathconf()`). Эти методы также можно использовать, чтобы выяснить, какие возможности, указанные в SUSv3, поддерживаются реализацией. В некоторых случаях возможность определить конкретное ограничение с помощью любого из этих методов может и не представиться. Тогда, чтобы установить ограничение, которого должно придерживаться приложение, следует прибегать к специальным методам.

Дополнительная информация

Основные вопросы, рассмотренные в данной главе, также освещаются в главе 2 издания [Stevens & Rago, 2005] и в главе 2 издания [Gallmeister, 1995]. В книге [Lewine, 1991] также предоставляются более ценные (хотя и немного устаревшие) основы. Некоторую

информацию о возможностях POSIX с замечаниями относительно glibc и подробностями, касающимися Linux, можно найти по адресу http://people.redhat.com/drepper/posix-option-groups.html. К данной теме также имеют отношение следующие страницы руководства по Linux: `sysconf(3)`, `pathconf(3)`, `feature_test_macros(7)`, `posixoptions(7)` и `standards(7)`.

Лучшими источниками информации (хотя иногда и сложными для понимания) являются соответствующие части SUSv3, особенно глава 2 из Base Definitions (XBD), и спецификации для `<unistd.h>`, `<limits.h>`, `sysconf()` и `fpathconf()`. Руководство по использованию SUSv3 предоставляется в издании [Josey, 2004].

11.7. Упражнения

11.1. Попробуйте запустить программу из листинга 11.1 в других реализациях UNIX, если у вас есть такая возможность.

11.2. Попробуйте запустить программу из листинга 11.2 в других файловых системах.

12 Информация о системе и процессе

В этой главе рассматриваются способы получения различной информации о системе и процессе. Основное внимание в ней уделяется файловой системе `/proc`. Кроме того, дается описание системного вызова `uname()`, используемого для извлечения различных идентификаторов системы.

12.1. Файловая система /proc

В старых реализациях UNIX не было простого способа выполнить интроспективный анализ атрибутов ядра для получения ответов на следующие вопросы.

- Сколько процессов запущено в системе и кто их владельцы?
- Какие файлы открыты процессом?
- Какие файлы в данный момент заблокированы и какие процессы удерживают эти блокировки?
- Какие сокеты используются в системе?

В некоторых старых реализациях эта проблема решалась тем, что привилегированным программам разрешалось анализировать структуры данных в ядре. Но такой подход имел несколько недостатков. В частности, он требовал специализированных знаний о структурах данных ядра, а эти структуры могли претерпевать изменения от одной версии ядра к другой, в силу чего программы, зависящие от этих структур, нужно было переделывать.

Чтобы предоставить более легкий доступ к информации ядра, во многих современных реализациях UNIX предусмотрена виртуальная файловая система `/proc`. Она находится в каталоге `/proc` и содержит различные файлы, предоставляющие информацию о ядре. Процессам можно беспрепятственно считывать эту информацию и в некоторых случаях вносить в нее изменения, используя обычные системные вызовы файлового ввода-вывода. Файловая система `/proc` называется виртуальной потому, что содержащиеся в ней файлы и подкаталоги не находятся на диске. Вместо этого ядро создает их на лету по мере обращения к ним процессов.

В этом разделе дается обзор файловой системы `/proc`. Конкретные `/proc`-файлы описываются в последующих главах. Хотя файловая система `/proc` предоставляется многими реализациями UNIX, ее описание в SUSv3 отсутствует, и все подробности, указанные в данной книге, относятся к операционной системе Linux.

12.1.1. Получение информации о процессе: /proc/PID

Для каждого процесса в системе ядро предоставляет соответствующий каталог по имени `/proc/`*`PID`*, где *`PID`* является идентификатором процесса. Внутри этого каталога находятся различные файлы и подкаталоги, содержащие информацию о процессе. Например, просмотрев файлы в каталоге `/proc/1`, можно получить информацию о процессе `init`, идентификатор которого всегда имеет значение `1`.

Среди файлов в каждом каталоге `/proc/PID` есть файл по имени `status`, предоставляющий множество данных о процессе:

```
$ cat /proc/1/status
Name:    init                    Имя исполняемого файла
State:   S (sleeping)            Состояние процесса
Tgid:    1                       ID группы потоков (обычный PID, getpid())
Pid:     1                       Фактически ID потока (gettid())
PPid:    0                       ID родительского процесса
TracerPid:    0                  PID отслеживающего процесса
                                 (0, если не отслеживается)
Uid:     0    0    0    0        Набор UID: реальный, действующий,
                                 сохраненный и файловой системы
Gid:     0    0    0    0        Набор GID: реальный, действующий,
                                 сохраненный и файловой системы
FDSize:  256                     Текущее количество выделенных дескрипторов файлов
Groups:                          Дополнительные групповые идентификаторы
VmPeak:       852 kB             Пиковое значение размера виртуальной памяти
VmSize:       724 kB             Текущее значение виртуальной памяти
VmLck:          0 kB             Заблокированная память
VmHWM:        288 kB             Пиковый размер резидентного набора
VmRSS:        288 kB             Текущий размер резидентного набора
VmData:       148 kB             Размер сегмента данных
VmStk:         88 kB             Размер стека
VmExe:        484 kB             Размер текстового сегмента (исполняемого кода)
VmLib:          0 kB             Размер кода совместно используемой библиотеки
VmPTE:         12 kB             Размер таблицы страниц (начиная с версии 2.6.10)
Threads:        1                Количество потоков в данной группе потоков
SigQ:    0/3067                  Текущее/максимальное количество сигналов
                                 в очереди (начиная с версии 2.6.12)
SigPnd:  0000000000000000        Маска сигналов, ожидающих по потокам
ShdPnd:  0000000000000000        Маска сигналов, ожидающих процесса
                                 (начиная с версии 2.6)
SigBlk:  0000000000000000        Маска заблокированных сигналов
SigIgn:  fffffffe5770d8fc        Маска игнорируемых сигналов
SigCgt:  00000000280b2603        Маска перехватываемых сигналов
CapInh:  0000000000000000        Маска наследуемых мандатов
CapPrm:  00000000ffffffff        Маска разрешенных мандатов
apEff:   00000000fffffeff        Маска действующих мандатов
CapBnd:  00000000ffffffff        Маска множества, ограничивающего мандаты
                                 (начиная с версии 2.6.26)
Cpus_allowed:   1                Маска разрешенных центральных процессоров
                                 (начиная с версии 2.6.24)
Cpus_allowed_list:   0           То же, что и выше, но в виде списка
                                 (начиная с версии 2.6.26)
Mems_allowed:   1                Маска разрешенных узлов памяти
                                 (начиная с версии 2.6.24)
Mems_allowed_list:   0           То же, что и выше, но в виде списка
                                 (начиная с версии 2.6.26)
voluntary_ctxt_switches: 6998    Преднамеренные переключения контекста
                                 (начиная с версии 2.6.23)
nonvoluntary_ctxt_switches: 107  Вынужденные переключения контекста
                                 (начиная с версии 2.6.23)
Stack usage:     8 kB            Метка наивысшего уровня использования стека
                                 (начиная с версии 2.6.32)
```

Эта информация была получена с использованием ядра версии 2.6.32. Из сопроводительных комментариев с метками «начиная с версии» видно, что формат со временем

изменялся и к нему в различных версиях ядра добавлялись новые поля (а иногда поля и удалялись). (Кроме отмеченных выше изменений, привнесенных Linux 2.6, в Linux 2.4 были добавлены поля `Tgid`, `TracerPid`, `FDSize` и `Threads`.)

Тот факт, что содержимое этого файла со временем изменяется, обуславливает следующий подход использования `/proc`-файлов: когда они состоят из множества записей, их нужно анализировать с оглядкой и искать в таком случае совпадение со строкой, содержащей конкретное строковое значение (например, `PPid:`), а не работать с файлом по логическим номерам строк.

В табл. 12.1 перечислены другие файлы, которые находятся в каждом каталоге `/proc/`*`PID`*.

Таблица 12.1. Отдельные файлы в каждом каталоге /proc/PID

Файл	**Описание (атрибут процесса)**
cmdline	Аргументы командной строки с \0 в качестве разделителя
cwd	Символьная ссылка на текущий рабочий каталог
environ	Пары вида ИМЯ=значение списка переменных среды с \0 в качестве разделителя
exe	Символьная ссылка на выполняемый файл
fd	Каталог, содержащий символьные ссылки на файлы, открытые данным процессом
maps	Отображения памяти
mem	Виртуальная память процесса (для получения правильного смещения перед вводом-выводом следует воспользоваться функцией lseek())
mounts	Точки монтирования для данного процесса
root	Символьная ссылка на корневой каталог
status	Различная информация (например, идентификаторы процесса, полномочия, использование памяти, сигналы)
task	Содержит по одному подкаталогу для каждого потока в процессе (Linux 2.6)

Каталог /proc/PID/fd

В каталоге `/proc/`*`PID`*`/fd` содержится по одной символьной ссылке для каждого файлового дескриптора, открытого процессом. Каждая из этих символьных ссылок имеет название, совпадающее с номером дескриптора, например, `/proc/1968/fd/1` является символьной ссылкой на стандартный вывод процесса 1968. Дополнительные сведения можно найти в разделе 5.11.

Для удобства, любой процесс может обратиться к своему собственному каталогу `/proc/`*`PID`* с помощью символьной ссылки `/proc/self`.

Потоки: каталог /proc/PID/task

В Linux 2.4 для соответствующей поддержки модели потоков POSIX добавилось понятие групп потоков. Поскольку некоторые атрибуты для потоков в группе потоков различаются, в Linux 2.4 добавился подкаталог `task`, расположенный в каталоге `/proc/`*`PID`*. Для каждого имеющегося в процессе потока ядро предоставляет подкаталог `/proc/`*`PID`*`/task/`*`TID`*, где *`TID`* является идентификатором потока. (То же самое число будет возвращено при вызове в потоке функции `gettid()`.)

В подкаталоге `/proc/`*`PID`*`/task/`*`TID`* находится набор файлов и каталогов, в точности похожий на расположенный в каталоге `/proc/`*`PID`*. Поскольку потоки совместно

используют большое количество атрибутов, множество сведений в этих файлах одинаково для каждого из потоков процесса. Но там, где есть для этого смысл, в таких файлах для каждого из потоков показывается различная информация. Например, в файлах `/proc/PID/task/TID/status` для группы потоков некоторые поля `State`, `Pid`, `SigPnd`, `SigBlk`, `CapInh`, `CapPrm`, `CapEff` и `CapBnd` могут иметь для каждого потока различные значения.

12.1.2. Системная информация, находящаяся в /proc

Доступ к информации, распространяющейся на всю систему, предоставляется в различных файлах и подкаталогах, находящихся в `/proc`. Некоторые из них показаны на рис. 12.1.

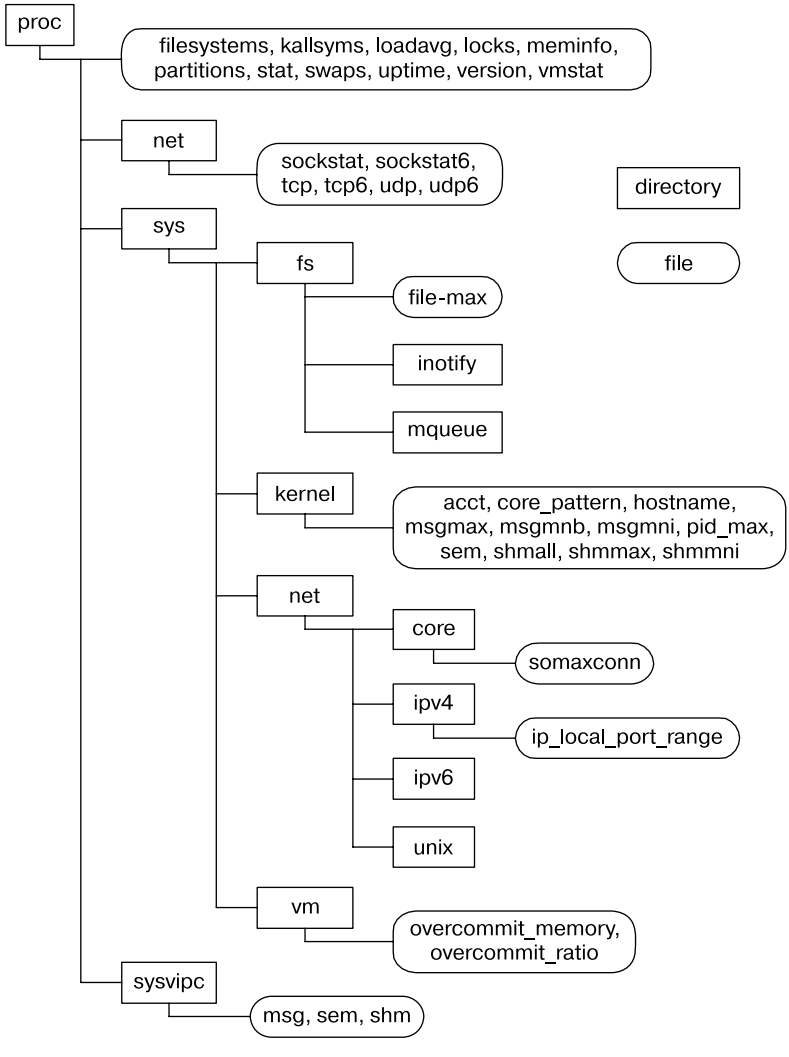


Рис. 12.1. Отдельные файлы и подкаталоги, находящиеся в /proc

Файлы, упомянутые на рис. 12.1, рассматриваются в разных местах данной книги. Основное назначение подкаталогов, перечисленных на рис. 12.1, сведено в табл. 12.2.

Таблица 12.2. Назначение отдельных подкаталогов, находящихся в /proc

Каталог	Информация, предоставляемая файлами в этом каталоге
/proc	Различная системная информация
/proc/net	Информация состояния сети и сокетов
/proc/sys/fs	Настройки, относящиеся к файловым системам
/proc/sys/kernel	Различные общие настройки ядра
/proc/sys/net	Настройки сети и сокетов
/proc/sys/vm	Настройки, касающиеся управления памятью
/proc/sysvipc	Информация об IPC-объектах System V

12.1.3 Доступ к файлам, находящимся в /proc

Доступ к файлам, находящимся в `/proc`, зачастую осуществляется с использованием сценариев оболочки (большинство `/proc`-файлов, хранящих множество значений, могут быть легко проанализированы с помощью таких языков написания сценариев, как Python или Perl). Например, содержимое `/proc`-файла можно изменить и просмотреть, используя следующие команды оболочки:

```
# echo 100000 > /proc/sys/kernel/pid_max
# cat /proc/sys/kernel/pid_max
100000
```

Доступ к `/proc`-файлам также может быть получен из программы с использованием обычных системных вызовов файлового ввода-вывода. При доступе к этим файлам применяются кое-какие ограничения.

- Некоторые `/proc`-файлы предназначены только для чтения, то есть они существуют лишь для отображения информации о ядре и не могут использоваться для ее изменения. Это справедливо для большинства файлов в каталогах `/proc/`*`PID`*.
- Некоторые `/proc`-файлы могут быть прочитаны только их владельцем (или привилегированным процессом). Например, все файлы, находящиеся в каталоге `/proc/`*`PID`*, являются собственностью пользователя, владеющего соответствующим процессом, и в отношении некоторых из них (например, `/proc/`*`PID`*`/environ`) права на чтение даются только владельцу файла.
- Кроме файлов в подкаталогах `/proc/`*`PID`*, большинство файлов в каталоге `/proc` являются собственностью суперпользователя (root), и те файлы, в которые разрешено вносить изменения, могут быть изменены только этим пользователем.

Обращение к файлам, находящимся в /proc/PID

Каталоги `/proc/`*`PID`* не существуют постоянно. Каждый из них появляется с созданием процесса с соответствующим идентификатором и исчезает, как только процесс завершится. То есть, определив факт существования конкретного каталога `/proc/`*`PID`*, нужно быть готовым обработать возможность того, что к моменту попытки открытия файла процесс уже мог завершиться и соответствующий каталог `/proc/`*`PID`* мог быть удален.

Пример программы

В листинге 12.1 показан способ чтения и изменения /proc-файла. Приведенная в нем программа считывает и отображает содержимое файла /proc/sys/kernel/pid_max. Если указан аргумент командной строки, программа обновляет файл, используя его значение. Этот файл (впервые появившийся в версии 2.6) указывает верхний предел для идентификаторов процесса (см. раздел 6.2). Пример работы программы выглядит следующим образом:

```
$ su                    Для обновления файла pid_max требуются соответствующие права
Password:
# ./procfs_pidmax 10000
Old value: 32768
/proc/sys/kernel/pid_max теперь содержит 10000
```

Листинг 12.1. Обращение к файлу /proc/sys/kernel/pid_max

sysinfo/procfs_pidmax.c

```
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_LINE 100

int
main(int argc, char *argv[])
{
    int fd;
    char line[MAX_LINE];
    ssize_t n;

    fd = open("/proc/sys/kernel/pid_max", (argc > 1) ? O_RDWR : O_RDONLY);
    if (fd == -1)
        errExit("open");

    n = read(fd, line, MAX_LINE);
    if (n == -1)
        errExit("read");

    if (argc > 1)
        printf("Old value: ");
    printf("%.*s", (int) n, line);

    if (argc > 1) {
        if (write(fd, argv[1], strlen(argv[1])) != strlen(argv[1]))
            fatal("write() failed");
        system("echo /proc/sys/kernel/pid_max now contains "
               "'cat /proc/sys/kernel/pid_max'");
    }

    exit(EXIT_SUCCESS);
}
```

sysinfo/procfs_pidmax.c

12.2. Идентификация системы: uname()

Системный вызов `uname()` возвращает идентифицирующую информацию о базовой системе, в которой выполняется приложение в структуре, указанной аргументом `utsbuf`.

```
#include <sys/utsname.h>

int uname(struct utsname *utsbuf);
```

Возвращает 0 при успешном завершении
или –1 при ошибке

Аргумент utsbuf является указателем на utsname-структуру, имеющую следующее определение:

```
#define _UTSNAME_LENGTH 65

struct utsname {
    char sysname[_UTSNAME_LENGTH];     /* Название реализации */
    char nodename[_UTSNAME_LENGTH];    /* Имя узла в сети */
    char release[_UTSNAME_LENGTH];     /* Идентификатор выпуска ОС */
    char version[_UTSNAME_LENGTH];     /* Версия ОС */
    char machine[_UTSNAME_LENGTH];     /* Оборудование, на котором
                                          запущена система */
#ifdef _GNU_SOURCE                     /* Далее следуют данные,
                                          характерные для Linux */
    char domainname[_UTSNAME_LENGTH];  /* Доменное имя хоста NIS */
#endif
};
```

В SUSv3 системный вызов uname() указан, но длина различных полей в структуре utsname не определена. Требуется только, чтобы строки завершались нулевым байтом. В Linux длина каждого из этих полей определена равна 65 байт, включая место для завершающего нулевого байта. В одних реализациях UNIX эти поля бывают короче, а в других (например, в Solaris) их длина доходит до 257 байт.

Поля sysname, release, version и machine структуры utsname автоматически заполняются ядром.

В Linux доступ к такой же информации, которая возвращается в полях sysname, release и version структуры utsname, дается в трех файлах каталога /proc/sys/kernel. Это файлы, предназначенные только для чтения, которые называются соответственно ostype, osrelease и version. Еще один файл, /proc/version, включает ту же информацию, что и эти три файла, а также сведения о компиляции ядра (то есть имя пользователя, выполнившего компиляцию, имя хоста, на котором она была выполнена, и версию gcc).

В поле nodename возвращается значение, установленное с использованием системного вызова sethostname() (подробности можно найти на странице руководства, посвященной этому системному вызову). Зачастую это имя похоже на префикс имени хоста из доменного имени системы в DNS.

В поле domainname возвращается значение, установленное с помощью системного вызова setdomainname() (подробности можно найти на соответствующей странице руководства). Это доменное имя хоста в сетевой информационной службе — Network Information Services (NIS) (не следует путать с доменным именем хоста в DNS).

Системный вызов gethostname(), являющийся противоположностью системного вызова sethostname(), извлекает имя хоста системы. Это имя можно также просмотреть и установить с помощью команды hostname(1) и характерного для Linux файла /proc/sys/kernel/hostname.

Системный вызов getdomainname(), будучи противоположностью системного вызова setdomainname(), извлекает доменное имя NIS. Это имя можно также просмотреть и установить

с помощью команды domainname(1) и характерного для ОС Linux файла /proc/sys/kernel/domainname.

Системные вызовы sethostname() и setdomainname() довольно редко применяются в прикладных программах. Обычно имя хоста и доменное имя NIS устанавливаются в ходе загрузки системы сценариями ее запуска.

Программа из листинга 12.2 выводит информацию, возвращаемую системным вызовом `uname()`. Пример вывода, который можно увидеть при запуске этой программы, имеет следующий вид:

```
$ ./t_uname
Node name:    tekapo
System name:  Linux
Release:      2.6.30-default
Version:      #3 SMP Fri Jul 17 10:25:00 CEST 2009
Machine:      i686
Domain name:
```

Листинг 12.2. Использование системного вызова uname()

sysinfo/t_uname.c

```
#define _GNU_SOURCE
#include <sys/utsname.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct utsname uts;

    if (uname(&uts) == -1)
        errExit("uname");

    printf("Node name: %s\n", uts.nodename);
    printf("System name: %s\n", uts.sysname);
    printf("Release: %s\n", uts.release);
    printf("Version: %s\n", uts.version);
    printf("Machine: %s\n", uts.machine);
#ifdef _GNU_SOURCE
    printf("Domain name: %s\n", uts.domainname);
#endif
    exit(EXIT_SUCCESS);
}
```

sysinfo/t_uname.c

12.3. Резюме

Файловая система `/proc` предоставляет прикладным программам ряд сведений о ядре. В каждом подкаталоге `/proc/`*`PID`* содержатся файлы и подкаталоги, предоставляющие информацию о процессе, чей идентификатор совпадает с *`PID`*. В других различных файлах и каталогах, находящихся в `/proc`, приводится общесистемная информация, которую программа может считать и в некоторых случаях изменить.

Системный вызов `uname()` позволяет нам уточнить реализацию UNIX и тип машины, на которой запущено приложение.

Дополнительная информация

Дополнительные сведения о файловой системе `/proc` можно найти на странице руководства `proc(5)`, в исходном файле ядра `Documentation/filesystems/proc.txt` и в различных файлах из каталога `Documentation/sysctl`.

12.4. Упражнения

12.1. Напишите программу, выводящую список идентификаторов процессов и имен команд для всех процессов, запущенных пользователем, который указан в аргументе командной строки программы. (Для этого вам может пригодиться функция `userIdFromName()` из листинга 8.1.) Эту задачу можно выполнить, исследовав строки `Name:` и `Uid:` всех имеющихся в системе файлов `/proc/`*`PID`*`/status`. Сквозной просмотр всех имеющихся в системе каталогов `/proc/`*`PID`* требует задействования функции `readdir(3)`, рассматриваемой в разделе 18.8. Обеспечьте возможность правильной обработки программой случаев исчезновения каталогов `/proc/`*`PID`* в период между обнаружением их существования и попыткой открытия соответствующего файла `/proc/`*`PID`*`/status`.

12.2. Напишите программу, выводящую на экран дерево, демонстрирующее иерархию родительско-дочерних отношений всех имеющихся в системе процессов, восходящую к `init`. Для каждого процесса программа должна вывести идентификатор процесса и выполняемую команду. Вывод программы должен быть похож на вывод команды `pstree(1)`, хотя совсем не обязательно, чтобы он был таким же сложным. Родитель каждого имеющегося в системе процесса может быть определен путем изучения строки `PPid:` всех имеющихся в системе файлов `/proc/`*`PID`*`/status`. Внимательно отнеситесь к обработке возможности исчезновения родителя процесса (и соответственно его каталога `/proc/`*`PID`*) в ходе сканирования всех каталогов `/proc/`*`PID`*.

12.3. Напишите программу, выводящую список всех процессов, у которых имеется открытый файл с указанным путевым именем. Эту задачу можно выполнить, изучив содержимое всех символьных ссылок `/proc/`*`PID`*`/fd/*`. Для этого могут потребоваться вложенные циклы, использующие функцию `readdir(3)` для сканирования всех каталогов `/proc/`*`PID`*, а затем содержимого всех записей `/proc/`*`PID`*`/fd` внутри каждого каталога `/proc/`*`PID`*. Для чтения содержимого символьной ссылки `/proc/`*`PID`*`/fd/`*`n` нужно задействовать функцию `readlink()`, рассмотренную в разделе 18.5.

13 Буферизация файлового ввода-вывода

Для достижения высокой скорости и эффективности работы системные вызовы ввода-вывода (то есть ядро) и функции ввода-вывода стандартной библиотеки языка C (то есть функции stdio) при работе с дисковыми файлами осуществляют буферизацию данных. В этой главе мы рассмотрим оба типа буферизации, а также то, как они влияют на производительность приложения. Здесь также описаны различные приемы настройки и отключения обоих типов буферизации и техника, называемая непосредственным вводом-выводом, применяемая при определенных обстоятельствах, чтобы избежать буферизации при работе в режиме ядра.

13.1. Буферизация файлового ввода-вывода при работе в режиме ядра: буферная кэш-память

При работе с файлами на диске системные вызовы `read()` и `write()` не инициируют непосредственный доступ к диску. Вместо этого они просто копируют данные между буфером в пространстве памяти пользователя и *буфером в буферном кэше* ядра. Например, следующий вызов переносит 3 байта данных из буфера в пространстве памяти пользователя в буфер в пространстве ядра:

```
write(fd, "abc", 3);
```

Сразу после этого происходит возвращение из системного вызова `write()`. Несколько позже ядро записывает (сбрасывает) свой буфер на диск. (В связи с этим говорится, что системный вызов не *синхронизирован* с дисковой операцией.) Если в данном промежутке времени какой-нибудь другой процесс предпримет попытку чтения этих байтов файла, ядро автоматически предоставит данные из буферной кэш-памяти, а не из файла (с уже устаревшим содержимым).

Аналогично для ввода ядро считывает данные с диска и сохраняет их в буфере ядра. Вызовы `read()` извлекают данные из этого буфера, пока он не будет исчерпан, после чего ядро считывает следующий сегмент файла в буферную кэш-память. (Это несколько упрощенное представление происходящего. В режиме последовательного доступа к файлу ядро обычно выполняет упреждающее чтение, пытаясь обеспечить считывание в буферную кэш-память следующих блоков файла еще до того, как они будут востребованы считывающим процессом. Более подробно упреждающее чтение рассматривается в разделе 13.5.)

Замысел заключается в попытке ускорить работу `read()` и `write()`, чтобы им не приходилось находиться в режиме ожидания завершения относительно медленных дисковых операций. Кроме того, такая конструкция повышает эффективность работы за счет сокращения количества переносов данных с диска, которые ядро должно выполнить.

Ядро Linux не накладывает никаких фиксированных ограничений на размер буферной кэш-памяти. Оно выделит столько страниц буферной кэш-памяти, сколько

понадобится, ограничившись при этом лишь объемом доступной физической памяти и потребностями в использовании физической памяти для других целей (например, для хранения текстовых страниц и страниц данных, требуемых выполняемым процессам). Если испытывается дефицит доступной памяти, ядро сбрасывает часть измененных страниц буферной кэш-памяти на диск с целью высвобождения этих страниц для их повторного использования.

Следует уточнить, что после выхода версии ядра 2.4 в Linux больше не создается отдельная буферная кэш-память. Вместо этого буферы файлового ввода-вывода включаются в страничную кэш-память, которая, к примеру, также содержит страницы из отображенных в памяти файлов. Тем не менее в изложении основного материала будет использоваться понятие буферной кэш-памяти, поскольку для реализаций UNIX оно более привычно.

Влияние размера буфера на производительность системных вызовов ввода-вывода

Независимо от того, выполняется 1000 записей одного байта или единая запись 1000 байт, ядро осуществляет одинаковое количество обращений к диску. Но последний вариант более предпочтителен, поскольку требует одного системного вызова, тогда как для первого варианта их требуется целая тысяча. Хотя системные вызовы выполняются намного быстрее дисковых операций, на них все же уходит довольно много времени, поскольку ядро должно системно перехватить вызов, проверить допустимость его аргументов и переместить данные между пространством пользователя и пространством ядра (подробности рассматриваются в разделе 3.1).

То, как размер буфера влияет на выполнение файлового ввода-вывода, можно проследить, запустив программу, показанную в листинге 4.1, с применением различных значений `BUF_SIZE`. (В константе `BUF_SIZE` указывается количество байтов, переносимых каждым вызовом `read()` и `write()`.) Время, требуемое программе для копирования файла размером 100 миллионов байт в Linux в файловой системе ext2 с использованием различных значений `BUF_SIZE`, перечислено в табл. 13.1. В дополнение к приведенной в этой таблице информации нужно заметить следующее.

- Столбцы *затрачиваемого времени* и *общего времени задействования центрального процессора* в пояснениях не нуждаются. Столбцы *времени задействования центрального процессора пользователем* и *системой* показывают, как общее время разбивается соответственно на время, затраченное на выполнение кода в пользовательском режиме, и время на выполнение кода ядра (то есть системных вызовов).
- Тест, по которому была сформирована табл. 13.1, выполнялся с использованием «ванильного» ядра версии 2.6.30 в файловой системе ext2 с размером блока 4096 байт.

  Когда говорится о том, что ядро «ванильное», это означает, что оно не подвергалось исправлениям. Оно отличается от ядер, предоставляемых большинством поставщиков, которые нередко включают различные исправления для устранения недостатков или добавления возможностей.

- В каждой строке показано усредненное значение для заданного размера буфера после 20 запусков. В этих тестах, а также в других, показанных далее в этой главе, перед каждым выполнением программы файловая система была размонтирована и снова смонтирована, чтобы гарантировать чистую буферную кэш-память, используемую для файловой системы. Замеры времени были выполнены с помощью команды оболочки `time`.

Таблица 13.1. Время, необходимое для дублирования файла длиной 100 миллионов байт

Размер BUF_SIZE	Время (в секундах)			
	Затрачиваемое	Задействования центрального процессора		
		Общее	Пользователем	Системой
1	107,43	107,32	8,20	99,12
2	54,16	53,89	4,13	49,76
4	31,72	30,96	2,30	28,66
8	15,59	14,34	1,08	13,26
16	7,50	7,14	0,51	6,63
32	3,76	3,68	0,26	3,41
64	2,19	2,04	0,13	1,91
128	2,16	1,59	0,11	1,48
256	2,06	1,75	0,10	1,65
512	2,06	1,03	0,05	0,98
1024	2,05	0,65	0,02	0,63
4096	2,05	0,38	0,01	0,38
16 384	2,05	0,34	0,00	0,33
65 536	2,06	0,32	0,00	0,32

Поскольку для различных размеров буферной памяти общий объем переносимых данных один и тот же (а стало быть, и одинаковое количество дисковых операций), информация в табл. 13.1 показывает наличие издержек на совершение вызовов `read()` и `write()`. При размере буферной памяти, равном 1 байту, для `read()` и `write()` совершается 100 миллионов вызовов. При размере буферной памяти, равном 4096 байт, количество обращений к каждому системному вызову снижается примерно до 24 000 и достигается производительность, близкая к оптимальной. После этого значения производительность существенно не улучшается, поскольку затраты на совершение системных вызовов `read()` и `write()` становятся несущественными по сравнению с временем, требуемым для копирования данных между пространством пользователя и пространством ядра и для выполнения фактического дискового ввода-вывода.

> Последние строки табл. 13.1 позволяют приблизительно оценить время, необходимое для переноса данных между пользовательским пространством памяти и пространством ядра, а также для осуществления файлового ввода-вывода. Поскольку количество системных вызовов в этих случаях относительно невелико, можно пренебречь их составляющей в затрачиваемом времени и времени задействования ЦП. Таким образом, можно сказать, что время задействования ЦП со стороны системы фактически является замером времени переноса данных между пользовательским пространством и пространством ядра. Значение затрачиваемого времени дает нам приблизительную оценку времени, необходимого для переноса данных на диск и с диска. (Как вскоре станет понятно, это в основном время, требуемое для считывания данных с диска.)

Таким образом, если переносится большой объем данных в файл или из файла, то буферизация данных в больших блоках и, в силу этого, выполнение меньшего количества системных вызовов позволяют нам существенно повысить производительность ввода-вывода.

Данные в табл. 13.1 относятся к целому ряду факторов: к времени выполнения системных вызовов `read()` и `write()`, времени переноса данных между буферами в пространстве памяти ядра и в пространстве пользовательской памяти, времени переноса данных между буферами ядра и диском. Давайте дополнительно рассмотрим послед-

ний фактор. Вполне очевидно, что перенос содержимого файла с вводимыми данными в буферную кэш-память неизбежен. Но мы уже видели, что возвращение из `write()` происходит сразу же после переноса данных из пользовательского пространства в буферную кэш-память ядра. Поскольку размер оперативной памяти в системе, используемой для тестирования (4 Гбайт), существенно превышает размер копируемого файла (100 Мбайт), можно предположить, что ко времени завершения работы программы файл с выводимыми данными фактически не будет записан на диск. Поэтому в качестве дальнейшего эксперимента мы запускаем программу, которая просто записывает произвольные данные в файл, используя различные размеры буферов `write()`. Результаты приведены в табл. 13.2.

Данные из табл. 13.2 также получены при использовании ядра версии 2.6.30 в файловой системе ext2 с размером блока 4096 байт. В каждой строке показаны усредненные значения после 20 запусков. Тестовая программа (`filebuff/write_bytes.c`) не приводится, но она доступна в исходном коде для этой книги.

Таблица 13.2. Время, требуемое для записи файла длиной 100 миллионов байт

Размер BUF_SIZE	Время (в секундах)			
	Затрачиваемое	Задействования центрального процессора		
		Общее	Пользователем	Системой
1	72,13	72,11	5,00	67,11
2	36,19	36,17	2,47	33,70
4	20,01	19,99	1,26	18,73
8	9,35	9,32	0,62	8,70
16	4,70	4,68	0,31	4,37
32	2,39	2,39	0,16	2,23
64	1,24	1,24	0,07	1,16
128	0,67	0,67	0,04	0,63
256	0,38	0,38	0,02	0,36
512	0,24	0,24	0,01	0,23
1024	0,17	0,17	0,01	0,16
4096	0,11	0,11	0,00	0,11
16 384	0,10	0,10	0,00	0,10
65 536	0,09	0,09	0,00	0,09

В табл. 13.2 показывается расход времени на совершение системных вызовов `write()` и на перенос данных из пространства пользователя в буферную кэш-память ядра с использованием различных размеров буферов для `write()`. Для бóльших размеров буфера заметна существенная разница с данными, показанными в табл. 13.1. Например, при размере буфера 65 536 байт затрачиваемое время в табл. 13.1 составляет 2,06 секунды, а в табл. 13.2 это же время равно 0,09 секунды. Дело в том, что в последнем случае дисковый ввод-вывод фактически не выполняется. Иными словами, основная часть времени в строках, соответствующих большим размерам буфера в табл. 13.1, затрачивается на считывание данных с диска.

Как будет показано в разделе 13.3, когда операции вывода намеренно блокируются до тех пор, пока данные не будут перенесены на диск, время, затрачиваемое на вызовы `write()`, существенно возрастает.

И наконец, стоит заметить, что представленные в табл. 13.2 данные (и последующая информация в табл. 13.3) являются всего лишь результатом одного (достаточно примитивного) теста производительности файловой системы. Кроме того, результаты в разных

файловых системах будут, по всей видимости, варьироваться. Оценочные тесты файловых систем можно проводить и по другим критериям, например по производительности при интенсивной нагрузке, инициированной множеством пользователей, по скорости создания и удаления файлов, по времени, требуемому для поиска файла в большом по размеру каталоге, по пространству, требуемому для хранения небольших файлов, или по обеспечению целостности файлов в случае отказа системы. Там, где решающее значение имеет производительность ввода-вывода или других операций, связанных с файловой системой, нет ничего лучше, чем тест целевой платформы, воспроизводящий поведение вашего приложения.

13.2. Буферизация в библиотеке stdio

Для того чтобы сократить количество системных вызовов, при работе с файлами на диске буферизация данных в большие блоки осуществляется внутри функций ввода-вывода библиотеки языка C (например, на `fprintf()`, `fscanf()`, `fgets()`, `fputs()`, `fputc()`, `fgetc()`). Таким образом, библиотека stdio берет на себя работу по буферизации данных для их вывода с помощью `write()` или ввода посредством `read()`.

Задание режима буферизации stdio-потока

Функция `setvbuf()` позволяет выбрать способ буферизации, которую будет применять библиотека stdio.

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Возвращает `0` при успешном завершении
или ненулевое значение при ошибке

Аргумент `stream` идентифицирует файловый поток, буферизация которого должна быть изменена. После того как поток открыт, функция `setvbuf()` может быть запущена до вызова в отношении этого потока любой другой функции stdio. Вызов `setvbuf()` влияет на поведение всех последующих stdio-операций, выполняемых над указанным потоком.

> Потоки, используемые библиотекой stdio, не нужно путать с фреймворком STREAMS для System V, который не реализован в «ванильном» ядре Linux.

Аргументы `buf` и `size` определяют буфер, используемый для потока, идентифицируемого аргументом `stream`. Эти аргументы могут быть указаны двумя способами.

- Если `buf` имеет ненулевое значение, то он указывает на блок памяти с размером в байтах, заданным в аргументе `size`. Этот блок будет использоваться в качестве буфера для `stream`. Поскольку буфер, на который указывает `buf`, затем используется библиотекой stdio, он должен быть либо статически, либо динамически выделен на куче (с помощью `malloc()` или подобной ей функции). Он не может быть выделен на стеке локальной переменной функции, поскольку это вызовет полный хаос, когда произойдет возврат из функции и будет освобожден соответствующий ей кадр стека.
- Если `buf` имеет значение `NULL`, библиотека stdio автоматически выделяет буфер для использования с потоком `stream` (если только не будет выбран рассматриваемый

далее ввод-вывод без применения буфера). В SUSv3 допускается, но не требуется использование реализацией аргумента `size` для определения размера этого буфера. В glibc-реализации в данном случае аргумент `size` игнорируется.

Аргумент `mode` указывает на тип буферизации и имеет одно из следующих значений.

- `_IONBF` — не выполнять буферизацию ввода-вывода. Каждый вызов библиотеки stdio приводит к немедленному системному вызову `write()` или `read()`. Аргументы `buf` и `size` игнорируются и могут быть указаны как `NULL` и `0` соответственно. Это настройка по умолчанию для stderr, что гарантирует немедленное появление сообщения об ошибке.
- `_IOLBF` — использовать построчную буферизацию ввода-вывода. Этот флаг задан по умолчанию для потоков, имеющих отношение к терминальным устройствам. Для выходных потоков данные буферизуются до тех пор, пока в выводе не появится символ новой строки (или пока не заполнится буфер). Для выходных потоков выполняется построчное считывание данных.
- `_IOFBF` — применять полностью буферизованный ввод-вывод. Данные считываются или записываются (с помощью вызовов `read()` или `write()`) блоками, равными размеру буфера. Для потоков, имеющих отношение к дисковым файлам, этот режим задан по умолчанию.

Использование `setvbuf()` продемонстрировано в следующем коде:

```
#define BUF_SIZE 1024
static char buf[BUF_SIZE];

if (setvbuf(stdout, buf, _IOFBF, BUF_SIZE) != 0)
    errExit("setvbuf");
```

Обратите внимание, что `setvbuf()` в случае ошибки возвращает ненулевое значение (не обязательно `-1`).

Функция `setbuf()` является надстройкой над `setvbuf()` и выполняет точно такую же задачу.

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

За исключением того, что не возвращается результат функции, вызов `setbuf(fp, buf)` является эквивалентом такого вызова:

```
setvbuf(fp, buf, (buf != NULL) ? _IOFBF: _IONBF, BUFSIZ);
```

Для аргумента `buf` определяется либо значение `NULL` для отказа от буферизации, либо указатель на буфер из `BUFSIZ` байтов, выделяемый вызывающим кодом. (Константа `BUFSIZ` определена в `<stdio.h>`. В реализации glibc она имеет весьма распространенное значение `8192`.)

Функция `setbuffer()` аналогична функции `setbuf()`, но позволяет вызывающему коду указать размер буфера `buf`.

```
#define _BSD_SOURCE
#include <stdio.h>

void setbuffer(FILE *stream, char *buf, size_t size);
```

Вызов функции `setbuffer(fp, buf, size)` является эквивалентом следующего вызова:

```
setvbuf(fp, buf, (buf != NULL) ? _IOFBF : _IONBF, size);
```

Функция `setbuffer()` не определена в SUSv3, но доступна в большинстве реализаций UNIX.

Сброс буфера stdio

Независимо от текущего режима буферизации, в любое время можно принудительно записать данные, находящиеся в выходном потоке stdio (то есть сбросить буфер ядра на диск посредством `write()`), воспользовавшись библиотечной функцией `fflush()`. Она сбрасывает буфер вывода для указанного потока.

```
#include <stdio.h>

int fflush(FILE *stream);
```

Возвращает при успешном завершении `0` или `EOF` при ошибке

Если для `stream` указано значение `NULL`, то `fflush()` сбрасывает на диск содержимое всех буферов stdio, которые связаны с потоками вывода.

Функция `fflush()` может также применяться к входному потоку. При этом отбрасывается весь буферизованный ввод. (Буфер будет заполнен заново при следующей попытке программы выполнить чтение из потока.)

Когда соответствующий поток закрывается, буфер stdio автоматически сбрасывается.

Во многих реализациях библиотек языка C, включая glibc, если stdin и stdout ссылаются на терминал, при каждом считывании ввода из stdin происходит скрытое выполнение `fflush(stdout)`. Это выражается в сбросе всех приглашений к вводу, которые записаны в stdout и не включают в себя завершающий символ новой строки (например, `printf("Date: ")`). Но такое поведение не указано в SUSv3 или C99 и реализовано не во всех библиотеках языка C. Для обеспечения отображения таких приглашений к вводу портируемые программы должны использовать явно указанные вызовы `fflush(stdout)`.

> В стандарте C99 изложены два требования для той ситуации, когда поток открыт как для ввода, так и для вывода. Во-первых, за операциями вывода не могут непосредственно следовать операции ввода без выполняемого между ними вызова fflush() или одной из функций позиционирования файлового указателя (fseek(), fsetpos() или rewind()). Во-вторых, за операцией ввода не может непосредственно следовать операция вывода без выполняемого между ними вызова одной из функций позиционирования файлового указателя, если только операция ввода не столкнулась с окончанием файла.

13.3. Управление буферизацией файлового ввода-вывода, осуществляемой в ядре

Сброс буферной памяти ядра для файлов вывода можно сделать принудительным. Иногда это необходимо, если приложение, прежде чем продолжить работу (например, процесс, журналирующий изменения базы данных), должно гарантировать фактическую запись вывода на диск (или как минимум в аппаратный кэш диска).

Перед тем как рассматривать системные вызовы, используемые для управления буферизацией в ядре, будет нелишним рассмотреть несколько относящихся к этому вопросу определений из SUSv3.

Синхронизированный ввод-вывод с обеспечением целостности данных и файла

В SUSv3 понятие *синхронизированного завершения ввода-вывода* означает «операцию ввода-вывода, которая либо привела к успешному переносу данных [на диск], либо была диагностирована как неудачная».

В SUSv3 определяются два различных типа завершений синхронизированного ввода-вывода. Различие между типами касается *метаданных* («данных о данных»), описывающих файл. Ядро хранит их вместе с данными самого файла. Подробности метаданных файла будут рассмотрены в разделе 14.4 при изучении индексных дескрипторов файлов. Пока же будет достаточно отметить, что файловые метаданные включают такую информацию, как сведения о владельце файла и его группе, полномочия доступа к файлу, размер файла, количество жестких ссылок на файл, метки времени, показывающие время последнего обращения к файлу, время его последнего изменения и время последнего изменения метаданных, а также указатели на блоки данных.

Первым типом завершения синхронизированного ввода-вывода в SUSv3 является *завершение с целостностью данных*. При обновлении данных файла должен быть обеспечен перенос информации, достаточной для того, чтобы позволить в дальнейшем извлечь эти данные для продолжения работы.

- Для операции чтения это означает, что запрошенные данные файла были перенесены (с диска) в процесс. Если есть отложенные операции записи, которые могут повлиять на запрошенные данные, данные будут перенесены на диск до выполнения чтения.
- Для операции записи это означает, что данные, указанные в запросе на запись, были перенесены (на диск), как и все метаданные файла, требуемые для извлечения этих данных. Ключевой момент, на который нужно обратить внимание: чтобы обеспечить извлечение данных из измененного файла, необязательно переносить все медатанные файла. В качестве примера атрибута метаданных измененного файла, который нуждается в переносе, можно привести его размер (если операция записи приводит к увеличению размера файла). В противоположность этому метки времени изменяемого файла не будут нуждаться в переносе на диск до того, как произойдет последующее извлечение данных.

Вторым типом завершения синхронизированного ввода-вывода, определенного в SUSv3, является *завершение с целостностью файла*. Это расширенный вариант завершения синхронизованного ввода-вывода с целостностью данных. Отличие этого режима заключается в том, что в ходе обновления файла все его метаданные переносятся на диск, даже если этого не требуется для последующего извлечения данных файла.

Системные вызовы для управления буферизацией, проводимой в ядре при файловом вводе-выводе

Системный вызов `fsync()` приводит к сбросу всех буферизованных данных и всех метаданных, которые связаны с открытым файлом, имеющим дескриптор `fd`. Вызов `fsync()` приводит файл в состояние целостности (файла) после завершения синхронного ввода-вывода.

Вызов `fsync()` возвращает управление только после завершения переноса данных на дисковое устройство (или по крайней мере в его кэш-память).

```
#include <unistd.h>

int fsync(int fd);
```

Возвращает при успешном завершении 0 или –1 при ошибке

Системный вызов fdatasync() работает точно так же, как и fsync(), но приводит файл в состояние целостности (данных) после после завершения синхронного ввода-вывода.

```
#include <unistd.h>

int fdatasync(int fd);
```

Возвращает при успешном завершении 0 или –1 при ошибке

Использование fdatasync() потенциально сокращает количество дисковых операций с двух, необходимых системному вызову fsync(), до одного. Например, если данные файла изменились, но размер остался прежним, вызов fdatasync() вызывает лишь принудительное обновление данных. (Выше уже отмечалось, что для завершения синхронной операции ввода-вывода с целостностью данных нет необходимости переносить изменение таких аттрибутов, как время последней модификации файла.) В отличие от этого вызов fsync() приведет также к принудительному переносу на диск метаданных.

Такое сокращение количества дисковых операций ввода-вывода будет полезным для отдельных приложений, для которых решающую роль играет производительность и неважно аккуратное обновление конкретных метаданных (например, отметок времени). Это может привести к существенным улучшениям производительности приложений, производящих несколько обновлений файла за раз. Поскольку данные и метаданные файла обычно располагаются в разных частях диска, обновление и тех и других потребует повторяющихся операций поиска вперед и назад по диску.

В Linux 2.2 и более ранних версиях fdatasync() реализован в виде вызова fsync(), поэтому не дает никакого прироста производительности.

Начиная с ядра версии 2.6.17, в Linux предоставляется нестандартный системный вызов sync_file_range(). Он позволяет более точно управлять процессом сброса данных файла на диск, чем fdatasync(). При вызове можно указать сбрасываемую область файла и задать флаги, устанавливающие условия блокировки данного вызова. Дополнительные подробности вы найдете на странице руководства sync_file_range(2).

Системный вызов sync() приводит к тому, что все буферы ядра, содержащие обновленную файловую информацию (то есть блоки данных, блоки указателей, метаданные и т. д.), сбрасываются на диск.

```
#include <unistd.h>

void sync(void);
```

В реализации Linux функция sync() возвращает управление только после того, как все данные будут перенесены на дисковое устройство (или как минимум в его кэш-память). Но в SUSv3 разрешается, чтобы sync() просто вносила в план перенос данных для операции ввода-вывода и возвращала управление до завершения этого переноса.

Постоянно выполняемый поток ядра обеспечивает сброс измененных буферов ядра на диск, если они не были явным образом синхронизированы в течение 30 секунд. Это делается для того, чтобы не допустить рассинхронизации данных буферов с соответствующим дисковым файлом на длительные периоды времени (и не подвергнуть их риску утраты при отказе системы). В Linux 2.6 эта задача выполняется потоком ядра pdflush. (В Linux 2.4 она выполнялась потоком ядра kupdated.)

Срок (в сотых долях секунды), через который измененный буфер должен быть сброшен на диск кодом потока pdflush, определяется в файле /proc/sys/vm/dirty_expire_centisecs. Дополнительные файлы в том же самом каталоге управляют другими особенностями операции, выполняемой потоком pdflush.

Включение режима синхронизации для всех записей: O_SYNC

Указание флага `O_SYNC` при вызове `open()` приводит к тому, что все последующие операции вывода выполняются в синхронном режиме:

```
fd = open(pathname, O_WRONLY | O_SYNC);
```

После этого вызова `open()` каждая проводимая с файлом операция `write()` автоматически сбрасывает данные и метаданные файла на диск (то есть записи выполняются как синхронизированные операции записи с целостностью файла).

В старых версиях системы BSD для обеспечения функциональных возможностей, включаемых флагом O_SYNC, использовался флаг O_FSYNC. В glibc флаг O_FSYNC определен как синоним O_SYNC.

Влияние флага O_SYNC на производительность

Использование флага `O_SYNC` (или же частые вызовы `fsync()`, `fdatasync()` или `sync()`) может сильно повлиять на производительность. В табл. 13.3 показано время, требуемое для записи 1 миллиона байт в только что созданный файл (в файловой системе ext2) при различных размерах буфера с выставленным и со сброшенным флагом `O_SYNC`. Результаты были получены (с помощью программы `filebuff/write_bytes.c`, предоставляемой в исходном коде для книги) с использованием «ванильного» ядра версии 2.6.30 и файловой системы ext2 с размером блока 4096 байт. В каждой строке приводится усредненное значение, полученное после 20 запусков для заданного размера буфера.

Таблица 13.3. Влияние флага O_SYNC на скорость записи 1 миллиона байт

BUF_SIZE	**Требуемое время (в секундах)**			
	Без использования O_SYNC		**С использованием O_SYNC**	
	Затрачиваемое	**Общее ЦП**	**Затрачиваемое**	**Общее ЦП**
1	0,73	0,73	1030	98,8
16	0,05	0,05	65,0	0,40
256	0,02	0,02	4,07	0,03
4096	0,01	0,01	0,34	0,03

Как видно, указание флага `O_SYNC` приводит к чудовищному увеличению затрачиваемого времени при использовании буфера размером 1 байт более чем в 1000 раз. Обратите также внимание на большую разницу, возникающую при выполнении записей с флагом `O_SYNC`, между затраченным временем и временем задействования ЦП. Она является последствием блокирования выполнения программы при фактическом сбросе содержимого каждого буфера на диск.

В результатах, показанных в табл. 13.3, не учтен еще один фактор, влияющий на производительность при использовании O_SYNC. Современные дисковые накопители обладают внутренней кэш-памятью большого объема, и по умолчанию установка флага O_SYNC просто приводит к переносу данных в эту кэш-память. Если отключить кэширование на диске (воспользовавшись командой hdparm -W0), влияние O_SYNC на производительность станет еще более существенным. При размере буфера 1 байт затраченное время возрастет с 1030 секунд до приблизительно 16 000 секунд. При размере буфера 4096 байт затраченное время возрастет с 0,34 секунды до 4 секунд. В итоге, если нужно выполнить принудительный сброс на диск буферов ядра, следует рассмотреть, можно ли спроектировать приложение с использованием бóльших по объему буферов для write() или же подумать об использовании вместо флага O_SYNC периодических вызовов fsync() или fdatasync().

Флаги O_DSYNC и O_RSYNC

В SUSv3 определены два дополнительных флага состояния открытого файла, имеющих отношение к синхронизированному вводу-выводу: O_DSYNC и O_RSYNC.

Флаг O_DSYNC приводит к выполнению в последующем синхронизированных операций записи с целостностью данных завершаемого ввода-вывода (подобно использованию fdatasync()). Эффект от его работы отличается от эффекта, вызываемого флагом O_SYNC, использование которого приводит к выполнению в последующем синхронизированных операций записи с целостностью файла (подобно fsync()).

Флаг O_RSYNC указывается совместно с O_SYNC либо с O_DSYNC и приводит к расширению поведения, связанного с этими флагами при выполнении операций чтения. Указание при открытии файла флагов O_RSYNC и O_DSYNC приводит к выполнению в последующем синхронизированных операций чтения с целостностью данных (то есть прежде чем будет выполнено чтение, из-за наличия O_DSYNC завершаются все ожидающие файловые записи). Указание при открытии файла флагов O_RSYNC и O_SYNC приводит к выполнению в последующем синхронизированных операций чтения с целостностью файла (то есть прежде, чем будет выполнено чтение, из-за наличия O_SYNC завершаются все ожидающие файловые записи).

До выхода версии ядра 2.6.33 флаги O_DSYNC и O_RSYNC в Linux не были реализованы и в заголовочных файлах glibc эти константы определялись как выставление флага O_SYNC. (В случае с O_RSYNC это было неверно, поскольку O_SYNC не влияет на какие-либо функциональные особенности операций чтения.)

Начиная с ядра версии 2.6.33, в Linux реализуется флаг O_DSYNC, а реализация флага O_RSYNC, скорее всего, будет добавлена в будущие выпуски ядра.

> До выхода ядра 2.6.33 в Linux отсутствовала полная реализация семантики O_SYNC. Вместо этого флаг O_SYNC был реализован как O_DSYNC. В приложениях, скомпонованных со старыми версиями GNU библиотеки C для старых ядер, в версиях Linux 2.6.33 и выше флаг O_SYNC по прежнему ведет себя как O_DSYNC. Это сделано для сохранения привычного поведения таких программ. (Для сохранения обратной бинарной совместимости в ядре 2.6.33 флагу O_DSYNC было присвоено старое значение флага O_SYNC, а новое значение O_SYNC включает в себя флаг O_DSYNC (на одной из машин это 04010000 и 010000 соответственно). Это позволяет приложениям, скомпилированным с новыми заголовочными файлами, получать в ядрах, вышедших до версии 2.6.33, по меньшей мере семантику O_DSYNC.)

13.4. Обзор буферизации ввода-вывода

На рис. 13.1 приведена схема буферизации, используемой (для файлов вывода) библиотекой stdio и ядром, а также показаны механизмы для управления каждым типом буферизации. Если пройтись по схеме вниз до ее середины, станет виден перенос поль-

зовательских данных функциями библиотеки stdio в буфер stdio, который работает в пользовательском пространстве памяти. Когда этот буфер заполнен, библиотека stdio прибегает к системному вызову `write()`, переносящему данные в буферную кэш-память ядра (находящуюся в памяти ядра). В результате ядро инициирует дисковую операцию для переноса данных на диск.

В левой части схемы на рис. 13.1 показаны вызовы, которые могут использоваться в любое время для явного принудительного сброса любого из буферов. В правой части показаны вызовы, которые могут применяться для автоматического выполнения сброса либо за счет выключения буферизации в библиотеке stdio, либо включением для системных вызовов файлового вывода синхронного режима выполнения, чтобы при каждом вызове `write()` происходил немедленный сброс на диск.

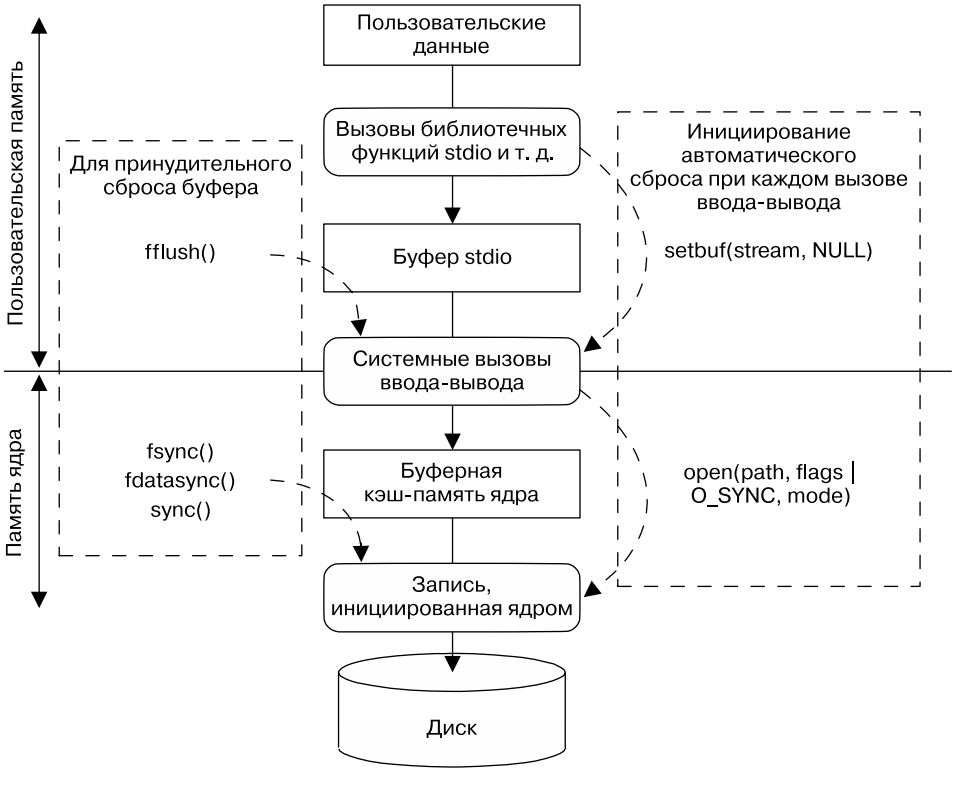


Рис. 13.1. Обзор буферизации ввода-вывода

13.5. Уведомление ядра о схемах ввода-вывода

Системный вызов `posix_fadvise()` позволяет процессу информировать ядро о предпочитаемой им схеме обращения к данным файла.

Ядро может (но не обязано) использовать информацию, предоставляемую системным вызовом `posix_fadvise()` для оптимизации задействования им буферной кэш-памяти, повышая тем самым производительность ввода-вывода для процесса и для системы в целом. На семантику программы вызов `posix_fadvise()` не влияет.

```
#define _XOPEN_SOURCE 600
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Возвращает при успешном завершении 0 или положительный номер ошибки при ее возникновении

Аргумент fd является дескриптором файла, идентифицирующим тот файл, о схеме обращения к которому нужно проинформировать ядро. Аргументы offset и len идентифицируют область файла, к которой относится уведомление: offset указывает на начальное смещение области, а len — на ее размер в байтах. Присвоение для len значения 0 говорит о том, что имеются в виду все байты, начиная с offset и заканчивая концом файла. (В версиях ядра до 2.6.6 значение 0 для len интерпретировалось буквально, как 0 байт.)

Аргумент advice показывает предполагаемый характер обращения процесса к файлу. Он определяется с одним из следующих значений.

- POSIX_FADV_NORMAL — у процесса нет особого уведомления, касающегося схем обращения. Это поведение по умолчанию, если для файла не дается никаких уведомлений. В Linux эта операция устанавливает для окна упреждающего считывания данных из файла его исходный размер (128 Кбайт).
- POSIX_FADV_SEQUENTIAL — процесс предполагает последовательное считывание данных от меньших смещений к бóльшим. В Linux эта операция устанавливает для окна упреждающего считывания данных из файла его удвоенное исходное значение.
- POSIX_FADV_RANDOM — процесс предполагает обращение к данным в произвольном порядке. В Linux этот вариант отключает упреждающее считывание данных из файла.
- POSIX_FADV_WILLNEED — процесс предполагает обращение к указанной области файла в ближайшее время. Ядро выполняет упреждающее считывание данных для заполнения буферной кэш-памяти данными файла в диапазоне, заданном аргументами offset и len. Последующие вызовы read() в отношении файла не блокируют дисковый ввод-вывод, а просто извлекают данные из буферной кэш-памяти. Ядро не дает никаких гарантий насчет продолжительности нахождения извлекаемых из файла данных в буферной кэш-памяти. Если при работе другого процесса или ядра возникнет особая потребность в памяти, то страница в конечном итоге будет повторно использована. Иными словами, если память остро востребована, нам нужно гарантировать небольшой разрыв по времени между вызовом posix_fadvise() и последующим вызовом (или вызовами) read(). (Функциональные возможности, эквивалентные операции POSIX_FADV_WILLNEED, предоставляет характерный для Linux системный вызов readahead().)
- POSIX_FADV_DONTNEED — процесс не предполагает в ближайшем будущем обращений к указанной области файла. Тем самым ядро уведомляется, что оно может высвободить соответствующие страницы кэш-памяти (если таковые имеются). В Linux эта операция выполняется в два этапа. Сначала, если очередь записи на базовом устройстве не переполнена серией запросов, ядро сбрасывает любые измененные страницы кэш-памяти в указанной области. Затем ядро предпринимает попытку высвободить все страницы кэш-памяти из указанной области. Для измененных страниц в данной области второй этап завершится успешно, только если они были записаны на базовое устройство в ходе первого этапа, то есть очередь записи на устройстве не переполнена. Так как приложение не может проверить состояние

очереди на устройстве, гарантировать освобождение страниц кэша можно, вызвав `fsync()` или `fdatasync()` в отношении дескриптора `fd` перед применением `POSIX_FADV_DONTNEED`.

- `POSIX_FADV_NOREUSE` — процесс предполагает однократное обращение к данным в указанной области файла, без ее повторного использования. Тем самым ядро уведомляется о том, что оно может высвободить страницы после однократного обращения к ним. В Linux эта операция в настоящее время остается без внимания.

Спецификация `posix_fadvise()` появилась только в SUSv3, и этот интерфейс поддерживается не всеми реализациями UNIX. В Linux вызов `posix_fadvise()` предоставляется, начиная с версии ядра 2.6.

13.6. Обход буферной кэш-памяти: непосредственный ввод-вывод

Начиная с версии ядра 2.4, Linux позволяет приложению обходить буферную кэш-память при выполнении дискового ввода-вывода, перемещая данные непосредственно из пользовательского пространства памяти в файл или на дисковое устройство. Иногда этот режим называют *непосредственным* или *необрабатываемым вводом-выводом*.

> Приведенная здесь информация относится исключительно к Linux и не стандартизирована в SUSv3. Тем не менее некоторые варианты непосредственного доступа к вводу-выводу в отношении устройств или файлов предоставляются большинством реализаций UNIX.

Иногда непосредственный ввод-вывод неверно понимается в качестве средства достижения высокой производительности ввода-вывода. Но для большинства приложений использование непосредственного ввода-вывода может существенно снизить производительность. Дело в том, что ядро выполняет несколько оптимизаций для повышения производительности ввода-вывода за счет использования буферной кэш-памяти, включая последовательное упреждающее чтение данных, выполнение ввода-вывода в кластерах, состоящих из дисковых блоков, и позволение процессам, обращающимся к одному и тому же файлу, совместно задействовать буферы в кэш-памяти. Все эти виды оптимизации при использовании непосредственного ввода-вывода утрачиваются. Он предназначен только для приложений со специализированными требованиями к вводу-выводу, например для систем управления базами данных, выполняющих свое собственное кэширование и оптимизацию ввода-вывода, и которым не нужно, чтобы ядро тратило время центрального процессора и память на выполнение таких же задач.

Непосредственный ввод-вывод можно выполнять либо в отношении отдельно взятого файла, либо в отношении блочного устройства (например, диска). Для этого при открытии файла или устройства с помощью вызова `open()` указывается флаг `O_DIRECT`.

Флаг `O_DIRECT` работает, начиная с версии ядра 2.4.10. Использование этого флага поддерживается не всеми файловыми системами и версиями ядра Linux. Большинство базовых файловых систем поддерживают флаг `O_DIRECT`, но многие файловые системы, не относящиеся к UNIX (например, VFAT), — нет. Можно проверить поддержку этой возможности, протестировав выбранную файловую систему (если файловая система не поддерживает `O_DIRECT`, вызов `open()` даст сбой с выдачей ошибки `EINVAL`) или исследовав на этот предмет исходный код ядра.

Если один процесс открыл файл с флагом O_DIRECT, а другой — обычным образом (то есть с использованием буферной кэш-памяти), то согласованность между содержимым буферной кэш-памяти и данными, считанными или записанными через непосредственный ввод/вывод, отсутствует. Подобного развития событий следует избегать.

Сведения об устаревшем (ныне нерекомендуемом) методе получения необрабатываемого (raw) доступа к дисковому устройству можно найти на странице руководства raw(8).

Ограничения по выравниванию для непосредственного ввода-вывода

Поскольку непосредственный ввод-вывод (как на дисковых устройствах, так и в отношении файлов) предполагает непосредственное обращение к диску, при выполнении ввода-вывода следует соблюдать некоторые ограничения.

- Переносимый буфер данных должен быть выровнен по границе памяти, кратной размеру блока.
- Смещение в файле или в устройстве, с которого начинаются переносимые данные, должно быть кратно размеру блока.
- Длина переносимых данных должна быть кратной размеру блока.

Несоблюдение любого из этих ограничений влечет за собой возникновение ошибки EINVAL. В показанном выше перечне под размером блока подразумевается размер физического блока устройства (обычно это 512 байт).

При выполнении непосредственного ввода-вывода в Linux 2.4 накладывается больше ограничений, чем в Linux 2.6: выравнивание, длина и смещение должны быть кратны размеру логического блока используемой файловой системы. (Обычно размеры логических блоков в файловой системе равны 1024, 2048 или 4096 байт.)

Пример программы

В листинге 13.1 предоставляется простой пример использования O_DIRECT при открытии файла для чтения. Эта программа воспринимает до четырех аргументов командной строки, указывающих (в порядке следования) файл, из которого будут считываться данные, количество считываемых из файла байтов, смещение, к которому программа должна перейти, прежде чем начать считывание данных из файла, и выравнивание буфера данных, передаваемое read(). Последние два аргумента опциональны и по умолчанию настроены соответственно на значения нулевого смещения и 4096 байт.

Рассмотрим примеры того, что будет показано при запуске программы:

```
$ ./direct_read /test/x 512            Считывание 512 байт со смещения 0
Read 512 bytes                         Успешно
$ ./direct_read /test/x 256
ERROR [EINVAL Invalid argument] read   Длина не кратна 512
$ ./direct_read /test/x 512 1
ERROR [EINVAL Invalid argument] read   Смещение не кратно 512
$ ./direct_read /test/x 4096 8192 512
Read 4096 bytes                        Успешно
$ ./direct_read /test/x 4096 512 256
ERROR [EINVAL Invalid argument] read   Выравнивание не кратно 512
```

Программа в листинге 13.1 выделяет блок памяти, который выровнен по адресу, кратному ее первому аргументу, и для этого использует функцию memalign(). Функция memalign() рассматривалась в подразделе 7.1.4.

Листинг 13.1. Использование O_DIRECT для обхода буферной кэш-памяти

filebuff/direct_read.c

```
#define _GNU_SOURCE /* Получение определения O_DIRECT из <fcntl.h> */

#include <fcntl.h>
#include <malloc.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    ssize_t numRead;
    size_t length, alignment;
    off_t offset;
    void *buf;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file length [offset [alignment]]\n", argv[0]);
        length = getLong(argv[2], GN_ANY_BASE, "length");
        offset = (argc > 3) ? getLong(argv[3], GN_ANY_BASE, "offset") : 0;
        alignment = (argc > 4) ? getLong(argv[4], GN_ANY_BASE,
                    "alignment") : 4096;

        fd = open(argv[1], O_RDONLY | O_DIRECT);
        if (fd == -1)
            errExit("open");

        /* Функция memalign() выделяет блок памяти, выровненный по адресу,
           кратному ее первому аргументу. Следующее выражение обеспечивает
           выравнивание 'buf' по границе, кратной 'alignment',
           но не являющейся степенью двойки. Это делается для того, чтобы в случае,
           к примеру, запроса буфера с выравниванием, кратным 256 байтам,
           не происходило случайного получения буфера, выровненного также
           и по 512-байтовой границе. Приведение к типу '(char *)' необходимо
           для проведения с указателем арифметических операций (что невозможно
           сделать с типом 'void *', который возвращает memalign(). */

        buf = (char *) memalign(alignment * 2, length + alignment)
                + alignment;
        if (buf == NULL)
            errExit("memalign");

        if (lseek(fd, offset, SEEK_SET) == -1)
            errExit("lseek");

        numRead = read(fd, buf, length);
        if (numRead == -1)
            errExit("read");

        printf("Read %ld bytes\n", (long) numRead);

        exit(EXIT_SUCCESS);
}
```

filebuff/direct_read.c

13.7. Смешивание библиотечных функций и системных вызовов для файлового ввода-вывода

Для выполнения ввода-вывода в отношении одного и того же файла можно совмещать использование системных вызовов и стандартных функций библиотеки языка C. Помочь нам в выполнении этой задачи могут функции `fileno()` и `fdopen()`.

```
#include <stdio.h>

int fileno(FILE *stream);
```

Возвращает при успешном завершении дескриптор файла, или `-1` при ошибке

```
FILE *fdopen(int fd, const char *mode);
```

Возвращает при успешном завершении (новый) указатель файла или `NULL` при ошибке

Для данного потока `stream` функция `fileno()` возвращает соответствующий файловый дескриптор (то есть тот самый, который библиотека stdio открыла для этого потока). Этот файловый дескриптор затем может использоваться привычным образом с такими системными вызовами ввода-вывода, как `read()`, `write()`, `dup()` и `fcntl()`.

Функция `fdopen()` является обратной функции `fileno()`. Для заданного дескриптора файла она создает соответствующий поток, использующий этот дескриптор для своего ввода-вывода. Аргумент `mode` имеет то же предназначение, что и в функции `fopen()`, например `r` для чтения, `w` для записи или `a` для добавления. Если этот аргумент не соответствует режиму доступа файлового дескриптора `fd`, функция `fdopen()` дает сбой.

Функция `fdopen()` особенно пригодится для дескрипторов, ссылающихся на необычные файлы. В последующих главах вы увидите, что системные вызовы для создания сокетов и конвейеров всегда возвращают файловые дескрипторы. Чтобы применять библиотеку stdio с файлами этих типов, для создания соответствующего файлового потока следует воспользоваться функцией `fdopen()`.

При задействовании функций библиотеки stdio в сочетании с системными вызовами ввода-вывода для выполнения этих операций в отношении дисковых файлов нужно учитывать вопросы буферизации. Системные вызовы ввода-вывода переносят данные непосредственно в буферную кэш-память ядра, а библиотека stdio, прежде чем вызвать `write()`, ждет, пока буфер потока в пользовательском пространстве заполнится, и только затем переносит его в буферную кэш-память ядра. Рассмотрим следующий код, используемый для записи в стандартный вывод:

```
printf("To man the world is twofold, ");
write(STDOUT_FILENO, "in accordance with his twofold attitude.\n", 41);
```

Обычно вывод `printf()` появляется, как правило, после вывода `write()`, следовательно, этот код выдает такой вывод:

```
in accordance with his twofold attitude.
To man the world is twofold,
```

Чтобы избежать этой проблемы, возникающей при смешивании системных вызовов и функций stdio для ввода-вывода, может потребоваться грамотное использование функ-

ции fflush(). Буферизацию также можно отключить с помощью функций setvbuf() или setbuf(), но это может повлиять на производительность ввода-вывода в приложении, поскольку в дальнейшем каждая операция вывода приведет к выполнению системного вызова write().

В SUSv3 приводится (длинный) список требований к приложениям, в которых допустимо смешивать системные вызовы и функции stdio для ввода-вывода. Подробности можно найти в разделе Interaction of File Descriptors and Standard I/O Streams в главе General Information тома System Interfaces (XSH).

13.8. Резюме

Буферизация входных и выходных данных выполняется ядром, а также библиотекой stdio. В некоторых случаях может понадобиться предотвратить буферизацию, но при этом нужно учитывать влияние, оказываемое на производительность приложения. Для управления буферизацией, выполняемой в ядре и осуществляемой библиотечными функциями, и для однократных сбросов буферов можно использовать разнообразные системные вызовы и библиотечные функции.

Для уведомления ядра о предпочитаемой схеме обращения к данным из указанного файла процесс может воспользоваться функцией posix_fadvise(). Ядро может применить эту информацию для оптимизации применения буферной кэш-памяти, повысив таким образом производительность ввода-вывода.

Характерный для Linux флаг O_DIRECT, используемый при системном вызове open(), позволяет специализированным приложениям обходить буферную кэш-память.

Функции fileno() и fdopen() помогают решить задачу смешивания системных вызовов и стандартных библиотечных функций языка C, чтобы выполнять ввод-вывод в отношении одного и того же файла. Для заданного потока функция fileno() возвращает соответствующий дескриптор файла, а функция fdopen() выполняет обратную операцию, создавая новый поток, который использует указанный открытый дескриптор файла.

Дополнительная информация

Описание реализации и преимуществ использования буферной кэш-памяти в System V приводится в издании [Bach, 1986]. В книгах [Goodheart & Cox, 1994] и [Vahalia, 1996] также дается описание целесообразности применения и реализации буферной кэш-памяти в System V. Дополнительную информацию, характерную для Linux, можно найти в изданиях [Bovet & Cesati, 2005] и [Love, 2010].

13.9. Упражнения

13.1. Используя встроенную команду оболочки time, попробуйте замерить время работы программы из листинга 4.1 (copy.c) в своей системе:
   1) проведите эксперименты с использованием различных размеров файлов и буферов памяти. Размер буфера памяти можно задать при компилировании программы с помощью ключа -DBUF_SIZE=nbytes;
   2) добавьте флаг O_SYNC в системный вызов open(). Определите, насколько это повлияет на скорость при различных размерах буферной памяти;
   3) попробуйте выполнить тесты по замеру времени в нескольких файловых системах (например, ext3, XFS, Btrfs и JFS). Будут ли результаты похожи друг на друга?

Будет ли совпадать динамика при переходе от небольших к большим размерам буферов памяти?

13.2. Замерьте время работы программы `filebuff/write_bytes.c` (предоставляемой в исходном коде, распространяемом для этой книги) для различных размеров буферов памяти и файловых систем.

13.3. Каким будет эффект использования следующих инструкций?

```
fflush(fp);
fsync(fileno(fp));
```

13.4. Объясните, почему вывод при выполнении следующего кода изменяется в зависимости от того, куда перенаправляется стандартный вывод — на терминал или в дисковый файл.

```
printf("If I had more time, \n");
write(STDOUT_FILENO, "I would have written you a shorter letter.\n", 43);
```

13.5. Команда `tail [ -n num ] file` выводит последние `num` строк (по умолчанию десять) указанного файла. Реализуйте эту команду, используя системные вызовы ввода-вывода (`lseek()`, `read()`, `write()` и т. д.). Чтобы реализация работала эффективно, не забудьте про рассмотренные в этой главе вопросы буферизации.

14 Файловые системы

В главах 4, 5 и 13 мы рассмотрели файловый ввод-вывод, уделив особое внимание обычным (то есть дисковым) файлам. В этой и последующих главах мы более подробно разберем некоторые темы, связанные с файлами.

- В текущей главе речь идет о файловых системах.
- В главе 15 описаны различные атрибуты файла, включая метки времени, принадлежность и права доступа.
- В главах 16 и 17 обсуждаются две новые особенности системы Linux 2.6: расширенные атрибуты и списки контроля доступа (ACL). Расширенные атрибуты — это способ привязки произвольных метаданных к файлу. Списки контроля доступа — это расширенный вариант традиционной UNIX-модели прав доступа к файлу.
- В главе 18 рассмотрены каталоги и ссылки.

Основная часть главы посвящена файловым системам, которые представляют собой упорядоченные наборы файлов и каталогов. Мы рассмотрим некоторые понятия, относящиеся к файловым системам, используя в отдельных случаях в качестве конкретного примера традиционную для Linux файловую систему ext2. Кроме того, вкратце будут описаны некоторые журналируемые файловые системы, доступные в Linux.

В завершение главы мы рассмотрим системные вызовы, которые используются для монтирования и размонтирования файловой системы, а также библиотечные функции, применяемые для получения информации о смонтированных файловых системах.

14.1. Специальные файлы устройств

В текущей главе часто упоминаются дисковые устройства, поэтому мы начнем с краткого рассмотрения понятия *«файл устройства»*.

Специальный файл устройства соответствует какому-либо устройству в системе. Внутри ядра каждому типу устройства соответствует драйвер устройства, который обрабатывает для него все запросы на ввод-вывод. *Драйвер устройства* — это модуль программного кода ядра, реализующий набор операций, которые (как правило) соответствуют операциям ввода-вывода на связанном аппаратном средстве. API, предоставляемый драйверами устройств, является фиксированным и содержит операции, соответствующие системным вызовам `open()`, `close()`, `read()`, `write()`, `mmap()` и `ioctl()`. Тот факт, что каждый драйвер устройства обеспечивает единый интерфейс, скрывающий различия в работе отдельных устройств, позволяет добиться *универсальности ввода-вывода* (см. раздел 4.2).

Некоторые устройства являются *реальными*, например мыши, диски и USB-накопители, другие — *виртуальными*. Это означает, что им не соответствует никакое аппаратное средство, а вместо него ядро предоставляет (с помощью драйвера устройства) абстрактное устройство с API таким же, как у реального устройства.

Устройства можно подразделить на два типа.

- *Символьные* — обрабатывают данные посимвольно. Примеры символьных устройств: терминалы и клавиатуры.

- *Блочные* — обрабатывают за один заход один блок данных. Размер блока зависит от типа устройства, но обычно является кратным 512 байтам. Примеры блочных устройств: диски и USB-накопители.

Файлы устройств располагаются внутри файловой системы, подобно другим файлам, обычно в каталоге `/dev`. Суперпользователь может создать файл устройства с помощью команды `mknod`. Эту же задачу можно выполнить в привилегированной (`CAP_MKNOD`) программе, используя системный вызов `mknod()`.

> Мы не рассматриваем подробно системный вызов mknod() («создать индексный дескриптор файловой системы»), поскольку его применение очевидно и единственное его назначение в настоящее время состоит в создании файлов устройств, что не является необходимым для типичного приложения. Можно также использовать вызов mknod() для организации очередей FIFO (см. раздел 44.7), однако предпочтительнее использовать функцию mkfifo(). Исторически в некоторых реализациях UNIX вызов mknod() применялся также для создания каталогов, но теперь вместо него используется системный вызов mkdir(). Тем не менее в некоторых реализациях UNIX (но не в Linux) такая возможность вызова mknod() сохранена для обратной совместимости. Дальнейшие подробности см. на странице mknod(2) руководства к ОС.

В ранних версиях Linux каталог `/dev` содержал записи для всех возможных устройств в системе, даже если такие устройства фактически не были подключены к нему. Это означало, что каталог `/dev` мог содержать буквально тысячи неиспользуемых записей, замедляющих работу команд, которым было необходимо просматривать его содержимое. При этом было невозможно использовать содержимое для того, чтобы выяснить, какие устройства действительно есть в системе. В Linux 2.6 эта проблема решена за счет программы-менеджера udev, которая опирается на файловую систему sysfs, экспортирующую информацию об устройствах и других объектах ядра в пространство пользователя через фиктивную файловую систему, смонтированную в каталоге `/sys`.

> В статье [Kroah-Hartman, 2003] приведен обзор менеджера udev и указаны причины, по которым его следует считать лучше файловой системы devfs, призванной решать те же проблемы в Linux 2.4. Информацию о файловой системе sysfs можно найти в файле Documentation/filesystems/sysfs.txt Linux 2.6, а также в работе [Mochel, 2005].

Идентификаторы устройств

Каждый файл устройства имеет *старший идентификационный номер* и *младший идентификационный номер*. Старший номер идентифицирует общий класс устройства и используется ядром для поиска драйвера, который подходит для данного типа устройства. Младший номер уникальным образом идентифицирует устройство внутри общего класса. Старший и младший номера устройства можно вывести с помощью команды `ls -l`.

Старший и младший номера устройства записаны в индексном дескрипторе для данного файла устройства. (Индексные дескрипторы рассмотрены в разделе 14.4.) Каждый драйвер устройства регистрирует свою привязку к определенному старшему идентификационному номеру, и она обеспечивает соединение между специальным файлом устройства и его драйвером. Когда ядро отыскивает драйвер устройства, имя файла устройства не имеет значения.

В версии Linux 2.4 и более ранних общее количество устройств в системе ограничено тем обстоятельством, что старший и младший номера описаны восьмью битами. А тот факт, что старшие номера устройств фиксированы и выделяются централизованно (организацией Linux Assigned Names and Numbers Authority, см. www.lanana.org), еще сильнее усугубляет это ограничение. В версии Linux 2.6 это ограничение менее строгое за счет использования большего количества битов для хранения старшего и младшего идентификаторов устройств (12 и 20 бит соответственно).

14.2. Диски и разделы

Обычные файлы и каталоги располагаются, как правило, на жестких дисках. (Файлы и каталоги могут также храниться и на других устройствах, например на компакт-дисках, картах с флеш-памятью и на виртуальных дисках, но нас интересуют главным образом жесткие диски.) В следующих разделах вы увидите, каким образом диски организованы и разбиты на разделы.

Дисководы

Дисковод — это механическое устройство, состоящее из одной или нескольких пластин, которые вращаются с высокой скоростью (до нескольких тысяч оборотов в минуту). Информация, которая закодирована магнитным способом на поверхности диска, извлекается или изменяется с помощью головок чтения/записи, перемещающихся вдоль радиуса диска. Физически информация на поверхности диска размещена в виде набора концентрических кругов, называемых *дорожками*. Дорожки, в свою очередь, разделены на *секторы*, каждый из которых состоит из последовательности *физических* блоков. Размер физического блока обычно равен 512 байтам (или кратному значению) и представляет собой наименьший блок информации, который привод способен прочитать или записать.

И хотя современные диски работают быстро, на чтение и запись информации все так же требуется существенное время. Сначала головка диска должна переместиться к соответствующей дорожке (время поиска), а затем привод должен дождаться, пока необходимый сектор окажется под головкой (задержка из-за вращения) и требуемые блоки будут переданы (время передачи). Общее время, которое необходимо для выполнения подобной операции, обычно составляет несколько миллисекунд. Для сравнения: современные ЦПУ способны выполнить за это время миллионы инструкций.

Разделы диска

Каждый диск имеет один или несколько (неперекрывающихся) *разделов*. Каждый раздел воспринимается ядром как отдельное устройство, расположенное в каталоге `/dev`.

> Системный администратор задает количество, тип и размеры разделов на диске с помощью команды fdisk. Команда fdisk –l выводит список всех разделов диска. В характерном для Linux файле /proc/partitions перечислены старшие и младшие номера устройств, размеры и названия всех дисковых разделов системы.

Дисковый раздел может содержать информацию любого типа, но обычно содержит что-либо из перечисленного ниже:

- *файловую систему*, которая упорядочивает файлы и каталоги, как описано в разделе 14.3;
- *область данных*, которая доступна в качестве устройства с прямой пересылкой данных, как описано в разделе 13.6 (эту технологию используют некоторые системы управления базами данных);
- *область подкачки*, которая применяется ядром для управления памятью.

Область подкачки создается с помощью команды `mkswap(8)`. Привилегированный (`CAP_SYS_ADMIN`) процесс может использовать системный вызов `swapon()` для уведомления ядра о том, что дисковый раздел следует задействовать в качестве области подкачки. Системный вызов `swapoff()` выполняет функцию преобразования, говоря ядру о том, чтобы оно прекратило использование дискового раздела в качестве области подкачки. Эти системные вызовы не регламентированы в стандарте SUSv3, но все же присутствуют во многих реализациях UNIX. Дополнительную информацию см. на страницах руководства `swapon(2)` и `swapon(8)`.

Особый файл Linux /proc/swaps можно применять для отображения информации об областях подкачки, задействованных в данный момент в системе. В числе этой информации указан размер каждой области подкачки, а также использованной доли этой области.

14.3. Файловые системы

Файловая система — это упорядоченный набор обычных файлов и каталогов. Файловая система создается с помощью команды `mkfs`.

Одной из сильных сторон Linux является возможность поддержки самых разных файловых систем, в число которых входят следующие:

- традиционная файловая система ext2;
- различные файловые UNIX-системы, например Minix, System V и BSD;
- файловые системы, разработанные корпорацией Microsoft: FAT, FAT32 и NTFS;
- файловая система ISO 9660 для компакт-дисков;
- файловая система HFS компьютеров Apple Macintosh;
- ряд сетевых файловых систем, включая широко используемую систему NFS компании Sun, систему SMB, разработанную компаниями IBM и Microsoft, систему NCP компании Novell, а также файловую систему Coda, созданную в университете Carnegie Mellon;
- некоторые журналируемые файловые системы, в число которых входят ext3, ext4, Reiserfs, JFS, XFS и Btrfs.

Типы файловых систем, которые в данный момент распознаны ядром, можно просмотреть в особом файле Linux `/proc/filesystems`.

В версии Linux 2.6.14 появилось средство Filesystem in Userspace (FUSE, «файловая система в пространстве пользователя»). Этот механизм добавляет в ядро перехватчики (hooks), которые позволяют полностью реализовать файловую систему с помощью программы из пространства пользователя, и при этом нет необходимости в исправлении или перекомпиляции ядра. Дополнительные подробности см. на сайте fuse.sourceforge.net.

Файловая система ext2

Долгие годы наиболее используемой файловой системой в Linux была ext2 — вторая расширенная файловая система, наследница ext — исходной файловой системы Linux. С недавнего времени вместо ext2 все чаще используются различные файловые системы с журналированием. Иногда бывает удобно описывать понятия, относящиеся к типичной файловой системе, с помощью терминов для какой-либо конкретной реализации системы. С этой целью далее в главе мы используем в различных примерах систему ext2.

Файловая система ext2 была создана Реми Кардом (Rémy Card). Ее исходный код небольшой (около 5000 строк на языке C) и представляет собой модель для различных реализаций других файловых систем. Главная веб-страница сайта, посвященного системе ext2, находится по адресу e2fsprogs.sourceforge.net/ext2.html. На этом сайте есть хорошая обзорная статья, описывающая реализацию файловой системы ext2. В онлайн-книге Дэвида Раслинга (David Rusling) The Linux Kernel («Ядро Linux»), доступной на сайте www.tldp.org, также описана файловая система ext2.

Структура файловой системы

Основной единицей для выделения пространства в файловой системе является *логический* блок. Он представляет собой множество смежных физических блоков на дисковом

устройстве, на котором располагается данная файловая система. Так, например, размер логического блока в файловой системе ext2 равен 1024, 2048 или 4096 байтам. (Размер логического блока указывается в качестве аргумента команды `mkfs(8)`, которая используется для создания файловой системы.)

> Привилегированная (CAP_SYS_RAWIO) программа может использовать операцию FIBMAP ioctl(), чтобы определить физическое расположение указанного блока для какого-либо файла. Третий аргумент вызова является целым числом, которое определяется в ходе вызова. До осуществления вызова следует передать в этот аргумент номер логического блока (номер первого логического блока равен 0); после вызова ему присваивается номер начального физического блока, в котором хранится указанный логический блок.

На рис. 14.1 показана связь между разделами диска и файловыми системами, а также отмечены части (типичной) файловой системы.

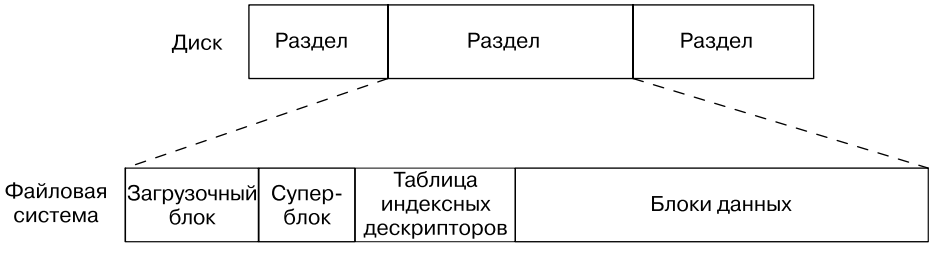


Рис. 14.1. Структура разделов диска и файловой системы

Файловая система состоит из следующих частей.

- *Блок начальной загрузки*. Всегда является первым блоком файловой системы. Блок начальной загрузки не используется файловой системой; он содержит информацию, которая применяется для загрузки операционной системы. И хотя для загрузки операционной системы необходим лишь один блок начальной загрузки, такой блок есть в каждой файловой системе (большинство этих блоков остается неиспользованным).
- *Суперблок*. Это единичный блок, который следует сразу за блоком начальной загрузки. Он содержит такую информацию о параметрах файловой системы, как:
  - размер таблицы индексных дескрипторов;
  - размер логических блоков в данной файловой системе;
  - размер файловой системы в логических блоках.

  Различные файловые системы, которые расположены на одном физическом устройстве, могут обладать разными типами и размерами, а также иметь различающиеся параметры (например, размер блока). Это одна из причин разбиения диска на несколько разделов.
- *Таблица индексных дескрипторов*. Каждый файл или каталог в данной файловой системе обладает уникальной записью в таблице индексных дескрипторов. Эти записи содержат различную информацию о файле. Индексные дескрипторы рассмотрены подробнее в следующем разделе. Таблицу индексных дескрипторов иногда называют также *индексным списком*.
- *Блоки данных*. Основная часть пространства файловой системы используется для блоков данных, которые образуют файлы и каталоги, расположенные в данной файловой системе.

В случае с файловой системой ext2 картина немного сложнее, чем описанная выше. После блока начальной загрузки файловая система разбита на группы блоков одинакового размера. Каждая группа блоков содержит копию суперблока, информацию о параметрах группы блоков, а также таблицу индексных дескрипторов и блоки данных для этой группы блоков. За счет хранения всех блоков какого-либо файла внутри одной группы блоков файловая система ext2 стремится сократить время поиска при последовательном доступе к файлу. Дополнительную информацию см. в файле исходного программного кода Linux Documentation/filesystems/ext2.txt, в исходном коде программы dumpe2fs, которая является частью пакета e2fsprogs, а также в работе [Bovet & Cesati, 2005].

14.4. Индексные дескрипторы

Таблица индексных дескрипторов файловой системы содержит по одному *индексному дескриптору* на каждый файл, расположенный в данной файловой системе. Индексные дескрипторы идентифицируются с помощью номеров в порядке их следования в таблице индексных дескрипторов. *Номер индексного дескриптора* (или *индексный номер*) файла — это первое поле, которое выводит команда `ls -li`. Информация, которую хранит индексный дескриптор, включает в себя следующее.

- Тип файла (то есть обычный файл, каталог, символическая ссылка, символьное устройство).
- Владелец (называется также идентификатором пользователя или UID) данного файла.
- Группа (называется также идентификатором группы или GID) для данного файла.
- Права доступа для трех категорий пользователей: *владельца* (иногда называемого *пользователем*), *группы* и *остальных* (всего остального мира). Подробности см. в разделе 15.4.
- Три метки времени: время последнего доступа к файлу (отображается с помощью команды `ls -lu`), время последнего изменения файла (это время по умолчанию отображает команда `ls -l`), а также время последнего изменения статуса (последнего изменения информации индексного дескриптора, отображается с помощью команды `ls -lc`). Следует отметить, что, подобно другим реализациям UNIX, в большинстве файловых систем Linux не записывается время создания файла.
- Количество жестких ссылок на файл.
- Размер файла в байтах.
- Количество блоков, фактически отведенных для данного файла; за единицу измерения принят блок размером 512 байт. Соответствие между этим числом и размером файла в байтах может быть непростым, поскольку файл способен содержать дыры (см. раздел 4.7), и поэтому для него потребуется меньше выделенных блоков, чем можно было бы ожидать, исходя из его номинального размера в байтах.
- Указатели на блоки данных для этого файла.

Индексные дескрипторы и указатели на блоки данных в файловой системе ext2

Подобно большинству файловых систем UNIX, файловая система ext2 не хранит блоки данных какого-либо файла рядом друг с другом или в порядке их следования (но все же пытается размещать их близко друг к другу). Для локализации блоков данных файла ядро хранит набор указателей в индексном дескрипторе. Система, которая используется для этого в файловой системе ext2, показана на рис. 14.2.

За счет избавления от необходимости смежного хранения блоков удается добиться более эффективного использования пространства в файловой системе. При этом, в частности,

снижается степень фрагментации свободного дискового пространства — потерь, которые вызваны наличием многочисленных несмежных фрагментов свободного пространства, которые слишком малы для того, чтобы их использовать. Если выразиться иначе, то можно сказать, что за преимущество эффективного использования свободного дискового пространства приходится расплачиваться фрагментацией файлов на занятом пространстве диска.

В файловой системе ext2 каждый индексный дескриптор содержит 15 указателей. Первые 12 из них (на рис. 14.2 они пронумерованы от 0 до 11) указывают на положение первых 12 блоков файла в файловой системе. Следующий указатель — это *указатель на блок указателей*, который сообщает расположение 13-го и последующих блоков данных файла. Количество указателей в этом блоке зависит от размера блока в данной файловой системе. Для каждого указателя необходимо 4 байта, и поэтому всего может быть от 256 (для блока размером 1024 байта) до 1024 указателей (для блока размером 4096 байт). Это позволяет использовать довольно большие файлы. Для файлов большего размера 14-й указатель (отмечен на схеме числом 13) является *двойным косвенным указателем* — он указывает на блок указателей, которые, в свою очередь, указывают на блоки указателей, указывающие на блоки данных файла. А если когда-либо возникнет необходимость в действительно огромном файле, то существует следующий уровень: последний указатель в индексном дескрипторе является *тройным косвенным указателем*.

Такая система, которая выглядит сложной, призвана удовлетворить ряд требований. Во-первых, она позволяет добиться фиксированного размера структуры индексного дескриптора и в то же время допускает произвольный размер файлов. Кроме того, она позволяет файловой системе хранить блоки файла в виде несмежных блоков, благодаря чему возможен произвольный доступ к данным с помощью команды `lseek()`; ядру необходимо лишь определить, по какому указателю (или указателям) следовать. И наконец, для небольших файлов, которые составляют подавляющее большинство от общего числа файлов во многих файловых системах, такая схема разрешает быстрый доступ к блокам данных файла через прямые указатели индексного дескриптора.

Для примера здесь выполнена оценка одной системы, содержащую более чем 150 000 файлов. Около 30 % этих файлов имели размер не более 1000 байт каждый, а 80 % файлов занимали 10 000 байт и менее. Если принять размер блока равным 1024 байтам, для всех файлов из второй группы можно было бы использовать всего 12 прямых указателей, которые могут ссылаться на блоки, содержащие в общей сложности 12 288 байт. При использовании блока размером 4096 байт этот предел возрастает до 49 152 байт (и он охватывает 95 % файлов в данной системе).

Такая схема допускает также наличие файлов гигантских размеров; при размере блока 4096 байт самый большой теоретически возможный размер файла составляет чуть более $1024 \times 1024 \times 1024 \times 4096$ байт, или около 4 Тбайт (4096 Гбайт). (Я говорю «чуть более», поскольку имеются блоки, на которые указывают прямые, косвенные и двойные косвенные указатели. Но их количество несущественно по сравнению с диапазоном, для которого можно использовать тройной косвенный указатель.)

Еще одним преимуществом, которое предоставляет такая схема, является то, что файлы могут обладать дырами, как описано в разделе 4.7. Вместо выделения блоков с пустыми байтами для дыр в файле файловой системе достаточно пометить (значением 0) соответствующие указатели в индексном дескрипторе и в блоках косвенного указателя, чтобы показать, что они не ссылаются на актуальные блоки диска.

14.5. Виртуальная файловая система

Каждая файловая система, которая доступна в Linux, отличается деталями своей реализации. К числу таких различий относятся, например, способы выделения блоков для файла и организация каталогов. Если бы каждой программе, которая работает с файлами, потребовалось вникать в особенности каждой файловой системы, то тогда задача по написанию программ, работающих во всех файловых системах, стала бы практически неосуществимой. *Виртуальная файловая система* (VFS, virtual file system, которую иногда называют также *виртуальным коммутатором файлов*) — это функция ядра, которая решает названную проблему, создавая уровень абстракции для операций файловой системы (рис. 14.3). Принципы, лежащие в основе виртуальной файловой системы, просты.

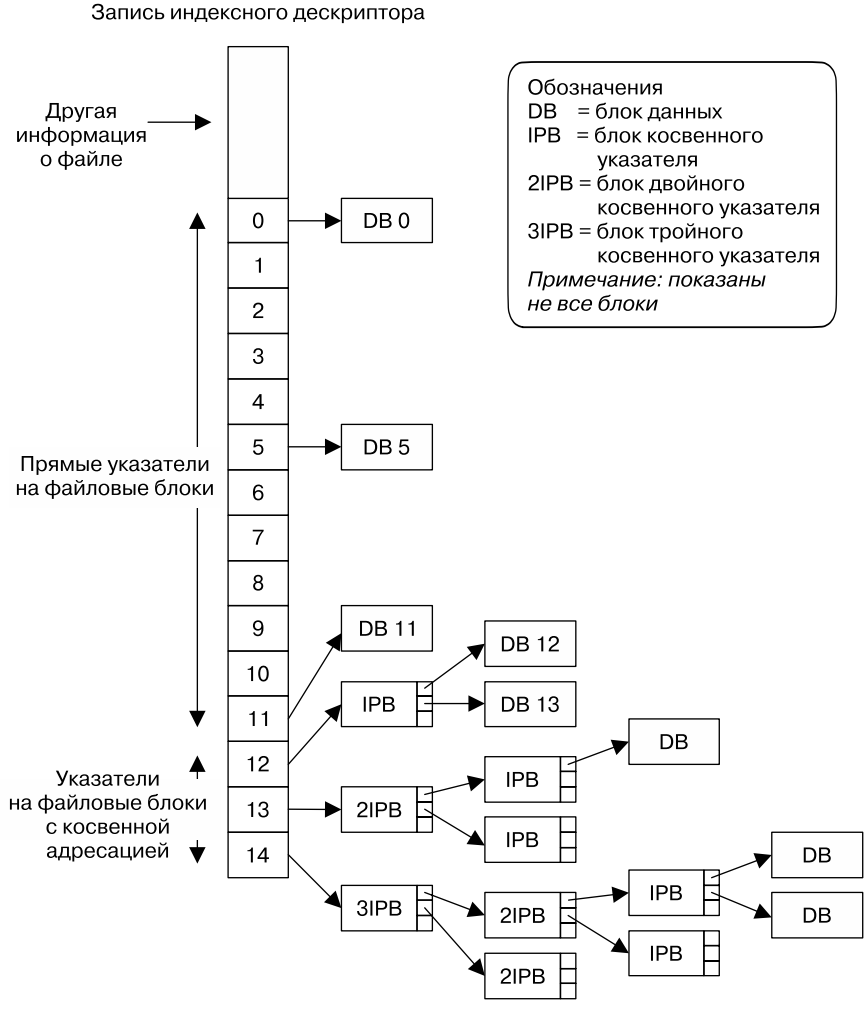


Рис. 14.2. Структура файловых блоков для файла в файловой системе ext2

- Виртуальная файловая система определяет обобщенный интерфейс для операций файловой системы. Все программы, которые работают с файлами, выражают свои операции в терминах данного обобщенного интерфейса.
- Каждая файловая система обеспечивает реализацию интерфейса виртуальной файловой системы.

Согласно этой схеме программам необходимо понимать только VFS-интерфейс. Они могут игнорировать детали реализации отдельных файловых систем.

Интерфейс виртуальной файловой системы содержит операции, соответствующие всем обычным системным вызовам для работы с файловыми системами и каталогми: `open()`, `read()`, `write()`, `lseek()`, `close()`, `truncate()`, `stat()`, `mount()`, `umount()`, `mmap()`, `mkdir()`, `link()`, `unlink()`, `symlink()` и `rename()`.

Уровень абстракции VFS очень близок к традиционной модели файловой системы UNIX. Естественно, некоторые файловые системы — в особенности не относящиеся к семейству UNIX — поддерживают не все операции виртуальной файловой системы (например, файловая система VFAT, разработанная компанией Microsoft, не поддерживает символические ссылки, созданные с помощью команды `symlink()`). В таком случае основная файловая система возвращает обратно на уровень VFS код ошибки, сообщающий об отсутствии поддержки, а виртуальная система, в свою очередь, возвращает этот код ошибки в приложение.

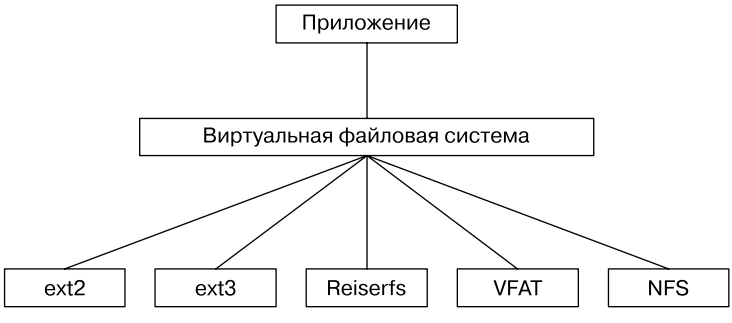


Рис. 14.3. Виртуальная файловая система

14.6. Журналируемые файловые системы

Файловая система ext2 является хорошим примером традиционной файловой системы UNIX, и для нее характерны типичные ограничения таких файловых систем: после системного сбоя необходимо выполнить проверку согласованности (`fsck`) или перезагрузку для обеспечения целостности системы. Это необходимо, поскольку в момент сбоя могло быть не завершено обновление какого-либо файла и метаданные файловой системы (записи каталогов, информация индексных дескрипторов и указатели на блоки данных) могут оказаться несогласованными. Если такие несоответствия не устранить, то файловая система может быть повреждена еще сильнее. Проверка согласованности файловой системы гарантирует целостность метаданных файловой системы. Там, где это возможно, выполняются исправления; если же информацию невозможно извлечь (включая, возможно, и данные), то она отбрасывается.

Проблема в том, что для проверки целостности необходимо обследовать всю файловую систему. Для небольшой файловой системы на это может потребоваться от нескольких секунд до минут. В больших файловых системах на проверку могут уйти часы,

и это представляет серьезную проблему для систем, которые должны сохранять высокую доступность (например, сетевые серверы).

В журналируемых файловых системах устранена необходимость продолжительной проверки целостности файловой системы после системного сбоя. Журналируемая файловая система заносит (журналирует) все обновления метаданных в специальный файл журнала на диске до того, как они будут осуществлены фактически. Эти обновления заносятся в группы, связанные с обновлениями метаданных *(транзакции)*. Если происходит системный сбой во время транзакции, то при перезагрузке системы можно использовать журнал, чтобы быстро отменить все незавершенные обновления и вернуть систему в согласованное состояние. (Выражаясь языком, который применяют для баз данных, мы можем сказать, что журналируемая файловая система всегда гарантирует *фиксацию* транзакций метаданных файла как завершенного модуля.) Даже довольно большие файловые системы могут быть снова доступны уже через несколько секунд после системного сбоя, и это делает их весьма привлекательными в тех случаях, когда требуется повышенная доступность.

Самым заметным недостатком журналирования является то, что при этом затрачивается дополнительное время на обновления файлов, хотя при хорошей реализации эти издержки можно снизить.

> Некоторые журналируемые файловые системы гарантируют лишь согласованность метаданных файла. Поскольку они не следят за данными файла, эти данные могут быть потеряны при сбое. Файловые системы ext3, ext4 и Reiserfs обеспечивают возможность слежения за обновлениями данных, но, в зависимости от рабочей нагрузки, это может привести к снижению скорости ввода-вывода.

К числу журналируемых файловых систем, доступных в Linux, относятся следующие.

- Reiserfs. Это первая журналируемая файловая система, которая была интегрирована в ядро (в версии 2.4.1). Она обладает функцией, которая называется *упаковкой хвостов* (или *слиянием хвостов*): небольшие файлы (а также завершающие фрагменты больших файлов) упаковываются в те же дисковые блоки, что и метаданные файла. Поскольку во многих системах присутствует большое количество маленьких файлов (а некоторые приложения создают такие файлы), упомянутая функция позволяет высвободить существенный объем дискового пространства.
- Файловая система ext3 явилась результатом проекта по добавлению журналирования в систему ext2 с минимальными затратами. Миграция от файловой системы ext2 к ext3 осуществляется очень просто (нет необходимости в создании резервной копии файлов и ее восстановлении), возможно также выполнить миграцию в обратном направлении. Файловая система ext3 была интегрирована в ядро в версии 2.4.15.
- Файловая система JFS разработана компанией IBM. Она интегрирована в ядро в версии 2.4.20.
- Файловая система XFS (oss.sgi.com/projects/xfs/) была разработана в начале 90-х годов компанией Silicon Graphics (SGI) для ОС Irix, собственной реализации UNIX. В 2001 году файловая система XFS была портирована в Linux и стала доступна в качестве свободного программного обеспечения. Она была интегрирована в ядро в версии 2.4.24.

Поддержка различных файловых систем указывается с помощью параметров ядра, которые устанавливаются в меню File systems (Файловые системы) при конфигурировании ядра.

На момент написания книги известно, что ведется работа над двумя другими файловыми системами, которые обеспечивают журналирование и некоторые другие расширенные функции.

- Файловая система ext4 (ext4.wiki.kernel.org/) является наследницей файловой системы ext3. Первые фрагменты ее реализации были добавлены в ядро в версии 2.6.19, а в более поздних версиях ядра были добавлены различные функции. В число планируемых (или уже реализованных) функций файловой системы ext4 входят экстенты (резервирование смежных блоков для хранения данных), а также другие функции, которые призваны снизить фрагментацию файлов, выполнять дефрагментацию сетевых файловых систем, ускорить проверку файловой системы и поддерживать наносекундные метки времени.
- Файловая система Btrfs (B-tree FS; btrfs.wiki.kernel.org) — это новая файловая система, которая с самого начала разрабатывается для обеспечения широкого ряда современных функций, таких как экстенты, запись снимков состояния системы (такая функция эквивалентна метаданным и журналированию), контрольные суммы для данных и для метаданных, проверка сетевых файловых систем, дефрагментация сетевых файловых систем, эффективное использование пространства за счет упаковки небольших файлов, и такое же эффективное индексирование каталогов. Эта файловая система интегрирована в ядро в версии 2.6.29.

14.7. Иерархия одиночного каталога и точки монтирования

В Linux, как и в других UNIX-системах, все файлы из всех файловых систем располагаются в одном дереве каталогов. В основании этого дерева находится корневой каталог, / (слеш). Другие файловые системы *монтируются* в корневом каталоге и возникают как поддеревья в общей иерархии. Для монтирования файловой системы суперпользователь может применить такую команду:

```
$ mount device directory
```

Эта команда «прикрепляет» файловую систему к устройству `device` в указанном каталоге `directory` в иерархии каталогов — в *точке монтирования* данной файловой системы. Существует возможность изменения места, в котором производится монтирование: для этого выполняется размонтирование файловой системы с помощью команды `umount`, а затем она монтируется заново в другой точке.

> В Linux 2.4.19 и более поздних версиях картина усложняется. Теперь ядро поддерживает попроцессные пространства имен монтирования. Это означает, что каждый процесс потенциально обладает собственным набором точек монтирования файловой системы, и поэтому может видеть иерархию каталога отлично от других процессов. Более подробно мы объясним это при описании флага CLONE_NEWNS в разделе 28.2.1.

Чтобы вывести список смонтированных в данный момент файловых систем, можно использовать команду `mount` без аргументов, как в приведенном ниже примере (результат ее работы показан в сокращенном виде):

```
$ mount
/dev/sda6 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
/dev/sda8 on /home type ext3 (rw,acl,user_xattr)
/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 on /home/mtk/test type reiserfs (rw)
```

На рис. 14.4 показана часть структуры каталогов и файлов для системы, в которой была выполнена приведенная выше команда `mount`. На этой схеме показаны точки монтирования по отношению к иерархии каталога.

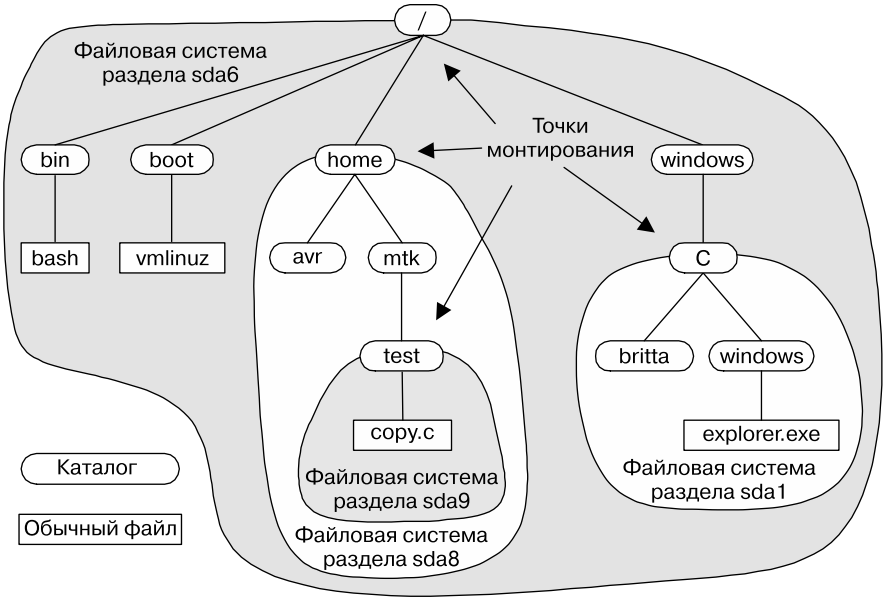


Рис. 14.4. Пример иерархии каталога, в которой показаны точки монтирования файловой системы

14.8. Монтирование и размонтирование файловых систем

Системные вызовы `mount()` и `umount()` позволяют привилегированному (`CAP_SYS_ADMIN`) процессу монтировать и размонтировать файловые системы. Однако они не стандартизированы в документе SUSv3, и их работа различна в разных реализациях UNIX и в разных файловых системах.

До рассмотрения этих системных вызовов полезно получить сведения о трех файлах, которые содержат информацию о файловых системах, смонтированных в данный момент или могущих быть смонтированными.

- Список смонтированных в данный момент файловых систем можно считать из характерного для Linux виртуального файла `/proc/mounts`. Этот файл является интерфейсом для структур данных ядра и поэтому он всегда содержит точную информацию о смонтированных файловых системах.

  После появления упомянутой выше функции попроцессных пространств имен монтирования у каждого процесса теперь есть файл /proc/PID/mounts, в котором перечислены точки монтирования, составляющие его пространство имен монтирования. Файл /proc/mounts является всего лишь символической ссылкой на /proc/self/mounts.

- Команды `mount(8)` и `umount(8)` автоматически заводят файл `/etc/mtab`, который содержит информацию, сходную с информацией в файле `/proc/mounts`, но являющуюся более

детальной. В частности, файл `/etc/mtab` содержит специфичные для файловой системы параметры, передаваемые команде `mount(8)`, которые не показаны в файле `/proc/mounts`. Однако поскольку системные вызовы `mount()` и `umount()` не обновляют файл `/etc/mtab`, информация в нем может оказаться неточной, если какое-либо приложение, которое монтирует или размонтирует устройства, не обновит его.

- Файл `/etc/fstab`, который обслуживается системным администратором, содержит описания всех доступных файловых систем в системе. Он используется командами `mount(8)`, `umount(8)` и `fsck(8)`.

Формат файлов `/proc/mounts`, `/etc/mtab` и `/etc/fstab` одинаков и описан на странице руководства `fstab(5)`. Приведу пример строки из файла `/proc/mounts`:

```
/dev/sda9 /boot ext3 rw 0 0
```

Эта строка содержит шесть полей, таких как:

- имя монтируемого устройства;
- точка монтирования этого устройства;
- тип файловой системы;
- флаги монтирования. В приведенном примере флаг `rw` означает, что файловая система была смонтирована для чтения/записи;
- число, которое используется для управления операцией резервного копирования файловой системы с помощью команды `dump(8)`. Данное поле и следующее за ним используются только в файле `/etc/fstab`; для файлов `/proc/mounts` и `/etc/mtab` эти поля всегда равны 0;
- число, которое используется для управления порядком проверки файловых систем командой `fsck(8)` во время загрузки системы.

На страницах руководства `getfsent(3)` и `getmntent(3)` документированы функции, которые можно использовать для чтения записей из этих файлов.

14.8.1. Монтирование файловой системы: mount()

Системный вызов `mount()` монтирует файловую систему, содержащуюся на устройстве, указанном в аргументе `source`, в каталог *(точку монтирования)*, указанный в аргументе `target`.

```
#include <sys/mount.h>

int mount(const char *source, const char *target, const char *fstype,
          unsigned long mountflags, const void *data);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Для первых двух аргументов использованы имена `source` и `target`, поскольку системный вызов `mount()` может выполнять и другие операции, помимо монтирования дисковой файловой системы в каталоги.

Аргумент `fstype` является строкой, которая идентифицирует тип файловой системы, расположенной на устройстве, например, ext4 или btrfs.

Аргумент `mountflags` является битовой маской, составленной с помощью команды ИЛИ (`|`) для нуля или для нескольких флагов, показанных в табл. 14.1. Эти флаги будут описаны подробнее далее.

Завершающий аргумент системного вызова `mount()` — `data` — является указателем на буфер с информацией, интерпретация которой зависит от файловой системы.

Для большинства типов файловых систем этот аргумент представляет собой строку, в которой через запятую приведены значения параметров. Полный перечень этих параметров можно найти на странице руководства `mount(8)` (или в документации к файловой системе, если она не описана на странице `mount(8)`).

Таблица 14.1. Значения флагов mountflags системного вызова mount()

Флаг	Назначение
MS_BIND	Создать связанную точку монтирования (начиная с версии Linux 2.4)
MS_DIRSYNC	Сделать обновления каталогов синхронными (начиная с версии Linux 2.6)
MS_MANDLOCK	Разрешить обязательную блокировку файлов
MS_MOVE	Автоматически переносить точку монтирования в новое местоположение
MS_NOATIME	Не обновлять время последнего доступа для файлов
MS_NODEV	Не разрешать доступ к каталогм
MS_NODIRATIME	Не обновлять время последнего доступа для каталогов
MS_NOEXEC	Не разрешать выполнение программ
MS_NOSUID	Отключить программы с полномочиями setuid и setgid
MS_RDONLY	Монтировать только для чтения; создавать или изменять файлы нельзя
MS_REC	Рекурсивное монтирование (начиная с версии Linux 2.6.20)
MS_RELATIME	Обновлять время последнего доступа, только если оно старше времени последнего изменения или последнего изменения статуса (начиная с версии Linux 2.4.11)
MS_REMOUNT	Повторное монтирование с новыми аргументами mountflags и data
MS_STRICTATIME	Всегда обновлять время последнего доступа (начиная с версии Linux 2.6.30)
MS_SYNCHRONOUS	Сделать все обновления файлов и каталогов синхронными

Аргумент `mountflags` является битовой маской флагов, которые влияют на выполнение системного вызова `mount()`. Для этого аргумента можно не указывать флаг или же использовать следующие.

- `MS_BIND` (начиная с версии Linux 2.4) — создает связанную точку монтирования. Мы описываем эту функцию в разделе 14.9.4. Если указан этот флаг, аргументы `fstype`, `mountflags` и `data` игнорируются.
- `MS_DIRSYNC` (начиная с версии Linux 2.6) — делает обновление каталогов синхронным. Это напоминает действие флага `open()` `O_SYNC` (см. раздел 13.3), но распространяется только на обновления каталогов. Описанный ниже флаг `MS_SYNCHRONOUS` обеспечивает расширенную функциональность по сравнению с флагом `MS_DIRSYNC`, позволяя синхронное обновление как файлов, так и каталогов. Флаг `MS_DIRSYNC` по-

зволяет какому-либо приложению (например, `open(pathname, O_CREAT)`, `rename()`, `link()`, `unlink()`, `symlink()` и `mkdir()`) убедиться в том, что обновления каталога синхронизированы, не затрачивая ресурсов на синхронизацию всех обновлений файлов. Назначение флага `FS_DIRSYNC_FL` (см. раздел 15.5) подобно флагу `MS_DIRSYNC`, с тем отличием, что флаг `FS_DIRSYNC_FL` можно применять к отдельным каталогам. Кроме того, в Linux системный вызов `fsync()`, примененный к файловому дескриптору, который указывает на каталог, позволяет выполнять синхронизацию обновлений каталогов. (Эта особенность работы системного вызова `fsync()` в Linux не отражена в стандарте SUSv3.)

- `MS_MANDLOCK` — разрешает обязательное блокирование записи для файлов в данной файловой системе. Мы рассмотрим блокирование записи в главе 51.
- `MS_MOVE` — автоматически перемещает существующую точку монтирования, указанную в аргументе `source`, в новое местоположение, определяемое аргументом `target`. Это соответствует параметру `-move` системного вызова `mount(8)`. Действие эквивалентно размонтированию поддерева с его последующим монтированием в другом месте, за исключением того, что здесь нет такого момента времени, когда поддерево является размонтированным. Аргумент `source` должен быть строкой, которая указана в качестве аргумента `target` для предыдущего вызова `mount()`. Когда этот флаг указан, аргументы `fstype`, `mountflags` и `data` игнорируются.
- `MS_NOATIME` — не обновлять время последнего доступа для файлов в данной файловой системе. Назначение этого флага, а также описанного ниже флага `MS_NODIRATIME`, состоит в том, чтобы избежать избыточного доступа к диску, который необходим для обновления индексного дескриптора файла всякий раз, когда происходит доступ к файлу. Для некоторых приложений отслеживание метки времени не является критичным, и за счет устранения этой операции можно существенно увеличить производительность. Назначение флага `MS_NOATIME` такое же, как у флага `FS_NOATIME_FL` (см. раздел 15.5), с тем лишь отличием, что флаг `FS_NOATIME_FL` можно применять для отдельных файлов. Linux обеспечивает подобную функциональность также с помощью флага `O_NOATIME open()`, который задает такое поведение для отдельных открытых файлов (см. раздел 4.3.1).
- `MS_NODEV` — не разрешает доступ к блочным и к символьным устройствам в данной файловой системе. Это функция безопасности, предназначенная для того, чтобы запретить пользователям выполнение таких действий, как вставка съемного диска, содержащего специальные файлы устройств, которые могли бы разрешить произвольный доступ к системе.
- `MS_NODIRATIME` — не обновлять время последнего доступа для каталогов в данной файловой системе. (Этот флаг обеспечивает часть функциональности, если сравнить его с флагом `MS_NOATIME`, который не допускает обновление времени последнего доступа для всех типов файлов.)
- `MS_NOEXEC` — запретить выполнение программ (или сценариев) из этой файловой системы. Эта возможность удобна, если файловая система содержит исполняемые файлы не из Linux.
- `MS_NOSUID` — отключить программы с полномочиями setuid и setgid в данной файловой системе. Это функция безопасности, которая не позволяет пользователям запускать программы с полномочиями setuid и setgid со съемных устройств.
- `MS_RDONLY` — монтировать файловую систему только для чтения, чтобы исключить возможность создания новых файлов или изменения уже существующих.
- `MS_REC` (начиная с версии Linux 2.4.11) — этот флаг используется в сочетании с другими флагами (например, с `MS_BIND`), чтобы рекурсивно выполнить монтирование для всех точек монтирования в поддереве.

- MS_RELATIME (начиная с версии Linux 2.6.20) — обновить метку времени последнего доступа для файлов в данной файловой системе, только если текущее значение этой метки меньше или равно метке времени либо последнего изменения, либо последнего изменения статуса. За счет этого можно добиться некоторого выигрыша в производительности (как для флага MS_NOATIME), но рассматриваемый флаг удобен для программ, которым необходимо знать, происходило ли чтение файла с момента его последнего обновления. Начиная с версии Linux 2.6.30, поведение, которое обеспечивается флагом MS_RELATIME, принято по умолчанию (если не указан флаг MS_NOATIME), а флаг MS_STRICTATIME необходим для восстановления «классического» поведения. Кроме того, начиная с версии Linux 2.6.30, метка времени последнего доступа обновляется всегда, если ее значение более чем на 24 часа отстоит от текущего времени, даже если это значение является более поздним, чем метки времени последнего изменения и последнего изменения статуса. (Это удобно для некоторых системных программ, проверяющих каталоги с целью обнаружения файлов, к которым недавно был осуществлен доступ.)
- MS_REMOUNT — изменить аргументы mountflags и data для файловой системы, которая уже смонтирована (например, сделать доступной для записи файловую систему, которая предназначалась только для чтения). При использовании этого флага аргументы source и target должны быть такими же, как и в исходном системном вызове mount(), при этом аргумент fstype игнорируется. Этот флаг устраняет необходимость в размонтировании и повторном монтировании диска, что может оказаться невозможным в некоторых случаях. Например, мы не можем размонтировать файловую систему, файлы которой открыты каким-либо процессом, или его текущий рабочий каталог находится внутри этой файловой системы (это всегда верно для корневой файловой системы). Еще одним примером необходимости использовать флаг MS_REMOUNT являются (размещаемые в памяти) файловые системы tmpfs (см. раздел 14.10), которые невозможно размонтировать, не потеряв их содержимое. Не все флаги mountflags допускают модификацию; см. подробности на странице руководства mount(2).
- MS_STRICTATIME (начиная с версии Linux 2.6.30) — всегда обновлять метку времени последнего доступа, когда осуществляется доступ к файлам данной файловой системы. Такой режим применялся по умолчанию до версии Linux 2.6.30. Если указан флаг MS_STRICTATIME, то будут проигнорированы флаги MS_NOATIME и MS_RELATIME, если они также присутствуют среди аргументов mountflags.
- MS_SYNCHRONOUS — сделать синхронным все обновления файлов и каталогов данной файловой системы. (Применительно к файлам действует так, словно файлы уже открыты с помощью флага open() O_SYNC.)

  Начиная с версии ядра 2.6.15, в Linux присутствуют четыре новых флага монтирования для поддержки совместно используемых поддеревьев. Это флаги MS_PRIVATE, MS_SHARED, MS_SLAVE и MS_UNBINDABLE. (Данные флаги можно применять в сочетании с флагом MS_REC, чтобы распространить их действие на все вложенные точки монтирования в поддереве монтирования.) Совместно используемые поддеревья предназначены для использования с некоторыми усовершенствованными функциями файловой системы, например для попроцессных пространств имен монтирования (см. описание флага CLONE_NEWNS в разделе 28.2.1) или для реализации файловой системы в пространстве пользователя (FUSE, Filesystem in Userspace). Совместно используемое поддерево позволяет контролируемым образом распространить точки монтирования файловой системы среди пространств имен точек монтирования. Подробности о совместно используемых поддеревьях можно найти в файле исходного кода ядра Documentation/filesystems/sharedsubtree.txt, а также в работе [Viro & Pai, 2006].

Пример программы

Программа в листинге 14.1 обеспечивает интерфейс командного уровня для системного вызова `mount(2)`. Фактически это сырая версия команды `mount(8)`. Приведенный ниже сеанс работы в оболочке демонстрирует использование данной программы. Мы начинаем с создания каталога, который будет использован в качестве точки монтирования, и монтируем файловую систему:

```
$ su                          Необходимая привилегия для монтирования файловой системы
Password:
# mkdir /testfs
# ./t_mount -t ext2 -o bsdgroups /dev/sda12 /testfs
# cat /proc/mounts | grep sda12       Проверка установки
/dev/sda12 /testfs ext3 rw 0 0        Не показывать группы bsdgroups
# grep sda12 /etc/mtab
```

Мы обнаруживаем, что предыдущая команда `grep` ничего не выводит, поскольку наша команда не обновляет файл `/etc/mtab`. Продолжим, повторно смонтировав файловую систему только для чтения:

```
# ./t_mount -f Rr /dev/sda12 /testfs
# cat /proc/mounts | grep sda12            Проверка изменения
/dev/sda12 /testfs ext3 ro 0 0
```

Фрагмент `ro` в строке, отображаемой из файла `/proc/mounts`, указывает на то, что монтирование выполнено только для чтения.

Наконец, мы перемещаем точку монтирования в новое местоположение внутри иерархии каталога:

```
# mkdir /demo
# ./t_mount -f m /testfs /demo
# cat /proc/mounts | grep sda12            Проверка изменения
/dev/sda12 /demo ext3 ro 0
```

Листинг 14.1. Использование системного вызова mount()

filesys/t_mount.c

```
#include <sys/mount.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);

    fprintf(stderr, "Usage: %s [options] source target\n\n", progName);
    fprintf(stderr, "Available options:\n");
#define fpe(str) fprintf(stderr, "    " str)        /* Короче! */
    fpe("-t fstype             [e.g., 'ext2' or 'reiserfs']\n");
    fpe("-o data               [file system-dependent options,\n");
    fpe("                      e.g., 'bsdgroups' for ext2]\n");
    fpe("-f mountflags         can include any of:\n");
#define fpe2(str) fprintf(stderr, "            " str)
    fpe2("b - MS_BIND          create a bind mount\n");
    fpe2("d - MS_DIRSYNC       synchronous directory updates\n");
    fpe2("l - MS_MANDLOCK      permit mandatory locking\n");
    fpe2("m - MS_MOVE          atomically move subtree\n");
    fpe2("A - MS_NOATIME       don't update atime (last access time)\n");
```

```
    fpe2("V - MS_NODEV         don't permit device access\n");
    fpe2("D - MS_NODIRATIME    don't update atime on directories\n");
    fpe2("E - MS_NOEXEC        don't allow executables\n");
    fpe2("S - MS_NOSUID        disable set-user/group-ID programs\n");
    fpe2("r - MS_RDONLY        read-only mount\n");
    fpe2("c - MS_REC           recursive mount\n");
    fpe2("R - MS_REMOUNT       remount\n");
    fpe2("s - MS_SYNCHRONOUS   make writes synchronous\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    unsigned long flags;
    char *data, *fstype;
    int j, opt;

    flags = 0;
    data = NULL;
    fstype = NULL;

    while ((opt = getopt(argc, argv, "o:t:f:")) != -1) {
        switch (opt) {
            case 'o':
            data = optarg;
            break;

            case 't':
            fstype = optarg;
            break;

            case 'f':
            for (j = 0; j < strlen(optarg); j++) {
                switch (optarg[j]) {
                case 'b':    flags |= MS_BIND;            break;
                case 'd':    flags |= MS_DIRSYNC;         break;
                case 'l':    flags |= MS_MANDLOCK;        break;
                case 'm':    flags |= MS_MOVE;            break;
                case 'A':    flags |= MS_NOATIME;         break;
                case 'V':    flags |= MS_NODEV;           break;
                case 'D':    flags |= MS_NODIRATIME;      break;
                case 'E':    flags |= MS_NOEXEC;          break;
                case 'S':    flags |= MS_NOSUID;          break;
                case 'r':    flags |= MS_RDONLY;          break;
                case 'c':    flags |= MS_REC;             break;
                case 'R':    flags |= MS_REMOUNT;         break;
                case 's':    flags |= MS_SYNCHRONOUS;     break;
                default:        usageError(argv[0], NULL);
                }
            }
            break;

            default:                usageError(argv[0], NULL);
        }
    }

    if (argc != optind + 2)
        usageError(argv[0], "Wrong number of arguments\n");
```

```
    if (mount(argv[optind], argv[optind + 1], fstype, flags, data) == -1)
        errExit("mount");

    exit(EXIT_SUCCESS);
}
```

filesys/t_mount.c

14.8.2. Размонтирование файловой системы: системные вызовы umount() и umount2()

Системный вызов `umount()` размонтирует смонтированную файловую систему.

```
#include <sys/mount.h>

int umount(const char *target);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

В аргументе `target` указывается точка монтирования файловой системы, которую следует размонтировать.

> В версии Linux 2.2 и более ранних можно идентифицировать файловую систему двумя способами: по точке монтирования или по имени устройства, содержащего эту файловую систему. Начиная с версии ядра 2.4 Linux не допускает использование второго способа, поскольку теперь файловую систему можно смонтировать в нескольких местах, вследствие чего указание файловой системы для аргумента target было бы неоднозначным. Мы подробно объясним этот момент в разделе 14.9.1.

Невозможно размонтировать файловую систему, которая *занята* — то есть если в файловой системе есть открытые файлы или же текущий рабочий каталог какого-либо процесса расположен где-либо в этой файловой системе. Применение системного вызова `umount()` к занятой файловой системе приведет к возникновению ошибки `EBUSY`.

Системный вызов `umount2()` является расширенной версией системного вызова `umount()`. Он позволяет более точно управлять работой с помощью аргумента `flags`.

```
#include <sys/mount.h>

int umount2(const char *target, int flags);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Аргумент `flags`, который является битовой маской, может не содержать значений или состоит из следующих флагов, объединенных операцией ИЛИ:

- `MNT_DETACH` (начиная с версии Linux 2.4.11) — выполняет «ленивое» размонтирование. Точка монтирования помечается таким образом, чтобы новые процессы не могли иметь доступ к ней, однако те процессы, которые уже используют ее, могут продолжать ее использование. Файловая система фактически размонтируется, когда все процессы прекращают использование точки монтирования.
- `MNT_EXPIRE` (начиная с версии Linux 2.6.8) — помечает точку монтирования как просроченную. Если исходный системный вызов `umount2()` осуществлен с указанием

данного флага и точка монтирования не занята, то такой вызов завершается ошибкой EAGAIN, однако при этом точка монтирования помечается как просроченная. (Если точка монтирования занята, то тогда системный вызов завершается ошибкой EBUSY, а точка монтирования не помечается как просроченная.) Точка монтирования остается просроченной до тех пор, пока ее не использует никакой из процессов. Второй системный вызов umount2() с указанием флага MNT_EXPIRE размонтирует просроченную точку монтирования. Так обеспечивается механизм размонтирования файловой системы, которая не использовалась в течение некоторого интервала времени. Данный флаг нельзя указывать совместно с флагом MNT_DETACH или MNT_FORCE.

- MNT_FORCE — осуществляет принудительное размонтирование, даже если устройство занято (только для точек монтирования NFS). Использование этого флага может вызвать потерю данных.
- UMOUNT_NOFOLLOW (начиная с версии Linux 2.6.34) — не разыменовывает аргумент target, если это символическая ссылка. Флаг предназначен для использования в некоторых программах set-user-ID-root, которые разрешают непривилегированным пользователям выполнять размонтирование, для того чтобы избежать проблем с безопасностью, которые могли бы возникнуть в том случае, если символическая ссылка target изменена и указывает на другое местоположение.

14.9. Дополнительные функции монтирования

Теперь мы рассмотрим некоторые усовершенствованные функции, которые можно применять при монтировании файловых систем. Разберем использование большинства из них на примере команды mount(8). Таких же результатов можно добиться, если какая-либо программа осуществит системный вызов mount(2).

14.9.1. Монтирование файловой системы в нескольких точках монтирования

В ядре Linux версий до 2.4 файловую систему можно было монтировать только в одной точке монтирования. Начиная с версии 2.4, файловую систему можно монтировать в нескольких местах внутри файловой системы. Поскольку каждая точка монтирования отображает одно и то же поддерево, изменения, сделанные посредством одной точки монтирования, становятся видны и в остальных, как показано в следующем сеансе работы в оболочке:

```
$ su                          Привилегии, необходимые для использования mount(8)
Password:
# mkdir /testfs               Создание двух каталогов для точек монтирования
# mkdir /demo
# mount /dev/sda12 /testfs    Монтирование файловой системы
                              в первой точке монтирования
# mount /dev/sda12 /demo      Монтирование файловой системы
                              во второй точке монтирования
# mount | grep sda12          Проверка установки
/dev/sda12 on /testfs type ext3 (rw)
/dev/sda12 on /demo type ext3 (rw)
# touch /testfs/myfile        Осуществление изменений посредством
                              первой точки монтирования
# ls /demo                    Просмотр файлов во второй точке монтирования
lost+found myfile
```

Из вывода команды ls понятно, что изменение, выполненное посредством первой точки монтирования (/testfs), видно через вторую точку монтирования (/demo).

В подразделе 14.9.4 при описании связанных точек монтирования приводится один пример, демонстрирующий удобство монтирования файловой системы в нескольких точках.

Именно потому, что устройство можно смонтировать в нескольких точках, системный вызов umount() не может принимать имя устройства в качестве аргумента, начиная с версии Linux 2.4.

14.9.2. Создание стека монтирования в одной точке

В версиях ядра до 2.4 точку монтирования можно было использовать только один раз. Начиная с версии ядра 2.4, Linux позволяет создавать стек из нескольких точек монтирования в единичной точке монтирования. При каждом новом монтировании скрывается поддерево каталогов, которое было видимым ранее в данной точке монтирования. При размонтировании точки, размещенной на вершине стека, становится видимой скрытая ранее точка, как показано в следующем сеансе работы в оболочке:

```
$ su                                Привилегии, необходимые для использования mount(8)
Password:
# mount /dev/sda12 /testfs          Создание первой точки монтирования в /testfs
# touch /testfs/myfile              Создание файла в данном поддереве
# mount /dev/sda13 /testfs          Помещение второй точки в стек /testfs
# mount | grep testfs               Проверка установки
/dev/sda12 on /testfs type ext3 (rw)
/dev/sda13 on /testfs type reiserfs (rw)
# touch /testfs/newfile             Создание файла в данном поддереве
# ls /testfs                        Просмотр файлов в данном поддереве newfile
# umount /testfs                    Изъятие точки монтирования из стека
# mount | grep testfs
/dev/sda12 on /testfs type ext3 (rw)    Теперь в каталоге /testfs
                                        только одна точка монтирования
# ls /testfs                        Теперь видна предыдущая точка монтирования
lost+found myfile
```

Один из вариантов применения такого стека точек монтирования — создание новой точки в существующей точке монтирования, которая занята. Процессы, которые удерживают файловые дескрипторы открытыми, изолированы системным вызовом chroot(), или которые имеют текущие рабочие каталоги внутри старой точки монтирования, продолжают работать под этой старой точкой, а процессы, которые осуществляют новые доступы, используют новую точку монтирования. В сочетании с флагом MNT_DETACH это позволяет обеспечить плавную миграцию из какой-либо файловой системы без необходимости перевода этой системы в режим одиночного пользователя. Еще один пример использования стека точек монтирования мы увидим в разделе 14.10, когда будем рассматривать файловую систему tmpfs.

14.9.3. Флаги монтирования, которые являются параметрами конкретной точки монтирования

В версиях ядра до 2.4 существовало однозначное соответствие между файловыми системами и точками монтирования. Но, поскольку, начиная с версии Linux 2.4, такое соответствие больше не сохраняется, некоторые из значений mountflags, описанных в разделе 14.8.1, можно задавать отдельно для каждой точки монтирования. Это относится

к флагам MS_NOATIME (начиная с версии Linux 2.6.16), MS_NODEV, MS_NODIRATIME (начиная с версии Linux 2.6.16), MS_NOEXEC, MS_NOSUID, MS_RDONLY (начиная с версии Linux 2.6.26) и MS_RELATIME. В следующем сеансе работы с оболочкой показано, как это выглядит в случае с флагом MS_NOEXEC:

```
$ su
Password:
# mount /dev/sda12 /testfs
# mount -o noexec /dev/sda12 /demo
# cat /proc/mounts | grep sda12
/dev/sda12 /testfs ext3 rw 0 0
/dev/sda12 /demo ext3 rw,noexec 0 0
# cp /bin/echo /testfs
# /testfs/echo "Art is something which is well done"
Art is something which is well done
# /demo/echo "Art is something which is well done"
bash: /demo/echo: Permission denied
```

14.9.4. Связанные (синонимичные) точки монтирования

Начиная с версии ядра 2.4, Linux разрешает создание связанных точек монтирования. *Связанная точка монтирования* (которая создается с помощью флага mount() MS_BIND) позволяет смонтировать каталог или файл в каком-либо еще местоположении в иерархии файловой системы. Это приводит к тому, что такой каталог или файл становятся видимыми в обоих размещениях. Связанная точка монтирования чем-то похожа на жесткую ссылку, но отличается от нее следующими двумя особенностями.

- Связанное монтирование может «скрещивать» точки монтирования файловой системы (даже изолированные системным вызовом chroot).
- Можно создавать связанную точку монтирования для каталога.

Связанное монтирование можно осуществить из оболочки с помощью параметра --bind для системного вызова mount(8), как показано в приведенных ниже примерах.

В первом примере происходит связанное монтирование каталога в другое местоположение. Показано, что файлы, которые создаются в одном каталоге, становятся видны и в другом месте размещения:

```
$ su                    Привилегии, которые необходимы для выполнения
                        системного вызова mount(8)
Password:
# pwd
/testfs
# mkdir d1              Создание каталога, который будет связан
                        с другим местоположением
# touch d1/x            Создание файла в этом каталоге
# mkdir d2              Создание точки монтирования, с которой
                        будет связан каталог d1
# mount --bind d1 d2    Связанное монтирование: каталог d1 виден
                        через каталог d2
# ls d2                 Проверка возможности просмотра содержимого
                        каталога d1 через каталог d2
x
# touch d2/y            Создание второго файла в каталоге d2
# ls d1                 Проверка того, что это изменение можно увидеть
                        через каталог d1
x    y
```

Во втором примере выполняется связанное монтирование файла в другое местоположение. Показано, что изменения файла в одном каталоге становятся видны и в другой точке монтирования:

```
# cat > f1                          Создание файла, который будет связан
                                    с другим местоположением
Chance is always powerful. Let your hook be always cast.
Нажимаем клавиши Ctrl+D
# touch f2                          Это новая точка монтирования
# mount --bind f1 f2                Связывание f1 с f2
# mount | egrep '(d1|f1)'           Просмотр того, как выглядит монтирование
/testfs/d1 on /testfs/d2 type none (rw,bind)
/testfs/f1 on /testfs/f2 type none (rw,bind)
# cat >> f2                         Меняем файл f2
In the pool where you least expect it, will be a fish.
# cat f1                            Изменение можно увидеть и в исходном файле
Chance is always powerful. Let your hook be always cast.
In the pool where you least expect it, will be a fish.
# rm f2                             Выполнить невозможно, поскольку это точка монтирования
rm: cannot unlink `f2': Device or resource busy
# umount f2                         Поэтому выполняем размонтирование
# rm f2                             Теперь можно удалить файл f2
```

Одним из примеров, в котором можно было бы использовать связанное монтирование, является создание «клетки» системного вызова `chroot` (см. раздел 18.12). Вместо репликации множества стандартных каталогов (таких как `/lib`) в «клетку», мы можем всего лишь создать связанные точки монтирования для этих каталогов (которые, возможно, смонтированы только для чтения) внутри данной «клетки».

14.9.5. Рекурсивное связанное монтирование

По умолчанию, если мы создаем связанную точку монтирования для какого-либо каталога с использованием флага `MS_BIND`, то в новом местоположении будет смонтирован только этот каталог; если в каталоге-источнике присутствуют вложенные точки монтирования, они не будут реплицированы в целевой точке монтирования `target`. В версии Linux 2.411 добавлен флаг `MS_REC`, который можно с помощью команды ИЛИ использовать вместе с флагом `MS_BIND` как часть аргумента `flags` системного вызова `mount()`, чтобы вложенные точки монтирования были реплицированы в целевой точке. Такой вариант называется *рекурсивным связанным монтированием*. Команда `mount(8)` снабжена параметром `--rbind`, который позволяет достичь такого же результата из оболочки, как показано в приведенном ниже сеансе работы.

Начнем с создания дерева каталогов (`src1`), смонтированного в точке `top`. Это дерево содержит вложенную структуру (`src2`) в точке монтирования `top/sub`.

```
$ su
Password:
# mkdir top                 Это точка монтирования верхнего уровня
# mkdir src1                Будем монтировать это в точке top
# touch src1/aaa
# mount --bind src1 top     Создаем нормальное связанное монтирование
# mkdir top/sub             Создаем каталог для вложенного монтирования под точкой top
# mkdir src2                Будем монтировать это в точке top/sub
# touch src2/bbb
# mount --bind src2 top/sub Создаем нормальное связанное монтирование
# find top                  Проверяем содержимое дерева монтирования top
```

```
top
top/aaa
top/sub                          Это вложенное монтирование
top/sub/bbb
```

Теперь выполним еще одно связанное монтирование (`dir1`), использовав в качестве источника `top`. Поскольку это новое монтирование является нерекурсивным, вложенная точка монтирования не реплицируется.

```
# mkdir dir1
# mount --bind top dir1          Здесь мы используем обычное связанное монтирование
# find dir1
dir1
dir1/aaa
dir1/sub
```

Отсутствие строки `dir1/sub/bbb` среди результатов работы команды `find` говорит о том, что вложенная точка монтирования `top/sub` не была реплицирована.

Выполним теперь рекурсивное связанное монтирование (`dir2`), используя точку `top` как источник.

```
# mkdir dir2
# mount --rbind top dir2
# find dir2
dir2
dir2/aaa
dir2/sub
dir2/sub/bbb
```

Наличие строки `dir2/sub/bbb` среди результатов работы команды `find` говорит о том, что вложенная точка монтирования `top/sub` была реплицирована.

14.10. Файловая система виртуальной памяти: tmpfs

Все файловые системы, рассмотренные ранее в этой главе, размещаются на дисках. Однако Linux поддерживает также *виртуальные файловые системы*, которые располагаются в памяти. Для приложений они выглядят подобно любой другой файловой системе: к файлам и каталогм подобных систем можно применять все те же операции (`open()`, `read()`, `write()`, `link()`, `mkdir()` и т. д.). Но есть, однако, одно важное отличие: файловые операции осуществляются намного быстрее, поскольку не задействован доступ к диску.

Для Linux созданы различные файловые системы, основанные на использовании памяти. Самой разработанной из них к настоящему моменту является файловая система tmpfs, которая впервые появилась в версии Linux 2.4. Файловая система tmpfs отличается от других файловых систем, использующих память, тем, что она является файловой системой *виртуальной* памяти. Это означает, что она использует не только оперативную память, но также и область подкачки, если полностью исчерпана оперативная память. (Несмотря на то что файловая система tmpfs описана здесь как специфичная для Linux, в большинстве реализаций UNIX присутствует в какой-либо форме поддержка файловых систем на основе памяти.)

Файловая система tmpfs является необязательным компонентом ядра Linux, который конфигурируется с помощью параметра CONFIG_TMPFS.

Для создания файловой системы tmpfs используется команда такого типа:

```
# mount -t tmpfs source target
```

Имя аргумента `source` может быть любым, важно лишь, чтобы оно присутствовало в файле `/proc/mounts` и отображалось с помощью команд `mount` и `df`. Аргумент `target` — это, как обычно, точка монтирования файловой системы. Следует отметить, что не обязательно использовать сначала команду `mkfs` для создания файловой системы, поскольку ядро автоматически создает ее во время системного вызова `mount()`.

В качестве примера использования файловой системы tmpfs можно привести применение стека монтирования (чтобы нам не пришлось беспокоиться о том, что точка монтирования /tmp уже используется) и создание файловой системы tmpfs в точке монтирования `/tmp` следующим образом:

```
# mount -t tmpfs newtmp /tmp
# cat /proc/mounts | grep tmp
newtmp /tmp tmpfs rw 0 0
```

Команда, подобная приведенной выше (или эквивалентная запись в файле `/etc/fstab`), иногда применяется для улучшения производительности приложений (например, компиляторов), которые активно используют каталог `/tmp` для создания временных файлов.

По умолчанию файловой системе tmpfs разрешено занимать не более половины размера оперативной памяти, однако можно использовать параметр `size=nbytes mount`, чтобы задать другую верхнюю границу для размера этой файловой системы, либо при создании системы, либо во время ее последующего повторного монтирования. (Файловая система tmpfs потребляет лишь столько памяти и пространства подкачки, сколько необходимо в данный момент для содержащихся в ней файлов.)

Если мы размонтируем файловую систему tmpfs или если происходит системный сбой, то все данные в этой файловой системе утрачиваются; отсюда и ее название — tmpfs.

Помимо применения для пользовательских приложений, файловая система tmpfs служит также двум специальным целям.

- Невидимая файловая система tmpfs, смонтированная внутренним образом с помощью ядра, использовалась для реализации совместно используемой памяти System V, а также для совместно используемого анонимного распределения памяти.
- Файловая система tmpfs, смонтированная в точке `/dev/shm`, используется для реализации совместно используемой памяти и семафоров POSIX с помощью библиотеки glibc.

14.11. Получение информации о файловой системе: statvfs()

Библиотечные функции `statvfs()` и `fstatvfs()` получают информацию о смонтированной файловой системе.

```
#include <sys/statvfs.h>

int statvfs(const char *pathname, struct statvfs *statvfsbuf);
int fstatvfs(int fd, struct statvfs *statvfsbuf);
```

Обе функции возвращают `0` при успешном завершении и `-1` при ошибке

Единственное различие между этими двумя функциями состоит в том, каким образом идентифицируется файловая система. Для функции `statvfs()` используется аргумент `pathname`, чтобы указать имя любого файла в файловой системе. Для функции `fstatvfs()` указывается открытый дескриптор файла, `fd`, ссылающийся на какой-либо файл в файловой системе. Обе функции возвращают структуру `statvfs`, содержащую информацию о файловой системе в буфере, на который указывает аргумент `statvfsbuf`. Эта структура составлена следующим образом:

```
struct statvfs {
    unsigned long f_bsize;    /* Размер блока файловой системы (в байтах) */
    unsigned long f_frsize;   /* Фундаментальный размер блока
                                 файловой системы (в байтах) */
    fsblkcnt_t f_blocks;      /* Общее количество блоков в файловой
                                 системе (в единицах 'f_frsize') */
    fsblkcnt_t f_bfree;       /* Общее количество свободных блоков */
    fsblkcnt_t f_bavail;      /* Количество свободных блоков, доступных
                                 для непривилегированного процесса */
    fsfilcnt_t f_files;       /* Общее количество индексных дескрипторов */
    fsfilcnt_t f_ffree;       /* Общее количество свободных индексных дескрипторов */
    fsfilcnt_t f_favail;      /* Количество индексных дескрипторов,
                                 доступных для непривилегированного
                                 процесса (задается в 'f_ffree' в Linux) */
    unsigned long f_fsid;     /* Идентификатор файловой системы */
    unsigned long f_flag;     /* Флаги монтирования */
    unsigned long f_namemax;  /* Максимальная длина имен файлов
                                 для данной файловой системы */
};
```

Назначение большинства полей в структуре `statvfs` ясно из сопровождающих комментариев. Отметим некоторые особенности, касающиеся отдельных полей.

- Типы данных `fsblkcnt_t` и `fsfilcnt_t` являются целочисленными и определены стандартом SUSv3.
- Для большинства файловых систем Linux значения `f_bsize` и `f_frsize` одинаковы. Однако некоторые файловые системы поддерживают фрагменты блоков, которые могут быть использованы для выделения меньших единиц хранения в конце файла, если не требуется полный блок. Это устраняет потерю пространства, которая возникла бы при выделении полного блока. В подобных файловых системах параметр `f_frsize` задает размер фрагмента, а `f_bsize` — размер целого блока. (Представление о фрагментах в файловых системах UNIX впервые появилось в начале 1980-х годов в файловой системе 4.2BSD Fast File System, которая описана в работе [McKusick et al., 1984].)
- Многие «родные» файловые системы UNIX и Linux поддерживают представление о резервировании некоторой части блоков файловой системы для суперпользователя на тот случай, когда файловая система становится заполненной. Суперпользователь по-прежнему может войти в систему и принять меры по устранению данной проблемы. Если в файловой системе есть зарезервированные блоки, то разность значений полей `f_bfree` и `f_bavail` в структуре `statvfs` сообщит нам, сколько блоков зарезервировано.
- Флаг `f_flag` является битовой маской флагов, используемых для монтирования файловой системы; то есть содержит информацию, подобную аргументу `mountflags`, передаваемому в системный вызов `mount(2)`. Однако константы, применяемые для битов данного поля, имеют имена, начинающиеся с префикса `ST_`, а не `MS_`, который используется в аргументе `mountflags`. Согласно стандарту SUSv3 необходимы лишь

константы ST_RDONLY и ST_NOSUID, однако реализация библиотеки glibc поддерживает полный набор констант с именами, соответствующими константам MS_*, описанным для аргумента mountflags системного вызова mount().

- Поле f_fsid используется в некоторых реализациях UNIX для возврата уникального идентификатора файловой системы — например, значения, которое основано на идентификаторе устройства, содержащего данную файловую систему. В большинстве файловых систем Linux это поле содержит 0.

Стандарт SUSv3 описывает обе функции: statvfs() и fstatvfs(). В Linux (как и в некоторых других реализациях UNIX) эти функции размещены слоем выше над довольно похожими системными вызовами statfs() и fstatfs(). (В некоторых реализациях UNIX системный вызов statfs() есть, а вызов statvfs() отсутствует.) Принципиальные отличия (помимо некоторой разницы в названиях полей) заключаются в следующем.

- Функции statvfs() и fstatvfs() возвращают поле f_flag, которое сообщает информацию о флагах монтирования файловой системы. (В реализации библиотеки glibc эта информация извлекается путем сканирования файла /proc/mounts или /etc/mtab.)
- Системные вызовы statfs() и fstatfs() возвращают поле f_type, которое сообщает тип файловой системы (так, например, значение 0xef53 говорит о том, что файловая система — ext2).

Подкаталог filesys ресурса, содержащего программный код примеров для данной книги, содержит файлы t_statvfs.c и t_statfs.c, демонстрирующие применение функций statvfs() и statfs().

14.12. Резюме

Устройства представлены записями в каталоге /dev. Каждое устройство имеет соответствующий драйвер устройства, который реализует стандартный набор операций, включающий в себя такие, которые соответствуют системным вызовам open(), read(), write() и close(). Устройство может быть реальным, и тогда присутствует соответствующее ему аппаратное устройство, или виртуальным, и тогда аппаратное устройство отсутствует, но, несмотря на это, ядро предоставляет драйвер устройства, который реализует такой же API, какой есть у реального устройства.

Жесткий диск имеет один или несколько разделов, каждый из которых может содержать файловую систему. Файловая система — это упорядоченный набор обычных файлов и каталогов. В Linux реализованы различные файловые системы, в число которых входит традиционная файловая система ext2. По своей концепции эта система напоминает ранние файловые системы UNIX, состоящие из загрузочного блока, суперблока, таблицы индексных дескрипторов и области данных, которая содержит блоки файловых данных. Каждый файл имеет запись в таблице индексных дескрипторов файловой системы. Такая запись содержит разнообразную информацию о файле: его тип, размер, количество ссылок, имя владельца, права доступа, метки времени и указатели на блоки данных этого файла.

В Linux представлен ряд журналируемых файловых систем, например, Reiserfs, ext3, ext4, XFS, JFS и Btrfs. Журналируемая файловая система заносит обновления метаданных (а в некоторых файловых системах также и обновления данных) в журнал до фактического выполнения обновлений файла. Это означает, что в случае системного сбоя можно воспользоваться информацией из файла журнала, чтобы быстро вернуть систему в согласованное состояние. Ключевым преимуществом журналируемых файловых систем является то, что они устраняют длительную проверку целостности

файловой системы, которая необходима в обычных файловых системах UNIX после системного сбоя.

Все файловые системы в Linux монтируются в единственном дереве каталогов, в корне которого располагается каталог `/`. Местоположение в дереве каталогов, где монтируется файловая система, называется точкой монтирования.

Привилегированный процесс может монтировать и размонтировать файловую систему с помощью системных вызовов `mount()` и `umount()`. Информацию о смонтированной файловой системе можно извлечь с помощью функции `statvfs()`.

Дополнительная информация

Детальные сведения об устройствах и о драйверах устройств см. в работе [Bovet & Cesati, 2005] и в особенности в [Corbet et al., 2005]. Некоторую полезную информацию об устройствах можно почерпнуть в ресурсном файле ядра `Documentation/devices.txt`.

Дополнительная информация о файловых системах содержится в нескольких книгах. Работа [Tanenbaum, 2007] является общим введением в структуру и реализацию файловых систем. Работа [Bach, 1986] представляет собой введение в реализацию файловых систем UNIX, ориентированных главным образом на версию System V. В работах [Vahalia, 1996] и [Goodheart & Cox, 1994] также описана реализация файловых систем в версии System V. Работы [Love, 2010] и [Bovet & Cesati, 2005] излагают реализацию в Linux виртуальной файловой системы.

Документацию по различным файловым системам можно найти в ресурсном подкаталоге ядра `Documentation/filesystems`. Можно также поискать отдельные сайты, описывающие большинство реализаций файловых систем, доступных в Linux.

14.13. Упражнение

14.1. Напишите программу, которая измеряет время, необходимое для создания и последующего удаления большого количества однобайтных файлов из одного каталога. Эта программа должна создавать файлы с именами в виде `xNNNNNN`, где `NNNNNN` заменяется случайным шестизначным числом. Файлы должны создаваться в случайном порядке в соответствии с генерируемыми именами, а затем их следует удалить в порядке возрастания чисел в именах (то есть в порядке, который отличен от порядка, в котором они были созданы). Количество файлов (`NF`) и каталог, в котором они должны быть созданы, следует указывать в командной строке. Измерьте время, которое требуется для различных значений `NF` (например, в диапазоне от 1000 до 20 000) и для разных файловых систем (например, для ext2, ext3 и XFS). Какую зависимость вы обнаружите в каждой файловой системе при увеличении числа `NF`? Как можно сравнить различные файловые системы? Изменятся ли результаты, если создать файлы в порядке возрастания их номеров (x000001, x000001, x0000002 и т. д.), а затем удалить их в том же порядке? Если да, то в чем может быть причина или причины? Опять-таки будут ли результаты различными для разных файловых систем?

15 Атрибуты файла

В данной главе мы рассмотрим различные атрибуты файлов (их метаданные). Начнем с описания системного вызова `stat()`, возвращающего структуру, содержащую большинство атрибутов, в число которых входят метки времени, информация о владельце и о правах доступа к файлу. Затем перейдем к рассмотрению различных системных вызовов, применяемых для изменения этих атрибутов. (Разговор о правах доступа к файлу продолжится в главе 17, где рассмотрены списки контроля доступа.) Завершим данную главу описанием флагов индексных дескрипторов (известных также как расширенные атрибуты файла в файловой системе ext2), которые управляют различными аспектами обработки файлов ядром.

15.1. Извлечение информации о файле: stat()

Системные вызовы `stat()`, `lstat()` и `fstat()` извлекают информацию о файле, в основном из индексного дескриптора файла.

```
#include <sys/stat.h>

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Все вызовы возвращают `0` при успешном завершении и `-1` при ошибке

Три этих системных вызова различаются только способом указания файла:

- вызов `stat()` возвращает информацию об именованном файле;
- вызов `lstat()` подобен вызову `stat()`, но если именованный файл является символической ссылкой, то возвращается информация о самой ссылке, а не о файле, на которую она указывает;
- вызов `fstat()` возвращает информацию о файле, к которому обращается открытый файловый дескриптор.

Системным вызовам `stat()` и `lstat()` не требуются права на доступ к самому файлу. Однако необходимо, чтобы у всех родительских каталогов, указанных в переменной `pathname`, было право на выполнение (поиск). Системный вызов `fstat()` всегда завершается успешно, если ему передан корректный дескриптор файла.

Все эти вызовы возвращают в буфер структуру `stat`, на которую указывает переменная `statbuf`. Форма данной структуры такова:

```
struct stat {
    dev_t       st_dev;       /* Идентификатор устройства,
                                 на котором находится файл */
    ino_t       st_ino;       /* Номер индексного дескриптора файла */
    mode_t      st_mode;      /* Тип файла и права доступа */
```

```
    nlink_t     st_nlink;     /* Количество (жестких) ссылок на файл */
    uid_t       st_uid;       /* Пользовательский ID владельца файла */
    gid_t       st_gid;       /* Групповой ID владельца файла */
    dev_t       st_rdev;      /* Идентификаторы файлов устройств */
    off_t       st_size;      /* Общий размер файла (в байтах) */
    blksize_t   st_blksize;   /* Оптимальный размер блока
                                 для ввода-вывода (в байтах) */
    blkcnt_t    st_blocks;    /* Количество отведенных блоков (по 512 байт) */
    time_t      st_atime;     /* Время последнего доступа к файлу */
    time_t      st_mtime;     /* Время последнего изменения файла */
    time_t      st_ctime;     /* Время последнего изменения статуса */
};
```

Различные типы данных, которые используются для представления полей в структуре `stat`, определены в стандарте SUSv3. Дополнительную информацию об этих типах см. в подразделе 3.6.2.

> Согласно стандарту SUSv3 если системный вызов lstat() применяется к символической ссылке, то он в обязательном порядке возвращает корректную информацию только в поле st_size, а также в компонент типа файла (записанный в краткой форме) поля st_mode. Остальные поля (например, поля с метками времени) не обязаны содержать корректную информацию. Это позволяет отказаться от их поддержки, чтобы повысить эффективность. В частности, целью ранних стандартов UNIX являлась возможность реализации символической ссылки либо как индексного дескриптора, либо как элемента в каталоге. В более поздних версиях невозможно реализовать все поля, которые необходимы структуре stat. (Во всех главных современных реализациях UNIX символические ссылки представлены как индексные дескрипторы.) В Linux системный вызов lstat() возвращает информацию во все поля структуры stat, когда применен к символической ссылке.

Ниже мы рассмотрим более подробно некоторые поля структуры `stat` и в завершение — пример программы, отображающей всю структуру `stat`.

Идентификаторы устройств и индексный дескриптор

Поле `st_dev` идентифицирует устройство, на котором находится файл. Поле `st_ino` содержит индексный дескриптор этого файла. Комбинация значений `st_dev` и `st_ino` уникальным образом идентифицирует файл во всех файловых системах. В типе `dev_t` записаны старший и младший номера устройства (см. раздел 14.1).

Если это индексный дескриптор устройства, то поле `st_rdev` содержит старший и младший номера данного устройства.

Старший и младший номера, содержащиеся в значении `dev_t`, можно извлечь с помощью двух макросов: `major()` и `minor()`. Заголовочный файл, который необходим для объявления этих макросов, отличается для разных реализаций UNIX. В Linux они объявляются с помощью файла `<sys/types.h>`, если определен макрос `_BSD_SOURCE`.

Величина целочисленных значений, возвращаемых макросами `major()` и `minor()`, различна для разных реализаций UNIX. Для совместимости мы всегда приводим возвращаемые значения к типу `long` при их выводе на печать (см. подраздел 3.6.2).

Принадлежность файла

Поля `st_uid` и `st_gid` идентифицируют соответственно владельца (пользовательский ID) и группу (групповой ID), которым принадлежит файл.

Счетчик ссылок

Поле `st_nlink` — это количество (жестких) ссылок на файл. Они подробно описаны в главе 13.

Тип файла и права доступа

Поле `st_mode` является битовой маской, которая служит двум целям: идентификации типа файла и указанием прав доступа к нему. Биты этого поля располагаются так, как показано на рис. 15.1.

Рис. 15.1. Битовая маска поля st_mode

Тип файла можно извлечь из данного поля с помощью операции И, задействуя константу `S_IFMT`. (В Linux для обозначения типа файла использованы четыре бита поля `st_mode`. Но, поскольку в стандарте SUSv3 не оговорено, каким образом представлять тип файла, детали могут быть различными в разных реализациях.) Затем результат можно сравнить с набором констант, чтобы определить тип файла. Например, так:

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
    printf("regular file\n");
```

Поскольку данная операция довольно обычная, для упрощения написанного выше применяется стандартный макрос:

```
if (S_ISREG(statbuf.st_mode))
    printf("regular file\n");
```

Полный набор макросов для типа файла (определенных в файле `<sys/stat.h>`) показан в табл. 15.1. Все эти макросы определены в стандарте SUSv3 и присутствуют в Linux. В некоторых реализациях UNIX определены дополнительные типы файлов (например, `S_IFDOOR` для файлов интерфейса `door` в ОС Solaris). Значение типа `S_IFLNK` возвращается только по вызовам `lstat()`, поскольку вызовы `stat()` всегда следуют по символическим ссылкам.

Исходный стандарт POSIX.1 не определяет константы, приведенные в первом столбце табл. 15.1, несмотря на то что многие из них присутствуют в большинстве реализаций UNIX. Согласно стандарту SUSv3 эти константы необходимы.

> Чтобы получить определения S_IFSOCK и S_ISSOCK() из файла <sys/stat.h>, следует либо определить проверочный макрос функции _BSD_SOURCE, либо задать для _XOPEN_SOURCE значение, превышающее или равное 500. (Эти правила немного отличаются для версий библиотеки glibc: в ряде случаев для _XOPEN_SOURCE следует задать значение, которое больше или равно 600.)

Таблица 15.1. Макросы для проверки типов файлов в поле st_mode структуры stat

Константа	**Проверочный макрос**	**Тип файла**
S_IFREG	S_ISREG()	Обычный файл
S_IFDIR	S_ISDIR()	Каталог
S_IFCHR	S_ISCHR()	Символьное устройство
S_IFBLK	S_ISBLK()	Блочное устройство
S_IFIFO	S_ISFIFO()	Очередь FIFO или канал
S_IFSOCK	S_ISSOCK()	Сокет
S_IFLNK	S_ISLNK()	Символическая ссылка

Двенадцать нижних битов поля `st_mode` определяют права доступа к файлу. Эти права мы рассмотрим в разделе 15.4. А сейчас надо отметить лишь, что девять младших значащих битов определяют права доступа на чтение, запись и выполнение для каждой из трех категорий: владельца, группы и остальных.

Размер файла, количество отведенных блоков и оптимальный размер блока для ввода-вывода

Для обычных файлов поле `st_size` — общий размер файла в байтах. Для символической ссылки это поле содержит длину (в байтах) имени пути, на который указывает ссылка. Для объекта из совместно используемой (разделяемой) памяти (см. главу 50) данное поле содержит размер такого объекта.

Поле `st_blocks` сообщает общее количество блоков, отведенных для файла. За единицу принимается блок в 512 байт. В это общее количество включено пространство, отведенное для блоков указателей (см. рис. 14.2). Выбор 512-байтного блока в качестве единицы измерения обусловлен историческими причинами — это наименьший размер блока для любой из файловых систем, которые были реализованы в UNIX. Многие современные файловые системы используют логические блоки большего размера. Так, например, для файловой системы ext2 значение `st_blocks` всегда является кратным 2, 4 или 8, в зависимости от размера логического блока файловой системы ext2 — 1024, 2048 или 4096 байтов.

> Стандарт SUSv3 не определяет единицы, в которых измеряется значение st_blocks, предоставляя возможность реализовать единицу, отличную от 512 байт. В большинстве версий UNIX применяются все же блоки по 512 байт, однако в версии HP-UX 11 используются блоки, характерные для ОС (например, размером 1024 байта в ряде случаев).

В поле `st_blocks` записано количество реально выделенных дисковых блоков. Если файл содержит дыры (см. раздел 4.7), то данное значение окажется меньше, чем можно было бы ожидать, исходя из количества байтов (`st_size`), соответствующего файлу. (Команда, отображающая информацию об использовании диска, `du -k file`, сообщает размер фактически выделенного пространства для файла в килобайтах; то есть значение, рассчитанное на основе величины `st_blocks`, а не `st_size`.)

Название поля `st_blksize` может сбить с толку. Это не размер блока базовой файловой системы, а оптимальный размер (в байтах) блока для операций ввода-вывода применительно к файлам данной файловой системы. Ввод-вывод с помощью блоков меньшего размера, чем данный, является менее эффективным (см. раздел 13.1). Типичное значение, возвращаемое константой `st_blksize`, составляет 4096.

Метки времени

Поля `st_atime`, `st_mtime` и `st_ctime` содержат соответственно время последнего доступа к файлу, время последнего изменения и время последнего изменения статуса. Эти поля имеют тип `time_t` — стандартный для UNIX формат времени в секундах, прошедших с начала «эры UNIX» — 1 января 1970 года. Подробнее о данных полях мы поговорим в разделе 15.2.

Пример программы

Программа в листинге 15.1 использует системный вызов `stat()` для извлечения информации о файле, чье имя передано в командную строку. Если указан параметр командной строки `-l`, то программа задействует системный вызов `lstat()`, чтобы мы смогли извлечь информацию о символической ссылке, а не о файле, на который она указывает. Данная программа выводит все поля возвращаемой структуры `stat`. (Объяснение того, почему

мы присваиваем полям `st_size` и `st_blocks` тип `long long`, см. в разделе 5.10.) Функция `filePermStr()`, примененная в этой программе, показана в листинге 15.4.

Приведу пример использования данной программы:

```
$ echo 'All operating systems provide services for programs they run' > apue
$ chmod g+s apue          Установка бита set-group-ID; отражается
                          на времени последнего изменения статуса
$ cat apue                Отражается на времени последнего доступа к файлу
All operating systems provide services for programs they run
$ ./t_stat apue
File type:                regular file
Device containing i-node: major=3 minor=11
I-node number:            234363
Mode:                     102644 (rw-r—r--)
    special bits set:     set-GID
Number of (hard) links:   1
Ownership:                UID=1000 GID=100
File size:                61 bytes
Optimal I/O block size:   4096 bytes
512B blocks allocated:    8
Last file access:         Mon Jun 8 09:40:07 2011
Last file modification:   Mon Jun 8 09:39:25 2011
Last status change:       Mon Jun 8 09:39:51 2011
```

Листинг 15.1. Извлечение информации о файле из структуры stat и ее интерпретация

files/t_stat.c

```
#define _BSD_SOURCE     /* Берем major() и minor() из файла <sys/types.h> */
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include "file_perms.h"
#include "tlpi_hdr.h"

static void
displayStatInfo(const struct stat *sb)
{
    printf("File type:                ");

    switch (sb->st_mode & S_IFMT) {
    case S_IFREG:  printf("regular file\n");           break;
    case S_IFDIR:  printf("directory\n");              break;
    case S_IFCHR:  printf("character device\n");       break;
    case S_IFBLK:  printf("block device\n");           break;
    case S_IFLNK:  printf("symbolic (soft) link\n");   break;
    case S_IFIFO:  printf("FIFO or pipe\n");           break;
    case S_IFSOCK: printf("socket\n");                 break;
    default:       printf("unknown file type?\n");     break;
    }

    printf("Device containing i-node: major=%ld   minor=%ld\n",
                (long) major(sb->st_dev), (long) minor(sb->st_dev));

    printf("I-node number:            %ld\n", (long) sb->st_ino);
    printf("Mode:                     %lo (%s)\n",
             (unsigned long) sb->st_mode, filePermStr(sb->st_mode, 0));
```

```
    if (sb->st_mode & (S_ISUID | S_ISGID | S_ISVTX))
        printf("  special bits set:     %s%s%s\n",
                (sb->st_mode & S_ISUID) ? "set-UID " : "",
                (sb->st_mode & S_ISGID) ? "set-GID " : "",
                (sb->st_mode & S_ISVTX) ? "sticky " : "");

    printf("Number of (hard) links:    %ld\n", (long) sb->st_nlink);

    printf("Ownership:                 UID=%ld    GID=%ld\n",
            (long) sb->st_uid, (long) sb->st_gid);

    if (S_ISCHR(sb->st_mode) || S_ISBLK(sb->st_mode))
        printf("Device number (st_rdev): major=%ld; minor=%ld\n",
                (long) major(sb->st_rdev), (long) minor(sb->st_rdev));

    printf("File size:            %lld bytes\n", (long long) sb->st_size);
    printf("Optimal I/O block size:  %ld bytes\n", (long) sb->st_blksize);
    printf("512B blocks allocated:   %lld\n", (long long) sb->st_blocks);
    printf("Last file access:        %s", ctime(&sb->st_atime));
    printf("Last file modification:  %s", ctime(&sb->st_mtime));
    printf("Last status change:      %s", ctime(&sb->st_ctime));
}

int
main(int argc, char *argv[])
{
    struct stat sb;
    Boolean statLink;     /* Истина, если указано "-l" (то есть использовать lstat) */
    int fname;            /* Место аргумента filename в массиве argv[] */

    statLink = (argc > 1) && strcmp(argv[1], "-l") == 0;
                          /* Простой синтаксический анализ для "-l" */
    fname = statLink ? 2 : 1;

    if (fname >= argc || (argc > 1 && strcmp(argv[1], "–help") == 0))
        usageErr("%s [-l] file\n"
                "        -l = use lstat() instead of stat()\n", argv[0]);

    if (statLink) {
        if (lstat(argv[fname], &sb) == -1)
            errExit("lstat");
    } else {
        if (stat(argv[fname], &sb) == -1)
            errExit("stat");
    }
    displayStatInfo(&sb);

    exit(EXIT_SUCCESS);
}
```

files/t_stat.c

15.2. Файловые метки времени

Поля `st_atime`, `st_mtime` и `st_ctime` структуры `stat` содержат файловые метки времени. В эти поля записывается соответственно время последнего доступа к файлу, время последнего изменения файла и время последнего изменения статуса файла (то есть по-

следнего изменения информации в файловом дескрипторе). Метки времени выражаются в секундах, прошедших с начала «эры UNIX» (1 января 1970 года; см. раздел 10.1).

Большинство нативных файловых систем Linux и UNIX поддерживают все поля меток времени, однако некоторые не-UNIX системы могут этого не делать.

В табл. 15.2 подытожена информация о том, какие поля меток времени (а в ряде случаев и аналогичные поля родительского каталога) меняются различными системными вызовами и библиотечными функциями, описанными в данной книге. В шапке этой таблицы буквами `a`, `m` и `c` обозначены поля `st_atime`, `st_mtime` и `st_ctime` соответственно. В большинстве случаев для соответствующей метки времени с помощью системного вызова задается значение текущего времени. Исключение составляет системный вызов `utime()` и подобные ему вызовы (рассмотренные в подразделах 15.2.1 и 15.2.2), которые можно использовать для того, чтобы явно указать произвольные значения для времени последнего доступа к файлу и времени его изменения.

Таблица 15.2. Действие различных функций на метки времени

Функция	**Файл или каталог**			**Родительский каталог**			**Примечания**
	a	**m**	**c**	**a**	**m**	**c**	
chmod()			*				То же, что и для fchmod()
chown()			*				То же, что и для lchown() или fchown()
exec()	*						
link()			*		*	*	Влияет на родительский каталог второго аргумента
mkdir()	*	*	*		*	*	
mkfifo()	*	*	*		*	*	
mknod()	*	*	*		*	*	
mmap()	*	*	*				Метки st_mtime и st_ctime изменяются только при обновлениях флага MAP_SHARED
msync()		*	*				Меняется только при изменении файла
open(), creat()	*	*	*		*	*	При создании нового файла
open(), creat()		*	*				При усечении существующего файла
pipe()	*	*	*				
read()	*						То же, что и для readv(), pread() или preadv()
readdir()	*						Функция readdir() может буферизовать записи каталога; метки времени обновляются только при чтении каталога

Продолжение ⇗

Таблица 15.2 *(продолжение)*

Функция	Файл или каталог			Родительский каталог			Примечания
	a	m	c	a	m	c	
removexattr()			*				То же, что и для fremovexattr() или lremovexattr()
rename()			*		*	*	Влияет на метки времени в обоих родительских каталогах; в стандарте SUSv3 не закреплено изменение метки st_ctime для файла, однако следует отметить, что в некоторых реализациях это происходит
rmdir()					*	*	То же, что и для remove(directory)
sendfile()	*						Изменяется метка времени для входного файла
setxattr()			*				То же, что и для fsetxattr() или lsetxattr()
symlink()	*	*	*		*	*	Устанавливает метки времени для ссылки (а не для целевого файла)
truncate()		*	*				То же, что и для ftruncate(); метки времени меняются только при изменении размера файла
unlink()			*		*	*	То же, что и для remove(file); метка времени st_ctime файла меняется, если предыдущий счетчик ссылок был > 1
utime()	*	*	*				То же, что и для utimes(), futimes(), futimens(), lutimes() или utimensat()
write()		*	*				То же, что и для writev(), pwrite() или pwritev()

В подразделе 14.8.1 и разделе 15.5 описаны параметры системного вызова `mount(2)` и пофайловые флаги, предотвращающие обновление времени последнего доступа к файлу. Флаг `O_NOATIME` вызова `open()`, описанный в разделе 4.3.1, также служит подобной цели. В ряде приложений это может помочь повысить производительность, поскольку снижает количество дисковых операций, которые необходимы при доступе к файлу.

Несмотря на то что в большинстве систем UNIX не записывается время создания файла, в новейших BSD-системах это время заносится в структуру stat в поле st_birthtime.

Наносекундные метки времени

Начиная с версии 2.6, Linux поддерживает наносекундную точность для трех полей с метками времени в структуре `stat`. Наносекундное разрешение повышает точность программ, которым необходимо принимать решения на основе относительного порядка следования меток времени файла (например, для команды `make(1)`).

В стандарте SUSv3 не предусмотрены наносекундные метки времени, эта спецификация добавлена в стандарт SUSv4.

Не все файловые системы поддерживают наносекундные метки времени. Файловые системы JFS, XFS, ext4 и Btrfs поддерживают их, а ext2, ext3 и Reiserfs — нет.

В API glibc (начиная с версии 2.3) каждое поле меток времени определяется как структура `timespec` (мы рассмотрим ее, когда будем говорить о системном вызове `utimensat()` далее в этом разделе), которая представляет время в виде секундного и наносекундного компонентов. Соответствующие макроопределения позволяют увидеть второй компонент таких структур благодаря использованию традиционных имен полей (`st_atime`, `st_mtime` и `st_ctime`). Доступ к наносекундным компонентам можно получить с помощью таких имен полей, как `st_atim.tv_nsec` (для наносекундного компонента времени последнего доступа к файлу).

15.2.1. Изменение меток времени файла с помощью системных вызовов utime() и utimes()

Метки времени последнего доступа к файлу или его изменения, хранящиеся в индексном дескрипторе файла, можно явным образом изменить с помощью системного вызова `utime()` или другого из связанного набора системных вызовов. Такие программы, как `tar(1)` и `unzip(1)`, используют эти системные вызовы для переустановки меток времени файла при распаковке архива.

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *buf);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Аргумент `pathname` идентифицирует файл, метки времени которого мы желаем изменить. Если этот аргумент является символической ссылкой, она разыменовывается. Аргумент `buf` может либо быть равен `NULL`, либо являться указателем на структуру `utimbuf`:

```
struct utimbuf {
    time_t actime;    /* Время доступа */
    time_t modtime;   /* Время изменения */
};
```

Поля в этой структуре приводят время в секундах, прошедшее с начала «эры UNIX» (см. раздел 10.1).

Характер работы системного вызова `utime()` определяется двумя различными случаями.

- Если для аргумента `buf` задано значение `NULL`, то для времени последнего доступа и для времени последнего изменения задается значение текущего времени. В данном случае либо действующий UID для процесса должен соответствовать идентификатору пользователя (владельца) для файла, либо процесс должен обладать правами на запись файла (что логично, поскольку процесс с разрешением на запись файла мог бы повлечь за собой другие системные вызовы, которые привели бы к побочному эффекту изменения этих меток времени файла), либо указанный процесс должен быть привилегированным (`CAP_FOWNER` или `CAP_DAC_OVERRIDE`). (Если точнее, то в Linux с UID файла сравнивается пользовательский идентификатор процесса в данной файловой системе, а не действующий UID, как описано в разделе 9.5.).

- Если аргумент `buf` определен как указатель на структуру `utimbuf`, то время последнего доступа и время изменения обновляются с применением значений соответствующих полей этой структуры. В таком случае действующий UID для процесса должен совпадать с UID для файла (обладать правами на запись файла здесь недостаточно), либо вызывающий процесс должен быть привилегирован (`CAP_FOWNER`).

Чтобы изменить только одну метку времени файла, следует сначала использовать системный вызов `stat()` для извлечения обеих меток, применить одно из значений времени для инициализации структуры `utimbuf`, а затем установить по желанию значение второй. Этот алгоритм продемонстрирован в приведенном ниже коде, который устанавливает для времени последнего изменения файла значение времени последнего доступа к нему:

```
struct stat sb;
struct utimbuf utb;

if (stat(pathname, &sb) == -1)
    errExit("stat");
utb.actime = sb.st_atime;        /* Оставить время доступа без изменений */
utb.modtime = sb.st_atime;
if (utime(pathname, &utb) == -1)
    errExit("utime");
```

При успешном завершении вызова `utime()` для времени последнего изменения статуса всегда устанавливается значение текущего времени.

В Linux есть также заимствованный из системы BSD системный вызов `utimes()`, выполняющий задачу, сходную с задачей `utime()`.

```
#include <sys/time.h>

int utimes(const char *pathname, const struct timeval tv[2]);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Самым заметным различием между системными вызовами `utime()` и `utimes()` является то, что второй позволяет указывать значения времени с микросекундной точностью (структура `timeval` описана в разделе 10.1). Это обеспечивает (частичный) доступ к наносекундной точности, которую имеют метки времени в Linux 2.6. Новое время доступа к файлу указывается в поле `tv[0]`, а новое время изменения файла — в `tv[1]`.

Пример использования системного вызова utimes() есть в файле files/t_utimes.c, представленном в исходном коде к данной книге.

Библиотечные функции `futimes()` и `lutimes()` работают подобно системному вызову `utimes()`. Они отличаются от него аргументом, служащим для указания файла, метки времени которого следует изменить.

```
#include <sys/time.h>

int futimes(int fd, const struct timeval tv[2]);
int lutimes(const char *pathname, const struct timeval tv[2]);
```

Оба вызова возвращают `0` при успешном завершении и `-1` при ошибке

Для функции `futimes()` файл указывается через открытый файловый дескриптор, `fd`.

Для функции `lutimes()` файл указывается через путь, с тем отличием от `utimes()`, что если данный путь оказывается символической ссылкой, то она не разыменовывается; вместо этого меняются метки времени самой ссылки.

Функция `futimes()` поддерживается, начиная с версии 2.3 библиотеки glibc, функция `lutimes()` — начиная с версии 2.6.

15.2.2. Изменение меток времени файла с помощью системного вызова utimensat() и функции futimens()

Системный вызов `utimensat()` (поддерживается, начиная с версии ядра 2.6.22) и библиотечная функция `futimens()` (поддерживается с версии glibc 2.6) позволяют в расширенном диапазоне задавать метки времени последнего доступа к файлу или времени его последнего изменения. К числу преимуществ данных интерфейсов относятся следующие.

- Можно задавать метки времени с наносекундной точностью. Это лучше микросекундной точности, которая обеспечивается системным вызовом `utimes()`.
- Есть возможность независимого задания меток времени (то есть по одной). Как было показано ранее, для изменения только одной метки времени с помощью старых интерфейсов необходимо сначала выполнить системный вызов `stat()`, чтобы извлечь значение другой метки времени, а затем указать извлеченное значение вместе с меткой времени, чье значение следует изменить. (Это может привести к состоянию соперничества, если какой-либо другой процесс выполнил операцию, которая обновила метку времени, вклинившись между этими двумя шагами.)
- Можно независимо указывать для любой метки времени значение текущего времени. Чтобы изменить текущее время только для одной метки с помощью старых интерфейсов, необходимо задействовать системный вызов `stat()` для извлечения информации о метке времени, значение которой следует оставить без изменений, а также функцию `gettimeofday()` для получения текущего времени.

Эти интерфейсы не описаны в стандарте SUSv3, но включены в стандарт SUSv4.

Системный вызов `utimensat()` обновляет метки времени файла, указанного в аргументе `pathname`, присваивая им значения, передаваемые в массиве `times`.

```
#define _XOPEN_SOURCE 700      /* Или define _POSIX_C_SOURCE >= 200809 */
#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname,
              const struct timespec times[2], int flags);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Если для аргумента `times` указано значение `NULL`, обе метки времени файла обновляются, принимая значение текущего времени. Если аргумент `times` не равен `NULL`, то новая метка времени последнего доступа указывается в элементе `times[0]`, а новая метка времени последнего изменения — в элементе `times[1]`. Каждый элемент массива `times` является структурой следующего вида:

```
struct timespec {
    time_t tv_sec;      /* Секунды ('time_t' является целочисленным типом) */
    long tv_nsec;       /* Наносекунды */
};
```

Поля в этой структуре указывают время в секундах и наносекундах, прошедших прошедших с начала «эры UNIX» (см. раздел 10.1).

Чтобы задать для одной метки времени текущее время, следует передать специальное значение `UTIME_NOW` в соответствующее поле `tv_nsec`. Если же нужно оставить одну из меток времени без изменений, то специальное значение `UTIME_OMIT` требуется передать в соответствующее поле `tv_nsec`. В обоих случаях игнорируется значение в соответствующем поле `tv_sec`.

В качестве аргумента `dirfd` можно либо передать значение `AT_FDCWD`, и тогда аргумент `pathname` будет интерпретирован так же, как и для системного вызова `utimes()`, либо передать файловый дескриптор, указывающий на каталог. Назначение второго варианта описано в разделе 18.11.

Аргумент `flags` может быть равен либо 0, либо `AT_SYMLINK_NOFOLLOW`. Последнее означает, что аргумент `pathname` не следует разыменовывать, если он является символической ссылкой (то есть метки времени самой символической ссылки не следует менять). В противоположность этому системный вызов `utimes()` всегда разыменовывает символические ссылки.

В приведенном ниже фрагменте кода для времени последнего доступа устанавливается значение текущего времени, а время последнего изменения остается без изменений:

```
struct timespec times[2];

times[0].tv_sec = 0;
times[0].tv_nsec = UTIME_NOW;
times[1].tv_sec = 0;
times[1].tv_nsec = UTIME_OMIT;
if (utimensat(AT_FDCWD, "myfile", times, 0) == -1)
    errExit("utimensat");
```

Права доступа при изменении меток времени с помощью системного вызова `utimensat()` (и функции `futimens()`) подобны тем, которые используются в старых API, и подробно описаны на странице `utimensat(2)` руководства.

Библиотечная функция `futimens()` обновляет метки времени файла, на который ссылается дескриптор открытого файла `fd`.

```
#include _GNU_SOURCE
#include <sys/stat.h>

int futimens(int fd, const struct timespec times[2]);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Аргумент `times` функции `futimens()` применяется так же, как и в системном вызове `utimensat()`.

15.3. Принадлежность файла

С каждым файлом связаны идентификатор пользователя (UID) и идентификатор группы (GID). Эти идентификаторы определяют, какому пользователю и какой группе принадлежит файл. Сейчас мы рассмотрим правила, которые определяют принадлежность новых файлов, а также опишем системные вызовы, используемые для изменения принадлежности файла.

15.3.1. Принадлежность новых файлов

При создании нового файла его идентификатор пользователя заимствуется от действующего ID пользователя для процесса. Идентификатор группы для нового файла может быть взят либо от действующего идентификатора группы для процесса (эквивалент принятого по умолчанию поведения версии System V), либо от идентификатора группы для родительского каталога (поведение BSD). Последний вариант удобен для создания каталогов проектов, в которых все файлы принадлежат какой-либо группе и доступны для ее участников. Какое из этих двух значений используется в качестве идентификатора группы для нового файла — зависит от различных факторов. В их число входит тип файловой системы, в которой создается новый файл. Начнем с правил, принятых для ext2 и для ряда других систем.

Следует уточнить, что для Linux все случаи употребления терминов «действующий идентификатор пользователя» или «действующий идентификатор группы» в данном разделе фактически относятся к идентификаторам пользователя или группы для файловой системы (см. раздел 9.5).

При монтировании файловой системы ext2 можно указать для команды `mount` один из параметров: `-o grpid` (или его синоним `-o bsdgroups`) или `-o nogrpid` (или его синоним `-o sysvgroups`). (В противном случае по умолчанию принимается `-o nogrpid`.) Если указан `-o grpid`, то новый файл всегда наследует идентификатор группы от родительского каталога. Если `-o ogrpid`, то по умолчанию новый файл заимствует идентификатор группы от действующего идентификатора группы для процесса. Тем не менее если для каталога установлен бит set-group-ID (с помощью команды `chmod g+s`), то идентификатор группы для данного файла наследуется от родительского каталога. Эти правила подытожены в табл. 15.3.

В разделе 18.6 будет показано, что, когда бит set-group-ID установлен для каталога, он устанавливается и для новых подкаталогов, создаваемых внутри данного. Таким же образом поведение set-group-ID, описанное в основном тексте, распространяется на все дерево каталогов.

Таблица 15.3. Правила, определяющие принадлежность к группе для созданного нового файла

Параметр монтирования файловой системы	**Установлен ли бит set-group-ID для родительского каталога?**	**Принадлежность к группе для нового файла наследуется от**
-o grpid -o bsdgroups	(Игнорируется)	Идентификатора группы для родительского каталога
-o nogrpid -o sysvgroups (по умолчанию)	Нет	Действующего идентификатора группы для процесса
	Да	Идентификатора группы для родительского каталога

На момент написания книги файловыми системами, которые поддерживают параметры монтирования `grpid` и `nogrpid`, являются ext2, ext3, ext4 и (с версии Linux 2.6.14) XFS. Другие файловые системы следуют правилам `nogrpid`.

15.3.2. Изменение принадлежности файла: системные вызовы chown(), fchown() и lchown()

Системные вызовы `chown()`, `lchown()` и `fchown()` изменяют владельца (идентификатор пользователя) и группу (идентификатор группы) файла.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

#define _XOPEN_SOURCE 500        /* Или: #define _BSD_SOURCE */
#include <unistd.h>

int lchown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Все вызовы возвращают `0` при успешном завершении и `-1` при ошибке

Отличия между этими тремя системными вызовами похожи на те, что существуют в семействе системных вызовов `stat()`:

- системный вызов `chown()` изменяет принадлежность файла, указанного в аргументе `pathname`;
- системный вызов `lchown()` делает то же, но если аргумент `pathname` оказывается символической ссылкой, меняется принадлежность ссылочного файла, а не того файла, на который указывает ссылка;
- системный вызов `fchown()` изменяет принадлежность файла, на который ссылается открытый файловый дескриптор, `fd`.

Для файла аргумент `owner` задает новый UID, а аргумент `group` — новый GID. Изменить лишь один из этих идентификаторов можно так: указать значение `-1` для другого аргумента, чтобы оставить его без изменений.

> До версии Linux 2.2 системный вызов chown() не разыменовывал символические ссылки. Начиная с Linux 2.2, семантика системного вызова chown() изменилась, и был добавлен новый системный вызов lchown(), чтобы обеспечить поведение старого системного вызова chown().

Только привилегированный процесс (`CAP_CHOWN`) может использовать системный вызов `chown()` для изменения идентификатора пользователя файла. Непривилегированный процесс может задействовать системный вызов `chown()` для изменения GID для файла, которым он обладает (то есть действующий UID для процесса совпадает с UID этого файла) для любой из групп, членом которой он является. Привилегированный процесс может изменить идентификатор группы файла на любое значение.

Если владелец файла или группа изменились, то тогда снимаются биты прав доступа set-user-ID и set-group-ID. Это мера предосторожности, которая гарантирует, что обычный пользователь не сможет установить бит set-user-ID (или set-group-ID) для исполняемого файла, чтобы затем каким-либо образом сделать данный файл принадлежащим привилегированному пользователю (или группе) и тем самым задействовать этот привилегированный экземпляр при выполнении файла.

> В стандарте SUSv3 не оговорено, следует ли снимать биты set-user-ID и set-group-ID, когда суперпользователь изменяет владельца или группу для исполняемого файла. В Linux 2.0 эти биты снимались в подобном случае, а в некоторых ранних версиях ядра 2.2 (до 2.2.12) — нет. Более поздние версии ядра 2.2 вернулись к поведению версии 2.0, при котором изменения, выполняемые суперпользователем, трактуются подобно действиям любого пользователя. Такой вариант поддерживается в последующих версиях ядра. (Однако если для изменения принадлежности файла мы используем команду chown(1) под корневой учетной записью, то после вызова chown(2) команда chown задействует системный вызов chmod(), чтобы заново установить биты set-user-ID и set-group-ID.)

При изменении владельца или группы для файла бит прав доступа set-group-ID не снимается, если бит разрешения на исполнение для группы уже снят или если принадлежность каталога меняется. В обоих случаях бит set-group-ID используется с целью, которая отличается от создания программы для работы с битом set-group-ID, и поэтому его нежелательно отключать. К таким вариантам применения бита set-group-ID относятся следующие:

- если бит разрешения на исполнение для группы снят, то тогда бит прав доступа set-group-ID применяется для включения принудительной блокировки файла (рассмотрена в разделе 51.4);
- в случае с каталогом бит set-group-ID служит для управления принадлежностью новых файлов, создаваемых в этом каталоге (см. подраздел 15.3.1).

Использование команды `chown()` продемонстрировано в листинге 15.2. Эта программа позволяет пользователю изменять владельца и группу для произвольного количества файлов, передаваемых в виде аргументов командной строки. (Данная программа задействует функции `userIdFromName()` и `groupIdFromName()` из листинга 8.1 для конвертирования имен пользователя и группы в соответствующие числовые идентификаторы.)

Листинг 15.2. Изменение владельца и группы для файла

files/t_chown.c

```
#include <pwd.h>
#include <grp.h>
#include "ugid_functions.h"          /* Объявление функций userIdFromName()
                                        и groupIdFromName() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    uid_t uid;
    gid_t gid;
    int j;
    Boolean errFnd;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s owner group [file...]\n"
                 "        owner or group can be '-', "
                 "meaning leave unchanged\n", argv[0]);

    if (strcmp(argv[1], "-") == 0) {      /* "-" ==> не менять владельца */
        uid = -1;
    } else {                      /* Преобразовать имя пользователя в UID */
        uid = userIdFromName(argv[1]);
        if (uid == -1)
            fatal("No such user (%s)", argv[1]);
    }

    if (strcmp(argv[2], "-") == 0) {      /* "-" ==> не менять группу */
        gid = -1;
    } else {                      /* Преобразовать имя группы в GID */
        gid = groupIdFromName(argv[2]);
        if (gid == -1)
            fatal("No group user (%s)", argv[1]);
    }

    /* Изменить принадлежность всех файлов, указанных в остальных аргументах */
```

```
    errFnd = FALSE;
    for (j = 3; j < argc; j++) {
        if (chown(argv[j], uid, gid) == -1) {
            errMsg("chown: %s", argv[j]);
            errFnd = TRUE;
        }
    }

    exit(errFnd ? EXIT_FAILURE : EXIT_SUCCESS);
}
```

files/t_chown.c

15.4. Права доступа к файлу

В этом разделе мы опишем схему прав доступа, которая применяется к файлам и каталогам. Несмотря на то что мы говорим здесь о правах доступа главным образом в отношении обычных файлов и каталогов, описываемые правила применимы для всех типов файлов, включая устройства, очереди FIFO и сокеты доменов UNIX. Более того, объекты межпроцессного взаимодействия в системах System V и POSIX (совместно используемая память, семафоры и очереди сообщений) также имеют маски прав доступа, а правила, применяемые для этих объектов, подобны правилам для файлов.

15.4.1. Права доступа к обычным файлам

Как отмечалось в разделе 15.1, нижние 12 битов поля `st_mode` структуры `stat` задают права доступа для файла. Первые три из этих битов являются специальными и называются set-user-ID, set-group-ID и бит закрепления (закрепляющий бит) (на рис. 15.1 они обозначены буквами U, G и T). Подробнее о них мы поговорим в подразделе 15.4.5. Остальные девять битов формируют маску, определяющую права доступа, которые предоставляются различным категориям пользователей, получающих доступ к файлу. Маска прав доступа к файлу разделяет объекты на три категории.

- *Владелец* (известный также как *пользователь*). Такие права доступа предоставлены владельцу данного файла.

  Термин «пользователь» используется такими командами, как chmod(1), которые обозначают буквой u эту категорию прав доступа.

- *Группа*. Такие права доступа предоставляются пользователям, входящим в группу файла.
- *Остальные*. Права доступа, предоставляемые всем остальным пользователям.

Каждой категории пользователей могут быть предоставлены следующие три права доступа:

- *чтение:* содержимое файла можно читать;
- *запись:* содержимое файла можно изменять;
- *выполнение:* данный файл можно выполнить (то есть это программа или сценарий). Для запуска файла сценария (например, bash) необходимо наличие прав на чтение и выполнение.

Права доступа и принадлежность файла можно просмотреть с помощью команды `ls -l`, как показано в следующем примере:

```
$ ls -l myscript.sh
-rwxr-x---    1 mtk      users          1667 Jan 15 09:22 myscript.sh
```

Права доступа к файлу отображаются как `rwxr-x---` (дефис, с которого начинается эта строка, сообщает тип файла: обычный файл). Для интерпретации данной строки следует разбить эти девять символов на блоки по три символа, которые будут указывать на предоставленные права доступа: чтение, запись или выполнение. Первый блок сообщает о правах доступа для владельца; ему разрешены чтение, запись и выполнение. Следующий блок сообщает о правах доступа для группы: разрешены чтение и выполнение, но не запись. Последний блок сообщает права доступа для остальных пользователей: им не предоставлено никаких прав.

Заголовочный файл `<sys/stat.h>` определяет константы, которые с помощью операции И (`&`) можно объединить со значением поля `st_mode` структуры `stat`, чтобы проверить, какие именно биты прав доступа установлены. (Эти константы определены также благодаря подключению файла `<fcntl.h>`, прототипирующий системный вызов `open()`.) Данные константы приведены в табл. 15.4.

Помимо показанных в табл. 15.4, определены три константы, чтобы уравнять маски всех трех прав доступа для каждой категории — владельца, группы и остальных: `S_IRWXU` (0700), `S_IRWXG` (070) и `S_IRWXO` (07).

Таблица 15.4. Константы для битов прав доступа к файлу

Константа	Восьмеричное значение	Бит прав доступа
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Закрепляющий
S_IRUSR	0400	Пользователь: чтение
S_IWUSR	0200	Пользователь: запись
S_IXUSR	0100	Пользователь: выполнение
S_IRGRP	040	Группа: чтение
S_IWGRP	020	Группа: запись
S_IXGRP	010	Группа: выполнение
S_IROTH	04	Остальные: чтение
S_IWOTH	02	Остальные: запись
S_IXOTH	01	Остальные: выполнение

Заголовочный файл в листинге 15.3 объявляет функцию `filePermStr()`, которая после принятия маски прав доступа к файлу возвращает статически размещенное строковое представление этой маски в таком же стиле, какой используется командой `ls(1)`.

Листинг 15.3. Заголовочный файл для file_perms.c

files/file_perms.h

```
#ifndef FILE_PERMS_H
#define FILE_PERMS_H
#include <sys/types.h>
#define FP_SPECIAL 1    /* Включить в возвращаемую строку информацию о битах
                           set-user-ID, set-group-ID и закрепляющем */

char *filePermStr(mode_t perm, int flags);
#endif
```

files/file_perms.h

Если флаг FP_SPECIAL установлен в качестве аргумента filePermStr() flags, то возвращаемая строка содержит параметры битов set-user-ID, set-group-ID и закрепляющего опять-таки в стиле команды ls(1).

Реализация функции filePermStr() приведена в листинге 15.4. Мы применяем эту функцию в программе из листинга 15.1.

Листинг 15.4. Преобразование маски прав доступа к файлу в строку

files/file_perms.c

```
#include <sys/stat.h>
#include <stdio.h>
#include "file_perms.h"          /* Интерфейс для данной реализации */

#define STR_SIZE sizeof("rwxrwxrwx")

char *          /* Возвращает вместо маски прав доступа к файлу
                   строку в стиле ls(1) */
filePermStr(mode_t perm, int flags)
{
    static char str[STR_SIZE];

    snprintf(str, STR_SIZE, "%c%c%c%c%c%c%c%c%c",
         (perm & S_IRUSR) ? 'r' : '-', (perm & S_IWUSR) ? 'w' : '-',
         (perm & S_IXUSR) ?
             (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
             (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
         (perm & S_IRGRP) ? 'r' : '-', (perm & S_IWGRP) ? 'w' : '-',
         (perm & S_IXGRP) ?
             (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
             (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
         (perm & S_IROTH) ? 'r' : '-', (perm & S_IWOTH) ? 'w' : '-',
         (perm & S_IXOTH) ?
             (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 't' : 'x') :
             (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 'T' : '-'));

    return str;
}
```

files/file_perms.c

15.4.2. Права доступа к каталогам

Для каталогов применяется та же схема прав доступа, что и для файлов. Однако три варианта прав доступа интерпретируются иначе.

- *Чтение.* Содержимое (то есть список имен файлов) каталога можно вывести (например, с помощью команды ls).

  Экспериментируя с проверкой поведения бита разрешения на чтение каталога, имейте в виду, что в ряде версий Linux создается псевдоним команды ls, включающий флаги (например, –F), которому необходим доступ к информации индексных дескрипторов файлов в данном каталоге, а для этого требуется разрешение на выполнение применительно к каталогу. Для гарантии использования «чистой» команды ls можно указать полный путь для нее (/bin/ls).

- *Запись.* В данном каталоге можно создавать файлы или удалять их из него. Обратите внимание: в этом случае не обязательно иметь какие-либо права доступа к файлу, чтобы удалить его.

- *Выполнение*. К файлам в каталоге разрешен доступ. Разрешение на выполнение применительно к каталогу иногда называют разрешением на *поиск*.

При доступе к файлу разрешение на выполнение необходимо для всех каталогов, которые содержатся в имени пути. Так, например, для чтения файла `/home/mtk/x` потребовалось бы разрешение на выполнение каталогов `/`, `/home` и `/home/mtk` (а также разрешение на чтение самого файла `x`). Если текущим рабочим каталогом является `/home/mtk/sub1` а мы осуществляем доступ по относительному имени пути `../sub2/x`, то в этом случае необходимо иметь разрешение на выполнение для каталогов `/home/mtk` и `/home/mtk/sub2` (но не для каталога `/` или `/home`).

Право доступа на чтение каталога позволяет лишь увидеть список имен файлов в этом каталоге. Следует иметь разрешение на выполнение для каталога, чтобы получить доступ к его содержимому или к информации индексных дескрипторов файлов в данном каталоге.

И наоборот, при наличии разрешения на выполнение для каталога, но отсутствии права доступа на чтение имеется доступ к файлу в этом каталоге, если известно имя файла, однако нельзя вывести содержимое (то есть имена других файлов) данного каталога. Это простой и часто используемый метод контроля доступа к содержимому общедоступного каталога.

Чтобы добавлять файлы в каталоге или удалять их, необходимо иметь разрешение на выполнение и запись для данного каталога.

15.4.3. Алгоритм проверки прав доступа

Ядро проверяет права доступа к файлу всякий раз при указании имени пути в системном вызове, который осуществляет доступ к файлу или к каталогу. Если имя пути, переданного системному вызову, содержит префикс каталога, то, помимо проверки необходимых прав доступа к самому файлу, ядро проверяет также разрешение на выполнение для каждого каталога в таком префиксе. Проверки прав доступа выполняются благодаря использованию действующих идентификаторов пользователя, группы и добавочной группы для процесса. (Если быть абсолютно точным, то для проверки прав доступа к файлу в Linux вместо соответствующих действующих идентификаторов задействуются идентификаторы пользователя и группы для данной файловой системы, как описано в разделе 9.5.)

> Как только файл открывается с помощью системного вызова open(), последующие системные вызовы (такие как read(), write(), fstat(), fcntl() и mmap()), которые работают с возвращенным дескриптором файла, не выполняют проверку прав доступа.

Правила, применяющие ядро при проверке прав доступа, выглядят так.

1. Если процесс привилегирован, предоставляется полный доступ.
2. Если действующий UID для процесса совпадает с идентификатором пользователя (владельца) файла, то предоставляется доступ в соответствии с правами доступа *владельца* файла. Например, право доступа на чтение предоставляется, если в маске прав доступа к файлу установлен бит разрешения на чтение для владельца; в противном случае такое разрешение не предоставляется.
3. Если действующий GID или идентификатор любой добавочной группы для процесса совпадает с идентификатором группы (владельца группы) для файла, то доступ предоставляется в соответствии с правами доступа *группы* для данного файла.
4. В остальных случаях доступ к файлу предоставляется в соответствии с правами доступа для *остальных*.

> В программном коде ядра перечисленные выше проверки сконструированы таким образом, чтобы проверка привилегированности процесса выполнялась только в том случае, если процессу не предоставлены необходимые права доступа в результате какой-либо другой проверки. Это

сделано во избежание излишней установки флага учета процессов ASU, который указывает на то, что данный процесс воспользовался привилегиями суперпользователя (см. раздел 28.1).

Проверки прав доступа для владельца, группы и остальных выполняются по порядку и прекращаются, как только обнаруживается применимое правило. Это может привести к неожиданным последствиям: если, например, права доступа для группы превышают права владельца, то последний будет фактически иметь меньше прав доступа к файлу, чем участники группы, как показано в следующем примере:

```
$ echo 'Hello world' > a.txt
$ ls -l a.txt
-rw-r--r--   1 mtk    users      12 Jun 18 12:26 a.txt
$ chmod u-rw a.txt              Лишаем владельца прав на чтение и запись
$ ls -l a.txt
----r--r--   1 mtk    users      12 Jun 18 12:26 a.txt
$ cat a.txt
cat: a.txt: Permission denied   Владелец больше не может читать файл
$ su avr                        Становимся кем-либо…
Password:
$ groups                        …кто входит в группу, владеющую этим файлом…
users staff teach cs
$ cat a.txt                     …и поэтому может его читать
Hello world
```

Подобные замечания применимы, если предоставлено больше прав доступа для остальных, чем для владельца или группы.

Поскольку информация о правах доступа к файлу и о его принадлежности содержится в индексном дескрипторе файла, все имена файлов (ссылки), которые указывают на один и тот же индексный дескриптор, будут совместно использовать эту информацию.

В Linux 2.6 реализованы списки контроля доступа (ACLs), позволяющие задавать права доступа к файлу для отдельного пользователя или группы. Если файл имеет такой список, то применяется измененная версия алгоритма, приведенного выше. Данные списки будут описаны в главе 17.

Проверка прав доступа для привилегированных процессов

Выше было сказано, что если процесс является привилегированным, то при проверке прав доступа ему предоставляется полный доступ. Необходимо добавить одну оговорку к этому утверждению. В случае с файлом, который не является каталогом, Linux предоставляет разрешение на выполнение для привилегированного процесса, только если такое разрешение предоставлено по меньшей мере одной категории прав доступа для данного файла. В ряде других реализаций UNIX привилегированный процесс может выполнять файл, даже если никакой из категорий не предоставлено разрешение на выполнение. При доступе к каталогу привилегированному процессу всегда предоставляется разрешение на выполнение (поиск).

Можно перефразировать наше описание привилегированного процесса в терминах двух возможностей процесса Linux: CAP_DAC_READ_SEARCH и CAP_DAC_OVERRIDE (см. раздел 39.2). Процесс с возможностью CAP_DAC_READ_SEARCH всегда обладает разрешением на чтение любого типа файла, а также всегда имеет права доступа на чтение и выполнение для каталога (то есть всегда может иметь доступ к файлам в каталоге и читать список файлов в каталоге). Процесс с возможностью CAP_DAC_OVERRIDE всегда обладает разрешением на чтение и запись любого типа файла, а также обладает разрешением на выполнение, если файл является каталогом или если разрешение на выполнение предоставлено по меньшей мере одной категории прав доступа для данного файла.

15.4.4. Проверка доступности файла: системный вызов access()

Как отмечалось в разделе 15.4.3, *действующие* идентификаторы пользователя и группы, а также добавочной группы используются для определения прав доступа, которыми обладает процесс при доступе к файлу. У программы (работающей, например, с полномочиями setuid и setgid) есть возможность проверить доступность файла на основе *реальных* идентификаторов пользователя и группы для процесса.

Системный вызов `access()` проверяет доступность файла, указанного в аргументе `pathname`, на основе реальных идентификаторов пользователя и группы (а также идентификаторов добавочных групп) для процесса.

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Возвращает `0`, если предоставлены все права доступа, и `-1` в противном случае

Если аргумент `pathname` является символической ссылкой, системный вызов `access()` разыменовывает ее.

Аргумент `mode` является битовой маской, состоящей из одной или из нескольких констант, приведенных в табл. 15.5, которые объединены с помощью операции ИЛИ (`|`). Если все права доступа, указанные в данном аргументе, предоставлены файлу с именем пути `pathname`, то системный вызов `access()` возвращает `0`; если недоступно хотя бы одно из запрашиваемых прав доступа (или если возникла ошибка), то системный вызов `access()` возвращает `-1`.

Таблица 15.5. Константы mode для системного вызова access()

Константа	Описание
F_OK	Существует ли файл?
R_OK	Можно ли читать файл?
W_OK	Можно ли записывать файл?
X_OK	Можно ли выполнять файл?

Наличие временно́го интервала между системным вызовом `access()` и последующей операцией над файлом означает следующее: нет никакой гарантии того, что информация, возвращенная системным вызовом `access()`, останется истинной к моменту выполнения операции (вне зависимости от того, насколько краток этот интервал). Такая ситуация может привести к возникновению брешей в системе безопасности ряда приложений.

Допустим, к примеру, что у нас есть команда установки бита set-user-ID-root, которая применяет системный вызов `access()` для проверки доступности файла программой, использующей реальный идентификатор пользователя и выполняющей операцию над файлом (например, `open()` или `exec()`), если он доступен.

Проблема заключается вот в чем: если передаваемое системному вызову `access()` имя пути является символической ссылкой, а злоумышленнику удается до начала второго этапа изменить данную ссылку так, чтобы она указывала на другой файл, то это может привести к тому, что команда set-user-ID-root будет работать с файлом, у которого нет

прав доступа для реального идентификатора пользователя. (Это пример состояния соперничества, возникающего на основе значений времени проверки и времени использования, как сказано в разделе 38.6.) Исходя из вышесказанного рекомендуется всецело избегать применения системного вызова `access()` (см., например, работу [Borisov, 2005]). В только что приведенном примере мы можем осуществить это, временно изменив действующий (или относящийся к файловой системе) идентификатор пользователя для процесса set-user-ID, пытающегося выполнить желаемую операцию (например, `open()` или `exec()`), а затем проверить возвращенное значение и параметр `errno`, чтобы установить, не была ли связана ошибка выполнения операции с нарушением прав доступа.

GNU-библиотека C предлагает аналогичную нестандартную функцию euidaccess() (синонимичное название — eaccess()), которая проверяет права доступа к файлу с помощью действующего идентификатора пользователя для процесса.

15.4.5. Биты set-user-ID, set-group-ID и закрепляющий

Помимо девяти битов, служащих для указания прав доступа владельца, группы и остальных, маска прав доступа содержит три дополнительных бита, называемых *set-user-ID* (бит 04000), *set-group-ID* (бит 02000) и *закрепляющий* (бит 01000). Мы уже говорили в разделе 9.3 о применении первых двух битов для создания привилегированных программ. Бит set-group-ID служит еще двум целям, которые мы описываем в другом месте: управлению принадлежностью к группе для новых файлов, создаваемых в каталоге, смонтированной с параметром `nogrpid` (подраздел 15.3.1), а также осуществлению принудительной блокировки файла (раздел 51.4). Здесь же мы ограничимся разговором об использовании бита закрепления.

В ранних реализациях UNIX данный бит служил как средство более быстрого выполнения часто применяемых программ. Если он был установлен для файла программы, то при первом запуске копия ее текста сохранялась в области подкачки — закреплялась в ней и загружалась быстрее при последующих выполнениях. В современных реализациях UNIX системы управления памятью более сложные, и поэтому приведенный вариант использования закрепляющего бита устарел.

Имя константы для бита закрепления, которое приведено в табл. 15.4, S_ISVTX, происходит от его альтернативного названия «бит сохраненного текста» (saved-text).

В современных реализациях UNIX (включая также Linux) бит закрепления служит совершенно другой цели. Для каталогов он действует как флаг *запрещения удаления*. Если бит установлен для каталога, то непривилегированный процесс может расцеплять (`unlink()`, `rmdir()`) и переименовывать (`rename()`) файлы в данном каталоге, только если у него есть право записи для каталога *и* он является владельцем либо файла, либо каталога. (Процесс с возможностью `CAP_FOWNER` может обходиться без последней проверки принадлежности.) Это позволяет создать каталог, который задействуют одновременно несколько пользователей. Каждый из них может создавать и удалять собственные файлы в данном каталоге, но не может удалять файлы, принадлежащие другим пользователям. По данной причине бит закрепления обычно установлен для каталога `/tmp`.

Закрепляющий бит для файла устанавливается с помощью команды `chmod` (`chmod +t file`) или системного вызова `chmod()`. Если бит закрепления задан для файла, то команда `ls -l` выведет в поле прав доступа на выполнение для остальных пользователей строчную или прописную букву T, в зависимости от того, установлен этот бит или нет, как показано ниже:

```
$ touch tfile
$ ls -l tfile
-rw-r--r--   1 mtk            users            0 Jun 23 14:44 tfile
$ chmod +t tfile
$ ls -l tfile
-rw-r--r-T   1 mtk            users            0 Jun 23 14:44 tfile
$ chmod o+x tfile
$ ls -l tfile
-rw-r--r-t   1 mtk            users            0 Jun 23 14:44 tfile
```

15.4.6. Маска режима создания файла процесса: umask()

Теперь рассмотрим более подробно права доступа, которые назначаются новому файлу или каталогу. Для новых файлов ядро использует права, указанные в аргументе `mode` системного вызова `open()` или `creat()`. Для новых каталогов эти права устанавливаются в соответствии с аргументом `mode` команды `mkdir()`. Однако указанные параметры изменяются с помощью маски режима создания файла, которая известна как `umask`. Этот параметр — атрибут процесса, указывающего, какой из битов прав доступа следует всегда *отключать* при создании данным процессом новых файлов или каталогов.

Зачастую процесс задействует атрибут `umask`, который он наследует от своей родительской оболочки. Следствием этого (как правило, желательным) является то, что пользователь может управлять данным атрибутом в программах, выполняемых из оболочки, используя встроенную в оболочку одноименную команду, изменяющую атрибут `umask` для процесса оболочки.

Файлы инициализации в большинстве оболочек по умолчанию устанавливают для атрибута `umask` восьмеричное значение `022 (----w--w-)`. Оно указывает на то, что право на запись должно быть всегда отключено для группы и для остальных пользователей. Следовательно, если учесть, что аргумент `mode` системного вызова `open()` равен `0666` (то есть чтение и запись разрешены для всех пользователей, что типично), то новые файлы создаются с правами доступа на чтение и запись для владельца, а для всех остальных — только с правом доступа на чтение (команда `ls -l` покажет это как `rw-r--r--`). Подобным же образом, если учесть, что для аргумента `mode` системного вызова `mkdir()` установлено значение `0777` (то есть все права доступа предоставлены всем пользователям), новые каталоги создаются с предоставлением всех прав доступа владельцу, а группам и остальным пользователям предоставляются только права доступа на чтение и выполнение (`rwxr-xr-x`).

Системный вызов `umask()` изменяет атрибут `umask` для процесса на значение, указанное в аргументе `mask`.

```
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

Всегда успешно возвращает параметр `umask` предыдущего процесса

Аргумент `mask` можно указывать либо как восьмеричное число, либо в виде строки, объединяющей с помощью операции ИЛИ (`|`) константы, приведенные в табл. 15.4.

Вызов `umask()` всегда завершается успешно и возвращает предыдущее значение параметра `umask`.

Листинг 15.5 иллюстрирует применение системного вызова `umask()` в сочетании с вызовами `open()` и `mkdir()`. При запуске данной программы мы увидим следующее.

```
$ ./t_umask
Requested file perms: rw-rw----     Это то, что мы запросили
Process umask:        ----wx-wx     Это то, что мы отклонили
Actual file perms:    rw-r-----     В результате вышло так

Requested dir. perms: rwxrwxrwx
Process umask:        ----wx-wx
Actual dir. perms:    rwxr--r--
```

В листинге 15.5 мы задействуем системные вызовы mkdir() и rmdir() для создания и удаления каталога, а также системный вызов unlink() для удаления файла. Эти системные вызовы описаны в главе 18.

Листинг 15.5. Использование системного вызова umask()

files/t_umask.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "file_perms.h"
#include "tlpi_hdr.h"

#define MYFILE    "myfile"
#define MYDIR        "mydir"
#define FILE_PERMS     (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
#define DIR_PERMS      (S_IRWXU | S_IRWXG | S_IRWXO)
#define UMASK_SETTING  (S_IWGRP | S_IXGRP | S_IWOTH | S_IXOTH)

int
main(int argc, char *argv[])
{
    int fd;
    struct stat sb;
    mode_t u;

    umask(UMASK_SETTING);

    fd = open(MYFILE, O_RDWR | O_CREAT | O_EXCL, FILE_PERMS);
    if (fd == -1)
        errExit("open-%s", MYFILE);
    if (mkdir(MYDIR, DIR_PERMS) == -1)
        errExit("mkdir-%s", MYDIR);

    u = umask(0);      /* Извлекает (и очищает) значение параметра umask */

    if (stat(MYFILE, &sb) == -1)
        errExit("stat-%s", MYFILE);
    printf("Requested file perms: %s\n", filePermStr(FILE_PERMS, 0));
    printf("Process umask:        %s\n", filePermStr(u, 0));
    printf("Actual file perms:    %s\n\n", filePermStr(sb.st_mode, 0));

    if (stat(MYDIR, &sb) == -1)
        errExit("stat-%s", MYDIR);
    printf("Requested dir. perms: %s\n", filePermStr(DIR_PERMS, 0));
    printf("Process umask:        %s\n", filePermStr(u, 0));
    printf("Actual dir. perms:    %s\n", filePermStr(sb.st_mode, 0));
    if (unlink(MYFILE) == -1)
        errMsg("unlink-%s", MYFILE);
```

```
    if (rmdir(MYDIR) == -1)
        errMsg("rmdir-%s", MYDIR);
    exit(EXIT_SUCCESS);
}
```

files/t_umask.c

15.4.7. Изменение прав доступа к файлу: системные вызовы chmod() и fchmod()

Системные вызовы `chmod()` и `fchmod()` изменяют права доступа к файлу.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

#define _XOPEN_SOURCE 500        /* Или #define _BSD_SOURCE */
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

Оба вызова возвращают `0` при успешном завершении и `-1` при ошибке

Системный вызов `chmod()` изменяет права доступа к файлу, указанному в аргументе `pathname`. Если данный аргумент является символической ссылкой, то системный вызов `chmod()` изменяет права доступа к файлу, на который она указывает, а не права доступа к самой ссылке. (Символическая ссылка всегда создается с правами доступа на чтение, запись и выполнение, предоставленными всем пользователям, и эти права нельзя изменить. Они игнорируются при разыменовании ссылки.)

Системный вызов `fchmod()` изменяет права доступа к файлу, указанному с помощью открытого файлового дескриптора `fd`.

Аргумент `mode` задает новые права доступа к файлу как число (восьмеричное) либо в виде маски, сформированной с помощью операции ИЛИ (`|`) из битов прав доступа, приведенных в табл. 15.4. Для изменения прав доступа к файлу необходимо, чтобы процесс был привилегированным (`CAP_FOWNER`) либо чтобы его действующий UID совпадал с владельцем (UID) для файла. (Если абсолютно точно, то в Linux в случае непривилегированного процесса с идентификатором пользователя файла должен совпадать пользовательский идентификатор в файловой системе процесса, а не его действующий UID, как описано в разделе 9.5.)

Чтобы предоставить всем пользователям только право доступа на чтение файла, можно использовать следующий системный вызов:

```
if (chmod("myfile", S_IRUSR | S_IRGRP | S_IROTH) == -1)
    errExit("chmod");
/* Или эквивалент — chmod("myfile", 0444); */
```

Порядок изменения избранных битов прав доступа к файлу таков: сначала следует извлечь существующие значения, применив системный вызов `stat()`, изменить необходимые биты, а затем выполнить системный вызов `chmod()`, чтобы обновить права доступа:

```
struct stat sb;
mode_t mode;
```

```
if (stat("myfile", &sb) == -1)
    errExit("stat");
mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;
    /* Разрешить владельцу запись, другим пользователям запретить чтение,
       остальные биты не менять */
if (chmod("myfile", mode) == -1)
    errExit("chmod");
```

Приведенный выше код эквивалентен следующей команде оболочки:

```
$ chmod u+w,o-r myfile
```

В подразделе 15.3.1 мы отметили: если каталог размещен в файловой системе ext2, смонтированной с параметром `-o bsdgroups`, или в какой-либо файловой системе, смонтированной с параметром `-o sysvgroups`, и для этого каталога установлен бит прав доступа set-group-ID, то создаваемые в данном каталоге новые файлы наследуют свою принадлежность от родительского каталога, а не от действующего GID создающего процесса. Но может случиться так, что идентификатор группы для такого файла не совпадает ни с одним из GID создающего процесса. По этой причине, когда непривилегированный процесс (то есть не обладающий возможностью `CAP_FSETID`) совершает системный вызов `chmod()` (или `fchmod()`) для файла, идентификатор группы которого не совпадает с действующим GID или идентификатором добавочной группы для процесса, ядро всегда очищает бит прав доступа set-group-ID. Это мера безопасности, призванная запретить пользователю создание программы с правами доступа set-group-ID для группы, к которой он не принадлежит. Следующие команды оболочки демонстрируют попытку взлома, которая пресекается данной мерой:

```
$ mount | grep test          Каталог /test смонтирован с параметром –o bsdgroups
/dev/sda9 on /test type ext3 (rw,bsdgroups)
$ ls -ld /test               Каталог имеет корневой идентификатор
                             GID, который доступен для записи всем
drwxrwxrwx   3 root    root        4096 Jun 30 20:11 /test
$ id                         Я обычный пользователь, не входящий в корневую группу
uid=1000(mtk) gid=100(users) groups=100(users),101(staff),104(teach)
$ cd /test
$ cp ~/myprog .              Скопируем сюда несколько вредоносных программ
$ ls -l myprog               Здорово! Они в корневой группе!
-rwxr-xr-x   1 mtk     root        19684 Jun 30 20:43 myprog
$ chmod g+s myprog           Смогу ли я присвоить корневые права доступа?
$ ls -l myprog               Хм, никак…
-rwxr-xr-x   1 mtk     root        19684 Jun 30 20:43 myprog
```

15.5. Флаги индексного дескриптора (расширенные атрибуты файла в файловой системе ext2)

Ряд файловых систем Linux допускают устанавливать для файлов и каталогов различные *флаги индексного дескриптора*. Эта функция является нестандартным расширением Linux.

В современных версиях BSD подобная функция флагов индексного дескриптора реализована в виде указания флагов файла с помощью команд chflags(1) и chflags(2).

Первой файловой системой Linux, которая стала поддерживать флаги индексного дескриптора, оказалась ext2, и поэтому такие флаги иногда называют *расширенными атрибутами файла в файловой системе ext2*. Впоследствии поддержка флагов индексного дескриптора была добавлена и в другие системы: Btrfs, ext3, ext4, Reiserfs (с версии Linux 2.4.19), XFS (с версий Linux 2.4.25 и 2.6) и JFS (с версии Linux 2.6.17).

Количество флагов индексного дескриптора немного различается для разных файловых систем. Чтобы использовать флаги индексного дескриптора в системе Reiserfs, необходимо при ее монтировании указать параметр mount –o attrs.

Напрямую из оболочки флаги индексного дескриптора можно установить и просмотреть с помощью команд `chattr` и `lsattr`, как показано в следующем примере:

```
$ lsattr myfile
-------- myfile
$ chattr +ai myfile            Установить флаги Append Only и Immutable
$ lsattr myfile
----ia-- myfile
```

Внутри какой-либо программы флаги индексного дескриптора можно извлечь и изменить с помощью системного вызова `ioctl()`, что будет подробно описано ниже.

Основная часть флагов индексного дескриптора предназначена для использования с обычными файлами, хотя некоторые из этих флагов можно задействовать также (или только) и для каталогов. В табл. 15.6 приведены доступные флаги. Указано соответствующее имя флага (оно определено в файле `<linux/fs.h>`), применяемое при выполнении системного вызова `ioctl()` из программ, а также буква параметра, используемого с командой `chattr`.

Таблица 15.6. Флаги индексного дескриптора

Константа	Параметр команды chattr	Назначение
FS_APPEND_FL	a	Только добавление (необходима привилегия)
FS_COMPR_FL	c	Задействовать сжатие файла (не реализовано)
FS_DIRSYNC_FL D	D	Синхронное обновление каталогов (начиная с версии Linux 2.6)
FS_IMMUTABLE_FL	i	Неизменяемый (необходима привилегия)
FS_JOURNAL_DATA_FL	j	Задействовать журналирование данных (необходима привилегия)
FS_NOATIME_FL	A	Не обновлять время последнего доступа к файлу
FS_NODUMP_FL	d	Без дампа
FS_NOTAIL_FL	t	Без упаковки хвостов
FS_SECRM_FL	s	Защищенное удаление (не реализовано)
FS_SYNC_FL	S	Синхронное обновление файла (и каталога)
FS_TOPDIR_FL	T	Считать каталогом верхнего уровня для стратегии Орлова (начиная с версии Linux 2.6)
FS_UNRM_FL	u	Можно восстановить удаленный файл (не реализовано)

До версии Linux 2.6.19 константы FS_*, показанные в табл. 15.6, не определялись в файле <linux/fs.h>. Вместо этого существовал набор заголовочных файлов, характерных для какой-либо файловой системы, и данные файлы задавали для нее имена констант, значения которых были одинаковыми. Так, в файловой системе ext2 была константа EXT2_APPEND_FL, определенная в файле <linux/ext2_fs.h>; файловая система Reiserfs имела константу REISERFS_APPEND_FL, определенную с тем же значением в файле <linux/reiser_fs.h> и т. д. Поскольку каждый из этих заголовочных файлов задает для соответствующих констант одинаковые значения, в старых версиях ОС, не использующих определения в файле <linux/fs.h>, можно включать любой заголовочный файл и применять имена, характерные для файловой системы.

Ниже приводится разъяснение различных флагов FL_*.

- FS_APPEND_FL — файл можно открыть для записи, только если установлен флаг O_APPEND (таким образом все обновления файла принудительно добавляются в его конец). Этот флаг можно использовать, например, для файла журнала. Устанавливать данный флаг могут только привилегированные (CAP_LINUX_IMMUTABLE) процессы.
- FS_COMPR_FL — хранить файл на диске в сжатом формате. Данная функция не реализована в стандартном виде ни в одной из основных файловых систем Linux. (Существуют версии, в которых она реализована для систем ext2 и ext3.) С учетом малой стоимости хранения на диске, избыточной нагрузки на ЦПУ при сжатии и распаковке, а также в связи с тем, что сжатие файла означает, что он перестает быть простым объектом с произвольным доступом к содержимому файла (с помощью функции lseek()), для многих приложений сжатие файла является нежелательным.
- FS_DIRSYNC_FL (с версии Linux 2.6) — сделать синхронным обновление каталогов (то есть open(pathname, O_CREAT), link(), unlink() и mkdir()). Эта функция аналогична механизму синхронного обновления файлов, описанному в разделе 13.3. Синхронное обновление каталогов точно так же отражается на производительности. Данный параметр можно применять только для каталогов. (Флаг монтирования MS_DIRSYNC, описанный в подразделе 14.8.1, обеспечивает подобную возможность, но по отношению к монтированию.)
- FS_IMMUTABLE_FL — сделать файл неизменяемым. Данные файла нельзя обновить (write() и truncate()), а изменения метаданных не допускаются (то есть chmod(), chown(), unlink(), link(), rename(), rmdir(), utime(), setxattr() и removexattr()). Установить данный флаг для файла могут только привилегированные процессы (CAP_LINUX_IMMUTABLE). Когда он установлен, даже привилегированный процесс не может изменить содержимое файла или его метаданные.
- FS_JOURNAL_DATA_FL — задействовать журналирование данных. Этот флаг поддерживается только в файловых системах ext3 и ext4. Они обеспечивают три уровня журналирования: *журнальное, упорядоченное* и *с обратной записью*. Все режимы заносят в журнал обновления метаданных файла, однако в *журнальном* дополнительно фиксируются обновления данных файла. В файловой системе, использующей журналирование в *упорядоченном* режиме или *с обратной записью*, привилегированный (CAP_SYS_RESOURCE) процесс может задействовать пофайловое журналирование обновлений данных с помощью установки этого флага. (Страница mount(8) руководства описывает различие между *упорядоченным режимом* и *обратной записью*.)
- FS_NOATIME_FL — не обновлять время последнего доступа к файлу при доступе к нему. Это позволяет избежать обновления индексного дескриптора файла при каждом доступе к файлу и таким образом повышает эффективность ввода-вывода (см. описание флага MS_NOATIME в подразделе 14.8.1).

- `FS_NODUMP_FL` — не включать данный файл в резервные копии, создаваемые с помощью команды `dump(8)`. Действие этого флага зависит от параметра `-h`, описанного на странице `dump(8)` руководства.
- `FS_NOTAIL_FL` — отключить упаковку хвостов. Этот флаг поддерживается только в файловой системе Reiserfs. Он отключает функцию, которая пытается упаковать небольшие файлы (и завершающие фрагменты больших файлов) в тот же дисковый блок, что и метаданные файла. Упаковку хвостов можно отключить и для файловой системы Reiserfs в целом, смонтировав ее с параметром `mount -notail`.
- `FS_SECRM_FL` — удалить файл «бесследно». Назначение данной нереализованной функции заключается в надежном удалении файла при его стирании; то есть сначала файл затирается другими данными, чтобы не позволить программе сканирования диска считать его или восстановить. (Задача по-настоящему надежного удаления файлов является довольно сложной: фактически может потребоваться многократная запись на магнитный носитель для надежного стирания записанных ранее данных; см. работу [Gutmann, 1996].)
- `FS_SYNC_FL` — сделать обновления файла синхронными. Применительно к файлам данный флаг обеспечивает синхронность операций записи в файл (как если бы был указан флаг `O_SYNC` для всех открытий данного файла). Применительно к каталогу обладает тем же действием, что и описанный выше флаг синхронных обновлений каталога.
- `FS_TOPDIR_FL` (с версии Linux 2.6) — помечает каталог для специальной обработки согласно стратегии *Орлова* для выделения блоков. Данная стратегия была создана под влиянием системы BSD. Она является видоизменением стратегии выделения блоков в файловой системе ext2, которая старается повысить шансы на то, чтобы взаимосвязанные файлы (например, расположенные внутри какого-либо каталога) были размещены на диске рядом друг с другом, что может уменьшить время поиска на диске. Подробности см. в работах [Corbet, 2002] и [Kumar et al., 2008]. Флаг `FS_TOPDIR_FL` работает только в файловой системе ext2 и ее потомках, ext3 и ext4.
- `FS_UNRM_FL` — допустить восстановление данного файла после его удаления. Эта функция не реализована, поскольку есть возможность реализовать механизмы восстановления файлов вне ядра.

В целом, если флаги индексного дескриптора установлены для каталога, их автоматически наследуют новые файлы и подкаталоги, создаваемые внутри данного каталога. Из этого правила есть исключения:

- флаг `FS_DIRSYNC_FL` (`chattr +D`), который можно применять только для каталога, наследуется только подкаталогами, создаваемыми в данном каталоге;
- когда флаг `FS_IMMUTABLE_FL` (`chattr +i`) устанавливается для каталога, он не наследуется файлами и каталогами, созданными внутри этого каталога, поскольку данный флаг не допускает добавления новых записей в каталоге.

Внутри программы флаги индексного дескриптора можно извлечь и изменить с помощью операций `ioctl()` `FS_IOC_GETFLAGS` и `FS_IOC_SETFLAGS`. (Эти константы определены в файле `<linux/fs.h>`.) Следующий код демонстрирует способ установки флага `FS_NOATIME_FL` для файла, определяемого по открытому файловому дескриптору `fd`:

```
int attr;
if (ioctl(fd, FS_IOC_GETFLAGS, &attr) == -1)    /* Извлечь текущие флаги */
    errExit("ioctl");
attr |= FS_NOATIME_FL;
if (ioctl(fd, FS_IOC_SETFLAGS, &attr) == -1)    /* Обновить флаги */
    errExit("ioctl");
```

Для изменения флагов индексного дескриптора файла необходимо, чтобы действующий UID для процесса соответствовал идентификатору пользователя (владельца) файла, либо чтобы процесс был привилегированным (`CAP_FOWNER`). (Если говорить абсолютно точно, то в Linux для непривилегированного процесса идентификатор пользователя в файловой системе процесса, а не его действующий UID должен совпадать с идентификатором пользователя файла, как сказано в разделе 9.5.)

15.6. Резюме

Системный вызов `stat()` извлекает информацию о файле (метаданные), основная часть которой берется из индексного дескриптора файла. К ней относятся сведения о принадлежности файла, о правах доступа к нему, а также метки времени.

Программа может обновить время последнего доступа к файлу и время его последнего изменения, выполнив системные вызовы `utime()`, `utimes()` или другие подобные интерфейсы.

Каждый файл имеет относящиеся к нему идентификаторы пользователя (владельца) и группы, а также набор битов прав доступа. Для организации прав доступа пользователи файла распределены по трем категориям: *владелец* (известный также как *пользователь*), *группа* и *остальные*. Каждой категории могут быть предоставлены следующие права доступа: *чтение*, *запись* и *выполнение*. Эта же схема используется для каталогов, хотя для них биты прав доступа имеют немного другое значение. Системные вызовы `chown()` и `chmod()` меняют принадлежность и права доступа к файлу. Системный вызов `umask()` задает маску битов прав доступа, которые всегда отключаются, когда вызывающий процесс создает файл.

Для файлов и каталогов задействуют три дополнительных бита прав доступа. Биты set-user-ID и set-group-ID можно применить к программным файлам, чтобы создать программы, которые вызывают появление у выполняющегося процесса привилегий, отличающихся от привилегий программного файла за счет присвоения другого идентификатора пользователя или группы. Для каталогов, расположенных в файловых системах, смонтированных с параметром `nogrpid (sysvgroups)`, бит set-group-ID можно использовать для контроля над тем, откуда будут наследовать GID новые файлы, создаваемые в данном каталоге: от действующего GID для процесса или же от GID для родительского каталога. Применительно к каталогам бит закрепления действует как флаг, запрещающий удаление.

Флаги индексного дескриптора управляют различными вариантами поведения файлов и каталогов. Несмотря на то что изначально они были определены для файловой системы ext2, теперь эти флаги поддерживаются и в некоторых других системах.

15.7. Упражнения

15.1. Раздел 15.4 содержит несколько утверждений о правах доступа, необходимых для различных операций в файловой системе. Воспользуйтесь командами оболочки или напишите программы, чтобы проверить или получить ответ на следующее:
   1) при удалении всех прав доступа владельца к файлу такой файл не разрешает доступ для владельца, хотя у группы и у остальных пользователей доступ все же сохраняется;

2) в каталоге, для которого есть разрешение на чтение, но нет разрешения на выполнение, можно вывести имена файлов, однако к самим файлам доступа нет вне зависимости от прав, назначенных им;
3) какие права доступа необходимы для родительского каталога и для самого файла, чтобы иметь возможность создать новый файл, открыть файл для чтения, открыть файл для записи или удалить файл? Какие права доступа необходимы для исходного и целевого каталогов при переименовании файла? Если при переименовании уже существует целевой файл, какие права доступа необходимы для него? Как установка закрепляющего бита (`chmod +t`) для каталога повлияет на операции переименования и удаления?

15.2. Как вы думаете, изменится ли какая-либо из трех меток времени файла после выполнения системного вызова `stat()`? Если нет, объясните почему.

15.3. На компьютере, работающем под Linux 2.6, измените программу, приведенную в листинге 15.1 (`t_stat.c`) так, чтобы метки времени файла отображались с наносекундной точностью.

15.4. Системный вызов `access()` проверяет права доступа, используя реальные идентификаторы пользователя и группы для процесса. Напишите соответствующую функцию, которая выполняет такие проверки на основе действующих UID и GID для процесса.

15.5. Как отмечалось в подразделе 15.4.6, системный вызов `umask()` всегда задает маску `umask` для процесса и в то же время возвращает копию старой маски. Каким образом можно получить текущую копию маски `umask`, оставив ее без изменений?

15.6. Команда `chmod a+rX file` предоставляет право доступа на чтение всем категориям пользователей и подобным же образом предоставляет право доступа на выполнение всем категориям пользователей, если файл `file` является каталогом или если разрешение на выполнение предоставлено какой-либо категории пользователей файла, как показано в приведенном ниже примере:

```
$ ls -ld dir file prog
dr-------- 2 mtk users    48 May 4 12:28 dir
-r-------- 1 mtk users 19794 May 4 12:22 file
-r-x------ 1 mtk users 19336 May 4 12:21 prog
$ chmod a+rX dir file prog
$ ls -ld dir file prog
dr-xr-xr-x 2 mtk users    48 May 4 12:28 dir
-r--r--r-- 1 mtk users 19794 May 4 12:22 file
-r-xr-xr-x 1 mtk users 19336 May 4 12:21 prog
```

Напишите программу, использующую системный вызов `stat()` и команду `chmod()` для выполнения действий, эквивалентных команде `chmod a+rX`.

15.7. Напишите простую версию команды `chattr(1)`, которая изменяет флаги индексного дескриптора. Подробности об интерфейсе командной строки для команды `chattr` см. на странице `chattr(1)` руководства. (Вам не требуется реализация параметров `-R`, `-V` и `-v`.)

16 Расширенные атрибуты

В данной главе описаны расширенные атрибуты (extended attributes, EA), допускающие использование произвольных метаданных в виде пар «имя-значение», привязанных к индексным дескрипторам. Расширенные атрибуты появились в версии Linux 2.6.

16.1. Обзор

Расширенные атрибуты служат для реализации списков контроля доступа (см. главу 17) и возможностями файла (см. главу 39). Однако организация данных атрибутов является настолько общей, что позволяет задействовать их и для других целей. Так, например, можно применить их для записи номера версии файла, информации о MIME-типе или кодовой таблице файла, а также для создания графического значка (или указателя на него).

EA не определены в стандарте SUSv3. Тем не менее подобная функция представлена в некоторых других реализациях UNIX, в особенности в современных версиях ОС BSD (см. `extattr(2)`), Solaris 9 и выше (см. `fsattr(5)`).

Расширенным атрибутам требуется поддержка со стороны основной файловой системы. Такую поддержку обеспечивают файловые системы Btrfs, ext2, ext3, ext4, JFS, Reiserfs и XFS.

> Данная поддержка является необязательной для файловой системы и контролируется с помощью параметров конфигурации ядра в меню File systems (Файловые системы). EA поддерживаются файловой системой Reiserfs, начиная с версии Linux 2.6.7.

Пространство имен для расширенных атрибутов

Имена расширенных атрибутов представлены в виде пары `namespace.name`. Компонент `namespace` предназначен для разграничения расширенных атрибутов по функционально различным классам. Компонент `name` уникальным образом идентифицирует расширенный атрибут внутри данного пространства `namespace`.

Для компонента `namespace` поддерживаются четыре значения (типа EA): `user`, `trusted`, `system` и `security`. Они используются следующим образом.

- Расширенными атрибутами типа `user` могут управлять непривилегированные процессы при определенных условиях: для извлечения значения EA `user` необходимо иметь разрешение на чтение файла; для изменения значения данного атрибута необходимо иметь разрешение на запись. (Отсутствие необходимых прав доступа приведет к возникновению ошибки `EACCES`.) Для привязки расширенных атрибутов `user` к файлу в файловой системе ext2, ext3, ext4 или Reiserfs необходимо, чтобы основная файловая система была смонтирована с ключом `user_xattr`:

  ```
  $ mount -o user_xattr device directory
  ```

- Расширенные атрибуты `trusted` похожи на атрибуты `user` в том, что ими могут управлять пользовательские процессы. Однако для управления EA `trusted` такой процесс должен быть привилегированным (`CAP_SYS_ADMIN`).

- Расширенные атрибуты `system` используются ядром для привязки системных объектов к файлу. В настоящее время единственным поддерживаемым типом объектов является список контроля доступа (см. главу 17).
- Расширенные атрибуты `security` применяются для хранения меток, предназначенных для модулей защиты операционной системы, а также для привязки возможностей к исполняемым файлам (см. подраздел 39.3.2). Изначально EA `security` были разработаны для поддержки Linux с улучшенной защитой (Security-Enhanced Linux, SELinux, https://www.nsa.gov/what-we-do/research/selinux/).

Индексный дескриптор может обладать несколькими связанными с ним расширенными атрибутами, из одного пространства имен или из разных. Имена данных атрибутов внутри каждого такого пространства являются отдельными наборами. В пространствах имен `user` и `trusted` имена EA могут быть произвольными строками. В пространстве имен `system` допускаются только имена, явным образом разрешенные ядром (то есть такие, которые используются в списках контроля доступа).

> Файловая система JFS поддерживает еще одно пространство имен, os2, которое не реализовано в других файловых системах. Это пространство имен предназначено для поддержки расширенных атрибутов, унаследованных от операционной системы OS/2. Для создания EA os2 процесс не обязан быть привилегированным.

Создание и просмотр расширенных атрибутов из оболочки

Для установки и просмотра расширенных атрибутов файла можно использовать команды оболочки `setfattr(1)` и `getfattr(1)`:

```
$ touch tfile
$ setfattr -n user.x -v "The past is not dead." tfile
$ setfattr -n user.y -v "In fact, it's not even past." tfile
$ getfattr -n user.x tfile        Извлечение значения одного EA
# file: tfile                     Информационное сообщение от команды getfattr
user.x="The past is not dead."    Команда getfattr command выводит пустую
                                  строку после каждого атрибута файла
$ getfattr -d tfile               Дамп значений всех EA типа user
# file: tfile
user.x="The past is not dead."
user.y="In fact, it's not even past."

$ setfattr -n user.x tfile        Изменение значения EA на пустую строку
$ getfattr -d tfile
# file: tfile
user.x
user.y="In fact, it's not even past."

$ setfattr -x user.y tfile        Удаление EA
$ getfattr -d tfile
# file: tfile
user.x
```

Одна из особенностей, которая продемонстрирована в приведенном выше сеансе работы в оболочке, заключается в следующем: значение расширенного атрибута может быть пустой строкой, и это не то же самое, что неопределенный расширенный атрибут. (В конце сеанса работы значение атрибута `user.x` является пустой строкой, а атрибут `user.y` не определен.)

По умолчанию команда `getfattr` выводит значения EA только для пространства имен `user`. Ключ `-m` можно использовать для указания шаблона регулярного выражения, выбирающего имена расширенных атрибутов, которые необходимо отобразить:

```
$ getfattr -m 'pattern' file
```

Значением по умолчанию для шаблона `pattern` является `^user\.`. Можно вывести все EA в файл с помощью следующей команды:

```
$ getfattr -m - file
```

16.2. Подробности реализации расширенных атрибутов

Здесь мы более подробно рассмотрим материал предыдущего раздела, добавив некоторые частности о реализации расширенных атрибутов.

Ограничения расширенных атрибутов user

Расширенными атрибутами `user` могут быть снабжены только файлы и каталоги. Другие типы файлов исключены по следующим причинам.

- Для символической ссылки все права доступа задействованы для всех пользователей, и эти права невозможно изменить. (Права доступа к такой ссылке не имеют смысла в Linux, как детально разъясняется в разделе 18.2.) То есть права нельзя использовать для того, чтобы не позволить произвольным пользователям назначать символической ссылке расширенный атрибут `user`. Решением данной проблемы является запрет для всех пользователей на создание EA `user` для символической ссылки.
- Для файлов устройств, сокетов и очередей FIFO с помощью прав доступа контролируется предоставление пользователям доступа к объекту, расположенному на нижнем уровне, для выполнения операций ввода-вывода. При манипулировании этими правами для управления созданием EA `user` может возникнуть конфликт.

Более того, непривилегированному процессу не разрешено назначать расширенный атрибут `user` для каталога, владельцем которого является другой пользователь, если для данного каталога установлен бит закрепления (см. подраздел 15.4.5). Это не позволяет произвольным пользователям назначать EA для таких каталогов, как `/tmp`, общедоступных для записи (в связи с чем могли бы позволить произвольным пользователям манипулировать расширенными атрибутами этого каталога), но имеющих установленный бит закрепления, чтобы не разрешить пользователям удаление файлов, владельцами которых являются другие пользователи в этом каталоге.

Ограничения реализации

Виртуальная файловая система Linux VFS накладывает следующие ограничения на расширенные атрибуты в любых файловых системах:

- длина имени такого атрибута ограничена 255 символами;
- размер его значения ограничен 64 Кбайт.

Кроме того, ряд файловых систем накладывают более строгие ограничения на размер и количество EA, которые могут быть ассоциированы с файлом:

- в файловых системах ext2, ext3 и ext4 общее количество байтов, примененных именами и значениями всех расширенных атрибутов какого-либо файла, ограничено размером одного блока логического диска (см. раздел 14.3): 1024, 2048 или 4096 байтами;
- в файловой системе JFS существует верхний предел 128 Кбайт для общего количества байтов, используемых именами и значениями всех EA какого-либо файла.

16.3. Системные вызовы для манипуляции расширенными атрибутами

В этом разделе мы рассмотрим системные вызовы, которые задействуются для обновления, извлечения и удаления расширенных атрибутов.

Создание и изменение расширенных атрибутов

С помощью системных вызовов `setxattr()`, `lsetxattr()` и `fsetxattr()` устанавливается значение какого-либо EA файла.

```
#include <sys/xattr.h>

int setxattr(const char *pathname, const char *name, const void *value,
             size_t size, int flags);
int lsetxattr(const char *pathname, const char *name, const void *value,
             size_t size, int flags);
int fsetxattr(int fd, const char *name, const void *value,
             size_t size, int flags);
```

Все вызовы возвращают `0` при успешном завершении и `-1` при ошибке

Различия между представленными тремя вызовами похожи на те, что существуют между вызовами `stat()`, `lstat()` и `fstat()` (см. раздел 15.1):

- системный вызов `setxattr()` идентифицирует файл по константе `pathname` и разыменовывает имя файла, если это символическая ссылка;
- системный вызов `lsetxattr()` идентифицирует файл по константе `pathname`, но не разыменовывает символические ссылки;
- системный вызов `fsetxattr()` идентифицирует файл по открытому дескриптору `fd`.

Эти же различия применимы и к другим группам системных вызовов, описываемых далее в данном разделе.

Аргумент `name` — строка с завершающим нулем, которая определяет имя расширенного атрибута. Аргумент `value` — указатель на буфер, определяющий новое значение для EA. Аргумент `size` задает длину данного буфера.

По умолчанию эти системные вызовы создают новый расширенный атрибут с заданным именем `name`, если такого атрибута еще нет, или изменяют значение уже существующего атрибута. Аргумент `flags` обеспечивает более тонкую настройку указанного поведения. Для него можно либо указать значение `0`, чтобы добиться поведения по умолчанию, либо присвоить одну из следующих констант:

- `XATTR_CREATE` — приводит к ошибке (`EEXIST`), если EA с заданным именем `name` уже существует;
- `XATTR_REPLACE` — приводит к ошибке (`ENODATA`), если EA с именем `name` еще не создан.

Рассмотрим пример использования системного вызова `setxattr()` для создания расширенного атрибута `user`.

```
char *value;

value = "The past is not dead.";

if (setxattr(pathname, "user.x", value, strlen(value), 0) == -1)
    errExit("setxattr");
```

Извлечение значения расширенного атрибута

С помощью системных вызовов `getxattr()`, `lgetxattr()` и `fgetxattr()` можно извлечь значение EA.

```
#include <sys/xattr.h>

ssize_t getxattr(const char *pathname, const char *name, void *value,
                 size_t size);
ssize_t lgetxattr(const char *pathname, const char *name, void *value,
                 size_t size);
ssize_t fgetxattr(int fd, const char *name, void *value,
                 size_t size);
```

Все вызовы возвращают `0` при успешном завершении и `-1` при ошибке

Аргумент `name` является строкой с завершающим нулем, идентифицирующей расширенный атрибут, чье значение мы желаем извлечь. Данное значение возвращается в буфер, на который указывает аргумент `value`. Буфер должен быть выделен вызывающим процессом, а его длину следует указать в аргументе `size`. При успешном завершении эти системные вызовы возвращают количество байтов, скопированное в аргумент `value`.

Если файл не имеет атрибута с указанным именем `name`, то данные системные вызовы приводят к ошибке `ENODATA`. Если значение `size` чересчур мало — то к ошибке `ERANGE`.

Можно указать для аргумента `size` значение `0`, и тогда аргумент `value` будет проигнорирован, но системный вызов по-прежнему возвращает размер значения расширенного атрибута. Таким образом обеспечивается механизм определения размера буфера `value`, который необходим для того, чтобы последующий вызов фактически извлек это значение EA. Однако обратите внимание: у нас по-прежнему нет гарантии того, что возвращаемый размер будет достаточно большим при последующей попытке извлечения значения. Другой процесс мог тем временем присвоить данному атрибуту большее значение или даже удалить сам атрибут.

Удаление расширенного атрибута

Системные вызовы `removexattr()`, `lremovexattr()` и `fremovexattr()` удаляет расширенный атрибут файла.

```
#include <sys/xattr.h>

int removexattr(const char *pathname, const char *name);
int lremovexattr(const char *pathname, const char *name);
int fremovexattr(int fd, const char *name);
```

Все вызовы возвращают `0` при успешном завершении и `-1` при ошибке

Строка с завершающим нулем, переданная аргументу `name`, идентифицирует EA, который нужно удалить. Попытка удаления несуществующего атрибута приведет к ошибке `ENODATA`.

Извлечение имен всех расширенных атрибутов, связанных с файлом

Системные вызовы `listxattr()`, `llistxattr()` и `flistxattr()` возвращают список, содержащий имена всех расширенных атрибутов, связанных с файлом.

```
#include <sys/xattr.h>

ssize_t listxattr(const char *pathname, char *list, size_t size);
ssize_t llistxattr(const char *pathname, char *list, size_t size);
ssize_t flistxattr(int fd, char *list, size_t size);
```

Все вызовы при успешном завершении возвращают количество байтов, скопированных в список, или `-1` при ошибке

Список имен расширенных атрибутов возвращается в виде последовательности строк с завершающим нулем в буфер, на который указывает аргумент `list`. Размер данного буфера должен быть задан в аргументе `size`. При успешном завершении эти системные вызовы возвращают количество байтов, скопированных в список `list`.

У системного вызова `getxattr()` можно указать нулевое значение аргумента `size`. В таком случае аргумент `list` игнорируется, однако система может вернуть размер буфера, который мог бы понадобиться для последующего вызова, чтобы на самом деле извлечь список имен расширенных атрибутов (предполагая, что он не изменился).

Для извлечения списка имен EA, связанных с файлом, необходимо лишь обеспечить доступность файла (то есть у нас должно быть право доступа на выполнение ко всем каталогам, которые есть в имени пути). Никаких прав доступа к самому файлу не требуется.

По соображениям безопасности из перечня расширенных атрибутов, приводящихся в списке `list`, могут быть исключены атрибуты, для доступа к которым у вызывающего процесса нет прав. Например, многие системы не включают атрибуты `trusted` в список, возвращаемый системным вызовом `listxattr()`, выполненным непривилегированным процессом. Однако заметьте: в предыдущем предложении сказано «могут быть исключены» — это указывает на то, что какая-либо реализация файловой системы не обязана так поступать. Следовательно, необходимо допускать возможность того, что при последующем системном вызове `getxattr()`, использующем имя расширенного атрибута из списка `list`, может произойти ошибка, поскольку данный процесс не обладает привилегиями, необходимыми для получения значения этого атрибута. (Подобная ошибка могла бы произойти также в том случае, если бы другой процесс удалил какой-либо атрибут между моментами вызовов `listxattr()` и `getxattr()`.)

Пример программы

Приведенная в листинге 16.1 программа извлекает и отображает имена и значения всех расширенных атрибутов файлов, перечисленных в командной строке. Для каждого файла программа задействует системный вызов `listxattr()`, чтобы извлечь имена всех EA, связанных с данным файлом, а затем осуществляет цикл, выполняя системный вызов `getxattr()` для каждого имени, чтобы извлечь соответствующее значение. По умолчанию значения атрибутов отображаются как простой текст. Если указать параметр `-x`, то значения атрибутов будут отображаться как шестнадцатеричные строки. Приведенный ниже сеанс работы в оболочке демонстрирует использование данной программы:

```
$ setfattr -n user.x -v "The past is not dead." Tfile
$ setfattr -n user.y -v "In fact, it's not even past." Tfile
$ ./xattr_view tfile
tfile:
        name=user.x; value=The past is not dead.
        Name=user.y; value=In fact, it's not even past.
```

Листинг 16.1. Отображение расширенных атрибутов файла

xattr/xattr_view.c

```
#include <sys/xattr.h>
#include "tlpi_hdr.h"

#define XATTR_SIZE 10000

static void
usageError(char *progName)
{
    fprintf(stderr, "Usage: %s [-x] file…\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    char list[XATTR_SIZE], value[XATTR_SIZE];
    ssize_t listLen, valueLen;
    int ns, j, k, opt;
    Boolean hexDisplay;

    hexDisplay = 0;
    while ((opt = getopt(argc, argv, "x")) != -1) {
        switch (opt) {
        case 'x': hexDisplay = 1;       break;
        case '?': usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);
    for (j = optind; j < argc; j++) {
        listLen = listxattr(argv[j], list, XATTR_SIZE);
        if (listLen == -1)
            errExit("listxattr");

        printf("%s:\n", argv[j]);

        for (ns = 0; ns < listLen; ns += strlen(&list[ns]) + 1) {
            printf("        name=%s; ", &list[ns]);

            valueLen = getxattr(argv[j], &list[ns], value, XATTR_SIZE);
            if (valueLen == -1) {
                printf("couldn't get value");
            } else if (!hexDisplay) {
                printf("value=%.*s", (int) valueLen, value);
            } else {
                printf("value=");
                for (k = 0; k < valueLen; k++)
                    printf("%02x ", (unsigned int) value[k]);
            }

            printf("\n");
        }
```

```
            printf("\n");
        }

        exit(EXIT_SUCCESS);
}
```

xattr/xattr_view.c

16.4. Резюме

Начиная с версии 2.6, Linux поддерживает расширенные атрибуты, которые допускают использование произвольных метаданных для файла, представленных в виде пар «имя-значение».

16.5. Упражнение

16.1. Напишите программу, которую можно использовать для создания или изменения EA типа `user` (то есть простую версию системного вызова `setfattr(1)`). Имя файла, а также имя и значение расширенного атрибута должны передаваться программе как аргументы командной строки.

17 Списки контроля доступа

В разделе 15.4 описана традиционная для UNIX (и Linux) организация файловых прав доступа. Для многих приложений эта схема является достаточной. Однако отдельным приложениям требуется более точное управление правами доступа, которые предоставляются отдельным пользователям и группам. Для удовлетворения таких требований во многих версиях UNIX реализовано расширение традиционной модели файловых прав доступа, известное как списки контроля доступа (англ. access control lists, ACLs, ACL-списки). Они позволяют указывать файловые права доступа для пользователя или группы, для произвольного количества пользователей или групп. В Linux данная модель реализована начиная с версии ядра 2.6.

Поддержка списков контроля доступа является необязательной для каждой файловой системы и контролируется с помощью параметров конфигурации ядра в меню File systems. Поддержка данных списков в файловой системе Reiserfs доступна с версии ядра 2.6.7.

Чтобы иметь возможность создавать списки контроля доступа в файловых системах ext2, ext3, ext4 или Reiserfs, следует монтировать файловую систему командой mount с параметром -o acl.

Списки контроля доступа никогда не были формально стандартизированы для UNIX-систем. Была предпринята попытка осуществить это в виде временных стандартов POSIX.1e и POSIX.2c, которые описывали соответственно спецификацию интерфейса прикладного программирования (API) и команды оболочки для ACL-списков (а также другие особенности, например возможности). В конечном итоге данная попытка провалилась, а временные стандарты были аннулированы. Тем не менее именно они (обычно в их последней версии, Draft 17 («Черновик № 17»)) лежат в основе ACL-списков во многих реализациях UNIX (включая Linux). Однако, поскольку существует множество различий между реализациями ACL-списков (что является отчасти следствием незавершенности временных стандартов), написание портируемых программ, использующих такие списки, представляет некоторые сложности.

В данной главе приводится описание списков контроля доступа и краткое руководство по их применению. Описан также ряд библиотечных функций, применяемых для операций со списками и их извлечения. Мы не станем углубляться в подробности работы всех этих функций, поскольку их очень много. (Дополнительную информацию см. на страницах руководства.)

17.1. Обзор

ACL-список представляет собой ряд записей, каждая из которых задает права доступа к файлу для отдельного пользователя или группы (рис. 17.1).

Записи ACL-списка

Каждая запись содержит следующие части:

- *тип тега*, указывающий на применимость данной записи к пользователю, группе или к какой-либо другой категории пользователей;

Рис. 17.1. Список контроля доступа

- необязательный *тег-спецификатор*, который идентифицирует отдельного пользователя или группу (то есть UID или GID);
- *набор прав доступа*, определяющий права доступа (чтение, запись и выполнение), предоставляемые данной записью.

Тип тега имеет одно из следующих значений.

- `ACL_USER_OBJ` — эта запись определяет права доступа, предоставляемые владельцу файла. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа *владельца (пользователя)*.
- `ACL_USER` — эта запись определяет права доступа, предоставляемые пользователю, который идентифицируется по тегу-спецификатору. ACL-список может содержать от нуля до нескольких записей `ACL_USER`, однако для отдельного пользователя можно определить не более одной такой записи.
- `ACL_GROUP_OBJ` — эта запись определяет права доступа, предоставляемые группе файла. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа *группы*, если только список не содержит также запись `ACL_MASK`.
- `ACL_GROUP` — эта запись определяет права доступа, предоставляемые группе, которая идентифицируется по тегу-спецификатору. ACL-список может содержать от нуля до нескольких записей `ACL_GROUP`, однако для отдельной группы можно определить не более одной такой записи.
- `ACL_MASK` — эта запись определяет максимальные права доступа, которые могут быть предоставлены записями `ACL_USER`, `ACL_GROUP_OBJ` и `ACL_GROUP`. ACL-список содержит не более одной записи `ACL_MASK`. Если он содержит запись `ACL_USER` или `ACL_GROUP`, то наличие записи `ACL_MASK` является обязательным. Мы подробнее поговорим об этом теге чуть ниже.
- `ACL_OTHER` — эта запись определяет права доступа, предоставляемые пользователям, не соответствующим другим записям в списке контроля доступа. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа для *остальных*.

Тег-спецификатор задействуется только для записей `ACL_USER` и `ACL_GROUP`. Он определяет либо идентификатор пользователя, либо идентификатор группы.

Минимальный и расширенный списки контроля доступа

Минимальный ACL-список семантически эквивалентен обычному набору прав доступа к файлу. Он содержит в точности три записи, по одной для каждого из трех типов: `ACL_USER_OBJ`, `ACL_GROUP_OBJ` и `ACL_OTHER`. В расширенный список добавлены записи `ACL_USER`, `ACL_GROUP` и `ACL_MASK`.

Одной из причин для различения минимального и расширенного ACL-списков является то, что последний обеспечивает семантическое расширение традиционной модели прав доступа. Еще одна причина связана с реализацией таких списков в Linux. Они реализованы как системные расширенные атрибуты (см. главу 16). EA, применяемый для организации списка контроля доступа к файлу, называется `system.posix_acl_access`. Этот атрибут необходим, только если у файла есть ACL-список. Информация о правах доступа для минимального списка может храниться (и действительно хранится) в обычных битах прав доступа к файлу.

17.2. Алгоритм проверки прав доступа с помощью списков контроля доступа

Проверка прав доступа к файлу, который снабжен ACL-списком, выполняется при тех же условиях, что и в традиционной модели (см. подраздел 15.4.3). Проверки производятся в следующем порядке, пока не будет удовлетворен какой-либо из критериев.

1. Если процесс является привилегированным, то предоставляются все права. Существует одно исключение из этого утверждения, аналогичное описанному в подразделе 15.4.3 для традиционной модели прав доступа. Когда привилегированный процесс запускает выполняемый файл, такому процессу предоставляется право на выполнение, только если оно предоставлено по меньшей мере одной ACL-записью для данного файла.
2. При совпадении действующего UID для процесса с владельцем (идентификатором пользователя) файла этому процессу предоставляются права доступа, указанные в записи `ACL_USER_OBJ`. (Если выражаться точно, то в Linux для проверок, описанных в данном разделе, используются идентификаторы, относящиеся к файловой системе процесса, а не его действующие идентификаторы, как сказано в разделе 9.5.)
3. Если действующий UID совпадает с тегом-спецификатором в одной из записей `ACL_USER`, то данному процессу предоставляются права, указанные в этой записи, с применением маски (операции И) к значению `ACL_MASK`.
4. Если один из идентификаторов группы для процесса (то есть действующий GID или любой идентификатор дополнительной группы) совпадает с группой файла (это соответствует записи `ACL_GROUP_OBJ`) или с тегом-спецификатором какой-либо записи `ACL_GROUP`, то права доступа определяются на основе проверки следующих условий, пока не будет обнаружено совпадение:
   1) если один из GID для процесса совпадает с группой файла, а запись `ACL_GROUP_OBJ` предоставляет требуемые права доступа, то данная запись определяет предоставляемые права. Они ограничиваются с учетом маски (операция И), примененной к значению `ACL_MASK` при его наличии;
   2) если один из GID для процесса совпадает с тегом-спецификатором `ACL_GROUP` в записи для этого файла, а сама запись предоставляет требуемые права доступа, то данная запись определяет предоставленные права. Они ограничиваются с учетом маски (операция И), примененной к значению `ACL_MASK`;
   3) в противном случае доступ запрещается.

5. В остальных случаях процессу предоставляются права доступа, указанные в записи ACL_OTHER.

Можно пояснить правила, относящиеся к GID, с помощью нескольких примеров. Допустим, у нас есть файл с идентификатором группы 100, причем этот файл защищен списком ACL, приведенным на рис. 17.1. Если бы процесс с GID 100 выполнил вызов access(file, R_OK), то этот вызов завершился бы успешно (то есть вернул бы 0). (Системный вызов access() описан в подразделе 15.4.4.) С другой стороны, несмотря на то, что запись ACL_GROUP_OBJ предоставляет все права доступа, вызов access(file, R_OK | W_OK | X_OK) завершился бы с ошибкой (то есть вернул бы –1 и значение EACCES для переменной errno), поскольку к правам доступа ACL_GROUP_OBJ применена маска (операция И) ACL_MASK, которая запрещает выполнение.

Приведем еще один пример на основе рис. 17.1. Допустим, у нас есть процесс с GID 102, причем среди идентификаторов дополнительных групп есть также идентификатор 103. Для такого процесса вызовы access(file, R_OK) и access(file, W_OK) завершились бы успешно, поскольку они соответствовали бы записи ACL_GROUP для идентификаторов 102 и 103. С другой стороны, системный вызов access(file, R_OK | W_OK) привел бы к ошибке, так как здесь нет совпадающей записи ACL_GROUP, которая предоставляла бы права доступа на чтение и запись одновременно.

17.3. Длинная и краткая текстовые формы списков контроля доступа

При манипулировании ACL-списками с помощью команд setfacl и getfacl (мы опишем их очень скоро) или некоторых библиотечных функций указывают текстовые представления записей ACL-списка. Для таких текстовых представлений допускаются два формата.

- ACL-список в *длинной текстовой форме* содержит по одной записи ACL на строку и может также включать комментарии, которые начинаются с символа # и продолжаются до конца строки. Команда getfacl выводит ACL-списки в длинной текстовой форме. Для команды setfacl с параметром –M acl-file, берущим ACL-спецификацию из файла, необходимо, чтобы эта спецификация была представлена в длинной текстовой форме.
- ACL-списки в *краткой текстовой форме* состоят из последовательности ACL-записей, разделенных запятыми.

В обеих формах каждая запись состоит из трех частей, разделенных двоеточиями:

```
tag-type:[tag-qualifier]: permissions
```

Часть *tag-type* содержит одно из значений, показанных в первом столбце табл. 17.1. Далее может следовать необязательная часть *tag-qualifier*, которая идентифицирует пользователя или группу по имени или числовому ID. Эта часть присутствует только для записей ACL_USER и ACL_GROUP. Ниже приведены все краткие текстовые формы ACL-списков, соответствующие традиционной маске прав доступа 0650:

```
u::rw-,g::r-x,o::---
u::rw,g::rx,o::-
user::rw,group::rx,other::-
```

В следующей краткой текстовой форме ACL-списка присутствуют два имени пользователей, имя группы и запись маски:

```
u::rw,u:paulh:rw,u:annabel:rw,g::r,g:teach:rw,m::rwx,o::-
```

Таблица 17.1. Расшифровка текстовых форм записи ACL-списка

Текстовые формы тега	Есть ли тег-спецификатор?	Соответствующий тип тега	Запись для...
u, user	Нет	ACL_USER_OBJ	Владельца файла (пользователя)
u, user	Да	ACL_USER	Указанного пользователя
g, group	Нет	ACL_GROUP_OBJ	Группы файла
g, group	Да	ACL_GROUP	Указанной группы
m, mask	Нет	ACL_MASK	Маски класса группы
o, other	Нет	ACL_OTHER	Остальных пользователей

17.4. Запись ACL_MASK и класс группы для ACL-списка

Если ACL-список содержит запись `ACL_USER` или `ACL_GROUP`, то должен включать запись `ACL_MASK`. Если же не содержит этих двух записей, то запись `ACL_MASK` является необязательной.

Запись `ACL_MASK` действует подобно верхней границе прав доступа, предоставляемых записями ACL-списка в так называемом *классе группы* — наборе всех записей `ACL_USER`, `ACL_GROUP` и `ACL_GROUP_OBJ` в списке ACL.

Цель записи `ACL_MASK` — обеспечить согласованное поведение при запуске приложений, которые не используют ACL-списки. В качестве примера, демонстрирующего необходимость записи с маской, предположим, что ACL-список файла содержит следующие записи:

```
user::rwx            # ACL_USER_OBJ
user:paulh:r-x       # ACL_USER
group::r-x           # ACL_GROUP_OBJ
group:teach:--x      # ACL_GROUP
other::--x           # ACL_OTHER
```

Предположим теперь, что программа выполняет следующий системный вызов `chmod()`:

```
chmod(pathname, 0700);    /* Установить права доступа rwx------ */
```

Для приложения, не работающего с ACL-списками, это означает «Запретить доступ всем, кроме владельца файла». Такая семантика должна сохраняться даже при наличии ACL-списков. В случае отсутствия записи `ACL_MASK` данное поведение можно реализовать различными способами, но каждый из них сталкивается с трудностями (представлены ниже).

- Простого изменения записей `ACL_GROUP_OBJ` и `ACL_USER_OBJ` так, чтобы у них была маска `---`, окажется недостаточно, поскольку у пользователя `paulh` и у группы `teach` по-прежнему остались бы некоторые права доступа к файлу.
- Другой возможностью могло бы стать применение новой группы с иным набором прав доступа (то есть с полным запретом доступа) для всех записей `ACL_USER`, `ACL_GROUP`, `ACL_GROUP_OBJ` и `ACL_OTHER` в списке ACL:

```
user::rwx                       # ACL_USER_OBJ
user:paulh:---                  # ACL_USER
group::---                      # ACL_GROUP_OBJ
group:teach:---                 # ACL_GROUP
other::---                      # ACL_OTHER
```

Проблема такого подхода заключается в том, что приложение, которое не использует ACL-списки, могло бы непреднамеренно нарушить семантику прав доступа, установленную приложениями, применяющими ACL-списки, поскольку (например) следующий вызов не вернул бы записи ACL_USER и ACL_GROUP ACL-списка в их исходное состояние:

```
chmod(pathname, 751);
```

- Чтобы избежать таких проблем, можно было бы сделать запись `ACL_GROUP_OBJ` ограничивающим набором для всех записей `ACL_USER` и `ACL_GROUP`. Однако это означало бы, что права доступа `ACL_GROUP_OBJ` необходимо устанавливать в сочетании со всеми правами, предоставляемыми записями `ACL_USER` и `ACL_GROUP`. Это конфликтовало бы с использованием записи `ACL_GROUP_OBJ` для определения прав доступа, согласующихся с группой файла.

Для решения данных проблем предназначена запись `ACL_MASK`. Она обеспечивает механизм, позволяющий реализовать традиционное назначение операций `chmod()`, не разрушая семантики прав доступа к файлу, заданной приложениями, применяющими ACL-списки. Когда ACL-список имеет запись `ACL_MASK`:

- все изменения в традиционных правах доступа группы с помощью системного вызова `chmod()` изменяют параметры записи `ACL_MASK` (а не записи `ACL_GROUP_OBJ`);
- вызов `stat()` возвращает права доступа `ACL_MASK` (а не права доступа `ACL_GROUP_OBJ`), указанные в битах прав доступа для группы в поле `st_mode` (см. рис. 15.1).

В то время как запись `ACL_MASK` обеспечивает способ сохранения ACL-информации, которая видна для приложений, не использующих ACL-списки, обратное не гарантировано. Допустим, к примеру, что мы снабдили файл следующим ACL-списком:

```
user::rw-,group::---,mask::---,other::r—
```

Если затем мы выполним команду `chmod g+rw` для этого файла, то ACL-список станет таким:

```
user::rw-,group::---,mask::rw-,other::r—
```

В данном случае у группы по-прежнему нет доступа к файлу. Одним из обходных вариантов является изменение записи ACL для группы, чтобы предоставить все права доступа. Вследствие этого группа всегда будет получать все права доступа, которые предоставляет запись `ACL_MASK`.

17.5. Команды getfacl и setfacl

В сеансе оболочки можно использовать команду `getfacl`, чтобы увидеть ACL-список для файла.

```
$ umask 022                     Приводим umask в известное состояние
$ touch tfile                   Создаем новый файл
$ getfacl tfile
# file: tfile
# owner: mtk
```

```
# group: users
user::rwgroup::
r–
other::r–
```

Из результатов работы команды `getfacl` видно, что новый файл создается с минимальным ACL-списком. При выводе данного списка в текстовой форме команда `getfacl` предваряет его записи тремя строками, в которых показаны имя и принадлежность данного файла. Можно отменить вывод этих строк, если указать параметр `--omit-header`.

Продемонстрируем теперь, что изменения прав доступа к файлу, выполняемые с помощью обычной команды `chmod`, «пропускаются» через ACL-список.

```
$ chmod u=rwx,g=rx,o=x tfile
$ getfacl –omit-header tfile
user::rwx
group::r-x
other::--x
```

Команда `setfacl` модифицирует ACL-список для файла. Здесь мы используем команду `setfacl –m`, чтобы добавить записи `ACL_USER` и `ACL_GROUP` в список прав доступа:

```
$ setfacl –m u:paulh:rx,g:teach:x tfile
$ getfacl –omit-header tfile
user::rwx
user:paulh:r-x                          Запись ACL_USER
group::r-x
group:teach:--x                         Запись ACL_GROUP
mask::r-x                               Запись ACL_MASK
other::--x
```

Команда `setfacl` с параметром `–m` изменяет существующие записи ACL-списка или добавляет новые записи, если соответствующие записи с указанным типом тега и спецификатором еще не существуют. Можно дополнительно применить параметр `–R`, чтобы рекурсивно применить указанный ACL-список для всех файлов в дереве каталогов.

Из результатов работы команды `getfacl` видно, что параметр `setfacl` автоматически создал запись `ACL_MASK` для данного ACL-списка.

Добавление записей `ACL_USER` и `ACL_GROUP` превращает данный ACL-список в расширенный, и команда `ls –l` подтверждает это, поскольку после традиционной маски прав доступа следует знак +:

```
$ ls –l tfile
-rwxr-x–x+        1 mtk        users               0 Dec 3 15:42 tfile
```

Продолжим использование параметра `setfacl`, отключив все права доступа, кроме выполнения, в записи `ACL_MASK`, а затем просмотрим ACL-список еще раз с помощью команды `getfacl`:

```
$ setfacl –m m::x tfile
$ getfacl –omit-header tfile
user::rwx
user:paulh:r-x                      #effective:--x
group::r-x                          #effective:--x
group:teach:--x
mask::--x
other::--x
```

Комментарии `#effective:`, которые команда `getfacl` выводит после записей, относящихся к пользователю `paulh` и к группе файла (`group::`), информируют нас о том, что

после применения маски (операция И) к записи ACL_MASK фактические права доступа, предоставляемые каждой из этих записей, окажутся меньше указанных в записи.

Используем теперь команду ls -l для повторного просмотра традиционных битов прав доступа к файлу. Видно, что показанные биты прав доступа для класса группы отражают права доступа в маске ACL_MASK (--x), а не права из записи ACL_GROUP (r-x).

```
$ ls -l tfile
-rwx-x-x+        1 mtk        users                0 Dec 3 15:42 tfile
```

Команду setfacl -x можно применять для удаления записей из ACL-списка. Удалим записи, относящиеся к пользователю paulh и к группе teach (при удалении права доступа не указывают):

```
$ setfacl -x u:paulh,g:teach tfile
$ getfacl -omit-header tfile
user::rwx
group::r-x
mask::r-x
other::--x
```

Обратите внимание: во время этой операции команда setfacl автоматически подстраивает запись маски так, чтобы она соответствовала всем записям класса группы. (Здесь оказалась всего одна такая запись: ACL_GROUP_OBJ.) Если необходимо избежать подобной подстройки, то в команде setfacl следует указать параметр -n.

В заключение надо отметить, что команду setfacl с параметром -b можно использовать для удаления всех расширенных записей из ACL-списка, оставив лишь минимальные (то есть для пользователя, группы и остальных).

17.6. ACL-списки по умолчанию и создание файла

До сего момента разговора о списках ACL рассматривались списки *доступа*. Как видно из самого названия, данный список используется при определении разрешений, которые имеет процесс, получающий доступ к файлу, связанному с ACL-списком. Можно создать список второго типа для каталогов: ACL-список *по умолчанию*.

Данный список не играет роли при определении прав доступа, предоставляемых при доступе к каталогу. Напротив, его наличие или отсутствие определяет ACL-список (или списки), а также права доступа, которые относятся к файлам и к подкаталогам, создаваемым в данном каталоге. (ACL-список по умолчанию хранится в виде расширенного атрибута с именем system.posix_acl_default.)

Чтобы просмотреть и задать ACL-список по умолчанию для каталога, применяется параметр -d для команд getfacl и setfacl.

```
$ mkdir sub
$ setfacl -d -m u::rwx,u:paulh:rx,g::rx,g:teach:rwx,o::- sub
$ getfacl -d -omit-header sub
user::rwx
user:paulh:r-x
group::r-x
group:teach:rwx
mask::rwx             Команда setfacl создала запись ACL_MASK автоматически
other::---
```

Удалить ACL-список по умолчанию для каталога можно с помощью параметра setfacl -k.

Если у каталога есть ACL-список по умолчанию, то:

- новый подкаталог, создаваемый в данном каталоге, наследует ACL-список по умолчанию в качестве своего ACL-списка по умолчанию. Иными словами, ACL-списки по умолчанию распространяются на нижние уровни дерева каталогов по мере создания новых подкаталогов;
- новый файл или подкаталог, созданные в данном каталоге, наследуют ACL-список по умолчанию для каталога в качестве своего ACL-списка доступа. Для записей ACL-списка, соответствующих традиционным битам прав доступа, применяется маска (операция И) в сочетании с соответствующими битами аргумента `mode` системного вызова (`open()`, `mkdir()` и т. д.), использованного для создания файла или подкаталога. Под «соответствующими записями ACL-списка» подразумеваются:
  - `ACL_USER_OBJ`;
  - `ACL_MASK` или, если отсутствует запись `ACL_MASK`, `ACL_GROUP_OBJ`;
  - `ACL_OTHER`.

Если у каталога есть ACL-список по умолчанию, то маска `umask` для процесса (см. подраздел 15.4.6) не принимает участия в определении прав доступа для записей ACL-списка доступа нового файла, созданного в данном каталоге.

В качестве примера того, как новый файл наследует ACL-список доступа от ACL-списка по умолчанию для родительского каталога, допустим, что мы использовали следующий вызов `open()` для создания нового файла в каталоге, созданном выше:

```
open("sub/tfile", O_RDWR | O_CREAT,
        S_IRWXU | S_IXGRP | S_IXOTH);     /* rwx-x-x */
```

Новый файл имел бы следующий ACL-список доступа:

```
$ getfacl -omit-header sub/tfile
user::rwx
user:paulh:r-x                  #effective:--x
group::r-x                      #effective:--x
group:teach:rwx                 #effective:--x
mask::--x
other::---
```

Если у каталога нет ACL-списка по умолчанию, то:

- новые подкаталоги, созданные в этом каталоге, также не имеют ACL-списка по умолчанию;
- права доступа к новому файлу или к каталогу устанавливаются в соответствии с обычными правилами (см. подраздел 15.4.6): для прав доступа к файлу берется значение аргумента `mode` (переданное вызову `open()`, `mkdir()` и т. д.), кроме битов, которые отключены маской `umask` для процесса. Это приводит к тому, что новый файл обладает минимальным ACL-списком.

17.7. Границы реализации списка контроля доступа

Различные реализации файловых систем накладывают ограничения на количество записей в ACL-списке.

- В файловых системах ext2, ext3 и ext4 общее количество ACL-списков для файла регулируется требованием того, чтобы байты для всех имен и значений расширенных атрибутов файла помещались внутри одного логического блока диска (см. раздел 16.2).

Каждой записи ACL-списка необходимо 8 байт, и поэтому максимальное количество записей списка для файла чуть меньше восьмой части размера блока (поскольку дополнительно затрачивается пространство для хранения имени расширенного атрибута ACL-списка). Так, при 4096-байтовом размере блока максимально допустимое количество записей ACL-списка — около 500. (Ядра версий до 2.6.11 накладывали для файловых систем ext2 и ext3 произвольно установленное ограничение — 32 записи.)

- В файловой системе XFS ACL-список ограничен 25 записями.
- В файловых системах Reiserfs и JFS списки ACL могут содержать до 8191 записи. Этот предел является следствием ограничения размера (64 Кбайт), налагаемого виртуальной файловой системой на значение расширенного атрибута (см. раздел 16.2).

На момент написания книги файловая система Btrfs ограничивает размер списков ACL приблизительно 500 записями. Тем не менее, поскольку данная система пока еще находится в активной разработке, это значение может измениться.

Хотя большинство рассмотренных выше файловых систем допускает создание большого количества записей в списке ACL, этого следует избегать по следующим причинам:

- обслуживание длинных ACL-списков становится сложной задачей системного администрирования, которая потенциально может приводить к ошибкам;
- количество времени, необходимого для сканирования ACL-списка в поисках соответствующей записи (или нескольких записей в случае проверки идентификатора группы), линейно пропорционально количеству записей ACL-списка.

Как правило, можно довести количество записей ACL-списка до разумного предела, определив подходящие группы в системном файле группы (см. раздел 8.3) и применив такие группы внутри ACL-списка.

17.8. API для ACL-списков

Предварительный стандарт POSIX.1e определял большой набор функций и структур данных для работы с ACL-списками. Поскольку их очень много, невозможно подробно описать все эти функции в одной книге. Мы рассмотрим, где они применяются, а завершит главу пример программы.

Программы, которые используют API для ACL-списков, должны включать файл `<sys/acl.h>`. Может также потребоваться включение файла `<acl/libacl.h>`, если программа задействует различные Linux-расширения предварительного стандарта POSIX.1e. (Перечень таких расширений приведен на странице `acl(5)` руководства.) Программы, применяющие API, следует компилировать с параметром `-lacl`, чтобы выполнить привязку к библиотеке libacl.

Как уже отмечалось, в Linux ACL-списки реализованы с помощью расширенных атрибутов, а интерфейс API для этих списков реализован в виде набора библиотечных функций. Эти функции оперируют структурами данных в пространстве пользователя и, если необходимо, совершают вызовы getxattr() и setxattr(), чтобы извлечь и изменить дисковый расширенный атрибут system, хранящий представление списка ACL. Возможно также (но не рекомендуется) использование приложением вызовов getxattr() и setxattr() для работы с ACL-списками напрямую.

Обзор

Функции, которые образуют API для ACL-списков, перечислены на странице `acl(5)` руководства. На первый взгляд это изобилие функций и структур данных может вызвать недоумение. На рис. 17.2 проиллюстрированы взаимосвязи между разными структурами данных, а также указаны варианты применения многих функций для работы с ACL-списками.

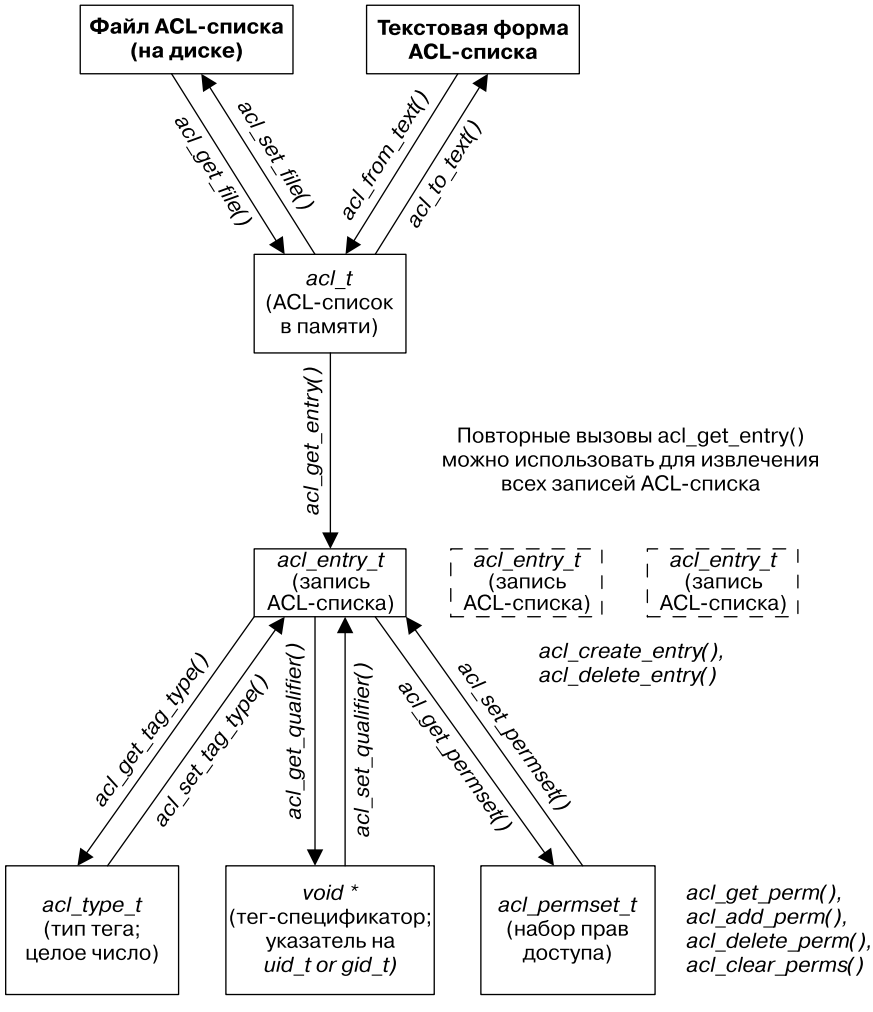


Рис. 17.2. Взаимосвязь между библиотечными ACL-функциями и структурами данных

Из рис. 17.2 видно, что API для ACL-списка рассматривает такой список как объект с иерархией:

- ACL-список состоит из одной или из нескольких ACL-записей;
- каждая ACL-запись состоит из типа тега, необязательного тега-спецификатора и набора прав доступа.

Сейчас мы бегло рассмотрим различные ACL-функции. В большинстве случаев мы не описываем ошибки, которые возвращает каждая функция. Функции, возвращающие целочисленное значение (`status`), как правило, возвращают `0` при успешном выполнении и `-1` в случае ошибки. Функции, возвращающие описатель (указатель), в случае ошибки возвращают `NULL`. Ошибки можно проанализировать обычным образом, используя значение `errno`.

Описатель — это абстрактный термин для обозначения какого-либо способа, примененного для соотнесения с объектом или со структурой данных. Представление описателя

зависит от реализации API. Он может быть, например, указателем, индексом массива или хеш-ключом.

Извлечение ACL-списка из файла в память

Функция `acl_get_file()` извлекает копию ACL-списка файла, идентифицируемого по аргументу `pathname`.

```
acl_t acl;
acl = acl_get_file(pathname, type);
```

Эта функция извлекает либо ACL-список доступа, либо ACL-список по умолчанию, в зависимости от того, какое значение указано для аргумента `type`: `ACL_TYPE_ACCESS` или `ACL_TYPE_DEFAULT`. В качестве результата функция `acl_get_file()` возвращает описатель (типа `acl_t`) для использования с другими ACL-функциями.

Извлечение записей ACL-списка, размещенного в памяти

Функция `acl_get_entry()` извлекает описатель (типа `acl_entry_t`), соотнесенный с одной из записей ACL-списка, размещенного в памяти и указанного с помощью аргумента `acl`. Данный описатель возвращается в то место, на которое указывает последний аргумент функции.

```
acl_entry_t entry;
status = acl_get_entry(acl, entry_id, &entry);
```

Аргумент `entry_id` определяет, описатель какой записи следует возвращать. Если для этого аргумента указано значение `ACL_FIRST_ENTRY`, то возвращается описатель первой записи в ACL-списке. Если для аргумента `entry_id` указано значение `ACL_NEXT_ENTRY`, то возвращается описатель записи, следующей за последней извлеченной записью ACL-списка. Таким образом, можно организовать цикл по всем записям в ACL-списке, указав для аргумента `type` при первом вызове значение `ACL_FIRST_ENTRY`, а при последующих вызовах — значение `ACL_NEXT_ENTRY`.

Функция `acl_get_entry()` возвращает `1` в случае успешного извлечения записи ACL-списка, `0` — если записей больше нет и `-1` в случае ошибки.

Извлечение и изменение атрибутов в записи ACL-списка

Функции `acl_get_tag_type()` и `acl_set_tag_type()` извлекают и меняют тип тега в записи ACL-списка, на которую указывает аргумент `entry`.

```
acl_tag_t tag_type;

status = acl_get_tag_type(entry, &tag_type);
status = acl_set_tag_type(entry, tag_type);
```

Аргумент `tag_type` имеет тип `acl_type_t` (целочисленным) и одно из следующих значений: `ACL_USER_OBJ`, `ACL_USER`, `ACL_GROUP_OBJ`, `ACL_GROUP`, `ACL_OTHER` или `ACL_MASK`.

Функции `acl_get_qualifier()` и `acl_set_qualifier()` извлекают и меняют тег-спецификатор в записи ACL-списка, на которую указывает аргумент `entry`. Рассмотрим пример: в нем мы полагаем, что данная запись уже определена нами как `ACL_USER` за счет соблюдения типа тега:

```
uid_t *qualp;                /* Указатель на UID */

qualp = acl_get_qualifier(entry);
status = acl_set_qualifier(entry, qualp);
```

Тег-спецификатор действителен, только если тип тега данной записи равен `ACL_USER` или `ACL_GROUP`. В первом случае указатель `qualp` является указателем на идентификатор пользователя (`uid_t *`), а во втором случае — на идентификатор группы (`gid_t *`).

Функции acl_get_permset() и acl_set_permset() извлекают и изменяют набор прав доступа в записи ACL-списка, на которую ссылается аргумент entry.

```
acl_permset_t permset;

status = acl_get_permset(entry, &permset);
status = acl_set_permset(entry, permset);
```

Тип данных acl_permset_t является описателем, ссылающимся на набор прав доступа. Следующие функции используются для работы с содержимым набора прав доступа:

```
int is_set;

is_set = acl_get_perm(permset, perm);
status = acl_add_perm(permset, perm);
status = acl_delete_perm(permset, perm);
status = acl_clear_perms(permset);
```

В каждом из перечисленных вызовов для аргумента perm указывается значение ACL_READ, ACL_WRITE или ACL_EXECUTE с обычным смыслом. Эти функции используются следующим образом:

- функция acl_get_perm() возвращает 1 (true), если право, указанное в аргументе perm, задействовано в наборе прав доступа, на который ссылается аргумент permset, и 0 — если это не так. Данная функция является расширением предварительного стандарта POSIX.1e в Linux;
- функция acl_add_perm() добавляет право доступа, указанное в аргументе perm, к набору прав, на который ссылается аргумент permset;
- функция acl_delete_perm() изымает право, указанное в аргументе perm, из набора прав доступа, на который ссылается аргумент permset (не является ошибкой удаление отсутствующего в наборе права);
- функция acl_clear_perms() удаляет все права доступа из набора, на который указывает аргумент permset.

Создание и удаление записей ACL-списка

Функция acl_create_entry() создает новую запись в существующем ACL-списке. Описатель, соотнесенный с новой записью, возвращается в то место, на которое указывает второй аргумент функции.

```
acl_entry_t entry;

status = acl_create_entry(&acl, &entry);
```

После этого новую запись можно заполнить с помощью функций, описанных ранее. Функция acl_delete_entry() удаляет запись из ACL-списка.

```
status = acl_delete_entry(acl, entry);
```

Обновление ACL-списка файла

Функция acl_set_file() является противоположностью функции acl_get_file(). Она обновляет расположенный на диске ACL-список с помощью содержимого ACL-списка, находящегося в памяти и указанного в аргументе acl.

```
int status;

status = acl_set_file(pathname, type, acl);
```

Аргумент type имеет значение либо ACL_TYPE_ACCESS, для обновления ACL-списка доступа, либо ACL_TYPE_DEFAULT, для обновления ACL-списка по умолчанию для каталога.

Преобразование ACL-списка, расположенного в памяти, в текстовую форму и обратно

Функция acl_from_text() переводит строку, содержащую длинную или краткую текстовую форму ACL-списка в ACL-список, расположенный в памяти, и возвращает описатель, который можно использовать для ссылки на данный список в последующих вызовах функции.

```
acl = acl_from_text(acl_string);
```

Функция acl_to_text() выполняет обратное преобразование, возвращая строку длинной текстовой формы для ACL-списка, на который указывает аргумент acl.

```
char *str;
ssize_t len;

str = acl_to_text(acl, &len);
```

Если аргумент len определен не как NULL, то буфер, на который он указывает, используется для возврата длины строки, являющейся результатом функции.

Другие функции из API для ACL-списков

В следующих абзацах описаны некоторые широко используемые ACL-функции, не показанные на рис. 17.2.

Функция acl_calc_mask(&acl) вычисляет и задает права доступа в записи ACL_MASK для ACL-списка, расположенного в памяти, на описатель которого указывает аргумент функции. Как правило, эта функция применяется при создании или при изменении ACL-списка. Маска прав доступа ACL_MASK вычисляется благодаря объединению прав доступа из всех записей ACL_USER, ACL_GROUP и ACL_GROUP_OBJ. Полезным свойством данной функции является то, что она создает запись ACL_MASK, если она еще не существует. То есть если мы добавляем записи ACL_USER и ACL_GROUP в ACL-список, который ранее был минимальным, то затем мы можем воспользоваться данной функцией, гарантирующей создание записи ACL_MASK.

Функция acl_valid(acl) возвращает 0, если ACL-список, на который указывает ее аргумент, является допустимым, или -1 в обратном случае. ACL-список является допустимым, если истинны все приведенные ниже утверждения:

- каждая из записей ACL_USER_OBJ, ACL_GROUP_OBJ и ACL_OTHER встречается только один раз;
- если присутствует любая из записей ACL_USER или ACL_GROUP, то есть и запись ACL_MASK;
- присутствует не более одной записи ACL_MASK;
- каждая запись ACL_USER обладает уникальным идентификатором пользователя;
- каждая запись ACL_GROUP обладает уникальным идентификатором группы.

Функции acl_check() и acl_error() (последняя является расширением Linux) представляют собой альтернативные варианты функции acl_valid(), которые в меньшей степени портируемы, но обеспечивают более точное описание ошибки в случае некорректно сформированного ACL-списка. Подробности см. на страницах руководства.

Функция acl_delete_def_file(pathname) удаляет ACL-список по умолчанию для каталога, на который указывает аргумент pathname.

Функция acl_init(count) создает новую пустую ACL-структуру, изначально содержащую пространство по меньшей мере для ACL-записей, количество которых равно count.

(Аргумент `count` является подсказкой для системы о предполагаемом использовании, это не жесткое ограничение.) Результатом функции является описатель для нового ACL-списка.

Функция `acl_dup(acl)` создает дубликат ACL-списка, на который указывает аргумент `acl`, и возвращает описатель для данного дубликата.

Функция `acl_free(handle)` высвобождает память, выделенную другими ACL-функциями. Так, например, необходимо применять эту функцию, чтобы высвободить память, которая была отведена в результате вызовов функций `acl_from_text()`, `acl_to_text()`, `acl_get_file()`, `acl_init()` и `acl_dup()`.

Пример программы

В листинге 17.1 продемонстрировано использование некоторых библиотечных функций для работы с ACL-списками. Эта программа извлекает и выводит список для файла (то есть представляет часть возможностей команды `getfacl`). Если в командной строке указан параметр `-d`, программа выводит ACL-список по умолчанию (для каталога), а не ACL-список доступа.

Рассмотрим пример использования данной программы:

```
$ touch tfile
$ setfacl -m 'u:annie:r,u:paulh:rw,g:teach:r' tfile
$ ./acl_view tfile
user_obj                    rw-
user            annie       r-
user            paulh       rw-
group_obj                   r-
group           teach       r-
mask                        rw-
other                       r-
```

Исходный код к данной книге содержит также программу acl/acl_update.c, которая выполняет обновление ACL-списка (то есть представляет часть функций команды setfacl).

Листинг 17.1. Вывод ACL-списка доступа или списка по умолчанию для файла

acl/acl_view.c

```
#include <acl/libacl.h>
#include <sys/acl.h>
#include "ugid_functions.h"
#include "tlpi_hdr.h"

static void
usageError(char *progName)
{
    fprintf(stderr, "Usage: %s [-d] filename\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    acl_t acl;
    acl_type_t type;
    acl_entry_t entry;
    acl_tag_t tag;
    uid_t *uidp;
    gid_t *gidp;
    acl_permset_t permset;
    char *name;
    int entryId, permVal, opt;
```

```
type = ACL_TYPE_ACCESS;
while ((opt = getopt(argc, argv, "d")) != -1) {
    switch (opt) {
    case 'd': type = ACL_TYPE_DEFAULT;     break;
    case '?': usageError(argv[0]);
    }
}

if (optind + 1 != argc)
    usageError(argv[0]);

acl = acl_get_file(argv[optind], type);
if (acl == NULL)
    errExit("acl_get_file");

/* Просмотр каждой записи в этом ACL-списке */
for (entryId = ACL_FIRST_ENTRY; ; entryId = ACL_NEXT_ENTRY) {

if (acl_get_entry(acl, entryId, &entry) != 1)
    break;           /* Выход при ошибке или по окончании записей */

    /* Извлечение и отображение типа тега */
    if (acl_get_tag_type(entry, &tag) == -1)
        errExit("acl_get_tag_type");
    printf("%-12s",  (tag == ACL_USER_OBJ) ?  "user_obj" :
                     (tag == ACL_USER) ?      "user" :
                     (tag == ACL_GROUP_OBJ) ? "group_obj" :
                     (tag == ACL_GROUP) ?     "group" :
                     (tag == ACL_MASK) ?      "mask" :
                     (tag == ACL_OTHER) ?      "other" : "???");

    /* Извлечение и отображение необязательного тега-спецификатора */
    if (tag == ACL_USER) {
        uidp = acl_get_qualifier(entry);
        if (uidp == NULL)
            errExit("acl_get_qualifier");
        name = userNameFromId(*uidp);
        if (name == NULL)
            printf("%-8d ", *uidp);
        else
            printf("%-8s ", name);
        if (acl_free(uidp) == -1)
            errExit("acl_free");
    } else if (tag == ACL_GROUP) {
        gidp = acl_get_qualifier(entry);
        if (gidp == NULL)
            errExit("acl_get_qualifier");
        name = groupNameFromId(*gidp);
        if (name == NULL)
            printf("%-8d ", *gidp);
        else
            printf("%-8s ", name);
        if (acl_free(gidp) == -1)
            errExit("acl_free");
    } else {
        printf("        ");
    }
    /* Извлечение и отображение прав доступа */
    if (acl_get_permset(entry, &permset) == -1)
        errExit("acl_get_permset");
```

```
        permVal = acl_get_perm(permset, ACL_READ);
        if (permVal == -1)
            errExit("acl_get_perm - ACL_READ");
        printf("%c", (permVal == 1) ? 'r' : '-');
        permVal = acl_get_perm(permset, ACL_WRITE);
        if (permVal == -1)
            errExit("acl_get_perm - ACL_WRITE");
        printf("%c", (permVal == 1) ? 'w' : '-');
        permVal = acl_get_perm(permset, ACL_EXECUTE);
        if (permVal == -1)
            errExit("acl_get_perm - ACL_EXECUTE");
        printf("%c", (permVal == 1) ? 'x' : '-');

        printf("\n");
    }

    if (acl_free(acl) == -1)
        errExit("acl_free");

    exit(EXIT_SUCCESS);
}
```

acl/acl_view.c

17.9. Резюме

Начиная с версии 2.6 Linux поддерживает списки контроля доступа (ACL-списки). Они расширяют традиционную модель прав доступа к файлам в UNIX, позволяя управлять разрешениями, предоставляемыми конкретным пользователям и группам.

Дополнительная информация

Последние версии (Draft 17) временных стандартов POSIX.1e и POSIX.2c доступны на веб-странице wt.tuxomania.net/publications/posix.1e/.

На странице `acl(5)` руководства приводится обзор ACL-списков, а также указания, относящиеся к портируемости различных библиотечных ACL-функций, реализованных в Linux.

Подробности о реализации ACL-списков и расширенных атрибутов в Linux можно найти в работе [Grьnbacher, 2003]. Ее автор, Андреас Грюнбахер (Andreas Grьnbacher) ведет также сайт http://acl.bestbits.at/, содержащий информацию о списках контроля доступа.

17.10. Упражнение

17.1. Напишите программу, которая отображает права доступа из записи ACL-списка, соответствующей какому-либо пользователю или группе. Эта программа должна принимать через командную строку два аргумента. Первым может быть либо буква `u`, либо буква `g`, указывающая на то, что именно идентифицирует второй аргумент — пользователя или группу. (Функции, определенные в листинге 8.1, можно использовать для того, чтобы разрешить ввод второго аргумента либо в числовом представлении, либо как имя.) Если запись ACL-списка, соответствующая указанному пользователю или группе, попадает в класс группы, то программа должна дополнительно отображать права доступа, которые были бы предоставлены после того, как эта запись будет изменена путем записи-маски.

18 Каталоги и ссылки

В данной главе мы завершаем обсуждение вопросов, относящихся к файлам, рассмотрев каталоги и ссылки. После обзора их реализации будут описаны системные вызовы для создания и удаления каталогов и ссылок. Затем будут представлены библиотечные функции, которые позволяют программам сканировать содержимое одиночного каталога, а также выполнять обход (то есть проверить каждый файл) каталога.

Каждый процесс имеет два атрибута, относящихся к каталогу: корневой каталог, задающий точку, относительно которой интерпретируются абсолютные имена путей, и рабочий каталог, задающий точку, относительно которой интерпретируются относительные имена путей. Мы рассмотрим системные вызовы, позволяющие процессу менять оба этих атрибута.

В завершение данной главы мы обсудим библиотечные функции, которые применяются для анализа имен путей и их разбора на компоненты, содержащие имена каталога и файла.

18.1. Каталоги и (жесткие) ссылки

Каталог хранится в файловой системе, подобно обычному файлу. Две особенности отличают каталог от обычного файла:

- для каталога указывается другой тип файла в записи индексного дескриптора (см. раздел 14.4);
- каталог является файлом с особым упорядочением. По сути, это таблица, состоящая из имен файлов и номеров индексных дескрипторов.

В большинстве нативных файловых систем Linux имена файлов могут быть длиной до 255 символов. Взаимосвязь между каталогами и индексными дескрипторами показана на рис. 18.1, иллюстрирующем часть содержимого таблицы индексных дескрипторов файловой системы и соответствующих файлов каталогов, которые предназначены для файла из примера (`/etc/passwd`).

Несмотря на то что процесс может открыть каталог, он не может применять системный вызов read() для чтения его содержимого. Для извлечения содержимого каталога процесс должен использовать системные вызовы и библиотечные функции, обсуждаемые далее в этой главе. (В некоторых реализациях UNIX можно выполнять системный вызов read() для каталога, но такое поведение не является портируемым.) Не может процесс и изменить содержимое каталога с помощью системного вызова write(); он может лишь косвенно (то есть через запрос к ядру) менять содержимое благодаря таким системным вызовам, как open() (для создания нового файла), link(), mkdir(), symlink(), unlink() и rmdir(). Все эти системные вызовы описаны далее в главе, за исключением open(), который был описан в разделе 4.3.)

Нумерация в таблице индексных дескрипторов начинается с 1, а не с 0, поскольку значение 0 в поле индексного дескриптора для каталога говорит о том, что данная запись не используется. Индексный дескриптор 1 применяется для маркировки неисправных блоков в файловой системе. Корневой каталог файловой системы (/) всегда хранится в записи индексного дескриптора 2 (как показано на рис. 18.1), чтобы ядро знало, откуда начинать анализ имени пути.

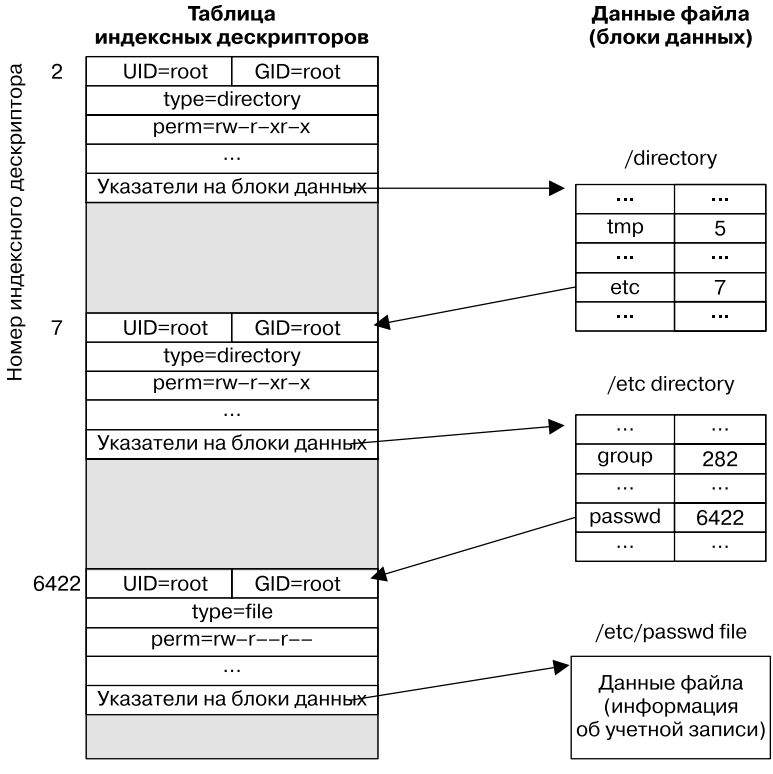


Рис. 18.1. Взаимосвязь между структурами индексных дескрипторов и каталогов для файла /etc/passwd

Если обратиться к списку информации, хранимой в индексном дескрипторе файла (раздел 14.4), можно увидеть, что индексный дескриптор не содержит имени файла; оно определяется только путем сопоставления внутри списка каталога. Это приводит к удобному следствию: можно создавать несколько имен — в том же каталоге или в каком-либо другом, — соотносящихся с одним и тем же индексным дескриптором. Такие имена называются *ссылками*, или *жесткими ссылками*, чтобы отличать их от символических ссылок, которые мы рассмотрим позже.

> Все нативные файловые системы Linux и UNIX поддерживают жесткие ссылки. Тем не менее многие файловые системы, отличные от UNIX (например, VFAT, разработанная компанией Microsoft), их не поддерживают. (Однако файловая система NTFS компании Microsoft все же поддерживает жесткие ссылки.)

Работая в оболочке, можно создать новые жесткие ссылки на существующий файл с помощью команды `ln`, как показано в следующем сеансе:

```
$ echo –n 'It is good to collect things,' > abc
$ ls –li abc
 122232 –rw-r–r--    1 mtk        users            29 Jun 15 17:07 abc
$ ln abc xyz
$ echo ' but it is better to go on walks.' >> xyz
$ cat abc
```

```
It is good to collect things, but it is better to go on walks.
$ ls –li abc xyz
 122232 –rw-r–r--    2 mtk        users            63 Jun 15 17:07 abc
 122232 –rw-r–r--    2 mtk        users            63 Jun 15 17:07 xyz
```

Номера индексных дескрипторов, выводимые (как первый столбец) командой `ls -li`, подтверждают то, что уже было видно из результатов работы команды `cat`: имена `abc` и `xyz` соотносятся с одной и той же записью индексного дескриптора и, следовательно, с одним и тем же файлом. В третьем поле, выводимом командой `ls -li`, можно увидеть счетчик ссылок для данного индексного дескриптора. После выполнения команды `ln abc xyz` счетчик ссылок для индексного дескриптора, соотносящегося с файлом `abc`, возрастает до 2, поскольку теперь на этот файл ссылаются два имени. (Такой же счетчик ссылок отображается для файла `xyz`, поскольку он соотносится с тем же индексным дескриптором.)

Если удалить одно из этих имен файлов, второе имя и сам файл останутся:

```
$ rm abc
$ ls –li xyz
 122232 –rw-r–r–    1 mtk        users            63 Jun 15 17:07 xyz
```

Запись индексного дескриптора и блоки данных для файла удаляются (освобождаются), только когда счетчик ссылок становится нулевым — то есть после удаления всех имен данного файла. Подведем итог: команда `rm` удаляет имя файла из списка каталога, уменьшает на 1 счетчик ссылок в соответствующем индексном дескрипторе и, если этот счетчик ссылок становится нулевым, освобождает индексный дескриптор и блоки данных, с которыми он соотносится.

Все имена (ссылки) для какого-либо файла являются эквивалентными: ни у одного из них (например, у первого) нет преимущества над остальными. Как мы увидели в приведенном выше примере, после удаления первого имени, относящегося к файлу, физический файл продолжает существовать, но доступен только путем использования другого имени.

На онлайн-форумах часто задают такой вопрос: «Как я могу в своей программе отыскать имя файла, связанное с файловым дескриптором X?» Краткий ответ: никак (по меньшей мере нельзя добиться портируемости и однозначности), поскольку дескриптор соотносится с индексным, а ему может соответствовать несколько имен файлов (или даже ни одного, как сказано в разделе 18.3).

> В Linux можно увидеть, какие файлы открыты процессом в настоящее время, с помощью команды readdir() (раздел 18.8), сканирующей содержимое характерного для Linux каталога /proc/PID/fd, содержащего символические ссылки для каждого файлового дескриптора, открытого процессом в данный момент. Для этой цели можно применять также инструменты lsof(1) и fuser(1), которые были портированы во многие UNIX-системы.

У жестких ссылок есть два ограничения, которые можно обойти за счет использования символических ссылок:

- поскольку записи каталога (жесткие ссылки) соотносятся с файлами только с помощью номера индексного дескриптора, а такие номера являются уникальными лишь в пределах одной файловой системы, жесткая ссылка должна находиться в той же файловой системе, что и файл, на который она указывает;
- невозможно создать жесткую ссылку на каталог. Это предотвращает создание циклических ссылок, которые создавали бы помехи в работе многих системных программ.

> В ранних реализациях UNIX суперпользователю разрешалось создавать жесткие ссылки на каталоги. Это было необходимо, поскольку в тех версиях ОС не было системного вызова mkdir(). Вместо него применяли системный вызов mknod(), а затем создавали ссылки

для записей . и .. ([Vahalia, 1996]). И хотя данная особенность уже не нужна, в некоторых современных реализациях UNIX она сохранена для обратной совместимости.

Результат, подобный жестким ссылкам на каталоги, можно получить с помощью связанного монтирования (см. подраздел 14.9.4).

18.2. Символические (мягкие) ссылки

Символическая ссылка, иногда называемая *мягкой ссылкой*, является специальным типом файла, данные которого — это имя другого файла. На рис. 18.2 проиллюстрирована ситуация, в которой две жесткие ссылки, `/home/erena/this` и `/home/allyn/that`, ссылаются на один файл, а символическая ссылка `/home/kiran/other` соотносится с именем `/home/erena/this`.

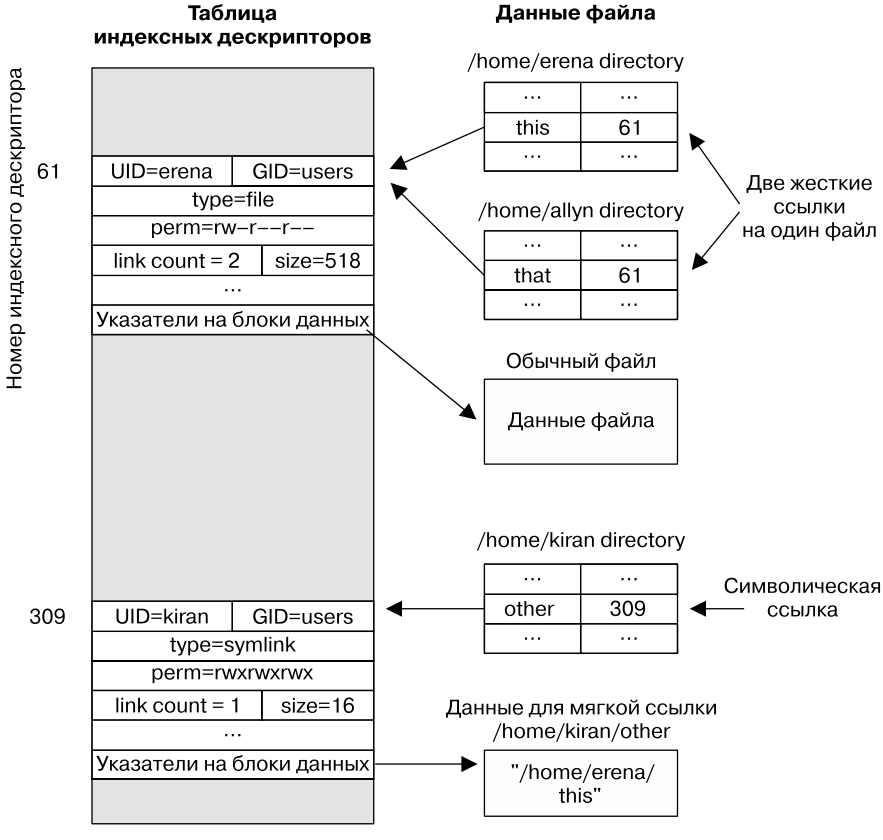


Рис. 18.2. Представление жестких и символических ссылок

В сеансе оболочки символические ссылки создаются с помощью команды `ln -s`. Команда `ls -F` отображает завершающий символ @ в конце символических ссылок.

Имя пути, с которым соотносится символическая ссылка, может быть либо абсолютным, либо относительным. Относительная символическая ссылка интерпретируется по отношению к местоположению самой ссылки.

Символические ссылки не обладают таким же статусом, что и жесткие ссылки. В частности, символическая ссылка не учитывается в счетчике ссылок для файла, на который указывает. (Таким образом, счетчик ссылок индексного дескриптора 61 на рис. 18.2 равен 2, а не 3.) Следовательно, если удалить имя файла, с которым соотносится символическая ссылка, то сама она останется, несмотря на то что ее нельзя будет разыменовать (проследовать по ней). О такой ссылке говорят, что она стала *зависшей*. Можно даже создать символическую ссылку на имя файла, который не существует на момент ее создания.

> Символические ссылки появились в стандарте 4.2BSD. И хотя они не были включены в стандарт POSIX.1-1990, впоследствии были приняты в стандарт SUSv1, а затем и в SUSv3.

Поскольку символическая ссылка соотносится с именем файла, а не с номером индексного дескриптора, ее можно использовать для указания на файл в иной файловой системе. Символические ссылки избавлены также от другого ограничения, характерного для жестких: можно создавать символические ссылки на каталоги. Такие инструменты, как `find` и `tar`, помогут различить жесткие и символические ссылки, а также либо по умолчанию не будут следовать по символическим, либо позволят избежать попадания в циклические ссылки, созданные с применением символических.

Можно создать цепочку символических ссылок (то есть `a` является символической ссылкой на `b`, которая является символической ссылкой на `c`). Когда такая ссылка указана в различных системных вызовах, работающих с файлами, ядро разыменовывает цепочки ссылок, чтобы добраться до окончательного файла.

Стандарт SUSv3 требует, чтобы реализация допускала по меньшей мере `_POSIX_SYMLOOP_MAX` разыменований для каждого компонента символической ссылки имени пути. Указанное значение `_POSIX_SYMLOOP_MAX` равно 8. Однако до версии ядра 2.6.18 Linux допускала не более пяти разыменований при следовании по цепочке символических ссылок. Начиная с указанной версии, Linux реализует минимальное число разыменований (8), указанное в стандарте SUSv3. Linux допускает также для всего имени пути общее число разыменований, равное 40. Эти пределы необходимы, чтобы избежать очень длинных цепочек из символических и циклических ссылок, вызывающих переполнение стека ядра при их анализе.

> Ряд файловых систем UNIX осуществляют оптимизацию, не упомянутую в основном тексте и не показанную на рис. 18.2. Когда общая длина строки, образующей содержимое символической ссылки, достаточно мала для того, чтобы уместиться в той части индексного дескриптора, которая обычно используется для указателей на данные, строка ссылки хранится там. Это позволяет избежать выделения блока на диске, а также ускоряет доступ к информации символической ссылки, поскольку она извлекается вместе с индексным дескриптором файла. Так, например, файловые системы ext2, ext3 и ext4 применяют данный метод для встраивания коротких символических строк в пределах 60 байт, которые обычно применяются для указателей на блоки данных. На практике такая оптимизация может оказаться весьма эффективной. Среди 20 700 символических ссылок в одной системе, исследованной автором, 97 % имели размер 60 байт и менее.

Интерпретация символической ссылки системными вызовами

Многие системные вызовы разыменовывают символические ссылки (следуют по ним) и таким образом работают с файлом, на который указывает ссылка. Отдельные вызовы не разыменовывают, а оперируют непосредственно ссылкой. При рассмотрении каждого системного вызова мы описываем его поведение по отношению к символическим ссылкам. Эти сведения подытожены также в табл. 18.1.

Таблица 18.1. Интерпретация символических ссылок различными функциями

Функция	Следует ли по ссылкам?	Примечания
access()	*	
acct()	*	
bind()	*	Сокеты домена UNIX имеют имена путей
chdir()	*	
chmod()	*	
chown()	*	
chroot()	*	
creat()	*	
exec()	*	
getxattr()	*	
lchown()		
lgetxattr()		
link()		См. раздел 18.3
listxattr()	*	
llistxattr()		
lremovexattr()		
lsetxattr()		
lstat()		
lutimes()		
open()	*	Если не указан флаг O_NOFOLLOW или O_EXCL | O_CREAT
opendir()	*	
pathconf()	*	
pivot_root()	*	
quotactl()	*	
readlink()		
removexattr()	*	
rename()		Ссылки не отслеживаются ни в одном из аргументов
rmdir()		Завершается с ошибкой ENOTDIR, если аргумент является символической ссылкой
setxattr()	*	
stat()	*	
statfs(), statvfs()	*	
swapon(), swapoff()	*	

Функция	Следует ли по ссылкам?	Примечания
truncate()	*	
unlink()		
uselib()	*	
utime(), utimes()	*	

В ряде случаев, когда необходимо обеспечить одинаковое функционирование как для файла, с которым соотносится символическая ссылка, так и для нее самой, применяют взаимоисключающие системные вызовы: один из них разыменовывает ссылку, а второй — нет, причем этот вызов снабжен префиксом в виде буквы `l`; например: `stat()` и `lstat()`.

Обычно выполняется следующее: символические ссылки, являющиеся частью имени пути, представляющей каталоги (то есть все компоненты, которые предшествуют завершающему слешу), всегда разыменовываются. Так, в имени пути `/somedir/somesubdir/file` компоненты `somedir` и `somesubdir` всегда будут разыменованы, если являются символическими ссылками, а разыменование компонента `file` будет зависеть от того, какому системному вызову было передано имя пути.

В разделе 18.11 мы описываем набор системных вызовов, добавленных в версии Linux 2.6.16. Они расширяют функциональность ряда интерфейсов, приведенных в табл. 18.1. Для некоторых из этих системных вызовов можно управлять вариантами обработки символических ссылок с помощью аргумента flags.

Права доступа к файлу и принадлежность символических ссылок

Принадлежность и права доступа к символической ссылке игнорируются большинством операций (символические ссылки всегда создаются с предоставлением всех прав доступа). Вместо них при определении того, допустима ли какая-либо операция, используются принадлежность и права доступа к файлу, на который указывает ссылка. Принадлежность символической ссылки имеет значение, только когда сама ссылка удаляется или переименовывается в каталоге, для которого установлен закрепляющий бит прав доступа (см. подраздел 15.4.5).

18.3. Создание и удаление (жестких) ссылок: системные вызовы link() и unlink()

Системные вызовы `link()` и `unlink()` создают и удаляют жесткие ссылки.

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если системному вызову `link()` передать в аргументе `oldpath` имя пути к существующему файлу, этот системный вызов создаст новую ссылку, используя имя файла, указанное в аргументе `newpath`. Если файл `newpath` уже существует, он не перезаписывается; вместо этого результатом является ошибка `EEXIST`.

В Linux системный вызов `link()` не разыменовывает символические ссылки. Если аргумент `oldpath` является такой ссылкой, то по аргументу `newpath` создается новая жесткая ссылка на файл, соотнесенный с символической. (Другими словами, `newpath` также является символической ссылкой на тот же файл, с которым соотносится `oldpath`.) Данное поведение не соответствует стандарту SUSv3: в нем сказано, что все функции, выполняющие анализ имени пути, должны осуществлять разыменование символических ссылок, если не указано иное (а для системного вызова `link()` не указано исключение). Большинство других реализаций UNIX работает согласно указаниям стандарта SUSv3. Заметным исключением является ОС Solaris, которая по умолчанию обеспечивает такое же поведение, как и в Linux, но может также работать и в соответствии со стандартом SUSv3, если использовать соответствующие параметры компилятора. Такое несоответствие между реализациями приводит к заключению: в портируемых приложениях следует избегать передачи символической ссылки в аргумент `oldpath`.

Стандарт SUSv4 признает несоответствие между существующими реализациями и говорит о том, что выбор варианта обработки символических ссылок системным вызовом link() определяется реализацией. Стандарт SUSv4 добавляет также спецификацию системного вызова linkat(), выполняющего ту же задачу, что и link(), но имеющего аргумент flags, который можно использовать для управления разыменованием символических ссылок. Подробности см. в разделе 18.11.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Системный вызов `unlink()` удаляет ссылку (имя файла) и, если это была последняя ссылка на данный файл, удаляет также и его. Если ссылка, указанная в аргументе `pathname`, не существует, системный вызов `unlink()` завершается ошибкой `ENOENT`.

Нельзя использовать системный вызов `unlink()` для удаления каталога; для этой задачи нужны системные вызовы `rmdir()` или `remove()`, которые мы рассмотрим в разделе 18.6.

В стандарте SUSv3 сказано, что, когда в аргументе pathname указан каталог, системный вызов unlink() должен завершаться ошибкой EPERM. Тем не менее в Linux системный вызов unlink() завершается в таком случае с ошибкой EISDIR. (Младший значащий бит явным образом допускает такое отклонение от стандарта SUSv3.) В портируемом приложении следует предусмотреть обработку каждой из этих ошибок.

Системный вызов `unlink()` не разыменовывает символические ссылки. Если аргумент `pathname` является такой ссылкой, то удаляется она сама, а не имя, на которое она указывает.

Открытый файл удаляется, только когда закрыты все файловые дескрипторы

Помимо отслеживания счетчика ссылок для каждого индексного дескриптора ядро считает также открытые файловые дескрипторы данного файла (см. рис. 5.2). Если удалена последняя ссылка на файл, но какие-либо процессы удерживают открытыми дескрипторы, относящиеся к этому файлу, то он не будет фактически удален до тех пор, пока не будут закрыты все дескрипторы. Данная особенность удобна, поскольку позволяет разорвать связь с файлом, не заботясь о том, открыт ли он каким-либо другим процессом. (Тем не

менее нельзя заново связать имя с открытым файлом, счетчик ссылок которого стал нулевым.) Кроме того, можно осуществить, например, такую хитрость: создать и открыть временный файл, немедленно разорвать связь с ним, а затем продолжить использовать его внутри программы, опираясь на тот факт, что данный файл будет удален, только когда мы закроем файловый дескриптор — либо явным образом, либо неявно, при выходе из программы. (Именно так работает функция `tmpfile()`, описанная в разделе 5.12.)

Программа в листинге 18.1 демонстрирует, что, даже когда удалена последняя ссылка на файл, сам он удаляется только после закрытия всех файловых дескрипторов, соотнесенных с ним.

Листинг 18.1. Удаление ссылки с помощью системного вызова unlink()

dirs_links/t_unlink.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define CMD_SIZE 200
#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int fd, j, numBlocks;
    char shellCmd[CMD_SIZE];     /* Команда, передаваемая вызову system() */
    char buf[BUF_SIZE];          /* Случайные байты для записи в файл */

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s temp-file [num-1kB-blocks] \n", argv[0]);
    numBlocks = (argc > 2) ? getInt(argv[2], GN_GT_0, "num-1kB-blocks")
                           : 100000;

    fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("open");

    if (unlink(argv[1]) == -1)         /* Удаляем имя файла */
        errExit("unlink");

    for (j = 0; j < numBlocks; j++)    /* Заполняем файл мусором */
        if (write(fd, buf, BUF_SIZE) != BUF_SIZE)
            fatal("partial/failed write");

    snprintf(shellCmd, CMD_SIZE, "df -k `dirname %s`", argv[1]);
    system(shellCmd);                /* Просмотр пространства,
                                        занятого в файловой системе */

    if (close(fd) == -1)             /* Теперь файл удален */
        errExit("close");
    printf("********** Closed file descriptor\n");

    system(shellCmd);      /* Повторный просмотр пространства,
                              занятого в файловой системе */
    exit(EXIT_SUCCESS);
}
```

dirs_links/t_unlink.c

Программа в листинге 18.1 принимает через командную строку два аргумента. Первый идентифицирует имя файла, который следует создать. Программа открывает этот файл, а затем немедленно разрывает связь с его именем. И хотя оно пропадает, сам файл продолжает существовать. Затем программа записывает произвольные блоки данных в этот файл. Количество их указано в необязательном втором аргументе командной строки. Здесь программа применяет команду `df(1)` для отображения величины занятого пространства в файловой системе.

Затем программа закрывает файловый дескриптор, и в этот момент происходит удаление файла. Далее она еще раз применяет команду `df(1)`, чтобы показать уменьшение величины использованного дискового пространства. Следующий сеанс работы в оболочке демонстрирует применение программы из листинга 18.1:

```
$ ./t_unlink /tmp/tfile 1000000
Filesystem           1K-blocks        Used Available Use% Mounted on
/dev/sda10             5245020     3204044   2040976  62% /
********** Closed file descriptor
Filesystem           1K-blocks        Used Available Use% Mounted on
/dev/sda10             5245020     2201128   3043892  42% /
```

В листинге 18.1 задействована функция system() для выполнения команды оболочки. Мы подробно опишем эту функцию в разделе 27.6.

18.4. Изменение имени файла: системный вызов rename()

Системный вызов `rename()` можно использовать как для переименования файла, так и для его перемещения в другой каталог той же файловой системы.

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `oldpath` — существующее имя пути, которое будет переименовано на указанное в аргументе `newpath`.

Системный вызов `rename()` оперирует записями каталога; он не перемещает данные файла. Переименование не отражается на других жестких ссылках, связанных с этим файлом, оно не влияет также ни на какие процессы, удерживающие открытыми дескрипторы для данного файла, поскольку такие дескрипторы соотносятся с открытыми файловыми дескрипторами, которые (после вызова `open()`) не связаны с именами файлов.

С применением системного вызова `rename()` связаны следующие правила.

- Если файл по адресу `newpath` уже существует, то перезаписывается.
- Если аргументы `newpath` и `oldpath` ссылаются на один и тот же файл, то никаких изменений не производится (и вызов завершается успешно). Данное правило противоречит здравому смыслу. Если отталкиваться от предыдущего пункта, то можно было бы ожидать, что при наличии двух имен файлов `x` и `y` системный вызов `rename("x", "y")` удалил бы имя `x`. Но это не так, если `x` и `y` являются ссылками на один и тот же файл.

Разумным объяснением этого правила, восходящим к реализации BSD, является, вероятно, желание упростить проверки, которые должно было выполнять ядро, чтобы такие системные вызовы, как rename("x", "x"), rename("x", "./x") и rename("x", "somedir/../x"), не удаляли файл.

- Системный вызов `rename()` не разыменовывает символические ссылки в обоих аргументах. Если аргумент `oldpath` является символической ссылкой, она переименовывается. Если аргумент `newpath` является символической ссылкой, то она обрабатывается как обычное имя пути, на которое следует переименовать имя `oldpath` (то есть существующая символическая ссылка `newpath` удаляется).
- Если аргумент `oldpath` ссылается на файл, который не является каталогом, то в этом случае в аргументе `newpath` нельзя указывать имя пути для каталога (ошибка `EISDIR`). Чтобы переименовать файл, указав на местоположение внутри каталога (то есть переместить файл в другой каталог), аргумент `newpath` должен содержать новое имя файла. Следующий системный вызов одновременно перемещает файл в другой каталог и меняет его имя:

```
rename("sub1/x", "sub2/y");
```

- При указании имени каталога в аргументе `oldpath` появляется возможность переименовать этот каталог. В таком случае ссылка `newpath` должна либо быть несуществующей, либо являться именем пустого каталога. Если аргумент `newpath` — существующий файл или существующий непустой каталог, то возникнет ошибка (соответственно `ENOTDIR` и `ENOTEMPTY`).
- Если аргумент `oldpath` является каталогом, то аргумент `newpath` не может содержать тот же префикс каталога, что и у аргумента `oldpath`. Например, мы не смогли бы переименовать `/home/mtk` на `/home/mtk/bin` (ошибка `EINVAL`).
- Файлы, с которыми соотносятся аргументы `oldpath` и `newpath`, должны располагаться в одной файловой системе. Это необходимо, поскольку каталог является списком жестких ссылок, указывающих на индексные дескрипторы в той же файловой системе, где расположен данный каталог. Как было сказано ранее, системный вызов `rename()` всего лишь оперирует содержимым списков каталога. При попытке переименовать файл с его переносом в другую файловую систему возникнет ошибка `EXDEV`. (Чтобы добиться желаемого результата, следует скопировать содержимое данного файла из одной файловой системы в другую, а затем удалить старый файл. Именно это выполняет команда `mv`.)

18.5. Работа с символическими ссылками: системные вызовы symlink() и readlink()

Теперь рассмотрим системные вызовы, которые применяются для создания символических ссылок и проверки их содержимого.

Системный вызов `symlink()` создает новую символическую ссылку `linkpath` для имени пути, указанного в аргументе `filepath`. (Чтобы удалить ее, используется системный вызов `unlink()`.)

```
#include <unistd.h>

int symlink(const char *filepath, const char *linkpath);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если имя пути, переданное в аргумент `linkpath`, уже существует, системный вызов завершается ошибкой (для переменной `errno` устанавливается значение `EEXIST`). Имя пути, указанное в аргументе `filepath`, может быть абсолютным или относительным.

Файл или каталог, указанные в аргументе `filepath`, могут и не существовать в момент совершения системного вызова. И даже если они существуют, ничто не препятствует их удалению впоследствии. Тогда ссылка `linkpath` становится *зависшей* и попытки ее разыменования с помощью других системных вызовов приведут к ошибке (как правило, `ENOENT`).

Если мы укажем символическую ссылку в качестве аргумента `pathname` системного вызова `open()`, то он откроет файл, на который указывает эта ссылка. Иногда может потребоваться извлечь содержимое самой ссылки, то есть имя пути, с которым она соотносится. Данную задачу выполняет системный вызов `readlink()`, помещающий копию строки символической ссылки в символьный массив, на который указывает аргумент `buffer`.

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buffer, size_t bufsiz);
```

Возвращает количество байтов, помещенных в массиве ***buffer***, при успешном завершении или `-1` при ошибке

Аргумент `bufsiz` является целым числом, которое используется, чтобы сообщить системному вызову `readlink()` количество байтов, доступных в массиве `buffer`.

Если не возникает ошибка, системный вызов `readlink()` возвращает количество байтов, фактически размещенных в массиве `buffer`. Если длина ссылки превышает величину `bufsiz`, в этот массив помещается обрезанная строка (а системный вызов `readlink()` возвращает размер этой строки — то есть `bufsiz`).

Поскольку в конце массива `buffer` не помещен завершающий нулевой байт, нет способа отличить обрезанную строку, возвращенную системным вызовом `readlink()`, от строки, которая в точности заполняет массив `buffer`. Один из вариантов проверки состоит в том, чтобы выделить массив `buffer` большего размера, а затем еще раз выполнить вызов `readlink()`. В другом варианте можно указать размер для имени пути с помощью константы `PATH_MAX` (описанной в разделе 11.1), определяющей длину самого длинного имени пути, который должна воспринимать программа.

Пример использования системного вызова `readlink()` приведен в листинге 18.4.

Стандарт SUSv3 определяет новое ограничение, SYMLINK_MAX, — его должна задавать конкретная реализация, чтобы указать максимальное количество байтов, которое может хранить символическая ссылка. Рекомендуемая величина данного ограничения — не менее 255 байт. На момент написания книги Linux не задает это ограничение. В основном тексте мы предлагаем использовать константу PATH_MAX, поскольку указанное ограничение должно быть по меньшей мере таким же, что и SYMLINK_MAX.

В стандарте SUSv2 для значения, возвращаемого системным вызовом readlink(), указан тип int, и многие современные реализации (а также версии Linux на основе библиотеки glibc) следуют этой спецификации. В стандарте SUSv3 тип возвращаемого значения изменен на ssize_t.

18.6. Создание и удаление каталогов: системные вызовы mkdir() и rmdir()

Системный вызов `mkdir()` создает новый каталог.

Аргумент `pathname` задает имя пути для нового каталога. Оно может быть относительным или абсолютным. Если файл с таким именем пути уже существует, системный вызов завершается с ошибкой `EEXIST`.

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Принадлежность нового каталога устанавливается в соответствии с правилами, описанными в подразделе 15.3.1.

Аргумент `mode` задает права доступа для нового каталога. (Мы описываем значение битов прав доступа для каталогов в подразделах 15.3.1, 15.3.2 и 15.4.5.) Эту битовую маску можно указать, применив операцию ИЛИ (`|`) с константами из табл. 15.4, но, как и для системного вызова `open()`, ее можно указать и восьмеричным числом. Значение, переданное в аргумент `mode`, объединяется с помощью операции И с маской процесса (подраздел 15.4.6). Кроме того, бит set-user-ID (`S_ISUID`) всегда отключен, поскольку для каталогов не имеет смысла.

Если в аргумент `mode` передан закрепляющий бит (`S_ISVTX`), то он будет установлен для нового каталога.

Значение бита set-group-ID (`S_ISGID`) в аргументе `mode` игнорируется. Вместо этого, если данный бит установлен для родительского каталога, он будет также установлен для нового каталога. В подразделе 15.3.1 было отмечено: установка бита прав доступа set-group-ID для каталога приводит к тому, что новые файлы, создаваемые в данном каталоге, заимствуют GID от идентификатора группы для каталога, а не от действующего GID для процесса. Системный вызов `mkdir()` распространяет бит прав доступа set-group-ID описанным здесь образом, чтобы все подкаталоги данного каталога вели себя одинаково.

Стандарт SUSv3 явно отмечает, что способ обращения системного вызова `mkdir()` с битами set-user-ID, set-group-ID и с закрепляющим битом определяется реализацией. В некоторых реализациях UNIX эти три бита всегда отключены для нового каталога.

Только что созданный каталог содержит две записи: `.` (точка), которая является ссылкой на сам каталог, и `..` (две точки) — это ссылка на родительский каталог.

> Стандарт SUSv3 не требует наличия записей . и .. в каталогах. Он требует лишь того, чтобы в реализации системы корректно интерпретировались эти точки, когда они появляются в именах путей. Портируемое приложение не должно полагаться на существование таких записей в каталоге.

Системный вызов `mkdir()` создает только последний компонент имени `pathname`. Другими словами, вызов `mkdir("aaa/bbb/ccc", mode)` завершится успешно, только если каталоги `aaa` и `aaa/bbb` уже существуют. (Это соответствует принятому по умолчанию действию команды `mkdir(1)`, однако данная команда имеет также параметр `-p` для создания всех промежуточных имен каталогов, если они не существуют.)

> GNU-библиотека C содержит функцию mkdtemp(template) — аналог функции mkstemp(), но для каталогов. Она создает каталог с уникальным именем, владельцу которого предоставлены права доступа на чтение, запись и выполнение, а всем остальным пользователям не предоставлено никаких прав. Вместо файлового дескриптора системный вызов mkdtemp() возвращает указатель на измененную строку, содержащую фактическое имя каталога в аргументе template. Стандарт SUSv3 не определяет эту функцию, и она недоступна во всех реализациях UNIX. Она определена в стандарте SUSv4.

Системный вызов `rmdir()` удаляет каталог, указанный в имени пути `pathname`, который может быть абсолютным или относительным.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Для успешного завершения системного вызова `rmdir()` необходимо, чтобы каталог был пустым. Если завершающий компонент имени `pathname` является символической ссылкой, она не разыменовывается; в результате возникает ошибка `ENOTDIR`.

18.7. Удаление файла или каталога: функция remove()

Библиотечная функция `remove()` удаляет файл или пустой каталог.

```
#include <stdio.h>

int remove(const char *pathname);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если аргумент `pathname` представляет имя файла, то функция `remove()` выполняет системный вызов `unlink()`; если данный аргумент является каталогом, то функция `remove()` выполняет системный вызов `rmdir()`.

Подобно системным вызовам `unlink()` и `rmdir()`, функция `remove()` не разыменовывает символические ссылки. Если аргумент `pathname` является такой ссылкой, то данная функция удаляет саму ссылку, а не файл, который соотносится с ней.

Если нужно удалить файл, чтобы создать новый файл с тем же именем, то использовать функцию `remove()` проще, чем создать программный код, который проверяет, указывает ли имя пути на файл или на каталог, а затем выполняет системный вызов `unlink()` или `rmdir()`.

Функция remove() была введена в стандартную библиотеку C, которая реализована как в системах UNIX, так и в других. В большинстве систем, отличных от UNIX, не поддерживаются жесткие ссылки, и поэтому удаление файлов с помощью функции unlink() не имело бы смысла.

18.8. Чтение каталогов: функции opendir() и readdir()

Библиотечные функции, описанные в этом разделе, можно использовать для открытия каталога и поочередного извлечения имен файлов, содержащихся в ней.

Библиотечные функции для чтения каталогов основаны на системном вызове getdents() (не являющемся частью стандарта SUSv3), однако обеспечивают интерфейс, более удобный в использовании. В Linux есть также системный вызов readdir(2) (в отличие от биб-

лиотечной функции readdir(3), описанной здесь), который выполняет ту же задачу, что и getdents(), но является устаревшим.

Функция `opendir()` открывает каталог и возвращает описатель, который можно применять для ссылки на каталог в последующих вызовах.

```
#include <dirent.h>

DIR *opendir(const char *dirpath);
```

Возвращает описатель потока каталога или `NULL` при ошибке

Функция `opendir()` открывает каталог, указанный в аргументе `dirpath`, и возвращает указатель на структуру типа `DIR`. Эта структура представляет собой так называемый *поток каталога*; вызывающий процесс передает данный описатель другим функциям, рассмотренным ниже. После получения результата функции `opendir()` поток каталога размещается в первой записи списка каталога.

Функция `fdopendir()` подобна функции `opendir()`, за исключением того, что каталог, для которого следует создать поток, указывается с помощью открытого файлового дескриптора `fd`.

```
#include <dirent.h>

DIR *fdopendir(int fd);
```

Возвращает описатель потока каталога или `NULL` при ошибке

Функция `fdopendir()` предназначена для того, чтобы приложения могли избежать режима соперничества, описанного в разделе 18.11.

После успешного завершения вызова функции `fdopendir()` этот файловый дескриптор находится под управлением системы, и программа не должна осуществлять доступ к нему иначе, чем с помощью функций, описанных в оставшейся части данного раздела.

Функция `fdopendir()` определена в стандарте SUSv4 (но не в стандарте SUSv3).

Функция `readdir()` считывает последовательные записи из потока каталога.

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

Возвращает указатель на статически выделенную структуру, описывающую следующую запись каталога, или `NULL` при достижении конца каталога или при ошибке

При каждом вызове функции `readdir()` происходит считывание следующего каталога из потока каталога, на который указывает аргумент `dirp`, и возвращается указатель на статически выделенную структуру типа `dirent`, содержащую следующую информацию о записи:

```
struct dirent {
    ino_t d_ino;       /* Номер индексного дескриптора файла */
    char d_name[];     /* Имя файла с завершающим нулем */
};
```

Данная структура перезаписывается при каждом вызове функции `readdir()`.

Я опустил в приведенном выше определении разные нестандартные поля структуры dirent для Linux, поскольку их использование делает приложение непортируемым. Наиболее интересным из этих нестандартных полей является поле d_type, которое присутствует также в BSD-ветках, но не в других UNIX-реализациях. Данное поле содержит значение, указывающее на тип файла, названного в аргументе d_name: DT_REG (обычный файл), DT_DIR (каталог), DT_LNK (символическая ссылка) либо DT_FIFO (очередь FIFO). (Приведенные имена аналогичны макроопределениям в табл. 15.1.) Применение информации из этого поля избавляет от необходимости совершать вызов lstat(), чтобы выяснить тип файла. Необходимо отметить, однако, что на момент написания книги это поле полностью поддерживается только в файловых системах Btrfs, ext2, ext3 и ext4.

Дальнейшую информацию о файле, на который указывает аргумент `d_name`, можно получить благодаря системному вызову `stat()` для имени пути, составленному с помощью аргумента `dirpath`, указанного для функции `opendir()`, сцепленного (с помощью слеша, `/`) со значением, возвращаемым в поле `d_name`.

Имена файлов, возвращаемые функцией `readdir()`, располагаются не в порядке сортировки, а в порядке появления в каталоге. (Это зависит от того порядка, в каком файловая система добавляет файлы в данный каталог и как она заполняет пустоты в списке каталога после удаления файлов.) (Команда `ls -f` перечисляет файлы в таком же неупорядоченном списке, какой был бы отображен функцией `readdir()`.)

Можно использовать функцию scandir(3) для извлечения упорядоченного списка файлов, который удовлетворяет критериям, задаваемым программистом; см. подробности на страницах руководства. И хотя функция scandir() не описана в стандарте SUSv3, она присутствует во многих реализациях UNIX.

При достижении конца каталога или при ошибке функция `readdir()` возвращает `NULL`; в последнем случае для указания на ошибку присваивается значение переменной `errno`. Для различения этих двух случаев можно написать такой код:

```
errno = 0;
direntp = readdir(dirp);

if (direntp == NULL) {
    if (errno != 0) {
        /* Обработка ошибки */
    } else {
        /* Достижение конца каталога */
    }
}
```

Если содержимое каталога меняется, пока программа сканирует его с помощью функции `readdir()`, то эта программа может не заметить изменений. Стандарт SUSv3 явным образом отмечает, что возникает неопределенность: возвратит ли функция `readdir()` имя файла, который был добавлен в данный каталог или был удален из него за время, прошедшее с момента последнего вызова функции `opendir()` или `rewinddir()`? Все имена файлов, которые не были добавлены или удалены с момента последнего вызова, гарантированно возвращаются.

Функция `rewinddir()` перемещает поток каталога обратно к началу, чтобы следующий вызов функции `readdir()` начинал работу с первого файла в данном каталоге.

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

Функция `closedir()` закрывает открытый поток каталога, на который указывает аргумент `dirp`, высвобождая ресурсы, использованные этим потоком.

```
#include <dirent.h>

int closedir(DIR *dirp);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Следующие две функции, `telldir()` и `seekdir()`, которые также определены в стандарте SUSv3, разрешают произвольный доступ внутри потока каталога. Дополнительную информацию об этих функциях см. на страницах руководства.

Потоки каталогов и файловые дескрипторы

Поток каталога обладает связанным с ним файловым дескриптором. Функция `dirfd()` возвращает дескриптор, относящийся к потоку каталога, на который указывает аргумент `dirp`.

```
#include <dirent.h>

int dirfd(DIR *dirp);
```

Возвращает файловый дескриптор при успешном завершении или `-1` при ошибке

Можно было бы, например, передать файловый дескриптор, возвращенный функцией `dirfd()`, функции `fchdir()` (см. раздел 18.10), чтобы изменить текущий рабочий каталог процесса на соответствующий. В другом варианте можно было бы передать файловый дескриптор как аргумент `dirfd` в одной из функций, описанных в разделе 18.11.

Функция `dirfd()` появляется также в версиях BSD, однако присутствует в немногих других реализациях. Она не определена в стандарте SUSv3, но описана в стандарте SUSv4.

Здесь уместно отметить: функция `opendir()` автоматически устанавливает флаг close-on-exec (`FD_CLOEXEC`) для файлового дескриптора, связанного с потоком каталога. Это дает гарантию того, что данный файловый дескриптор будет автоматически закрыт после выполнения вызова `exec()`. (Стандарт SUSv3 требует такого поведения.) Мы рассмотрим флаг close-on-exec в разделе 27.4.

Пример программы

В листинге 18.2 применены функции `opendir()`, `readdir()` и `closedir()` для вывода содержимого каталогов, указанных в командной строке (или в рабочем каталоге, если аргументы не указаны). Ниже приведен пример использования этой программы:

```
$ mkdir sub                 Создаем тестовый каталог
$ touch sub/a sub/b         Создаем несколько файлов в тестовом каталоге
$ ./list_files sub          Выводим содержимое каталога
sub/a
sub/b
```

Листинг 18.2. Сканирование каталога

dirs_links/list_files.c

```
#include <dirent.h>
#include "tlpi_hdr.h"

static void             /* Перечисляет все файлы в каталоге 'dirPath' */
listFiles(const char *dirpath)
{
    DIR *dirp;
    struct dirent *dp;
    Boolean isCurrent;          /* Истина, если 'dirpath' равен "." */

    isCurrent = strcmp(dirpath, ".") == 0;

    dirp = opendir(dirpath);
    if (dirp    == NULL) {
        errMsg("opendir failed on '%s'", dirpath);
        return;
    }

    /* Для каждой записи в этом каталоге выводим имя каталога + имя файла */

    for (;;) {
        errno = 0;    /* Чтобы отличить ошибку от достижения конца папки */
        dp = readdir(dirp);
        if (dp == NULL)
            break;

        if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
            continue;            /* Пропускаем . и .. */

        if (!isCurrent)
            printf("%s/", dirpath);
        printf("%s\n", dp->d_name);
    }

    if (errno != 0)
        errExit("readdir");

    if (closedir(dirp) == -1)
        errMsg("closedir");
}

int
main(int argc, char *argv[])
{
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [dir-path...]\n", argv[0]);

    if (argc == 1)        /* Аргументов нет – используем текущий каталог */
        listFiles(".");
    else
        for (argv++; *argv; argv++)
            listFiles(*argv);

    exit(EXIT_SUCCESS);
}
```

dirs_links/list_files.c

Функция readdir_r()

Функция `readdir_r()` является вариантом функции `readdir()`. Ключевым семантическим различием между этими функциями является то, что первая допускает многократный ввод, а вторая — нет. Это связано вот с чем: функция `readdir_r()` возвращает запись файла через аргумент `entry`, назначаемый вызывающим процессом, а функция `readdir()` возвращает информацию, задействуя указатель на статически выделенную структуру. Мы рассмотрим многократный ввод в подразделе 21.1.2 и разделе 31.1.

```
#include <dirent.h>

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

Возвращает `0` при успешном завершении
или положительное значение при ошибке

При заданном аргументе `dirp`, являющемся потоком каталога, открытым ранее с помощью функции `opendir()`, функция `readdir_r()` помещает информацию о следующей записи каталога в структуру `dirent`, на которую указывает аргумент `entry`. Кроме того, в аргумент `result` помещается указатель на эту структуру. Если достигнут конец потока каталога, в аргумент `result` помещается значение `NULL` (и функция `readdir_r()` возвращает `0`). В случае ошибки функция `readdir_r()` возвращает не `-1`, а положительное число, соответствующее одному из значений кода `errno`.

В Linux поле `d_name` структуры `dirent` определено как массив из 256 байт, длины которого достаточно для хранения самого длинного возможного имени файла. Несмотря на то что в ряде реализаций UNIX для размера поля `d_name` определено такое же значение, стандарт SUSv3 не оговаривает этот момент, в связи с чем в некоторых реализациях UNUX данное поле определено как однобайтный массив, а на вызывающую программу ложится задача по выделению структуры с корректным размером. При ее осуществлении следует задавать размер поля `d_name` на единицу больше (для завершающего нулевого байта), чем значение константы `NAME_MAX`. Таким образом, в портируемых приложениях структуру `dirent` следует выделять так:

```
struct dirent *entryp;
size_t len;

len = offsetof(struct dirent, d_name) + NAME_MAX + 1;
entryp = malloc(len);
if (entryp == NULL)
    errExit("malloc");
```

За счет использования макроопределения `offsetof()` (приводимого в файле `<stddef.h>`) удается избежать зависимостей, специфичных для конкретной реализации, выражающих число и размер полей структуры `dirent`, предшествующих полю `d_name` (которое всегда является последним полем в этой структуре).

> Макроопределение offsetof() принимает два аргумента — тип структуры и имя поля внутри нее — и возвращает значение типа size_t, которое является смещением (в байтах) данного поля относительно начала структуры. Это макроопределение необходимо, поскольку компилятор может вставить заполняющие байты в структуру, чтобы удовлетворить требованиям выравнивания для таких типов, как int, в результате чего смещение поля в структуре может оказаться больше, чем сумма размеров полей, которые ей предшествуют.

18.9. Обход дерева файлов: функция nftw()

Функция `nftw()` позволяет программе рекурсивно перемещаться по всему поддереву каталога, выполняя некую операцию (например, вызов некоторой функции, указанной программистом) для каждого файла в этом поддереве.

> Функция nftw() является улучшением старой функции ftw(), выполняющей похожую задачу. В новых приложениях следует использовать функцию nftw() (new ftw), поскольку она обеспечивает бульшую функциональность и предсказуемую обработку символических ссылок (стандарт SUSv3 разрешает функции ftw() либо следовать по символическим ссылкам, либо нет). Стандарт SUSv3 определяет обе функции, nftw() и ftw(), однако последняя считается устаревшей в стандарте SUSv4.
>
> GNU-библиотека C также обеспечивает API на основе BSD в виде функций fts (fts_open(), fts_read(), fts_children(), fts_set() и fts_close()). Они выполняют задачи подобно функциям ftw() и nftw(), но обеспечивают повышенную гибкость для приложения, просматривающего дерево. Тем не менее данный интерфейс не стандартизирован и присутствует лишь в ряде реализаций UNIX, которые не являются ветками BSD, и поэтому мы не рассматриваем его здесь.

Функция `nftw()` выполняет обход дерева каталога, указанного в аргументе `dirpath` и вызывает указанную программистом функцию `func` для каждого файла в дереве каталога.

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

int nftw(const char *dirpath,
         int (*func) (const char *pathname, const struct stat *statbuf,
                      int typeflag, struct FTW *ftwbuf),
         int nopenfd, int flags);

                    Возвращает 0 после успешного обхода всего дерева, -1 при ошибке
                  или первое ненулевое значение, возвращенное вызовом функции func
```

По умолчанию функция `nftw()` выполняет несортированный обход указанного дерева в прямом порядке, обрабатывая каждый каталог, прежде чем перейти к обработке файлов и подкаталогов внутри данного каталога.

При обходе дерева каталога функция `nftw()` открывает по меньшей мере один файловый дескриптор для каждого уровня этого дерева. Аргумент `nopenfd` задает максимальное количество файловых дескрипторов, которые может использовать функция `nftw()`. Если глубина каталога превосходит данный максимум, функция `nftw()` выполняет своего рода учет системных ресурсов, закрывая и заново открывая дескрипторы, чтобы избежать ситуации, при которой одновременно открыто более `nopenfd` дескрипторов (в результате функция работает медленнее). Необходимость этого аргумента была важна в старых реализациях UNIX; часть из них устанавливали предел — 20 открытых файловых дескрипторов на процесс. Современные реализации UNIX позволяют процессу открывать большое количество дескрипторов, и поэтому можно не скупиться на их количество (указав, скажем, 10 или больше).

Аргумент `flags` для функции `nftw()` создается с помощью операции ИЛИ (`|`), которая применяется к нескольким (или ни к одной) из перечисленных ниже констант, меняющих работу данной функции.

- FTW_CHDIR — выполнить функцию chdir() в каждом каталоге перед обработкой ее содержимого. Это удобно, если функция func предназначена для выполнения некой работы в том каталоге, где расположен файл, указанный в аргументе pathname.
- FTW_DEPTH — выполнить обход в обратном порядке в глубину дерева каталога. Это значит, что функция nftw() вызывает функцию func для всех файлов (и подкаталогов) внутри какого-либо каталога, прежде чем выполнить функцию func для самого каталога. (Название флага немного сбивает с толку: функция nftw() всегда выполняет обход дерева каталога сначала в глубину, а затем в ширину. Все, что делает данный флаг, заключается в изменении порядка обхода с прямого на обратный.)
- FTW_MOUNT — не перемещаться внутрь другой файловой системы. Следовательно, если один из подкаталогов данного дерева является точкой монтирования, его обход не выполняется.
- FTW_PHYS — по умолчанию функция nftw() разыменовывает символические ссылки. Данный флаг запрещает такое поведение. Вместо этого ссылка передается функции func со значением FTW_SL для аргумента typeflag, как описано ниже.

Для каждого файла функция nftw() передает четыре аргумента при вызове функции func. Первый из них, pathname, является именем пути к данному файлу. Это имя может быть абсолютным, если аргумент dirpath был указан как абсолютное имя пути, или относительным по отношению к текущему рабочему каталогу вызывающего процесса на момент вызова функции ntfw(), если аргумент dirpath выражен относительным именем пути. Второй аргумент, statbuf, является указателем на структуру stat (см. раздел 15.1), содержащую информацию о данном файле. Третий аргумент, typeflag, содержит дополнительную информацию о файле и имеет одно из следующих значений:

- FTW_D — каталог;
- FTW_DNR — каталог, который нельзя прочитать (и поэтому функция nftw() не обходит его каталоги-потомки);
- FTW_DP — мы выполняем обход каталога в обратном порядке (FTW_DEPTH), и текущий элемент является каталогом, файлы и подкаталоги которого уже были обработаны;
- FTW_F — файл любого типа, кроме каталога или символической ссылки;
- FTW_NS — вызов stat() для данного файла завершился ошибкой, вероятно, вследствие ограниченных прав доступа. Значение statbuf не определено;
- FTW_SL — символическая ссылка. Данное значение возвращается, только если функция nftw() вызвана с флагом FTW_PHYS;
- FTW_SLN — этот элемент является зависшей символической ссылкой. Данное значение появляется, только если не был указан флаг FTW_PHYS в качестве аргумента flags.

Четвертый аргумент функции func, ftwbuf, является указателем на структуру, определяемую следующим образом:

```
struct FTW {
    int base;       /* Смещение относительно базовой части имени пути */
    int level;      /* Глубина файла внутри обхода дерева */
};
```

Поле base данной структуры является целочисленным смещением компонента с именем файла (компонент, следующий за последним слешем) в аргументе pathname функции func. Поле level является глубиной этого элемента относительно начальной точки обхода (для которой уровень равен 0).

При каждом вызове функция func должна возвращать целочисленное значение, которое интерпретируется функцией nftw(). Если возвращен 0, то функция nftw() продолжает обход дерева, и если все вызовы функции func возвращают 0, то и сама nftw() возвращает 0 вызывающему процессу. Если возвращено ненулевое значение,

то функция nftw() немедленно прекращает обход дерева и возвращает упомянутое ненулевое значение.

Поскольку функция nftw() задействует динамически выделяемые структуры данных, единственный способ, с помощью которого программа может преждевременно прервать обход дерева, заключается в возврате ненулевого значения функцией func. Использование функции longjmp() (см. раздел 6.8) может привести к непредсказуемым результатам — по меньшей мере к утечке памяти в программе.

Пример программы

Листинг 18.3 демонстрирует применение функции nftw().

Листинг 18.3. Использование функции для обхода дерева каталога

dirs_links/nftw_dir_tree.c

```
#define _XOPEN_SOURCE 600
/* Получаем объявление функции nftw() и макроопределение S_IFSOCK */
#include <ftw.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s\n", msg);
    fprintf(stderr, "Usage: %s [-d] [-m] [-p]
          [directory-path]\n", progName);
    fprintf(stderr, "\t-d Use FTW_DEPTH flag\n");
    fprintf(stderr, "\t-m Use FTW_MOUNT flag\n");
    fprintf(stderr, "\t-p Use FTW_PHYS flag\n");
    exit(EXIT_FAILURE);
}

static int                      /* Функция, вызываемая функцией nftw() */
dirTree(const char *pathname, const struct stat *sbuf, int type,
        struct FTW *ftwb)
{
    switch (sbuf->st_mode & S_IFMT) {       /* Выводим тип файла */
    case S_IFREG:         printf("-"); break;
    case S_IFDIR:         printf("d"); break;
    case S_IFCHR:         printf("c"); break;
    case S_IFBLK:         printf("b"); break;
    case S_IFLNK:         printf("l"); break;
    case S_IFIFO:         printf("p"); break;
    case S_IFSOCK:        printf("s"); break;
    default:              printf("?"); break;
    /* Такое не должно произойти (в Linux) */
    }

    printf(" %s    ",
             (type == FTW_D)   ? "D  " : (type == FTW_DNR)  ? "DNR" :
             (type == FTW_DP)  ? "DP " : (type == FTW_F)    ? "F  " :
             (type == FTW_SL)  ? "SL " : (type == FTW_SLN)  ? "SLN" :
             (type == FTW_NS)  ? "NS " : "    ");

    if (type != FTW_NS)
        printf("%7ld ", (long) sbuf->st_ino);
```

```
    else
        printf("        ");

    printf(" %*s", 4 * ftwb->level, "");  /* Добавляем подходящий отступ */
    printf("%s\n",    &pathname[ftwb->base]);   /* Выводим базовое имя */
    return 0;
    /* Даем команду на продолжение работы функции nftw() */
}

int
main(int argc, char *argv[])
{
    int flags, opt;

    flags = 0;
    while ((opt = getopt(argc, argv, "dmp")) != -1) {
        switch (opt) {
        case 'd': flags |= FTW_DEPTH;        break;
        case 'm': flags |= FTW_MOUNT;        break;
        case 'p': flags |= FTW_PHYS;         break;
        default:    usageError(argv[0], NULL);
        }
    }

    if (argc > optind + 1)
        usageError(argv[0], NULL);

    if (nftw((argc > optind) ? argv[optind] : ".", dirTree, 10, flags)
          == -1) {
        perror("nftw");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

dirs_links/nftw_dir_tree.c

Программа в листинге 18.3 отображает снабженную отступами иерархию имен файлов в дереве каталога, по одному имени файла на строку, а также тип файла и номер индексного дескриптора. Можно использовать параметры командной строки, чтобы указать параметры аргумента `flags`, примененного для вызова функции `nftw()`. Следующий сеанс работы в оболочке демонстрирует примеры того, что мы увидим при запуске данной программы. Создадим сначала пустой подкаталог, который заполним файлами разных типов:

```
$ mkdir dir
$ touch dir/a dir/b           Создаем несколько обычных файлов
$ ln -s a dir/sl              и символическую ссылку,
$ ln -s x dir/dsl             а также зависшую символическую ссылку
$ mkdir dir/sub               и подкаталог
$ touch dir/sub/x             с собственным файлом,
$ mkdir dir/sub2              и еще один подкаталог,
$ chmod 0 dir/sub2            который нельзя прочесть
```

Используем теперь нашу программу для вызова функции `nftw()` с нулевым аргументом `flags`:

```
$ ./nftw_dir_tree dir
d D    2327983  dir
- F    2327984      a
- F    2327985      b
```

```
- F    2327984      sl      Символическая ссылка sl была представлена как a
l SLN  2327987      dsl
d D    2327988      sub
- F    2327989      x
d DNR  2327994      sub2
```

Из приведенного выше фрагмента видно, что для символической ссылки `sl` был выполнен разбор.

Применим теперь нашу программу для вызова функции `nftw()` с аргументом `flags`, который содержит значения `FTW_PHYS` и `FTW_DEPTH`:

```
$ ./nftw_dir_tree -p -d dir
- F    2327984      a
- F    2327985      b
l SL   2327986      sl     Символическая ссылка sl не была проанализирована
l SL   2327987      dsl
- F    2327989      x
d DP   2327988      sub
d DNR  2327994      sub2
d DP   2327983  dir
```

Из приведенного выше фрагмента видно, что разбор символической ссылки `sl` не был выполнен.

Флаг FTW_ACTIONRETVAL функции nftw()

Начиная с версии 2.3.3, библиотека glibc позволяет использовать дополнительный нестандартный флаг `flags`. Этот флаг, `FTW_ACTIONRETVAL`, изменяет способ интерпретации функцией `nftw()` значения, которое возвращено после вызова функции `func()`. Если указан данный флаг, функция `func()` должна возвращать одно из следующих значений:

- `FTW_CONTINUE` — продолжить обработку записей в дереве каталога, как при обычном возврате нулевого значения от функции `func()`;
- `FTW_SKIP_SIBLINGS` — не обрабатывать остальные записи в данном каталоге; возобновить обработку родительского каталога;
- `FTW_SKIP_SUBTREE` — если аргумент `pathname` является каталогом (то есть значение `typeflag` равно `FTW_D`), то не вызывать функцию `func()` для записей в этом каталоге. Обработка возобновляется в следующем каталоге, родительский каталог которого такой же, как у текущего;
- `FTW_STOP` — не обрабатывать дальнейшие записи в дереве каталога, как при обычном ненулевом результате функции `func()`. Процессу, который вызывал функцию `nftw()`, возвращается значение `FTW_STOP`.

Можно указать проверочное макроопределение `_GNU_SOURCE`, чтобы получить определение `FTW_ACTIONRETVAL` из файла `<ftw.h>`.

18.10. Текущий рабочий каталог процесса

Текущий рабочий каталог процесса задает исходную точку для анализа относительных имен путей, на которые ссылается процесс. Новый процесс наследует свой текущий рабочий каталог от родительского процесса.

Извлечение имени текущего рабочего каталога

Процесс может извлечь имя своего текущего рабочего каталога с помощью функции `getcwd()`.

```
#include <unistd.h>

char *getcwd(char *cwdbuf, size_t size);
```

Возвращает значение *cwdbuf* при успешном завершении или NULL при ошибке

Функция getcwd() помещает строку с завершающим нулем, содержащую абсолютное имя пути для текущего рабочего каталога, в выделенный буфер, на который указывает аргумент cwdbuf. Вызывающий процесс должен выделить буфер cwdbuf размером по меньшей мере size байт. (Обычно размер этого буфера задается с помощью константы PATH_MAX.)

При успешном завершении функция getcwd() возвращает указатель на cwdbuf в качестве результата работы функции. Если имя пути текущего рабочего каталога превышает величину size байт, функция getcwd() возвращает NULL, а переменной errno присваивается значение ERANGE.

В Linux/x86-32 функция getcwd() возвращает максимум 4096 (PATH_MAX) байт. Если длина имени текущего рабочего каталога (а также буфер cwdbuf и значение size) превышает данный предел, имя пути обрезается без уведомления об этом, в результате чего удаляются полные префиксы каталогов от *начала* строки (которая по-прежнему завершается нулем). Иными словами, мы не можем с уверенностью использовать функцию getcwd(), когда длина абсолютного имени пути для текущего рабочего каталога превышает данный предел.

По сути, функция getcwd() в Linux внутренним образом выделяет страницу виртуальной памяти для возвращаемого имени пути. В архитектуре x86-32 размер такой страницы равен 4096 байтам, однако в архитектурах с бóльшим размером страниц (например, в архитектуре Alpha с размером страницы 8192 байта) возможен возврат более длинных имен путей.

Если буфер cwdbuf равен NULL, а размер size нулевой, то оберточная функция для функции getcwd() из библиотеки glibc выделяет буфер необходимого размера и возвращает указатель на этот буфер в качестве результата функции. Чтобы избежать утечек памяти, вызывающий процесс должен впоследствии освободить данный буфер с помощью функции free(). В портируемых приложениях не следует полагаться на ее надежность. В большинстве других реализаций предложено более простое расширение спецификации стандарта SUSv3: если буфер cwdbuf равен NULL, то функция getcwd() выделяет size байт и использует этот буфер для возврата результата вызывающему процессу. Реализация функции getcwd() в библиотеке glibc тоже обеспечивает такую особенность.

GNU-библиотека C предоставляет также еще две функции для получения имени текущего рабочего каталога. Заимствованная из BSD-версии функция getwd(path) подвержена переполнению буфера, поскольку она не предлагает способа указания верхнего предела на размер возвращаемого имени пути. Функция get_current_dir_name() возвращает строку, содержащую имя текущего рабочего каталога. Эту функцию легко использовать, но она не является портируемой. Из соображений безопасности и портируемости функция getcwd() предпочтительнее, чем две названные (если мы стремимся избежать применения расширений библиотеки GNU).

Имея подходящие права доступа (грубо говоря, если мы являемся владельцем процесса или обладаем возможностью CAP_SYS_PTRACE), можно определить имя текущего рабочего каталога для любого процесса, прочитав (readlink()) содержимое специфичной для Linux символической ссылки /proc/*PID*/cwd.

Изменение текущего рабочего каталога

Системный вызов `chdir()` меняет текущий рабочий каталог вызывающего процесса на относительное или абсолютное имя пути, приведенное в аргументе `pathname` (который разыменовывается, если это символическая ссылка).

```
#include <unistd.h>

int chdir(const char *pathname);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Системный вызов `fchdir()` выполняет то же, что и вызов `chdir()`, только каталог указывается с помощью файлового дескриптора, полученного ранее при открытии каталога, задействуя системный вызов `open()`.

```
#define _XOPEN_SOURCE 500            /* Или: #define _BSD_SOURCE */
#include <unistd.h>

int fchdir(int fd);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Можно использовать системный вызов `fchdir()`, чтобы изменить текущий рабочий каталог процесса на другой, а затем вернуться к исходному местоположению, как показано ниже:

```
int fd;

fd = open(".", O_RDONLY);    /* Запоминаем, где мы находимся */
chdir(somepath);             /* Переходим в другое место */
fchdir(fd);                  /* Возвращаемся в исходный каталог */
close(fd);
```

Эквивалентный код, применяющий системный вызов `chdir()`, выглядит так:

```
char buf[PATH_MAX];

getcwd(buf, PATH_MAX);        /* Запоминаем, где мы находимся */
chdir(somepath);              /* Переходим в другое место */
chdir(buf);                   /* Возвращаемся в исходный каталог */
```

18.11. Работа с использованием файлового дескриптора каталога

Начиная с версии ядра 2.6.16 Linux предлагает ряд новых системных вызовов, выполняющие задачи, сходные с задачами традиционных вызовов, но обеспечивающие дополнительную функциональность, которая может быть удобна в отдельных приложениях. Эти системные вызовы приведены в табл. 18.2. Мы рассматриваем их в данной главе, поскольку они вносят изменения в традиционную семантику текущего рабочего каталога процесса.

Таблица 18.2. Системные вызовы, использующие файловый дескриптор каталога для интерпретации относительных имен пути

Новый интерфейс	Традиционный аналог	Примечания
faccessat()	access()	Поддерживает флаги AT_EACCESS и AT_SYMLINK_NOFOLLOW
fchmodat()	chmod()	
fchownat()	chown()	Поддерживает флаг AT_SYMLINK_NOFOLLOW
fstatat()	stat()	Поддерживает флаг AT_SYMLINK_NOFOLLOW
linkat()	link()	Поддерживает (начиная с версии Linux 2.6.18) флаг AT_SYMLINK_FOLLOW
mkdirat()	mkdir()	
mkfifoat()	mkfifo()	Библиотечная функция, основанная на mknodat()
mknodat()	mknod()	
openat()	open()	
readlinkat()	readlink()	
renameat()	rename()	
symlinkat()	symlink()	
unlinkat()	unlink()	Поддерживает флаг AT_REMOVEDIR
utimensat()	utimes()	Поддерживает флаг AT_SYMLINK_NOFOLLOW

Для описания этих системных вызовов возьмем конкретный пример: системный вызов `openat()`.

```
#define _XOPEN_SOURCE 700      /* Или define _POSIX_C_SOURCE >= 200809 */
#include <fcntl.h>

int openat(int dirfd, const char *pathname, int flags, … /* mode_t mode */);
```

Возвращает файловый дескриптор при успешном завершении или `-1` при ошибке

Системный вызов `openat()` подобен традиционному вызову `open()`, но снабжен дополнительным аргументом `dirfd`, применяемым следующим образом:

- если аргумент `pathname` задает относительное имя пути, то оно интерпретируется по отношению к каталогу, на который указывает открытый файловый дескриптор `dirfd`, а не по отношению к текущему рабочему каталогу процесса;
- если аргумент `pathname` задает относительное имя пути, а аргумент `dirfd` содержит специальное значение `AT_FDCWD`, то имя пути `pathname` интерпретируется по отношению к текущему рабочему каталогу процесса (то есть поведение такое же, как у вызова `open(2)`);
- если аргумент `pathname` задает абсолютное имя пути, то аргумент `dirfd` игнорируется.

Аргумент `flags` функции `openat()` служит тем же целям, что и в системном вызове `open()`. Тем не менее ряд системных вызовов, перечисленных в табл. 18.2, поддерживают аргумент `flags`, который не обеспечивается соответствующим традиционным системным

вызовом; назначение такого аргумента заключается в изменении семантики вызова. Наиболее часто задействован флаг AT_SYMLINK_NOFOLLOW; он указывает на то, что если аргумент pathname является символической ссылкой, то системный вызов должен оперировать этой ссылкой, а не файлом, на который она ссылается. (Системный вызов linkat() снабжен флагом AT_SYMLINK_FOLLOW, выполняющим обратное действие, меняя принятое по умолчанию поведение вызова linkat() таким образом, чтобы он разыменовывал аргумент oldpath, если он является символической ссылкой.) Подробности, относящиеся к другим флагам, см. на соответствующих страницах руководства.

Системные вызовы, приведенные в табл. 18.2, поддерживаются вследствие двух причин (опять-таки объяснение приводится на примере вызова openat()).

- Использование системного вызова openat() позволяет приложению избежать условий соперничества, которые могут возникнуть, если вызов open() применен для открытия файлов в каталогах, отличных от текущего рабочего каталога. Такое соперничество может появиться, поскольку параллельно с вызовом open() мог измениться какой-либо компонент префикса каталога в аргументе pathname. С помощью открытия файлового дескриптора для целевого каталога и передачи этого дескриптора вызову openat() можно избежать подобного соперничества.
- В главе 29 мы увидим, что рабочий каталог является атрибутом процесса, совместно применяемым всеми потоками процесса. Для ряда приложений удобно, если различные потоки обладают разными «виртуальными» рабочими каталогами. Приложение способно имитировать такую функциональность, используя вызов openat() в сочетании с файловыми дескрипторами каталога, с которыми работает приложение.

Данные системные вызовы не включены в стандарт SUSv3, но присутствуют в стандарте SUSv4. Чтобы сделать видимым объявление каждого из этих вызовов, следует указать значение не менее 700 для макроопределения _XOPEN_SOURCE (feature test macro, проверка функциональной возможности), прежде чем включать соответствующий заголовочный файл (то есть <fcntl.h> для функции open()). В качестве альтернативы можно указать в макроопределении _POSIX_C_SOURCE значение, которое больше или равно 200 809. (До выхода версии 2.10 библиотеки glibc было необходимо макроопределение _ATFILE_SOURCE, делающее видимым объявления этих системных вызовов.)

В ОС Solaris 9 и более поздних версиях присутствуют варианты некоторых интерфейсов, перечисленных в табл. 18.2, с немного отличающейся семантикой.

18.12. Изменение корневого каталога процесса: системный вызов chroot()

Каждый процесс обладает *корневым каталогом* — он представляет собой точку отсчета, от которой интерпретируются абсолютные имена путей (то есть начинающиеся с символа /). По умолчанию данным каталогом является реальный корневой каталог файловой системы. (Новый процесс наследует корневой каталог своего родителя.) Иногда бывает удобно, чтобы процесс изменил свой корневой каталог. Это может осуществить процесс с привилегией CAP_SYS_CHROOT с помощью системного вызова chroot().

Системный вызов chroot() изменяет корневой каталог процесса на каталог, указанный в аргументе pathname (который разыменовывается, если это символическая ссылка). Затем все абсолютные имена путей интерпретируются как начинающиеся с указанного местоположения в файловой системе. Иногда это называют заключением в клетку chroot, поскольку программа оказывается замкнутой внутри некоторой части файловой системы.

```
#define _BSD_SOURCE
#include <unistd.h>

int chroot(const char *pathname);
```

Возвращает 0 при успешном завершении или –1 при ошибке

Стандарт SUSv2 содержал спецификацию системного вызова chroot() (с пометкой LEGACY («устарело»)), которая была изъята из стандарта SUSv3. Тем не менее системный вызов chroot() присутствует в большинстве реализаций UNIX.

Системный вызов chroot() задействуется командой chroot, позволяющей выполнять команды оболочки в клетке chroot.

Корневой каталог любого процесса можно найти, прочитав (readlink()) содержимое характерной для Linux символической ссылки /proc/PID/root.

Классическим примером использования системного вызова chroot() является программа ftp. В качестве меры безопасности при анонимном входе пользователя по протоколу FTP программа ftp применяет системный вызов chroot(), чтобы в качестве корневого каталога для нового процесса задать каталог, который специально зарезервирован для анонимных подключений. По завершении вызова chroot() пользователь ограничен поддеревом нового корневого каталога и поэтому не может «бродить» по всей файловой системе. (Это опирается на следующий факт: корневой каталог является собственным родителем; то есть путь /.. — ссылка на /, в связи с чем при изменении каталога на / с последующей попыткой выполнить команду cd .. пользователь остается в том же самом каталоге.)

В некоторых реализациях UNIX (но не в Linux) допускается наличие нескольких жестких ссылок на каталог, и поэтому возможно создать жесткую ссылку внутри подкаталога на его родительский каталог (или на каталог более высокого уровня). В версиях операционной системы, позволяющих это, наличие жесткой ссылки, ведущей за пределы дерева каталогов, расположенных в клетке, разрушает данную клетку. Символические ссылки на каталоги, расположенные вне ее, не создают проблем: они интерпретируются внутри фреймворка нового корневого каталога процесса, поэтому не могут выходить за клетку chroot.

В нормальном режиме нельзя выполнять произвольные программы внутри клетки chroot. Это вызвано тем, что большинство программ динамически связано с совместно используемыми библиотеками. Следовательно, мы должны либо ограничиться выполнением статически привязанных программ, либо реплицировать внутри клетки стандартный набор системных каталогов, содержащих совместно используемые библиотеки (такие, например, как /lib и /usr/lib) (в этом отношении может оказаться полезной функция связанного монтирования, описанная в подразделе 14.9.4).

Системный вызов chroot() не был задуман как абсолютно защищенный механизм для создания клетки. Начнем с того, что существуют различные способы, с помощью которых привилегированная программа может последовательно применять дальнейший вызов chroot() для выхода из клетки. Например, программа с привилегиями CAP_MKNOD может задействовать системный вызов mknod(), чтобы создать файл запоминающего устройства (подобный файлу /dev/mem), предоставляя доступ к содержимому оперативной памяти, после чего становится возможным все. В целом, рекомендуется не размещать программы set-user-ID-root внутри файловой системы с клеткой chroot.

И даже в случае с непривилегированными программами следует проявлять осторожность, чтобы предупредить следующие возможные варианты выхода из клетки `chroot`.

- Вызов `chroot()` не меняет текущий рабочий каталог процесса. Следовательно, системный вызов `chroot()`, как правило, предваряется или завершается вызовом `chdir()` (то есть `chdir("/")` после вызова `chroot()`). Если это не выполнено, процесс может использовать относительные имена путей для доступа к файлам и каталогам вне клетки. (В некоторых BSD-ветках такая возможность предотвращается: если текущий рабочий каталог расположен вне нового дерева корневого каталога, то системный вызов `chroot()` делает его таким же, как корневой.)
- Если процесс удерживает открытый файловый дескриптор каталога, расположенного вне клетки, то можно воспользоваться комбинацией вызовов `fchdir()` и `chroot()`, чтобы выйти за ее пределы, как показано в следующем примере кода:

```
int fd;

fd = open("/", O_RDONLY);
chroot("/home/mtk");            /* Внутри клетки */
fchdir(fd);
chroot(".");                    /* Вне клетки */
```

Для предотвращения такой возможности следует закрывать все открытые файловые дескрипторы, указывающие на каталоги, расположенные вне клетки. (В ряде реализаций UNIX присутствует системный вызов fchroot(), который можно использовать, чтобы добиться результата, похожего на работу приведенного выше кода.)

- Но даже такого устранения описанных возможностей оказывается недостаточно для того, чтобы запретить произвольной непривилегированной программе (то есть такой, чьей работой мы не можем управлять) выход из клетки. Заключенный в нее процесс может использовать сокет домена UNIX для получения файлового дескриптора (от другого процесса), указывающего на каталог вне клетки. (Принцип передачи файловых дескрипторов между процессами с помощью сокета вкратце описан в подразделе 57.13.3.) Указав данный дескриптор при вызове `fchdir()`, программа может определить для себя текущий рабочий каталог вне клетки, а затем осуществлять доступ к любым файлам и каталогам с помощью относительных имен путей.

В отдельных BSD-ветках присутствует системный вызов jail(), устраняющий проблемы, описанные выше, а также ряд других за счет создания клетки, которая является защищенной даже для привилегированного процесса.

18.13. Анализ имени пути: функция realpath()

Библиотечная функция `realpath()` разыменовывает все символические ссылки в аргументе `pathname` (который является строкой с завершающим нулем) и выполняет анализ всех ссылок на `/.` и `/..`, чтобы выдать строку с завершающим нулем, содержащую соответствующее абсолютное имя пути.

Результирующая строка размещается в буфере, указанном в аргументе `resolved_path`, который должен быть символьным массивом, содержащим по меньшей мере `PATH_MAX` байт. При успешном завершении функция `realpath()` также возвращает указатель на эту проанализированную строку.

```
#include <stdlib.h>

char *realpath(const char *pathname, char *resolved_path);
```

При успешном завершении возвращает указатель на проанализированное имя пути или NULL при ошибке

Функция `realpath()`, реализованная в библиотеке glibc, позволяет вызывающему процессу указать для аргумента `resolved_path` значение `NULL`. В таком случае функция `realpath()` выделяет буфер размером до `PATH_MAX` байт для проанализированного имени пути и возвращает указатель на данный буфер. (Вызывающий процесс должен освободить этот буфер с помощью вызова `free()`.) В стандарте SUSv3 не описано такое расширение возможностей, но оно есть в стандарте SUSv4.

Программа в листинге 18.4 задействует вызов `readlink()` и функцию `realpath()` для чтения содержимого символической ссылки и для ее анализа с преобразованием в абсолютное имя пути. Ниже приведен пример использования этой программы:

```
$ pwd                                   Где мы находимся?
/home/mtk
$ touch x                               Создаем файл
$ ln -s x y                             и символическую ссылку на него
$ ./view_symlink y
readlink: y --> x
realpath: y --> /home/mtk/x
```

Листинг 18.4. Чтение и анализ символической ссылки

irs._links/view_symlink.c

```
#include <sys/stat.h>
#include <limits.h>             /* Для определения PATH_MAX */
#include "tlpi_hdr.h"

#define BUF_SIZE PATH_MAX

int
main(int argc, char *argv[])
{
    struct stat statbuf;
    char buf[BUF_SIZE];
    ssize_t numBytes;

    if (argc != 2 || strcmp(argv[1], "-help") == 0)
        usageErr("%s pathname\n", argv[0]);

    if (lstat(argv[1], &statbuf) == -1)
        errExit("lstat");

    if (!S_ISLNK(statbuf.st_mode))
        fatal("%s is not a symbolic link", argv[1]);

    numBytes = readlink(argv[1], buf, BUF_SIZE - 1);
    if (numBytes == -1)
        errExit("readlink");
    buf[numBytes] = '\0';    /* Добавляем завершающий нулевой байт */
    printf("readlink: %s --> %s\n", argv[1], buf);
```

```
    if (realpath(argv[1], buf) == NULL)
        errExit("realpath");
    printf("realpath: %s --> %s\n ", argv[1], buf);

    exit(EXIT_SUCCESS);
}
```

irs._links/view_symlink.c

18.14. Синтаксический разбор строк с именем пути: функции dirname() и basename()

Функции `dirname()` и `basename()` разбивают строку с именем пути на имя каталога и файловые имена частей. (Эти функции выполняют работу, сходную с командами `dirname(1)` и `basename(1)`.)

```
#include <libgen.h>

char *dirname(char *pathname);
char *basename(char *pathname);
```

Обе функции возвращают указатель на строку с завершающим нулем (которая может быть статически выделенной)

Так, например, для имени пути `/home/britta/prog.c` функция `dirname()` возвращает `/home/britta`, а функция `basename()` вернет `prog.c`. Благодаря сцеплению строки, возвращенной функцией `dirname()`, с символом слеша и со строкой, возвращенной функцией `basename()`, получается полное имя пути.

Обратите внимание на следующие моменты, касающиеся работы функций `dirname()` и `basename()`.

- Завершающие символы `/` в аргументе `pathname` игнорируются.
- Если аргумент `pathname` не содержит слешей, функция `dirname()` возвращает строку с точкой (`.`), а функция `basename()` возвращает имя `pathname`.
- Если аргумент `pathname` состоит только из слеша, то обе функции возвращают строку `/`. Применение описанного выше правила сцепления для получения имени пути из возвращенных строк привело бы к появлению строки `///`. Такое имя пути является корректным. Поскольку несколько последовательных символов `/` эквивалентны одному, имя пути `///` эквивалентно имени `/`.
- Если аргумент `pathname` является нулевым (`NULL`) указателем или пустой строкой, обе функции возвращают строку с точкой (`.`). (Сцепление таких строк приводит к имени пути `./.`, которое эквивалентно имени `.`, то есть текущего каталога.)

В табл. 18.3 показаны строки, возвращаемые функциями `dirname()` и `basename()` при различных именах пути.

Обе функции, `dirname()` и `basename()`, могут изменять строку, на которую указывает аргумент `pathname`. Таким образом, если желательно сохранить строку с именем пути, то следует передать ее копии функциям `dirname()` и `basename()`, как показано в листинге 18.5. Эта программа задействует функцию `strdup()` (которая вызывает функ-

цию `malloc()`), чтобы сделать копии строк, предназначенные для передачи функциям `dirname()` и `basename()`. Затем она применяет функцию `free()` для высвобождения пространства, использовавшегося дубликатами.

Таблица 18.3. Примеры строк, возвращаемых функциями dirname() и basename()

Строка имени пути	dirname()	basename()
/	/	/
/usr/bin/zip	/usr/bin	zip
/etc/passwd////	/etc	passwd
/etc////passwd	/etc	passwd
etc/passwd	etc	passwd
passwd	.	passwd
passwd/	.	passwd
..	.	..
NULL	.	.

Листинг 18.5. Использование функций dirname() и basename()

irs._links/t_dirbasename.c

```
#include <libgen.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *t1, *t2;
    int j;

    for (j = 1; j < argc; j++)    {
        t1 = strdup(argv[j]);
        if (t1 == NULL)
            errExit("strdup");
        t2 = strdup(argv[j]);
        if (t2 == NULL)
            errExit("strdup");

        printf("%s ==> %s + %s\n", argv[j],
               dirname(t1), basename(t2));
        free(t1);
        free(t2);
    }

    exit(EXIT_SUCCESS);
}
```

irs._links/t_dirbasename.c

И в завершение обратите внимание на то, что функции `dirname()` и `basename()` могут возвращать указатели на статически выделенные строки, которые могут быть изменены последующими вызовами этих же функций.

18.15. Резюме

Индексный дескриптор не содержит имени файла. Взамен этого имена назначаются файлам путем записей в каталогах, которые являются таблицами, перечисляющими имена файлов с соответствующими номерами индексных дескрипторов. Такие записи называют (жесткими) ссылками. Файл может обладать несколькими ссылками, статус которых одинаков. Ссылки создаются и удаляются с помощью системных вызовов `link()` и `unlink()`. Файл можно переименовать, используя системный вызов `rename()`.

Символическая (или мягкая) ссылка создается с помощью системного вызова `symlink()`. Символические ссылки чем-то напоминают жесткие, с тем отличием, что первые могут выходить за пределы файловой системы, а также ссылаться на каталоги. Символическая ссылка — всего лишь файл, содержащий имя другого файла; это имя можно извлечь, воспользовавшись системным вызовом `readlink()`. Символическая ссылка не учитывается счетчиком ссылок (целевого) индексного дескриптора и может оказаться зависшей, если имя файла, на который она ссылается, будет удалено. Ряд системных вызовов автоматически разыменовывают символические ссылки (то есть следуют по ним); остальные вызовы этого не делают. В некоторых случаях существуют две версии системного вызова: первая разыменовывает символические ссылки, а вторая — нет. В качестве примеров можно назвать вызовы `stat()` и `lstat()`.

Каталоги создаются с помощью системного вызова `mkdir()`, а удаляются вызовом `rmdir()`. Чтобы просканировать содержимое каталога, можно применить системные вызовы `opendir()`, `readdir()` и соответствующие функции. Функция `nftw()` позволяет программе выполнить обход всего дерева каталога, вызывая указанную программистом функцию для работы с каждым файлом в данном дереве.

Функцию `remove()` можно использовать для удаления файла (то есть ссылки) или пустого каталога.

Каждый процесс имеет корневой каталог. Он определяет местоположение, относительно которого интерпретируются абсолютные имена путей. У процесса есть также текущий рабочий каталог, определяющий местоположение, относительно которого интерпретируются относительные имена путей. Для изменения этих атрибутов применяются системные вызовы `chroot()` и `chdir()`. Функция `getcwd()` возвращает имя текущего рабочего каталога процесса.

В Linux присутствует набор системных вызовов (например, `openat()`), работающие подобно своим традиционным аналогам (то есть `open()`), за исключением того, что относительные имена путей могут интерпретироваться по отношению к каталогу, указанному в файловом дескрипторе, переданном вызову (а не с помощью текущего рабочего каталога процесса). Это удобно использовать, чтобы избежать некоторых типов соперничества, а также для реализации виртуальных рабочих каталогов для потоков.

Функция `realpath()` выполняет анализ имени пути — разыменование всех символических ссылок, а также приведение всех ссылок `.` и `..` к соответствующим каталогам, — чтобы получить абсолютное имя пути. Функции `dirname()` и `basename()` можно применять для разбора имени пути на компоненты, состоящие из имен каталога и файла.

18.16. Упражнения

18.1. В разделе 4.3.2 мы отметили, что нельзя открыть файл на запись, если он в данный момент выполняется (системный вызов `open()` возвращает `-1`, а переменной `errno` присваивается значение `ETXTBSY`). Тем не менее можно выполнить из оболочки следующее:

```
$ cc -o longrunner longrunner.c
$ ./longrunner &                    Оставляем работать в фоновом режиме
$ vi longrunner.c                   Вносим изменения в исходный код
$ cc -o longrunner longrunner.c
```

Последняя команда перезаписывает существующий исполняемый файл с таким же именем. Почему это возможно? (Подсказка: воспользуйтесь командой `ls -li`, чтобы увидеть номер индексного дескриптора исполняемого файла после каждой компиляции.)

18.2. Почему приводит к ошибке системный вызов `chmod()` в следующем коде:

```
mkdir("test", S_IRUSR | S_IWUSR | S_IXUSR);
chdir("test");
fd = open("myfile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
symlink("myfile", "../mylink");
chmod("../mylink", S_IRUSR);
```

18.3. Реализуйте функцию `realpath()`.

18.4. Измените программу из листинга 18.2 (`list_files.c`) так, чтобы вместо функции `readdir()` использовалась функция `readdir_r()`.

18.5. Реализуйте функцию, которая работает подобно функции `getcwd()`. Для решения данной задачи удобно воспользоваться таким приемом: можно выяснить имя текущего рабочего каталога с помощью функций `opendir()` и `readdir()`, чтобы выполнить обход каждой записи в родительском каталоге (`..`). Это необходимо для нахождения записи с такими же индексным дескриптором и номером устройства, что и у текущего рабочего каталога (то есть соответственно полей `st_ino` и `st_dev` в структуре `stat`, возвращаемой системными вызовами `stat()` и `lstat()`). Следовательно, есть возможность «собрать» путь каталога путем пошагового обхода его дерева (`chdir("..")`) и выполнения такого сканирования. Обход можно завершить, когда родительский каталог будет такой же, как текущий рабочий (как вы помните, `/..` — это то же, что и `/`). Вызывающий процесс должен оставаться в том же каталоге, откуда начал работу, вне зависимости от того, как завершила работу ваша функция `getcwd()`, успешно или с ошибкой (для данной цели удобно применить системный вызов `open()` в сочетании с функцией `fchdir()`).

18.6. Измените программу из листинга 18.3 (`nftw_dir_tree.c`) так, чтобы использовать флаг `FTW_DEPTH`. Обратите внимание на отличие в порядке обхода каталога.

18.7. Напишите программу, которая задействует функцию `nftw()` для обхода дерева каталога и выводит в результате количество и долю (в процентах) файлов различных типов (обычных, каталогов, символических ссылок и т. д.) в данном дереве.

18.8. Реализуйте функцию `nftw()`. (Для этого понадобятся, среди прочих, системные вызовы `opendir()`, `readdir()`, `closedir()` и `stat()`.)

18.9. В разделе 18.10 показаны два различных метода (с применением функций `fchdir()` и `chdir()` соответственно), позволяющие вернуться в предыдущий рабочий каталог после изменения текущего. Предположим, мы выполняем такую операцию многократно. Как вы думаете, какой метод окажется более эффективным? Почему? Напишите программу для обоснования вашего ответа.

19 Мониторинг событий файлов

Некоторым приложениям требуется иметь возможность осуществлять мониторинг файлов или каталогов, чтобы определить, произошли ли с объектами наблюдения те или иные события. Например, графический менеджер файлов должен быть в состоянии установить, когда пользователь добавляет или удаляет файлы из отображаемого в настоящий момент каталога, фоновый процесс может также проводить мониторинг собственного файла конфигурации, чтобы определять, был ли изменен этот файл.

Начиная с версии ядра 2.6.13, Linux предоставляет механизм `inotify`, позволяющий приложениям осуществлять мониторинг событий файлов. В данной главе описывается его использование.

Механизм `inotify` призван заменить устаревший механизм `dnotify`, предоставлявший лишь часть возможностей `inotify`. В конце главы мы приводим краткое описание `dnotify`, концентрируясь при этом на преимуществах `inotify`.

Механизмы `inotify` и `dnotify` — специфичные механизмы Linux. (Ряд других операционных систем предоставляют схожие механизмы. Например, BSD реализован интерфейс API `kqueue`.)

> Лишь в некоторых библиотеках реализованы интерфейсы API, по своим свойствам еще более абстрактные и портируемые, чем механизмы inotify и dnotify. Использование данных библиотек может быть предпочтительным для отдельных приложений. В части этих библиотек применяются механизмы inotify и dnotify, но только в тех системах, где указанные механизмы доступны. FAM (File Alteration Monitor (монитор изменения файлов) и Gamin — пример таких библиотек. См.: oss.sgi.com/projects/fam/.

19.1. Обзор

Далее перечислены ключевые шаги использования API `inotify`.

1. Приложение задействует системный вызов `inotify_init()` для создания *объекта* `inotify`. Системный вызов возвращает файловый дескриптор, служащий для ссылки на объект `inotify` при выполнении последующих операций.
2. Приложение информирует ядро о том, какие файлы ему необходимы, с помощью функции `inotify_and_watch()` для добавления элементов в список наблюдения объекта `inotify`, созданного в предыдущем шаге. Каждый элемент списка наблюдения состоит из путевого имени, а также битовой маски. Она определяет список событий, мониторинг которых необходимо осуществлять для данного имени. В результате выполнения функция `inotify_and_watch()` возвращает *дескриптор наблюдения*, использующийся впоследствии для ссылки на объект наблюдения (системный вызов `inotify_rm_watch()` выполняет обратную операцию — удаляет объект наблюдения, ранее уже добавленный в список).
3. Чтобы получить оповещения о событиях, приложение выполняет для дескриптора файла `inotify` операции чтения `read()`. Любая успешно выполненная операция воз-

вращает одну или несколько структур `inotify_event`, каждая из которых содержит информацию о произошедшем событии, имеющем отношение к одному из наблюдаемых с помощью объекта `inotify` путевых имен.

4. По завершении мониторинга приложение закрывает дескриптор файла `inotify`. Это позволяет удалить все элементы списка наблюдения, связанные с объектом `inotify`.

Механизм `inotify` может использоваться для мониторинга файлов и каталогов. При мониторинге каталога приложение будет оповещено о событиях, касающихся только самого каталога, но не файлов, хранящихся в нем.

Механизм мониторинга `inotify` не является рекурсивным. Если приложению необходимо осуществлять мониторинг по всему поддереву каталога, оно должно произвести вызовы `inotify_and_watch()` для каждого каталога в дереве.

Мониторинг файлового дескриптора `inotify` может проходить с помощью интерфейсов `select()`, `poll()`, `epoll` и, начиная с версии Linux 2.6.25, ввода-вывода, управляемого сигналом. Если события доступны для чтения, то вышеприведенные интерфейсы обозначают файловый дескриптор `inotify` как читаемый. Для получения дополнительной информации об указанных интерфейсах см. главу 59.

Механизм inotify — это необязательный компонент ядра Linux, настраиваемый с помощью опций CONFIG_INOTIFY и CONFIG_INOTIFY_USER.

19.2. Интерфейс inotify

Системный вызов `inotify_init()` создает новый объект `inotify`.

```
#include <sys/inotify.h>

int inotify_init(void);
```

Возвращает файловый дескриптор или `-1` при ошибке

В результате выполнения функция `inotify_init()` возвращает файловый дескриптор. Такой дескриптор является манипулятором, использующимся для ссылки на объект `inotify` при выполнении последующих операций.

Начиная с версии ядра 2.6.27, Linux поддерживает новый, нестандартный, системный вызов inotify_init1(). Он выполняет те же задачи, что и inotify_init(), но предоставляет при этом дополнительный аргумент flags, который может применяться для модификации поведения системного вызова. Поддерживается два флага. Флаг IN_CLOEXEC заставляет ядро установить флаг закрытия при исполнении exec (FD_CLOEXEC) для нового файлового дескриптора. Данный флаг полезен по тем же причинам, что и флаг open() O_CLOEXEC, описанный в подразделе 4.3.1. Флаг IN_NONBLOCK заставляет ядро установить флаг O_NONBLOCK на описание текущей процедуры открытия файла, чтобы последующие операции чтения были неблокирующими. Это позволяет не осуществлять дополнительные вызовы функции fcntl() для достижения того же результата.

Системный вызов `inotify_and_watch()` либо добавляет новый элемент в список наблюдения для объекта `inotify`, ссылка на который осуществляется с помощью файлового дескриптора `fd`, либо изменяет уже имеющийся элемент (рис. 19.1).

```
#include <sys/inotify.h>

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

Возвращает дескриптор наблюдения или -1 при ошибке

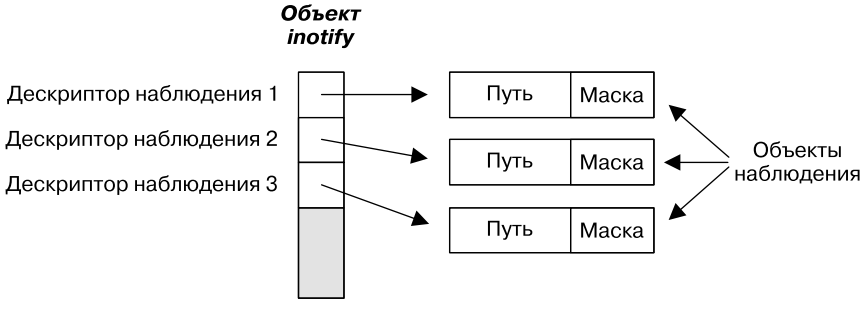


Рис. 19.1. Объект inotify и связанные структуры данных ядра

Аргумент `pathname` указывает на файл, для которого требуется создать или изменить элемент списка наблюдения. Вызывающий код должен иметь разрешение на доступ к этому файлу. (Проверка разрешения доступа выполняется единожды при осуществлении вызова `inotify_and_watch()`. Вызывающий код будет продолжать получать оповещения от файла на протяжении всего времени существования элемента списка наблюдения, даже если условия доступа впоследствии будут изменены, так как вызывающий код не повторял процедуру чтения разрешений доступа к файлу.)

Аргумент `mask` — это битовая маска, определяющая перечень событий, чей мониторинг осуществляется для файла, установленного аргументом `pathname`. Далее по тексту мы опишем подробнее значения битов, которые могут быть указаны в аргументе `mask`.

Если аргумент `pathname` не был ранее добавлен в файловый дескриптор `fd`, то вызов `inotify_and_watch()` создает новый элемент списка наблюдения и возвращает новый неотрицательный дескриптор наблюдения, который используется для ссылки на этот элемент при последующих операциях. Этот дескриптор уникален для данного объекта `inotify`.

Если же аргумент `pathname` уже был ранее добавлен в файловый дескриптор `fd`, то вызов `inotify_and_watch()` изменяет маску существующего элемента списка наблюдения на `pathname` и возвращает дескриптор наблюдения для этого элемента. (Данный дескриптор будет таким же, что и дескриптор, возвращенный функцией `inotify_and_watch()` при изначальном добавлении файла `pathname` в данный список). Мы более подробно расскажем о том, каким образом можно отредактировать маску, когда будем описывать флаг `IN_MASK_ADD` в следующем разделе.

Системный вызов `inotify_rm_watch()` удаляет элемент списка наблюдения, указанный дескриптором наблюдения `wd` объекта `inotify`, ссылка на который осуществляется с помощью дескриптора `fd`.

```
#include <sys/inotify.h>

int inotify_rm_watch(int fd, uint32_t wd);
```

Возвращает дескриптор наблюдения или -1 при ошибке

Аргумент `wd` — дескриптор наблюдения, возвращенный предшествующим вызовом `inotify_and_watch()`. (Тип данных `uint32_t` — это 32-разрядное целое число без знака.)

Удаление элемента списка наблюдения приведет к генерации для данного дескриптора наблюдения события `IN_IGNORED`. Мы более подробно расскажем об этом событии ниже.

19.3. События inotify

Когда мы создаем или изменяем элемент списка наблюдения с помощью функции `inotify_and_watch()`, аргумент битовая маска `mask` задает события, чей мониторинг должен осуществляться для указанного файла `pathname`. Биты событий, которые могут быть указаны в аргументе `mask`, перечислены в столбце In табл. 19.1.

Таблица 19.1. События inotify

Битовое значение	In	Out	Описание
IN_ACCESS	*	*	Файл был прочитан (read())
IN_ATTRIB	*	*	Метаданные файла изменены
IN_CLOSE_WRITE	*	*	Файл был открыт для записи, а потом закрыт
IN_CLOSE_ NOWRITE	*	*	Файл был открыт для записи, а потом закрыт
IN_CREATE	*	*	Файл/каталог создан внутри наблюдаемого каталога
IN_DELETE	*	*	Файл/каталог удален из наблюдаемого каталога
IN_DELETE_SELF	*	*	Наблюдаемый файл/каталог был удален
IN_MODIFY	*	*	Файл был изменен
IN_MOVE_SELF	*	*	Наблюдаемый файл/каталог был перемещен
IN_MOVED_FROM	*	*	Наблюдаемый файл/каталог был перемещен из наблюдаемого каталога
IN_MOVED_TO	*	*	Наблюдаемый файл/каталог был перемещен в наблюдаемый каталог
IN_OPEN	*	*	Файл был открыт
IN_ALL_EVENTS	*		Сокращение для всех вышеперечисленных событий ввода
IN_MOVE	*		Сокращение для IN_MOVED_FROM | IN_MOVED_TO
IN_CLOSE	*		Сокращение для IN_CLOSE_WRITE | IN_CLOSE_ NOWRITE
IN_DONT_FOLLOW	*		Не разыменовывать символьную ссылку (начиная с Linux 2.6.15)
IN_MASK_ADD	*		Добавить события в маску текущего элемента списка наблюдения для файла pathname
IN_ONESHOT	*		Наблюдать для файла pathname только одно событие
IN_ONLYDIR	*		Ошибка, если pathname не каталог (начиная с Linux 2.6.15)

Продолжение ⇗

Таблица 19.1 *(продолжение)*

Битовое значение	In	Out	Описание
IN_IGNORED		*	Элемент списка наблюдения был удален приложением или ядром
IN_ISDIR		*	Имя файла, возвращенное в name, — каталог
IN_Q_OVERFLOW		*	Переполнение очереди событий
IN_UNMOUNT		*	Файловая система, содержащая объект, была размонтирована

Значения большинства битов в табл. 19.1 ясны из их названий. Следующий список содержит пояснения некоторых деталей.

- Событие `IN_ATTRIB` происходит при изменении метаданных файла, таких как разрешения, права владения, счетчик ссылок, расширенные атрибуты, идентификатор пользователя, идентификатор группы.
- Событие `IN_DELETE_SELF` происходит при удалении объекта наблюдения (файла или каталога). Событие `IN_DELETE` происходит при удалении одного из файлов, содержащегося в объекте наблюдения, если объект наблюдения — каталог.
- Событие `IN_MOVE_SELF` происходит при переименовании объекта наблюдения. События `IN_MOVE_FROM` и `IN_MOVE_TO` происходят при перемещении некоего объекта, находящегося внутри каталога, за которым ведется наблюдение. При этом первое событие происходит с каталогом, содержащим старое имя, а второе — с каталогом, содержащим новое.
- Биты `IN_DONT_FOLLOW`, `IN_MASK_ADD`, `IN_ONESHOT` и `IN_ONLYDIR` не устанавливают события, за свершением которых требуется вести наблюдение. Данные биты контролируют работу вызова `inotify_and_watch()`.
- Бит `IN_DONT_FOLLOW` устанавливает, что аргумент `pathname` не подлежит разыменованию, если это символьная ссылка. Такое правило позволяет программе вести мониторинг самой символьной ссылки, а не файла, на который она указывает.
- При выполнении вызова `inotify_and_watch()` с параметром `pathname`, уже включенным в список наблюдения через файловый описатель `inotify`, по умолчанию произойдет перезапись имеющегося значения параметра `mask` данного элемента на новое, передаваемое с текущим вызовом. При указании бита `IN_MASK_ADD` текущая маска дополняется значением, переданным в атрибуте `mask` данного вызова.
- Бит `IN_ONESHOT` позволяет приложению осуществлять мониторинг только одного события `pathname`. После его свершения происходит автоматическое удаление соответствующего элемента списка наблюдения.
- Бит `IN_ONLYDIR` позволяет приложению осуществлять мониторинг, только если `pathname` — это каталог. Если нет, то вызов `inotify_and_watch()` завершается с ошибкой `ENOTDIR`. Использование данного флага позволяет избежать состояния гонки, которое могло бы возникнуть, если бы мы захотели удостовериться, что ведем наблюдение именно за каталогом.

19.4. Чтение событий inotify

После регистрации элементов в списке наблюдения приложение с помощью функции `read()` может определить, какие события произошли с файловым дескриптором `inotify`. Если к данному моменту никаких событий не случилось, то функция `read()` блокируется

до свершения какого-либо события (если для файлового описателя не был установлен флаг `O_NONBLOCK`, в случае чего функция `read()` завершается с ошибкой `EAGAIN` при отсутствии доступных событий).

После свершения событий каждый вызов функции `read()` возвращает буфер (см. рис. 19.2), содержащий одну или несколько структур следующего типа:

```
struct inotify_event {
    int      wd;       /* Дескриптор наблюдения, с которым произошло событие */
    uint32_t mask;     /* Биты, описывающие произошедшие события */
    uint32_t cookie;   /* Cookie для связанных событий (для rename()) */
    uint32_t len;      /* Размер поля 'name' */
    char     name[];   /* Опциональное имя файла с нуль-символом в конце */
};
```

Рис. 19.2. Входящий буфер, содержащий три структуры inotify_event

Поле `wd` сообщает нам дескриптор наблюдения, для которого произошло данное событие. Это поле содержит одно из значений, возвращенных предшествующим вызовом `inotify_and_watch()`. Поле `wd` полезно, когда приложение ведет мониторинг нескольких файлов или каталогов через один и тот же файловый дескриптор `inotify`, предоставляющий ссылку, позволяющую приложению определить, с каким конкретно файлом или каталогом произошло данное событие. (Однако для этого в приложении должна содержаться структура, связывающая дескрипторы наблюдения с путевыми именами).

Поле mask возвращает битовую маску, описывающую событие. Диапазон битов, которые могут появиться в поле mask, указаны в столбце Out табл. 19.1. Обратите внимание на приведенную ниже дополнительную информацию о некоторых битах.

- Событие IN_IGNORED генерируется при удалении элемента списка наблюдения. Это может произойти по нескольким причинам: приложение использовало вызов inotify_and_watch() для явного удаления элемента, либо элемент был имплицитно удален ядром, так как объект мониторинга был удален, либо файловая система, в которой данный объект находился, была размонтирована. Событие IN_IGNORED не генерируется, когда элемент списка наблюдения с установленным флагом IN_ONESHOT автоматически удаляется после произошедшего события.
- Если субъект события — каталог, то вдобавок к другим битам в поле mask также будет установлен бит IN_ISDIR.
- Событие IN_UNMOUNT информирует приложение, что файловая система, содержавшая наблюдаемый объект, была размонтирована. После этого события будет доставлено следующее событие, содержащее бит IN_IGNORED.
- Флаг IN_Q_OVERFLOW описывается в разделе 19.5, где ведется обсуждение ограничений очереди событий inotify.

Поле cookie используется для объединения связанных друг с другом событий. В настоящее время оно применяется только при переименовании файла. Когда происходит это событие для каталога, содержавшего переименовываемый файл, генерируется событие IN_MOVE_FROM, а затем событие IN_MOVE_TO генерируется для каталога, содержащего переименованный файл. (Если файлу присваивается новое имя в одном и том же каталоге, то оба события происходят с данным каталогом.) В поле cookie этих двух событий будет храниться одно и то же уникальное значение, что позволит приложению связать их.

Если событие происходит с файлом внутри наблюдаемого каталога, то поле name используется для возврата строки, идентифицирующей файл и завершающейся нуль-символом. Если же событие происходит с самим объектом наблюдения, то поле name не применяется, а поле len будет содержать 0.

Поле len показывает, сколько байтов было выделено для поля name. Это поле является обязательным, так как между окончанием строки, сохраненной в поле name и началом следующей структуры inotify_event, содержащейся в буфере, возвращенном функцией read(), может находиться несколько заполняющих байтов (см. рис. 19.2). Длина каждого события inotify, таким образом, может быть вычислена как sizeof(struct notify_event) + len.

Если буфер, переданный функции read(), слишком мал, чтобы вместить еще и следующую структуру inotify_event, то, в целях предупреждения приложения об этом, функция read() завершается с ошибкой EINVAL. (В версиях ядра, предшествовавших версии 2.6.21, функция read() в таком случае возвращала значение 0. Изменение в пользу применения константы EINVAL дает более четкое указание на допущенную программную ошибку.) Приложение может ответить повторной попыткой вызова функции read() с буфером увеличенного размера. Однако этой проблемы можно вовсе избежать, удостоверяясь, что буфер имеет достаточный размер для хранения хотя бы одного события: размер буфера, переданного функции read(), должен быть не менее (sizeof(struct inotify_event) + + NAME_MAX + 1) байт, где NAME_MAX — максимальная длина имени файла плюс один байт для завершающего нуль-символа.

Использование размера буфера, большего, чем минимальный, позволяет приложению эффективно получать информацию о нескольких событиях с помощью лишь одного вызова функции read(). Эта функция из файлового дескриптора inotify возвращает минимальное количество доступных событий и количество событий, которое может поместиться в предоставленный буфер.

Вызов ioctl(fd, FIONREAD, &numbytes) возвращает количество байтов, доступных в настоящий момент для чтения, а также объект inotify, на который ссылается файловый описатель fd.

События, считываемые из файлового дескриптора `inotify`, формируют собой упорядоченную очередь. Таким образом, например, гарантируется, что при переименовании файла событие `IN_MOVED_FROM` будет прочитано перед событием `IN_MOVED_TO`.

При добавлении нового события в конец очереди событий ядро объединит данное событие с событием, находящимся в хвосте очереди (таким образом, фактически получается, что новое в очередь не ставится), если оба этих события имеют одинаковые значения полей `wd`, `mask`, `cookie` и `name`. Так происходит потому, что многим приложениям нет необходимости знать о повторяющихся экземплярах одного и того же события, а отбрасывание лишних событий уменьшает количество памяти (ядра), необходимой для хранения очереди. Однако это значит, что мы не можем использовать `inotify` для достоверного определения того, сколько раз или как часто происходило повторяющееся событие.

Пример программы

Несмотря на то что в предшествующем описании много деталей, интерфейс API `inotify` на самом деле достаточно прост в обращении. В листинге 19.1 показано его применение.

Листинг 19.1. Использование интерфейса API inotify

inotify/demo_inotify.c

```
#include <sys/inotify.h>
#include <limits.h>
#include "tlpi_hdr.h"

static void             /* Отобразить информацию из структуры inotify_event*/
displayInotifyEvent(struct inotify_event *i)
{
    printf("    wd =%2d; ", i->wd);
    if (i->cookie > 0)
        printf("cookie =%4d; ", i->cookie);

    printf("mask = ");
    if (i->mask & IN_ACCESS)         printf("IN_ACCESS ");
    if (i->mask & IN_ATTRIB)         printf("IN_ATTRIB ");
    if (i->mask & IN_CLOSE_NOWRITE)  printf("IN_CLOSE_NOWRITE ");
    if (i->mask & IN_CLOSE_WRITE)    printf("IN_CLOSE_WRITE ");
    if (i->mask & IN_CREATE)         printf("IN_CREATE ");
    if (i->mask & IN_DELETE)         printf("IN_DELETE ");
    if (i->mask & IN_DELETE_SELF)    printf("IN_DELETE_SELF ");
    if (i->mask & IN_IGNORED)        printf("IN_IGNORED ");
    if (i->mask & IN_ISDIR)          printf("IN_ISDIR ");
    if (i->mask & IN_MODIFY)         printf("IN_MODIFY ");
    if (i->mask & IN_MOVE_SELF)      printf("IN_MOVE_SELF ");
    if (i->mask & IN_MOVED_FROM)     printf("IN_MOVED_FROM ");
    if (i->mask & IN_MOVED_TO)       printf("IN_MOVED_TO ");
    if (i->mask & IN_OPEN)           printf("IN_OPEN ");
    if (i->mask & IN_Q_OVERFLOW)     printf("IN_Q_OVERFLOW ");
    if (i->mask & IN_UNMOUNT)        printf("IN_UNMOUNT ");
    printf("\n");

    if (i->len > 0)
        printf("        name = %s\n", i->name);
}
```

```
#define BUF_LEN (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

int
main(int argc, char *argv[])
{
    int inotifyFd, wd, j;
    char buf[BUF_LEN];
    ssize_t numRead;
    char *p;
    struct inotify_event *event;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname... \n", argv[0]);

(1) inotifyFd = inotify_init();          /* Создать объект inotify */
    if (inotifyFd == -1)
        errExit("inotify_init");

    for (j = 1; j < argc; j++) {
(2)     wd = inotify_add_watch(inotifyFd, argv[j], IN_ALL_EVENTS);
        if (wd == -1)
            errExit("inotify_add_watch");
        printf("Watching %s using wd %d\n", argv[j], wd);
    }

    for (;;) {                           /* Читать события вечно */
(3)     numRead = read(inotifyFd, buf, BUF_LEN);
        if (numRead == 0)
            fatal("read() из inotify fd вернула 0!");

        if (numRead == -1)
            errExit("read");

        printf("Read %ld bytes from inotify fd\n", (long) numRead);

        /* Обработка всех событий из буфера, переданного read() */

        for (p = buf; p < buf + numRead; ) {
(4)         event = (struct inotify_event *) p;
            displayInotifyEvent(event);
            p += sizeof(struct inotify_event) + event->len;
        }
    }

    exit(EXIT_SUCCESS);
}
```

inotify/demo_inotify.c

Программа из листинга 19.1 выполняет следующие шаги:

- использует функцию `inotify_init()` для создания файлового дескриптора `inotify` (1);
- задействует функцию `inotify_add_watch()` для добавления элемента списка наблюдения для каждого из файлов, перечисленных в аргументах командной строки программы (2). Каждый из элементов списка осуществляет мониторинг всех возможных событий;
- выполняет бесконечный цикл, который:
  - читает буфер событий из файлового дескриптора `inotify` (3);

- вызывает функцию для `displayInotifyEvent()` отображения содержимого каждой структуры `inotify_event` в буфере ❹.

Следующая сессия оболочки демонстрирует применение программы в листинге 19.1. Мы запускаем экземпляр программы, работающей в фоновом режиме и осуществляющей мониторинг двух каталогов:

```
$ ./demo_inotify dir1 dir2 &
[1] 5386
Watching dir1 using wd 1
Watching dir2 using wd 2
```

Затем запускаем выполнение команд, генерирующих события в двух каталогах. Начинаем с создания файла с помощью команды `cat(1)`:

```
$ cat > dir1/aaa
Read 64 bytes from inotify fd
    wd = 1; mask = IN_CREATE
        name = aaa
wd = 1; mask = IN_OPEN
        name = aaa
```

Вышеприведенный вывод, созданный фоновой программой, показывает, что функция `read()` передала буфер, содержащий два события. Мы продолжим вводом некой информации в файл и завершающего *нуль-символа*:

```
Hello world
Read 32 bytes from inotify fd
    wd = 1; mask = IN_MODIFY
        name = aaa
Нажмите Ctrl-D
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CLOSE_WRITE
    name = aaa
```

Затем мы переименуем файл с переносом в другой каталог под наблюдением. Это приведет к генерации двух событий: одного для каталога, из которого файл переносится (дескриптор наблюдения 1) и второго — для каталога, в который файл будет перенесен (дескриптор наблюдения 2):

```
$ mv dir1/aaa dir2/bbb
Read 64 bytes from inotify fd
    wd = 1; cookie = 548; mask = IN_MOVED_FROM
        name = aaa
    wd = 2; cookie = 548; mask = IN_MOVED_TO
        name = bbb
```

Значения полей `cookie` обоих событий совпадают, что позволяет приложению связать их.

При создании подкаталога в одном из наблюдаемых каталогов маска генерируемого при этом события содержит бит `IN_ISDIR`, показывающий, что субъект события является каталогом:

```
$ mkdir dir2/ddd
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CREATE IN_ISDIR
        name = ddd
```

На данном этапе важно повторить, что мониторинг `inotify` не является рекурсивным. Если бы приложению потребовалось осуществлять мониторинг событий в только что

созданном каталоге, то ему нужно было бы совершить еще один вызов `inotify_add_watch()` с указанием путевого имени данного подкаталога.

Наконец, мы удаляем один из наблюдаемых подкаталогов:

```
$ rmdir dir1
Read 32 bytes from inotify fd
    wd = 1; mask = IN_DELETE_SELF
    wd = 1; mask = IN_IGNORED
```

Последнее событие, `IN_IGNORED`, было сгенерировано для информирования приложения о том, что ядро удалило данный элемент из списка наблюдения.

19.5. Ограничения очереди и файлы /proc

Создание очереди событий `inotify` требует определенного количества памяти ядра. По этой причине ядро устанавливает различные ограничения на использование механизма `inotify`. Суперпользователь может изменять их с помощью трех файлов в каталоге `/proc/sys/fs/inotify`:

- `max_queued_events` — при вызове функции `inotify_init()` это значение используется для установки верхнего ограничения количества событий, которые могут быть поставлены в очередь нового объекта `inotify`. Если ограничение достигнуто, то происходит генерация события `IN_Q_OVERFLOW` и удаление лишних событий. Поле `wd` события переполнения очереди будет иметь значение `-1`;
- `max_user_instances` — ограничение количества экземпляров объекта `inotify`, которые могут быть созданы для одного реального пользовательского идентификатора;
- `max_user_watches` — ограничение количества элементов списка наблюдения, которые могут быть созданы для одного реального пользовательского идентификатора.

По умолчанию значения, установленные в этих трех файлах, — `16384`, `128` и `8192` соответственно.

19.6. Старая система мониторинга событий файлов: dnotify

Linux предоставляет еще один механизм мониторинга событий файлов. Этот механизм под названием `dnotify` был доступен, начиная с версии ядра 2.4, однако после появления механизма `inotify` устарел. По сравнению с `inotify` механизм `dnotify` страдает целым рядом ограничений.

- Механизм `dnotify` оповещения о событиях реализован в форме отправки сигналов в приложение. Применение сигналов в качестве механизма оповещения усложняет разработку приложений (раздел 22.12). Кроме того, это затрудняет использование механизма `dnotify` в рамках библиотеки, так как вызывающая программа может поменять диспозицию сигнала (-ов) оповещения. В механизме `inotify` сигналы не задействуются. Единицей мониторинга механизма `dnotify` является каталог. Приложение получает оповещения, когда над любым файлом из данного каталога производится некое действие. В противоположность этому механизм `inotify` может использоваться для мониторинга каталогов или отдельных файлов.
- Для мониторинга каталога механизму `dnotify` необходимо, чтобы приложение открыло файловый дескриптор этого каталога. Применение файловых дескрипторов

приводит к возникновению двух проблем. Во-первых, из-за того, что файловая система, содержащая данный каталог, занята, она не может быть размонтирована. Во-вторых, так как для каждого каталога требуется отдельный файловый дескриптор, приложение может в конце концов потребить большое количество дескрипторов. Так как механизм `inotify` не использует файловые дескрипторы, описанных проблем удается избежать.

- Информация о событиях, предоставляемая механизмом `dnotify`, менее точна по сравнению с информацией, предоставляемой `inotify`. Так, при изменении файла в наблюдаемом каталоге `dnotify` сообщает нам, что произошло событие, однако не сообщает, какой файл стал частью данного события. Приложение может определить это перехватом информации о содержимом каталога. Более того, механизм `inotify` предоставляет более детализированную информацию о произошедшем событии по сравнению с `dnotify`.
- В некоторых случаях механизм `dnotify` не предоставляет надежную информацию о событиях файлов.

Дополнительную информацию о механизме `dnotify` можно найти на странице `fcntl(2)` руководства в разделе с описанием `F_NOTIFY`, а также в ресурсах с информацией о ядре `Documentation/dnotify.txt`.

19.7. Резюме

Специфичный механизм Linux `inotify` позволяет приложению получать уведомления, когда с набором наблюдаемых файлов и каталогов происходят различные события (файлы открываются, закрываются, создаются, удаляются, изменяются, переименовываются и т. д.). Механизм `inotify` вытеснил старый механизм `dnotify`.

19.8. Упражнение

19.1. Напишите программу, которая протоколировала бы все создания, удаления и переименования внутри каталога, указанного в аргументе командной строки программы. Она должна осуществлять мониторинг событий во всех подкаталогах внутри указанного каталога. Для получения списка всех подкаталогов вам потребуется воспользоваться функцией `nftw()` (см. раздел 18.9). При добавлении нового подкаталога в дерево или при удалении каталога набор наблюдаемых подкаталогов должен быть обновлен соответствующим образом.

20 Сигналы: фундаментальные концепции

В этой и двух следующих главах мы обсуждаем сигналы. Несмотря на то что сами фундаментальные концепции достаточно просты, наше обсуждение будет довольно длинным из-за большого количества деталей, которые следует охватить.

В этой главе мы рассмотрим:

- различные сигналы и их предназначение;
- обстоятельства, при которых ядро может сгенерировать сигнал процессу, а также системные вызовы, которые один процесс может использовать для отправки сигнала другому;
- каким образом процесс отвечает на сигнал по умолчанию, а также средства, с помощью которых процессы могут изменять свой ответ на сигнал, в частности, за счет использования обработчика сигнала — программно-определенной функции, активируемой автоматически в случае получения сигнала;
- использование сигнальной маски процесса для блокировки сигналов, а также связанного с ней понятия ожидающих сигналов;
- как процесс может приостановить выполнение и подождать доставки сигнала.

20.1. Концепции и общие сведения

Сигнал — это оповещение процесса о том, что произошло некое событие. Иногда сигналы также описываются как *программные прерывания*. Сигналы аналогичны аппаратным прерываниям в том смысле, что они останавливают нормальное выполнение программы. В большинстве случаев невозможно предсказать, когда именно будет доставлен тот или иной сигнал.

Один процесс может (при наличии необходимых разрешений) отправить сигнал другому процессу. При таком использовании сигналы могут рассматриваться как технология синхронизации либо даже как примитивная форма межпроцессного взаимодействия (IPC). Процесс также может отправить сигнал самому себе. Однако обычно источником большинства сигналов, отправляемых в процесс, является ядро. Далее перечислены несколько типов событий, свершение которых заставляет ядро сгенерировать сигнал для процесса.

- Произошло аппаратное исключение. Это значит, что аппаратное обеспечение зафиксировало неверное состояние и оповестило об этом ядро, которое, в свою очередь, направило соответствующий сигнал затронутому процессу. Примерами аппаратного исключения могут быть выполнение ошибочного машинного кода, деление на ноль, а также обращение к недоступному участку памяти.
- Пользователь ввел один из специальных символов терминала, генерирующих сигналы. Примерами таких символов могут быть символ *прерывания* (обычно Ctrl+C) и символ *приостановки* (обычно Ctrl+Z).
- Произошло программное событие. Например, появился ввод из файлового дескриптора, размер окна терминала был изменен, сработал таймер, для процесса было превышено временное ограничение ЦП либо был завершен дочерний процесс данного процесса.

Каждому сигналу присваивается уникальный идентификатор — (небольшое) целое число, нумерация задается последовательно по возрастанию, начиная с 1. Эти целые числа определены в файле `<signal.h>` с символьными именами в формате `SIGxxxx`. Поскольку

числа для каждого из сигналов отличаются в зависимости от реализации, в программах всегда применяются символьные имена. Например, когда пользователь вводит символ *прерывания*, процессу доставляется сигнал `SIGINT` (с номером 2).

Сигналы можно разделить на две большие категории. Первый набор сигналов состоит из *традиционных*, или *стандартных*, сигналов, которые используются ядром для оповещения процессов о свершении событий. В Linux стандартные сигналы пронумерованы от 1 до 31. Мы опишем их в этой главе. Второй набор состоит из сигналов *реального времени*, отличия которых от стандартных описаны в разделе 22.8.

Говорят, что сигнал *генерируется* каким-либо событием. После генерации происходит *доставка* в процесс, который затем выполняет определенные действия для ответа на полученный сигнал. После того, как сигнал был сгенерирован, и до его доставки, говорят, что сигнал находится в *состоянии ожидания*.

Как правило, сигнал в режиме ожидания доставляется в процесс при его следующем запланированном выполнении либо моментально, если процесс уже запущен (например, если процесс отправил сигнал самому себе). Но иногда мы должны быть уверены в том, что выполнение фрагмента кода не будет прервано доставкой сигнала. Для этого мы можем добавить сигнал в *сигнальную маску* процесса, то есть набор сигналов, доставка которых временно *заблокирована*. Если сигнал был сгенерирован, в то время как такие сигналы заблокированы, он остается в режиме ожидания до тех пор, пока не будет разблокирован (удален из сигнальной маски). Разные системные вызовы позволяют процессу добавлять и удалять сигналы из сигнальной маски. По получении сигнала процесс по умолчанию выполняет одно из нижеперечисленных действий, в зависимости от полученного сигнала.

- Сигнал *игнорируется*, то есть сбрасывается ядром и не влияет на процесс. (Процесс никогда даже не узнает о том, что данный сигнал был отправлен.)
  Процесс *завершается*. Иногда это называют *аварийным завершением процесса*, что противопоставляется нормальному завершению — при использовании функции `exit()`.
- Генерируется *файл дампа ядра*, и процесс завершается. Файл дампа ядра содержит отпечаток виртуальной памяти процесса, который может быть загружен в отладчик для изучения состояния процесса в момент завершения.
- Процесс *останавливается* — выполнение процесса приостанавливается.
- Выполнение процесса *возобновляется* после предшествовавшей приостановки.

Для каждого конкретного сигнала программа может изменить действие, выполняемое по его получении. Это называют установкой *диспозиции* сигнала. Программа может установить одну из нижеперечисленных диспозиций сигнала.

- Следует выполнить *действие по умолчанию*. Эта инструкция применяется для отмены изменения настройки диспозиции сигнала.
- Сигнал *игнорируется*. Эта инструкция используется для сигналов, действие по умолчанию для которых — завершение процесса.
- Выполняется *обработчик сигнала*.

Обработчик сигнала — функция, написанная программистом и выполняющая нужные действия при получении сигнала. Например, для оболочки установлен обработчик сигнала `SIGINT` (генерируемого символом *прерывания*, Ctrl+C), заставляющий оболочку прекратить выполнение текущей задачи и вернуть управление в основной цикл так, что пользователь опять видит на экране приглашение на ввод команды. Оповещение ядра о необходимости активации функции обработчика сигнала называют *установкой* или *настройкой* обработчика сигнала. Когда в ответ на получение сигнала программа активирует соответствующий обработчик, мы говорим, что сигнал был *обработан* или, что синонимично, *перехвачен*.

Обратите внимание, что невозможно установить диспозицию сигнала таким образом, чтобы он *завершал* процесс или *сбрасывал дамп ядра* (за исключением случая, когда это

является диспозицией по умолчанию). Максимум, как мы можем приблизиться к этому, — это установить обработчик сигнала, который вызывает одну из двух функций: `exit()` или `abort()`. Функция `abort()` (см. подраздел 21.2.2) генерирует для процесса сигнал `SIGABRT`, который приводит к сбросу дампа ядра и завершению процесса.

> Характерный для Linux файл /proc/PID/status содержит различные поля битовых масок, которые можно изучить для выяснения того, каким образом процесс обрабатывает сигналы. Битовые маски представлены в виде шестнадцатеричных чисел, при этом наименьший бит представляет сигнал номер 1, а следующий бит влево представляет сигнал номер 2 и т. д. Это поля SigPnd (ожидающие сигналы по потокам), ShdPnd (ожидающие сигналы процесса, начиная с версии Linux 2.6), SigBlk (заблокированные сигналы), SigIgn (игнорируемые сигналы), SigCgt (перехваченные сигналы). (Различие между сигналами SigPnd и ShdPnd станет ясным после того, как мы рассмотрим обработку сигналов в многопоточных процессах в разделе 33.2.) Аналогичная информация может быть получена с использованием различных параметров команды ps(1).

Сигналы, появившиеся в самых ранних реализациях UNIX, на сегодняшний день подверглись нескольким существенным изменениям. В ранних реализациях при определенных обстоятельствах сигналы могли быть потеряны (то есть не доставлены в целевой процесс). Более того, несмотря на то, что средства для блокировки сигналов при выполнении важного кода предоставлялись, они не были надежными. Эти проблемы были исправлены в системе 4.2BSD, которая предоставляла так называемые *надежные сигналы*. (Еще одной инновацией системы BSD была реализация дополнительных сигналов для поддержки управления задачами оболочки, которые будут описаны в разделе 34.7.)

В систему System V была добавлена надежная семантика сигналов, однако при этом использовалась модель, несовместимая с BSD. Эти несовместимости были разрешены только с появлением стандарта POSIX.1-1990, который установил спецификацию надежных сигналов, по большей части основанную на модели BSD.

Мы рассматриваем детали надежных и ненадежных сигналов в разделе 22.7, а также вкратце описываем старые API сигналов BSD в разделе 22.13.

20.2. Типы сигналов и действия по умолчанию

Ранее мы говорили, что стандартные сигналы Linux пронумерованы от 1 до 31. Однако на странице справочника `signal(7)` приводится больше имен сигнала. Для этого есть несколько причин. Одни имена — просто синонимы иных имен и определяются только для обеспечения совместимости файлов исходного кода с другими реализациями UNIX. Другие имена определены, но не используются. В следующем списке приведено описание различных сигналов.

- `SIGABRT` — процессу отправляется данный сигнал при вызове функции `abort()` (см. подраздел 21.2.2). По умолчанию данный сигнал завершает процесс с дампом ядра. Это позволяет достичь той цели, для которой предназначен вызов функции `abort()`: создание дампа ядра для отладки.
- `SIGALRM` — ядро генерирует данный сигнал по истечении времени таймера реального времени, установленного вызовом функции `alarm()` или `setitimer()`. Таймер реального времени отсчитывает время как настенные часы (то есть это привычное для человека понятие прошедшего времени). Для получения дополнительных сведений см. раздел 23.1.
- `SIGBUS` — этот сигнал («ошибка шины») генерируется для обозначения определенных ошибок доступа к памяти. Одна из таких ошибок может произойти при использовании отображения памяти с помощью функции `mmap()` во время попытки доступа к адресу, находящемуся вне пределов соответствующего файла отображаемой памяти (см. подраздел 45.4.3).

- SIGCHLD — этот сигнал отправляется (ядром) родительскому процессу при завершении одного из его потомков (как при вызове функции exit(), так и в результате завершения по сигналу). Может быть отправлен в процесс при завершении или возобновлении одного из дочерних процессов по сигналу. Сигнал SIGCHLD детально описан в разделе 26.3.
- SIGCLD — синоним SIGCHLD.
- SIGCONT — при отправке в остановленный процесс данный сигнал возобновляет его (то есть перепланирует запуск процесса позднее). При получении этого сигнала действующим процессом данный сигнал по умолчанию игнорируется. Процесс может перехватить этот сигнал — и выполнить некое действие при возобновлении. Более подробное описание SIGCONT дано в разделах 22.2 и 34.7.
- SIGEMT — в системах UNIX данный сигнал применяется для обозначения аппаратной ошибки, зависящей от реализации. В Linux же данный сигнал используется только в реализации Sun SPARC. Суффикс EMT восходит к emulator trap — ассемблерной инструкции на мини-ЭВМ Digital PDP-11.
- SIGFPE — генерируется для арифметических ошибок определенных видов, таких как деление на ноль. Суффикс FPE — сокращение от английского *floating-point exception* (исключение в операции с плавающей точкой), однако этот сигнал может быть сгенерирован и для ошибок в арифметических действиях с целыми числами. Точные детали того, когда генерируется этот сигнал, зависят от аппаратной архитектуры и настроек контрольных регистров ЦПУ. Например, на x86-32 целочисленное деление на ноль всегда вызывает SIGFPE, однако обработка деления на ноль числа с плавающей точкой зависит от того, установлен ли флаг исключения FE_DIVBYZERO. Если флаг этого исключения установлен (с помощью функции feenableexcept()), то деление на ноль числа с плавающей точкой генерирует сигнал SIGFPE. В противном случае выводится результат для операндов в соответствии со стандартом IEEE (представление бесконечности в формате числа с плавающей точкой). Для получения более подробной информации см. страницу справочника fenv(3) и файл <fenv.h>.
- SIGHUP — при отключении (отсоединении) терминала этот сигнал отправляется в процесс, управляющий терминалом. Мы рассматриваем концепцию контролирующего процесса и различные ситуации, при которых происходит отправка сигнала SIGHUP, в разделе 34.6. Сигнал SIGHUP также находит применение с демонами (например, init, httpd и inetd). Многие демоны настроены на повторную инициализацию и повторное чтение своих конфигурационных файлов при получении сигнала SIGHUP. Системный администратор запускает эти действия, вручную отправляя сигнал SIGHUP в фоновый процесс либо выполнив команду kill, либо путем запуска программы или сценария, выполняющего ту же операцию.
- SIGILL — этот сигнал направляется процессу, который пытается выполнить запрещенный (то есть неверно сформулированный) машинный код.
- SIGINFO — в Linux имя этого сигнала — синоним сигнала SIGPWR. В системах BSD сигнал SIGINFO, генерируемый при нажатии Ctrl+T, применяется для получения статусной информации о группе приоритетных процессов.
- SIGINT — когда пользователь вводит в терминал символ *прерывания* (обычно это Ctrl+C), драйвер терминала посылает этот сигнал группе приоритетных процессов. Действие по умолчанию для данного сигнала — завершение процесса.
- SIGIO — с помощью системного вызова fcntl() можно организовать генерацию данного сигнала при свершении события ввода-вывода (например, переход ввода в состояние «доступен») с некоторыми типами открытых дескрипторов файлов, например, для терминалов и сокетов. Эта возможность описана более подробно в разделе 63.3.
- SIGIOT — в Linux это синоним сигнала SIGABRT. В некоторых других реализациях UNIX данный сигнал указывает на аппаратную ошибку, зависящую от реализации.

- SIGKILL — это сигнал *императивного завершения процесса*. Данный сигнал не может быть заблокирован, проигнорирован или перехвачен обработчиком, а следовательно, всегда завершает процесс.
- SIGLOST — имя этого сигнала существует в Linux, однако оно не используется. В некоторых других реализациях UNIX клиент NFS посылает сигнал в локальные процессы, удерживающие блокировки, если клиенту NFS не удается восстановить удерживаемые процессом блокировки после восстановления удаленного сервера NFS, аварийно завершившего работу. (Эта функция не стандартизирована в спецификациях NFS.)
- SIGPIPE — генерируется, когда процесс пытается выполнить запись в канал (конвейер), FIFO или сокет, для которых нет соответствующего процесса чтения. Как правило, это происходит, если файловый дескриптор читающего процесса был закрыт для канала IPC (что бы это ни значило) . См. раздел 44.2 для получения более подробной информации.
- SIGPOLL — унаследован от System V, в Linux является синонимом сигнала SIGIO.
- SIGPROF — ядро генерирует этот сигнал по окончании времени профилирующего таймера, установленного вызовом функции setitimer() (см. раздел 23.1). Профилирующий таймер — это таймер, отсчитывающий время ЦПУ, затрачиваемое процессом. В отличие от виртуального (см. SIGVTALRM ниже), профилирующий таймер отсчитывает время ЦПУ, затрачиваемое как в режиме пользователя, так и в режиме ядра.
- SIGPWR — это сигнал *сбоя подачи питания*. В системах с источниками бесперебойного питания (ИБП) возможно установить фоновый процесс, который бы осуществлял мониторинг уровня заряда вспомогательного аккумулятора в случае сбоя подачи питания. Если заряд батареи близок к нулю (после длительного отсутствия подачи электрического тока), то процесс, осуществляющий мониторинг, отправляет сигнал SIGPWR в процесс init, интерпретирующий данный сигнал как требование завершить работу системы быстро и в установленном порядке.
- SIGQUIT — когда пользователь вводит символ выхода (обычно Ctrl+\) с клавиатуры, этот сигнал посылается группе приоритетных процессов. По умолчанию он завершает процесс и создает дамп ядра, который затем может быть использован для отладки. Применение SIGQUIT таким образом может быть полезным, если программа застряла в бесконечном цикле или не отвечает по другим причинам. Нажимая Ctrl+\, затем загружая получившийся дамп ядра с помощью отладчика gdb и используя команду backtrace для получения трассировки стека, мы можем выяснить, какая часть программного кода выполнялась. ([Matloff, 2008] описывает использование gdb.)
- SIGSEGV — очень распространенный сигнал. Генерируется, когда программа обращается по неверной ссылке на ячейку в памяти. Ссылка может быть неверной из-за того, что страница, на которую ссылаются, не существует (например, находится в неразмеченной области где-то между кучей и стеком), процесс пытался обновить участок в памяти только для чтения (например, текстовый сегмент программы или участок отображаемой памяти, помеченный «только для чтения») или процесс пытался получить доступ к части памяти ядра, работая в режиме пользователя (см. раздел 2.1). В языке C такие события часто являются результатом разыменования указателя, содержащего неверный адрес (например, неинициализированного указателя) или передачи неверного аргумента в вызов функции. Название сигнала происходит от термина *segmentation violation* — «нарушение сегментации».
- SIGSTKFLT — задокументирован в signal(7) как «ошибка стека на сопроцессоре». Сигнал определен, но не используется в Linux.
- SIGSTOP — сигнал императивной остановки процесса. Не может быть заблокирован, проигнорирован или перехвачен обработчиком, то есть он всегда останавливает процесс.
- SIGSYS — генерируется, если процесс совершает «плохой» системный вызов. Это значит, что процесс выполнил инструкцию, которая была интерпретирована как прерывание системного вызова, но номер связанного системного вызова не был допустимым.

- `SIGTERM` — стандартный сигнал, который применяется для завершения процесса и по умолчанию посылается командами `kill` и `killall`. Иногда пользователи явно посылают сигнал `SIGKILL` процессу, используя команды `kill -KILL` или `kill -9`. Но это, как правило, ошибка. У хорошо спроектированного приложения будет обработчик для `SIGTERM`, который вызывает корректное завершение приложения, позволяя ему стереть временные файлы и заблаговременно высвободить другие ресурсы. Завершение процесса с помощью `SIGKILL` обходит обработчик `SIGTERM`. Следовательно, мы всегда должны сначала попытаться завершить процесс, используя `SIGTERM`, и приберечь `SIGKILL` в качестве последнего средства для завершения вышедших из-под контроля процессов, не отвечающих на `SIGTERM`.
- `SIGTRAP` — сигнал используется для реализации точек прерывания отладчика и отслеживания системных вызовов, выполняемых утилитой `strace(1)`. Для получения более подробной информации см. страницу справочника, посвященную вызову `ptrace(2)`.
- `SIGTSTP` — это сигнал *стоп*, отправляемый группе приоритетных процессов, когда пользователь вводит с клавиатуры символ *приостановки* (обычно Ctrl+Z). В главе 34 в деталях описываются группы процессов (задания), собственно контроль задания, а также когда и как программе может потребоваться обработать данный сигнал. Имя этого сигнала происходит от английского *terminal stop* — «остановка с терминала».
- `SIGTTIN` — во время работы в оболочке, управляющей заданием, драйвер терминала отправляет данный сигнал группе фоновых процессов при попытке чтения (`read()`) с управляющего терминала. По умолчанию данный сигнал останавливает процесс.
- `SIGTTOU` — этот сигнал похож на `SIGTTIN`, но используется при попытке фонового задания осуществить вывод на управляющий терминал. При работе в оболочке, управляющей заданием, с включенным (возможно, посредством команды `stty tostop`) параметром терминала `TOSTOP` *(остановка вывода на терминал)*, драйвер терминала отправляет сигнал `SIGTTOU` группе фоновых процессов при попытке записи (`write()`) в терминал (см. подраздел 34.7.1). По умолчанию данный сигнал останавливает процесс.
- `SIGUNUSED` — как можно понять из имени, данный сигнал не применяется. Во многих архитектурах начиная с версии Linux 2.4 данный сигнал синонимичен `SIGSYS`. Иными словами, во многих архитектурах данный сигнал более не является неиспользуемым, однако имя этого сигнала сохраняется для гарантии обратной совместимости.
- `SIGURG` — этот сигнал отправляется в процесс для указания наличия на сокете доступных для чтения срочных данных (см. подраздел 57.13.1).
- `SIGUSR1` — этот сигнал вместе с `SIGUSR2` доступен для целей, определяемых программистом. Ядро никогда не генерирует эти сигналы для процесса. Процессы могут использовать их для оповещения друг друга о свершении событий или для синхронизации друг с другом. В ранних реализациях UNIX это были единственные два сигнала, которые можно было свободно задействовать в приложениях. (На самом деле процессы могут отправлять друг другу любые сигналы, однако это может привести к путанице, если ядро также генерирует один из этих сигналов для процесса.) В современных реализациях UNIX предоставляется широкий спектр сигналов реального времени, которые также доступны для целей, определяемых программистом (см. раздел 22.8).
- `SIGUSR2` — см. описание сигнала `SIGUSR1`.
- `SIGVTALRM` — ядро генерирует этот сигнал по окончании времени виртуального таймера, установленного вызовом функции `setitimer()` (см. раздел 23.1) Виртуальный таймер — это таймер, отсчитывающий время ЦПУ, затраченное процессом в режиме пользователя.
- `SIGWINCH` — в оконной среде данный сигнал отправляется группе приоритетных процессов при изменении размеров окна терминала (как следствие ручного изменения размеров окна пользователем или программного изменения размеров с помощью вызова функции `ioctl()`, как описано в разделе 58.9). Благодаря установке обработчика

этого сигнала такие программы, как `vi` или `less`, могут узнать о необходимости перерисовать вывод после изменения размеров окна.

- `SIXCPU` — данный сигнал направляется процессу по исчерпании выделенного ему времени ЦПУ (константа `RLIMIT_CPU` описывается в разделе 36.3).
- `SIGXFSZ` — этот сигнал направляется процессу при попытке (с помощью функций `write()` или `truncate()`) увеличить размер файла за пределы ресурса процесса (константа `RLIMIT_FSIZE` описывается в разделе 36.3).

В табл. 20.1 приводятся обобщенные сведения о сигналах в Linux. Обратите внимание на следующие детали об информации в таблице.

- В столбце «Номер сигнала» представлен номер, присвоенный данному сигналу в различных аппаратных архитектурах. Если не указано иное, сигналы имеют один и тот же номер во всех архитектурах. Архитектурные различия в номерах сигналов даются в скобках и встречаются в архитектурах Sun SPARC и SPARC64 (S), HP/Compaq/Digital Alpha (A), MIPS (M) и HP PA-RISC (P). В этом столбце приписка «н/опр» означает, что для данной архитектуры символ не определен.
- Столбец «SUSv3» показывает, стандартизирован ли сигнал по системе SUSv3.
- Столбец «По умолчанию» содержит информацию о действии сигнала по умолчанию. Условное обозначение «Заверш.» говорит о том, что сигнал завершает процесс, «Ядро» — что процесс создает файл дампа ядра и завершается, «Игнор.» — что сигнал игнорируется, «Стоп» означает, что сигнал останавливает процесс, а «Прод.» — что возобновляет остановленный процесс.

Некоторые из перечисленных ранее сигналов в табл. 20.1 не приводятся: SIGCLD (синоним SIGCHLD), SIGINFO (не используется), SIGIOT (синоним SIGABRT), SIGLOST (не используется) и SIGUNUSED (синоним SIGSYS во многих архитектурах).

Таблица 20.1. Сигналы Linux

Имя	Номер сигнала	Описание	SUSv3	По умолчанию
SIGABRT	6	Аварийно завершить процесс	+	Ядро
SIGALRM	14	Время таймера реального времени истекло	+	Заверш.
SIGBUS	7 (SAMP=10)	Ошибка доступа к памяти	+	Ядро
SIGCHLD	17 (SA=20, MP=18)	Дочерний процесс завершен или остановлен	+	Игнор.
SIGCONT	18 (SA=19, M=25, P=26)	Продолжить, если завершен	+	Прод.
SIGEMT	н/опр (SAMP=7)	Аппаратная ошибка		Заверш.
SIGFPE	8	Арифметическое исключение	+	Ядро
SIGHUP	1	Потеря соединения	+	Заверш.
SIGILL	4	Недопустимая инструкция	+	Ядро
SIGINT	2	Прерывание с терминала	+	Заверш.
SIGIO/SIGPOLL	29 (SA=23, MP=22)	Возможен ввод/вывод	+	Заверш.
SIGKILL	9	Императивное завершение	+	Заверш.

Имя	Номер сигнала	Описание	SUSv3	По умолчанию
SIGPIPE	13	Нарушенный канал	+	Заверш.
SIGPROF	27 (M=29, P=21)	Закончилось время профилирующего таймера	+	Заверш.
SIGPWR	30 (SA=29, MP=19)	Подача питания скоро прекратится		Заверш.
SIGQUIT	3	Выход с терминала	+	Ядро
SIGSEGV	11	Неверная ссылка памяти	+	Ядро
SIGSTKFLT	16 (SAM=н/опр, P=36)	Ошибка стека на сопроцессоре		Заверш.
SIGSTOP	19 (SA=17, M=23, P=24)	Императивная остановка	+	Стоп
SIGSYS	31 (SAMP=12)	Неверный системный вызов	+	Ядро
SIGTERM	15	Завершить процесс	+	Заверш.
SIGTRAP	5	Ловушка точки прерывания	+	Ядро
SIGTSTP	20 (SA=18, M=24, P=25)	Остановка с терминала	+	Стоп
SIGTTIN	21 (M=26, P=27)	Чтение с терминала ФП	+	Стоп
SIGTTOU	22 (M=27, P=28)	Запись в терминал ФП	+	Стоп
SIGURG	23 (SA=16, M=21, P=29)	Срочные данные на сокете	+	Игнор.
SIGUSR1	10 (SA=30, MP=16)	Пользовательский сигнал 1	+	Заверш.
SIGUSR2	12 (SA=31, MP=17)	Пользовательский сигнал 2	+	Заверш.
SIGVTALRM	26 (M=28, P=20)	Закончилось время виртуального таймера	+	Заверш.
SIGWINCH	28 (M=20, P=23)	Изменен размер окна терминала		Игнор.
SIGXCPU	24 (M=30, P=33)	Превышено ограничение времени ЦПУ	+	Ядро
SIGXFSZ	25 (M=31, P=34)	Превышено ограничение размера файла	+	Ядро

Обратите внимание на следующие замечания касаемо поведения по умолчанию некоторых сигналов из табл 20.1.

- В Linux 2.2 действие по умолчанию для сигналов `SIGXCPU`, `SIGXFSZ`, `SIGSYS` и `SIGBUS` — завершить процесс без создания файла дампа ядра. Начиная с версии ядра 2.4, Linux соответствует требованиям стандарта SUSv3, требующим, чтобы эти сигналы завершали процесс с генерацией дампа ядра. В некоторых других реализациях UNIX сигналы `SIGXCPU` и `SIGXFSZ` обрабатываются так же, как и в Linux 2.2.
- Сигнал `SIGPWR` в иных реализациях UNIX (где встречается) по умолчанию игнорируется.

- Сигнал SIGIO в некоторых реализациях UNIX (в частности, производных от BSD) по умолчанию игнорируется.
- Сигнал SIGEMT не утвержден стандартами, но встречается практически во всех реализациях UNIX. Однако в других реализациях он приводит к завершению с дампом ядра.
- В стандарте SUSv1 действие по умолчанию для сигнала SIGURG было указано как завершение процесса, оно является действием по умолчанию в некоторых старых реализациях UNIX. В стандарте SUSv2 была принята текущая спецификация (игнорирование).

20.3. Изменение диспозиций сигналов: signal()

В системах UNIX предоставляется два способа изменения диспозиции сигнала: signal() и sigaction(). Системный вызов signal(), описываемый в этом разделе, изначально был API для установки диспозиции сигнала, кроме того, данный вызов предлагает более простой интерфейс по сравнению с sigaction(). С другой стороны, sigaction() предоставляет функционал, недоступный в signal(). Существует несколько вариаций поведения функции signal() в различных реализациях UNIX (см. раздел 22.7), а это значит, что данная функция никогда не должна использоваться для установки обработчиков сигналов в переносимых программах. Из-за обозначенных проблем с переносимостью функция sigaction() является предпочтительным API для установки обработчиков сигналов. После того как мы объясним применение sigaction() в разделе 20.13, мы всегда будем задействовать данный вызов для установки обработчиков сигналов в примерах программ.

> Несмотря на то что функция signal() задокументирована во втором разделе справочных страниц Linux, на самом деле эта функция реализована в glibc как библиотечная, реализованная поверх системного вызова sigaction().

```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
```

Возвращает предыдущую диспозицию сигнала при успешном завершении или SIG_ERR при ошибке

Прототип функции signal() требует небольших пояснений. Первый аргумент, sig, обозначает сигнал, диспозицию которого мы хотим изменить. Второй аргумент, handler, — это адрес функции, которую нужно вызывать по получении данного сигнала. Эта функция ничего не возвращает (void) и принимает один целочисленный аргумент. Таким образом, обработчик сигнала имеет следующую общую форму:

```
void
handler(int sig)
{
    /* Код обработчика */
}
```

Мы описываем предназначение аргумента sig функции обработчика в разделе 20.4.

Возвращаемое значение функции signal() — это предыдущая диспозиция сигнала. Как и аргумент handler, данное значение — указатель на функцию, ничего не возвращающую и принимающую один целочисленный аргумент. Иными словами, мы могли бы написать код, подобный тому, что приведен ниже, для временной установки обработчика сигнала и последующего сброса его диспозиции до любого предшествовавшего значения:

```
void (*oldHandler)(int);

oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
    errExit("signal");
/* Сделать здесь что-то еще. Если в это время получен SIGINT,
   для обработки сигнала будет использован newHandler. */
if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("signal");
```

С помощью функции signal() невозможно получить текущую диспозицию сигнала без одновременного изменения ее значения. Чтобы сделать это, нам необходимо воспользоваться функцией sigaction().

Можно сделать прототип функции `signal()` более понятным, применяя следующее определение типа для указателя на функцию обработчика сигнала:

```
typedef void (*sighandler_t)(int)
```

Это позволяет переписать прототип функции `signal()` таким образом:

```
sighandler_t signal(int sig, sighandler_t handler);
```

Если определен макрос проверки возможностей _GNU_SOURCE, то библиотека glibc представляет нестандартный тип данных sighandler_t в заголовочном файле <signal.h>.

Вместо указания адреса функции в виде аргумента `handler` функции `signal()` можно указать одно из следующих значений.

- `SIG_DFL` — сбросить диспозицию сигнала до значения по умолчанию (см. табл. 20.1). Применяется для отмены эффекта предыдущего вызова функции `signal()`, изменившего диспозицию сигнала.
- `SIG_IGN` — игнорировать сигнал. При генерации сигнала для процесса ядро незаметно удаляет сгенерированный сигнал. Процесс никогда не узнает, что сигнал был сгенерирован.

Успешный вызов функции `signal()` возвращает предыдущее значение диспозиции сигнала, которое может представлять собой адрес ранее установленной функции обработчика либо одной из двух констант: `SIG_DFL` или `SIG_IGN`. При возникновении ошибки функция `signal()` возвращает значение `SIG_ERR`.

20.4. Введение в обработчики сигналов

Обработчик сигнала (также называемый *перехватчиком сигнала*) — это функция, вызываемая при получении указанного сигнала процессом. В этом разделе приводятся базовые сведения об обработчиках сигналов, а в главе 21 — уже более подробная информация.

Активация обработчика сигнала может в любое время прервать ход выполнения программы. Ядро осуществляет вызов обработчика от имени процесса, а по возвращении из обработчика выполнение программы возобновляется с той же точки, в которой оно было прервано. Эта последовательность проиллюстрирована на рис. 20.1.

Хотя обработчики сигнала могут выполнять практически любое действие, в целом они должны быть максимально простыми. Мы вернемся к этой теме в разделе 21.1.

В листинге 20.1 показан простой пример функции обработчика сигнала, а также основной программы, устанавливающей эту функцию в качестве обработчика сигнала `SIGINT`. (Драйвер терминала генерирует этот сигнал при вводе с клавиатуры символа

прерывания, обычно это сочетание клавиш Ctrl+C.) Обработчик просто выводит на печать сообщение — и возвращает управление в основную программу.

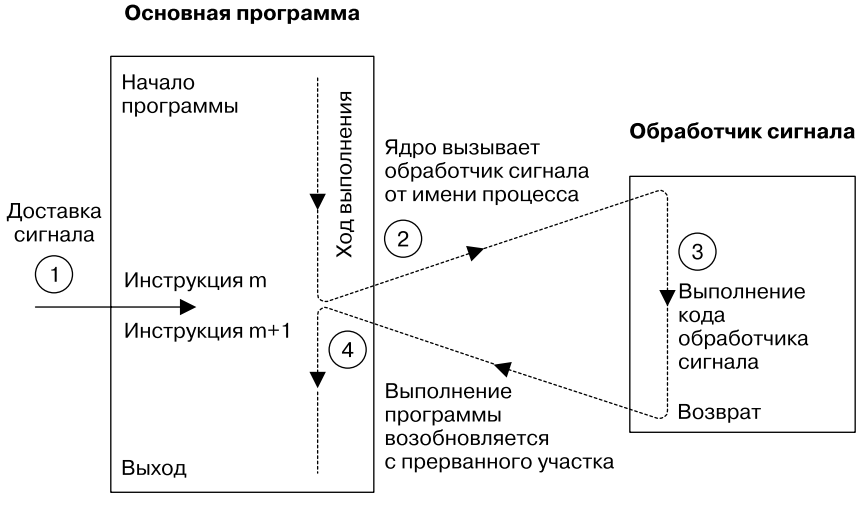


Рис. 20.1. Доставка сигнала и выполнение обработчика

Листинг 20.1. Установка обработчика для SIGINT

signals/ouch.c

```
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigHandler(int sig)
{
    printf("Ouch!\n");    /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */
}

int
main(int argc, char *argv[])
{
    int j;
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");
    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3);       /* Медленный цикл... */
    }
}
```

signals/ouch.c

Основная программа непрерывно выполняет цикл. При каждом проходе по циклу она увеличивает значение счетчика и распечатывает его значение, а затем «засыпает» на несколько секунд. (Для этого мы используем функцию `sleep()`, приостанавливающую выполнение вызывавшего ее кода на указанное количество секунд. Описание этой функции дано в подразделе 23.4.1.)

При запуске программы из листинга 20.1 на экране мы увидим примерно следующее:

```
$ ./ouch
0               Основная программа выполняет цикл, отображая увеличивающиеся числа
Нажмите Ctrl-C
Ouch!           Обработчик сигнала выполнен
1               Управление возвращено в основную программу
2
Снова нажмите Ctrl-C
Ouch!
3
Нажмите Ctrl-\ (символ выхода терминала)
Quit (core dumped)
```

После активации ядром обработчика сигнала, номер сигнала, ставшего причиной активации, передается в обработчик в виде целочисленного аргумента. (В листинге 20.1 — это аргумент `sig`.) Если перехватывается только один тип сигнала, то толку от этого аргумента мало. Однако мы можем установить один обработчик на перехват нескольких типов сигналов и использовать этот аргумент для определения того, какой именно сигнал стал причиной активации обработчика.

В листинге 20.2 приводится программа, устанавливающая один и тот же обработчик для сигналов `SIGINT` и `SIGQUIT`. (Сигнал `SIGQUIT` генерируется драйвером терминала, когда мы вводим символ *выхода*, обычно Ctrl+\.) Код обработчика различает два сигнала путем осмотра аргумента `sig` — и выполняет различные действия для каждого. В функции `main()` мы используем функцию `pause()` (см. раздел 20.14) для блокирования процесса до перехвата сигнала.

Следующий журнал сессии оболочки демонстрирует применение этой программы:

```
$ ./intquit
Нажмите Ctrl-C
Caught SIGINT (1)
Нажмите Ctrl-C снова
Caught SIGINT (2)
и еще раз
Caught SIGINT (3)
Нажмите Ctrl-\
Caught SIGQUIT — that's all folks!
```

В листингах 20.1 и 20.2 для вывода сообщений из обработчика сигнала мы используем функцию `printf()`. По причинам, которые мы обсуждаем в подразделе 21.1.2, в реальных приложениях никогда не применяются функции stdio из одного обработчика. Однако в различных примерах программ мы тем не менее будем вызывать функцию `printf()` из одного обработчика для обозначения факта его вызова.

Листинг 20.2. Установка одного и того же обработчика для двух разных сигналов

signals/intquit.c

```
#include <signal.h>
#include "tlpi_hdr.h"
static void
sigHandler(int sig)
{
    static int count = 0;

    /* НЕБЕЗОПАСНО: в этом обработчике используются функции, небезопасные
       для асинхронных сигналов (printf(), exit(); см. подраздел 21.1.2) */
    if (sig == SIGINT) {
        count++;
        printf("Caught SIGINT (%d)\n", count);
        return;           /* Возобновить выполнение в точке прерывания */
    }
```

```
    /* Должно быть SIGQUIT – распечатать сообщение и завершить процесс */
    printf("Caught SIGQUIT - that\'s all, folks!\n");
    exit(EXIT_SUCCESS);
}

int
main(int argc, char *argv[])
{
    /* Установить один обработчик для SIGINT и SIGQUIT */
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");
    if (signal(SIGQUIT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (;;)            /* Бесконечный цикл, ожидание сигналов */
        pause();        /* Блокировать до перехвата сигнала */
}
```

signals/intquit.c

20.5. Отправка сигналов: kill()

Один процесс может отправить сигнал другому процессу с помощью системного вызова `kill()`, который является аналогом команды оболочки `kill`. (Термин *kill* (с англ. — «убить») был выбран, потому что действием по умолчанию для большинства сигналов, доступных в ранних версиях UNIX, было завершение процесса.)

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Аргумент `pid` идентифицирует один или несколько процессов, в которые отправляется сигнал, задаваемый параметром `sig`. Четыре различных случая определяют, каким образом интерпретируется значение аргумента `pid`.

- Если `pid` больше `0`, то сигнал отправляется в процесс, идентификатор которого указан в аргументе `pid`.
- Если `pid` равен `0`, то сигнал отправляется во все процессы той же группы, что и вызывающий процесс, в том числе и в сам вызывающий процесс. (Стандарт SUSv3 устанавливает, что сигнал должен отправляться всем процессам группы за исключением «неуказанного набора системных процессов»; это же замечание касается всех остальных случаев.)
- Если `pid` меньше `-1`, то сигнал отправляется во все процессы группы, идентификатор которой равняется абсолютному значению аргумента `pid`. Такая отправка сигнала находит свое применение в управлении заданиями оболочки (см. раздел 34.7).
- Если `pid` равен `-1`, сигнал отправляется во все процессы, в которые у вызывавшего процесса есть разрешение отправлять сигналы, за исключением `init` (ID процесса равен 1) и вызывающего процесса. Если сигнал подается привилегированным процессом, он будет доставлен во все процессы в системе, кроме двух выше обозначенных. Сигналы, отправляемые таким образом, называются *сигналами оповещения*. (Стандарт SUSv3 не требует исключения вызывающего процесса из списка процессов, которые получат сигнал. В этом случае Linux следует семантике BSD.)

Если ни один из процессов не подходит под указанный в аргументе `pid` идентификатор, то функция `kill()` завершается с ошибкой и устанавливает для переменной `errno` значение `ESRCH` («Нет такого процесса»). Процессу для отправки сигнала другому требуются соответствующие разрешения. Далее перечислены правила доступа:

- Привилегированный (`CAP_KILL`) процесс может посылать сигналы в любой процесс.
- Процесс `init` (ID процесса 1), запущенный пользователем и группой `root`, — особый случай. В него можно посылать только те сигналы, для которых у него установлены обработчики. Это предотвращает возникновение ситуаций, когда системный администратор случайно аварийно завершает процесс `init`, фундаментальный для работы системы.
- Непривилегированный процесс может отправлять сигнал другому процессу, если реальное или эффективный ID пользователя процесса, отправляющего сигнал, совпадает с реальным или сохраненным установленным ID пользователя процесса, получающего сигнал, как показано на рис. 20.2. Это правило позволяет пользователям отправлять сигналы в запущенные ими программы с установленным ID пользователя независимо от текущего значения действующего ID пользователя целевого процесса. Более того, в Linux и других системах, предоставляющих системный вызов `setresuid()`, программа с установленным ID пользователя может извлечь преимущество из данного правила. Если установить с помощью `setresuid()` сохраненный ID пользователя в то же значение, что и реальный ID пользователя, то пользователь — владелец исполняемого файла не сможет оправлять этому процессу сигналы. (Стандарт SUSv3 устанавливает правила, изображенные на рис. 20.2, но в версиях ядра, предшествовавших 2.0, Linux следовала несколько иным правилам, как описано на странице справочника `kill(2)`.)
- Сигнал `SIGCONT` обрабатывается особым образом. Непривилегированный процесс может послать этот сигнал в любой процесс, запущенный в той же сессии, минуя проверку ID пользователей. Это позволяет оболочкам, управляющим заданиями, повторно запускать остановленные задания (группы процессов), даже если процессы задания изменили значения ID пользователя (иными словами, это привилегированные процессы, которые применили системные вызовы, описанные в разделе 9.7, для изменения учетных данных).

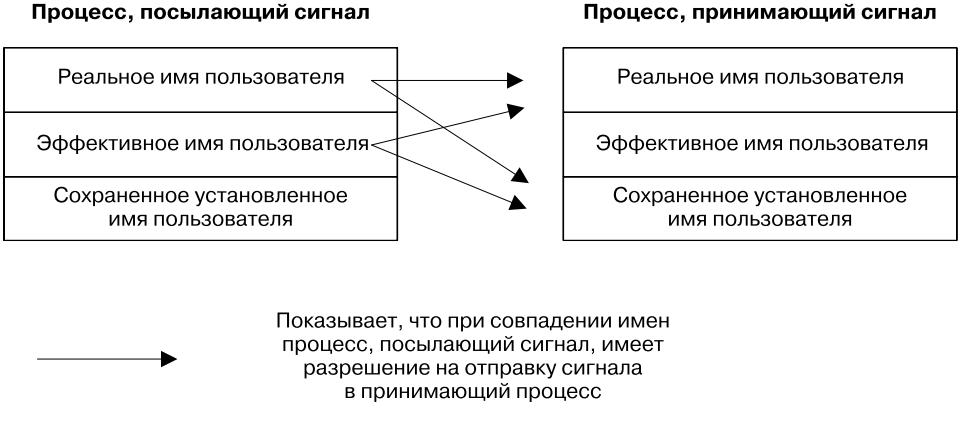


Рис. 20.2. Разрешения, необходимые непривилегированному процессу для отправки сигнала

Если у процесса нет разрешения на отправку сигнала в запрошенный идентификатор процесса `pid`, то вызов `kill()` завершается неудачно с установкой значения `EPERM` в `errno`. Если аргумент `pid` задает набор процессов (то есть значение `pid` — отрицательное число), вызов `kill()` завершается успешно, если сигнал может быть получен хотя бы одним из указанных процессов. Использование функции `kill()` продемонстрировано в листинге 20.3 далее.

20.6. Проверка существования процесса

Системный вызов `kill()` может также служить другой цели. Если параметр `sig` указан как `0` (так называемый *нулевой сигнал*), то никакой сигнал не отправляется. Вместо этого функция `kill()` лишь выполняет проверку ошибок, удостоверяясь в том, есть ли возможность отправки сигнала в процесс. То есть мы можем использовать нулевой сигнал для тестирования наличия процесса с указанным идентификатором. Если отправка нулевого сигнала завершается неудачно, с ошибкой `ESRCH`, то мы знаем, что процесс не существует. Если же вызов завершается с ошибкой `EPERM` (означающей, что процесс существует, но мы не обладаем разрешением на отправку в него сигналов) или завершается успешно, узнаем, что процесс существует.

Подтверждение существования конкретного идентификатора процесса не гарантирует, что конкретная программа все еще запущена. Так как ядро повторно использует идентификаторы процессов при рождении и завершении процессов, один и тот же идентификатор в разное время может соответствовать разным процессам. Более того, некий идентификатор процесса может существовать, но при этом быть «зомби» (иными словами, процесс уже завершился, однако его родительский процесс еще не осуществил вызов функции `wait()` для получения кода завершения этого дочернего процесса, как описано в разделе 26.2).

Для проверки того, запущен ли тот или иной процесс, могут применяться другие методы.

- *Системные вызовы wait()*. Эти вызовы описываются в главе 26. Они могут быть задействованы, только если наблюдаемый процесс является дочерним по отношению к вызывающему процессу.
- *Семафоры и исключающие файловые блокировки*. Если наблюдаемый процесс удерживает семафор или файловую блокировку, тот факт, что мы можем получить о них информацию, означает, что данный процесс завершен. Описание семафоров можно найти в главе 49, а файловых замков — в главе 51.
- *Каналы IPC, такие как простые поименованные каналы и каналы FIFO*. Мы настраиваем наблюдаемые процессы таким образом, чтобы они удерживали файловые дескрипторые открытыми на запись в канал на протяжении всей жизни процесса. Одновременно процесс, осуществляющий мониторинг, удерживает открытым файловый дескриптор для чтения из канала и знает, что наблюдаемый процесс завершился, если пишущий конец канала закрыт (так как был достигнут конец файла). Процесс, осуществляющий мониторинг, может определить это либо путем чтения собственного файлового дескриптора, либо путем мониторинга дескриптора с помощью одной из техник, описанных в главе 59.
- *Интерфейс /proc/PID*. Например, если существует идентификатор процесса 12345, значит, должен существовать и каталог `/proc/12345`, соответственно, мы можем проверить наличие этого каталога с помощью вызова `stat()`.

На все вышеприведенные тактики, за исключением последней, повторное использование операционной системой идентификаторов процессов не оказывает отрицательного влияния.

В листинге 20.3 демонстрируется применение функции `kill()`. Эта программа принимает два аргумента командной строки: идентификатор процесса и номер сигнала , а затем вызывает функцию `kill()` для отправки сигнала в заданный процесс. Если указан сигнал `0` (нулевой сигнал), то программа докладывает о существовании целевого процесса.

Листинг 20.3. Использование системного вызова kill()

signals/t_kill.c

```
#include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
```

```
    int s, sig;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid sig-num\n", argv[0]);

    sig = getInt(argv[2], 0, "sig-num");
    s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");
    } else {    /* Нулевой сигнал: проверка существования процесса */
        if (s == 0) {
            printf("Process exists and we can send it a signal\n");
        } else {
            if (errno == EPERM)
                printf("Process exists, but we don't have "
                       "permission to send it a signal\n");
            else if (errno == ESRCH)
                printf("Process does not exist\n");
            else
                errExit("kill");
        }
    }

    exit(EXIT_SUCCESS);
}
```

signals/t_kill.c

20.7. Другие способы отправки сигналов: raise() и killpg()

Иногда полезной практикой является отправка процессом сигнала самому себе. (Мы увидим пример этого в подразделе 34.7.3.) Эту задачу выполняет функция `raise()`.

```
#include <signal.h>

int raise(int sig);
```

Возвращает `0` при успешном завершении и ненулевое значение при ошибке

В программе с одним потоком вызов функции `raise()` аналогичен следующему вызову функции `kill()`:

```
kill(getpid(), sig);
```

В системах, поддерживающих потоки, вызов `raise(sig)` реализуется таким образом:

```
pthread_kill(pthread_self(), sig);
```

Функция `pthread_kill()` описывается в подразделе 33.2.3, а сейчас достаточно сказать: такая реализация означает, что сигнал будет доставлен именно в тот поток, из которого был выполнен вызов функции `raise()`. Но вызов `kill(getpid(), sig)` посылает сигнал

в вызывающий *процесс*, то есть сигнал может быть доставлен в любой поток в рамках процесса.

> Функция raise() появилась в языке C89. Стандарты языка C не затрагивают детали реализации операционной системы, такие как идентификаторы процессов, но функция raise() может быть описана в этом стандарте, так как ей не требуется указывать на идентификаторы процессов.

Когда процесс посылает сигнал самому себе с помощью функции `raise()` (или `kill()`), сигнал доставляется моментально (иными словами, перед тем, как функция `raise()` вернет управление в вызвавший ее код).

Обратите внимание, что функция `raise()` возвращает ненулевой результат (не обязательно `-1`) при возникновении ошибки. Единственная ошибка, которая может возникнуть при работе `raise()`, — это `EINVAL` при указании неверного значения `sig`. Таким образом, при указании одной из констант `SIGxxxx` нет необходимости проверять код возврата.

Функция `killpg()` посылает сигналы всем процессам в группе.

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Вызов функции `killpg()` эквивалентен следующему вызову функции `kill()`:

```
kill(-pgrp, sig);
```

Если аргумент `pgrp` указан как `0`, то сигнал посылается всем процессам той же группы процессов, что и вызывающий процесс. В стандарте SUSv3 это не прописано, однако такой случай интерпретируется в большинстве реализаций UNIX точно так же, как и в Linux.

20.8. Отображение описаний сигналов

Для каждого сигнала существует выводимое на печать описание. Все описания хранятся в массиве `sys_siglist`. Например, можно указать элемент массива `sys_siglist[SIGPIPE]` для получения описания сигнала `SIGPIPE` (нарушенный канал). Но вместо использования массива `sys_siglist` напрямую предпочтительнее вызывать функцию `strsignal()`.

```
#define _BSD_SOURCE
#include <signal.h>

extern const char *const sys_siglist[];

#define _GNU_SOURCE
#include <string.h>

char *strsingnal(int sig);
```

Возвращает указатель на строку с описанием сигнала

Функция `strsignal()` выполняет проверку попадания аргумента `sig` в границы массива, а затем возвращает указатель на пригодное для печати описание сигнала или на строку

с сообщением об ошибке, если был указан неверный номер сигнала. (В некоторых других реализациях UNIX функция `strsignal()` возвращает `NULL`, если аргумент `sig` неверный.)

Кроме проверки соблюдения границ, у функции `strsignal()` есть дополнительное преимущество над непосредственным применением массива `sys_siglist`. Оно состоит в том, что функция `strsignal()` чувствительна к локали (см. раздел 10.4), а это значит, что описания сигналов будут представлены на местном языке.

Пример использования функции `strsignal()` приведен в листинге 20.4 далее.

Функция `psignal()` выводит (в стандартное устройство вывода сообщений об ошибках) строку, соответствующую аргументу `msg`, добавляет двоеточие, а затем выводит описание сигнала `sig`. Как и функция `strsignal()`, функция `psignal()` чувствительна к локали.

```
#include <signal.h>

void psignal(int sig, const char *msg);
```

Несмотря на то что функции `psignal()`, `strsignal()` и массив `sys_siglist` не прописаны в SUSv3, они тем не менее доступны во многих реализациях UNIX. (В SUSv4 были добавлены спецификации функций `psignal()` и `strsignal()`.)

20.9. Наборы сигналов

Многие системные вызовы, связанные с сигналами, должны быть в состоянии представлять группу различных сигналов. Например, `sigaction()` и `sigprocmask()` позволяют программе указать группу сигналов, которые должны быть заблокированы процессом, тогда как `sigpending()` возвращает группу сигналов, находящихся в настоящее время в режиме ожидания процесса. (Перечисленные системные вызовы будут описаны позже.)

Несколько сигналов описываются структурой данных под названием *набор сигналов*, которая представлена системным типом данных `sigset_t`. SUSv3 устанавливает целый набор функций для управления наборами сигналов, сейчас мы перейдем к рассмотрению этих функций.

> В Linux, как и в большинстве других реализаций UNIX, тип данных sigset_t — это битовая маска. Однако стандарт SUSv3 этого не требует. Следовательно, набор сигналов может быть представлен структурой другого типа. Единственное, чего требует SUSv3, — чтобы тип данных был присваиваемым. Таким образом, он должен быть реализован через некий скалярный тип (например, целое число) или структуру C (возможно, содержащую массив целых чисел).

Функция `sigemptyset()` инициализирует набор сигналов, не содержащий членов. Функция `sigfillset()` инициализирует набор сигналов, содержащий все сигналы (в том числе все сигналы реального времени).

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t, *set);
```

Обе функции возвращают `0` при успешном завершении или `-1` при ошибке

Для инициализации набора сигналов *должна* быть использована одна из функций `sigemptyset()` и `sigfillset()`. Это необходимо потому, что язык C не инициализирует автоматические переменные, а инициализация статических переменных нулем, вероятно, не может считаться надежным способом указания пустого набора сигналов, так как наборы сигналов могут быть реализованы посредством структур, отличных от битовых масок. (По этой же причине неправильным шагом будет применение функции `memset(3)` для обнуления содержимого набора сигналов, чтобы пометить данный набор как пустой.)

После инициализации отдельные сигналы могут быть добавлены в набор с помощью функции `sigaddset()` и удалены — с помощью `sigdelset()`.

```
#include <signal.h>

int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

Обе функции возвращают `0` при успешном завершении или `-1` при ошибке

Для обеих функций `sigaddset()` и `sigdelset()` аргумент `sig` — это номер сигнала.

Функция `sigismember()` проверяет, является ли данный сигнал членом набора.

```
#include <signal.h>

int sigismember(sigset_t *set, int sig);
```

Возвращает `1`, если *`sig`* входит в набор *`set`*, `0` — если не входит, `-1` при ошибке

Функция `sigismember()` возвращает `1` (истина), если `sig` является членом `set`, `0` (ложь), если не является, и `-1` при ошибке (например, `sig` не является допустимым номером сигнала).

В GNU-библиотеке C реализованы три нестандартные функции, выполняющие задачи, которые дополняют стандартные функции наборов сигналов, описанные выше.

```
#define _GNU_SOURCE
#include <signal.h>

int sigandset(sigset_t *set, sigset_t *left, sigset_t *right);
int sigorset(sigset_t *set, sigset_t *left, sigset_t *right);
```

Обе функции возвращают `0` при успешном завершении или `-1` при ошибке

```
int sigisemptyset(const sigset_t *set);
```

Возвращает `1` если *`set`* пуст, иначе — `0`

Вышеприведенные функции выполняют следующие задачи:

- `sigandset()` — помещает пересечение наборов `left` и `right` в набор `dest`;
- `sigorset()` — помещает объединение наборов `left` и `right` в набор `dest`;
- `sigisemptyset()` — возвращает `1`, если `set` не содержит сигналов, `0` в противном случае.

Пример программы

Используя функции, рассмотренные в этом разделе, мы можем написать функции, приведенные в листинге 20.4, которые будем применять далее в различных программах. Пер-

вая, `printSigset()`, отображает сигналы, являющиеся членами указанного набора. В этой функции задействуется константа `NSIG`, определенная в файле `<signal.h>`, со значением, на единицу большим, чем самый большой номер сигнала. Мы используем константу `NSIG` в качестве верхней границы цикла при проверке всех номеров сигналов на членство в наборе.

> Несмотря на то, что константа NSIG не указана в стандарте SUSv3, она определена в большинстве реализаций UNIX. Тем не менее, возможно, потребуется установить некоторые параметры компилятора для используемой реализации, чтобы сделать эту константу видимой. Например, в Linux мы должны определить один из макросов тестирования возможности _BSD_SOURCE, _SVID_SOURCE или _GNU_SOURCE.

Функции `printSigMask()` и `printPendingSigs()` применяют функцию `printSigset()` для вывода сигнальной маски процесса и набора сигналов, ожидающих доставки процессу, соответственно. Функции `printSigMask()` и `printPendingSigs()` соответственно применяют системные вызовы `sigprocmask()` и `sigpending()`. Вызовы `sigprocmask()` и `sigpending()` описаны в разделах 20.10 и 20.11.

Листинг 20.4. Функции для отображения наборов сигналов

signals/signal_functions.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "signal_functions.h"    /* Объявляет определяемые здесь функции */
#include "tlpi_hdr.h"

/* ПРИМЕЧАНИЕ: Все следующие функции используют функцию fprintf(), не являющуюся
   безопасной для асинхронных сигналов (см. подраздел 21.1.2). Следовательно, эти
   функции также не являются безопасными для асинхронных сигналов (остерегайтесь
   беспорядочно вызывать их из обработчиков сигналов). */

void                /* Выводит список сигналов в наборе */
printSigset(FILE *of, const char *prefix, const sigset_t *sigset)
{
    int sig, cnt;
    cnt = 0;

    for (sig = 1; sig < NSIG; sig++) {
        if (sigismember(sigset, sig)) {
            cnt++;
            fprintf(of, "%s%d (%s)\n", prefix, sig, strsignal(sig));
        }
    }
    if (cnt == 0)
        fprintf(of, "%s<empty signal set>\n", prefix);
}

int    /* Напечатать маску заблокированных сигналов процесса */
printSigMask(FILE *of, const char *msg)
{
    sigset_t currMask;
    if (msg != NULL)
        fprintf(of, "%s", msg);
    if (sigprocmask(SIG_BLOCK, NULL, &currMask) == -1)
        return -1;
    printSigset(of, "\t\t", &currMask);
    return 0;
}
```

```
int         /* Распечатать сигналы, ожидающие доставки процессу */
printPendingSigs(FILE *of, const char *msg)
{
    sigset_t pendingSigs;
    if (msg != NULL)
        fprintf(of, "%s", msg);
    if (sigpending(&pendingSigs) == -1)
        return -1;
    printSigset(of, "\t\t", &pendingSigs);
    return 0;
}
```

signals/signal_functions.c

20.10. Сигнальная маска (блокирование доставки сигналов)

Для каждого процесса ядро хранит *сигнальную маску* — набор сигналов, доставка которых в процесс временно заблокирована. Если в процесс отправляется заблокированный сигнал, то доставка этого сигнала откладывается до тех пор, пока сигнал не будет разблокирован путем удаления из сигнальной маски процесса. (В подразделе 33.2.1 мы увидим, что на самом деле сигнальная маска — это атрибут потока и что каждый поток многопоточного процесса может независимо просматривать и изменять сигнальную маску с помощью функции `pthread_sigmask()`.)

Сигнал может быть добавлен в сигнальную маску одним из следующих способов.

- При активации обработчика сигнал, послуживший этому причиной, может быть автоматически добавлен в маску. Происходит это или нет, зависит от флагов, использованных при установке обработчика с помощью `sigaction()`.
- Когда обработчик установлен с помощью вызова `sigaction()`, можно указать дополнительный набор сигналов, которые подлежат блокировке при активации обработчика.
- Системный вызов `sigprocmask()` может использоваться в любое время для явного добавления и удаления сигналов из сигнальной маски.

Мы отложим обсуждение первых двух случаев до того момента, пока не изучим функцию `sigaction()` в разделе 20.13, а вызов `sigprocmask()` рассмотрим прямо сейчас.

```
#include <signal.h>

int sigprogmask(int how, const sigset_t *set, sigset_t *oldset);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Мы можем использовать функцию `sigprocmask()` для изменения сигнальной маски процесса, для получения существующей маски либо для совершения обоих действий. Аргумент `how` определяет изменения, вносимые функцией `sigprocmask()` в сигнальную маску.

- `SIG_BLOCK` — сигналы, включенные в набор сигналов, на который указывает аргумент `set`, добавляются в сигнальную маску. Иными словами, сигнальная маска — это объединение текущего значения маски и значения аргумента `set`.
- `SIG_UNBLOCK` — сигналы, указанные в наборе сигналов, на который указывает аргумент `set`, исключаются из сигнальной маски. Разблокирование незаблокированных сигналов не приводит к возврату кода ошибки.

- SIG_SETMASK — набор сигналов, на который указывает параметр set, присваивается сигнальной маске.

В каждом случае если значение аргумента oldset не равно NULL, то указывает на буфер sigset_t, используемый для возврата предыдущего значения сигнальной маски.

Если мы хотим получить сигнальную маску без внесения изменений, можно установить значение NULL для аргумента set, в этом случае аргумент how будет проигнорирован.

Для временного предотвращения доставки сигнала можно использовать последовательность вызовов, приведенных в листинге 20.5, для блокировки и последующей разблокировки сигнала путем сброса сигнальной маски к ее предыдущему состоянию.

Листинг 20.5. Временное блокирование доставки сигнала

```
sigset_t blockSet, prevMask;

/* Инициализировать набор сигналов сигналом SIGINT */
sigemptyset(&blockSet);
sigaddset(&blockSet, SIGINT);

/* Блокировать SIGINT, сохранить предыдущую сигнальную маску */
if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)
    errExit("sigprocmask1");

/* ... Код, который нельзя прервать сигналом SIGINT ... */

/* Восстановить предыдущую сигнальную маску, разблокировать SIGINT */
if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask2");
```

Стандарт SUSv3 устанавливает, что если один из ожидающих сигналов разблокирован вызовом функции sigprocmask(), то хотя бы один из этих сигналов будет доставлен до того, как вызов будет возвращен. Иными словами, если мы разблокируем ожидающий сигнал, то он будет незамедлительно доставлен в процесс.

Попытки заблокировать сигналы SIGKILL и SIGSTOP игнорируются без оповещения. Если мы попытаемся заблокировать эти сигналы, функция sigprocmask() не только не даст доступа к ним, но и не сгенерирует ошибку. Это значит, что мы можем использовать следующий код для блокировки всех сигналов, за исключением сигналов SIGKILL и SIGSTOP:

```
sigfillset(&blockSet);
if (sigprocmask(SIG_BLOCK, &blockSet, NULL) == -1)
    errExit("sigprocmask");
```

20.11. Ожидающие сигналы

Если процесс получает сигнал, который в данный момент подлежит блокированию, то он добавляется в набор ожидающих сигналов. Когда (и если) сигнал будет разблокирован в дальнейшем, он будет доставлен в процесс. Для определения того, какие сигналы процесса находятся в режиме ожидания, мы можем вызвать функцию sigpending().

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Системный вызов `sigpending()` возвращает набор сигналов, находящихся в режиме ожидания процесса в структуре `sigset_t`, на которую указывает аргумент `set`. После этого мы сможем проверить содержимое структуры `set` с помощью функции `sigismember()`, описанной в разделе 20.9.

Если диспозиция ожидающего сигнала изменяется, то при последующем разблокировании данный сигнал будет обработан в соответствии со вновь заданной диспозицией. Это может быть полезно для предотвращения доставки ожидающего сигнала путем изменения его диспозиции на `SIG_IGN` или `SIG_DFL` (если действие по умолчанию для данного сигнала — *игнорировать*). В результате сигнал удаляется из набора ожидающих сигналов и, таким образом, в процесс не доставляется.

20.12. Сигналы не ставятся в очередь

Набор ожидающих сигналов — это лишь маска, она показывает факт возникновения того или иного сигнала, но не количество возникновений. Иными словами, если один и тот же сигнал был сгенерирован несколько раз, будучи заблокированным, то он записывается в набор ожидающих сигналов, а затем доставляется, но лишь однажды. (Одно из различий между стандартными сигналами и сигналами реального времени заключается в том, что вторые ставятся в очередь, как описано в разделе 22.8.)

В листингах 20.6 и 20.7 показаны две программы, которые применяются для наблюдения того, как сигналы могут быть не поставлены в очередь. Программа в листинге 20.6 принимает четыре аргумента командной строки, следующим образом:

```
$ ./sig_sender PID num-sigs sig-num [sig-num-2]
```

Первый аргумент — это идентификатор процесса, в который программа должна отправлять сигналы. Второй определяет количество сигналов, подлежащих отправке в целевой процесс. Третий аргумент устанавливает номер сигнала, подлежащего отправке в целевой процесс. Если в качестве четвертого аргумента предоставляется номер сигнала, то программа отправляет один экземпляр этого сигнала после отправки сигналов, указанных предыдущими аргументами. В примере сессии оболочки ниже мы используем последний аргумент для отправки сигнала `SIGINT` в целевой процесс. Назначение отправки данного сигнала вы поймете в ближайшее время.

Листинг 20.6. Отправка нескольких сигналов

signals/sig_sender.c

```
include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numSigs, sig, j;
    pid_t pid;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid num-sigs sig-num [sig-num-2]\n", argv[0]);

    pid = getLong(argv[1], 0, "PID");
    numSigs = getInt(argv[2], GN_GT_0, "num-sigs");
    sig = getInt(argv[3], 0, "sig-num");
    /* Отправить сигналы получателю */
```

```
    printf("%s: sending signal %d to process %ld %d times\n",
            argv[0], sig, (long) pid, numSigs);

    for (j = 0; j < numSigs; j++)
        if (kill(pid, sig) == -1)
            errExit("kill");

    /* Если был указан четвертый аргумент командной строки, отправить этот сигнал */
    if (argc > 4)
        if (kill(pid, getInt(argv[4], 0, "sig-num-2")) == -1)
            errExit("kill");

    printf("%s: exiting\n", argv[0]);
    exit(EXIT_SUCCESS);
}
```

signals/sig_sender.c

Программа из листинга 20.7 предназначена для перехвата и передачи информации о сигналах, отправленных программой из листинга 20.6. Она выполняет такие шаги.

- Устанавливает обработчик для перехвата всех сигналов ❷. (Сигналы `SIGKILL` и `SIGSTOP` перехватить невозможно, но мы игнорируем ошибку, происходящую при попытке установить обработчик для этих сигналов.) Чаще всего обработчик ❶ просто подсчитывает сигналы с помощью массива. Если получен сигнал `SIGINT`, то обработчик устанавливает флаг (`gotSigint`), что заставляет программу выйти из основного цикла (цикл `while`, описываемый ниже). (Объяснение применения классификатора `volatile` и типа данных `sig_atomic_t`, использованного для объявления переменной `gotSigint` см. в подразделе 21.1.3.)
- Если в программу был передан аргумент командной строки, то программа блокирует все сигналы на протяжении количества секунд, установленных этим аргументом, а затем, перед разблокированием сигналов, отображает набор ожидающих сигналов ❸. Это позволяет нам отправлять сигналы в процесс перед тем, как он перейдет к следующему шагу.
- Программа выполняет цикл `while`, потребляющий процессорное время до тех пор, пока не установлен флаг `gotSigint` ❹. (В разделах 20.14 и 22.9 описывается применение функций `pause()` и `sigsuspend()`, являющихся более эффективными в отношении ресурсов процессора методами ожидания доставки сигнала.)
- После выхода из цикла `while` программа отображает количество всех полученных сигналов ❺.

Мы сначала используем эти две программы для иллюстрации того, что заблокированный сигнал доставляется только однажды, вне зависимости от того, сколько раз он был сгенерирован. Это делается путем установки времени сна получателя и отправки всех сигналов до окончания установленного временного интервала.

```
$ ./sig_receiver 15 &                Получатель блокирует сигналы на 15 секунд
[1] 5368
./sig_receiver: PID is 5386
./sig_receiver: sleeping for 15 seconds
$ ./sig_sender 5368 1000000 10 2     Отправка сигналов SIGUSR1 и сигнала SIGINT
./sig_sender: sending signal 10 to process 5368 1000000 times
./sig_sender: exiting
./sig_receiver: pending signals are:
                2 (Interrupt)
                10 (User defined signal 1)
./sig_receiver: signal 10 caught 1 time
[1]+ Done                ./sig_receiver 15
```

В аргументах командной строки для отправляющей программы были указаны сигналы SIGUSR1 и SIGINT, являющиеся в Linux/ x86 сигналами 10 и 2 соответственно.

Из программного вывода, приведенного выше, мы видим, что из миллиона отправленных нами сигналов получателю доставлен был только один.

Даже если процесс не блокирует сигналы, он может получить сигналов меньше, чем было отправлено. Это может произойти, если сигналы отправляются настолько быстро, что они прибывают до того, как выполнение получающего сигналы процесса может быть запланировано ядром. В результате, множественные сигналы записываются в набор ожидающих сигналов процесса лишь однажды. Если мы запустим программу из листинга 20.7 без аргументов командной строки (так, чтобы программа не блокировала сигналы и не переходила в режим сна), на экране мы увидим следующее:

```
$ ./sig_receiver &
[1] 5393
./sig_receiver: PID is 5393
$ ./sig_sender 5393 1000000 10 2
./sig_sender: sending signal 10 to process 5393 1000000 times
./sig_sender: exiting
./sig_receiver: signal 10 caught 52 times
[1]+ Done                    ./sig_receiver
```

Из миллиона отправленных сигналов только 52 были перехвачены процессом-получателем. (Точное количество перехваченных сигналов будет варьироваться в зависимости от причуд алгоритма планирования ядра.) Причина этого заключается в том, что каждый раз, когда программа-отправитель запускается по плану, она отправляет получателю несколько сигналов. Однако только один из этих сигналов помечается как ожидающий и доставляется лишь тогда, когда у получателя есть возможность запуститься.

Листинг 20.7. Перехват и подсчет сигналов

signals/sig_receiver.c

```
#define _GNU_SOURCE
#include <signal.h>
#include "signal_functions.h"     /* Объявление printSigset() */
#include "tlpi_hdr.h"

static int sigCnt[NSIG];          /* Считает количество доставок каждого сигнала */
static volatile sig_atomic_t gotSigint = 0;
        /* Устанавливается ненулевое значение в случае доставки SIGINT */

static void
(1) handler(int sig)
{
    if (sig == SIGINT)
        gotSigint = 1;
    else
        sigCnt[sig]++;
}

int
main(int argc, char *argv[])
{
    int n, numSecs;
    sigset_t pendingMask, blockingMask, emptyMask;
    printf("%s: PID is %ld\n", argv[0], (long) getpid());
```

```
❷ for (n = 1; n < NSIG; n++)    /* Один обработчик для всех сигналов */
     (void) signal(n, handler);   /* Игнорируем ошибки */

    /* Если указано время сна, временно заблокировать все сигналы, спать (пока другой
       процесс посылает нам сигналы), а затем отобразить маску ожидающих сигналов
       и разблокировать все сигналы */

❸ if (argc > 1) {
        numSecs = getInt(argv[1], GN_GT_0, NULL);

        sigfillset(&blockingMask);
        if (sigprocmask(SIG_SETMASK, &blockingMask, NULL) == -1)
            errExit("sigprocmask");
        printf("%s: sleeping for %d seconds\n", argv[0], numSecs);
        sleep(numSecs);

        if (sigpending(&pendingMask) == -1)
            errExit("sigpending");
        printf("%s: pending signals are: \n", argv[0]);
        printSigset(stdout, "\t\t", &pendingMask);
        sigemptyset(&emptyMask);        /* Разблокировать все сигналы */
        if (sigprocmask(SIG_SETMASK, &emptyMask, NULL) == -1)
            errExit("sigprocmask");
    }

❹ while (!gotSigint)           /* Выполнять цикл до перехвата SIGINT */
        continue;

❺ for (n = 1; n < NSIG; n++)   /* Распечатать количество перехваченных сигналов */
        if (sigCnt[n] != 0)
            printf("%s: signal %d caught %d time%s\n", argv[0], n,
                    sigCnt[n], (sigCnt[n] == 1) ? "" : "s");
    exit(EXIT_SUCCESS);
}
```

signals/sig_receiver.c

20.13. Изменение диспозиций сигналов: sigaction()

Системный вызов `sigaction()` является альтернативой функции `signal()` в части установки диспозиции сигнала. Несмотря на то что функция `sigaction()` в той или иной степени сложнее в использовании, чем функция `signal()`, взамен она предоставляет большую гибкость. В частности, функция `sigaction()` позволяет получать текущую диспозицию сигнала без ее изменения, а также устанавливать различные атрибуты для точного управления тем, что происходит при активации обработчика сигнала. Кроме того, как мы выясним после тщательного анализа в разделе 22.7, функция `sigaction()` обладает куда большими свойствами переносимости при установке обработчика сигнала, по сравнению с функцией `signal()`.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент sig означает сигнал, диспозицию которого мы хотим получить или изменить. Этим аргументом может быть любой сигнал, за исключением SIGKILL или SIGSTOP.

Аргумент act — это указатель на структуру, устанавливающую новую диспозицию сигнала. Если нам необходимо только лишь выяснить текущую диспозицию, то мы можем указать для этого аргумента значение NULL. Аргумент oldact — это указатель на структуру такого же типа, он используется для возврата информации о предыдущей диспозиции сигнала. Если нам это неинтересно, то мы можем задать для этого аргумента значение NULL. Структура, на которую указывают аргументы act и oldact, имеет следующий тип:

```
struct sigaction {
    void (*sa_handler)(int);   /* Адрес обработчика */
    sigset_t sa_mask;          /* Сигналы, блокируемые во время вызова обработчика */
    int sa_flags;              /* Флаги, контролирующие активацию обработчика */
    void (*sa_restorer)(void); /* Не для использования в приложениях */
};
```

На самом деле структура sigaction несколько более сложна, чем показано выше. Мы обсудим это более детально в разделе 21.4.

Поле sa_handler соотносится с аргументом handler, передаваемым функции signal(). В данном поле указывается адрес обработчика сигнала или одна из констант — SIG_IGN или SIG_DFL. Поля sa_mask и sa_flags, которые мы обсудим в ближайшее время, интерпретируются только в том случае, если поле sa_handler содержит адрес обработчика сигнала, иными словами — значение, отличное от констант SIG_IGN и SIG_DFL. Оставшееся поле, sa_restorer, не предназначено для использования в приложениях и в стандарте SUSv3 не устанавливается.

Поле sa_restorer задействуется внутренне для гарантии того, что по завершении работы обработчика сигнала осуществляется системный вызов специального назначения sigreturn(), восстанавливающий контекст выполнения процесса в той точке, на которой оно было прервано. Пример такого использования данной функции может быть найден в файле исходного кода glibc: sysdeps/unix/sysv/linux/i386/sigaction.c.

Поле sa_mask определяет список сигналов, блокируемых во время активации обработчика, определенного аргументом sa_handler. После активации обработчика все сигналы из этого набора, не являющиеся в данный момент частью сигнальной маски процесса, автоматически добавляются в маску до вызова обработчика. Они остаются в сигнальной маске процесса до тех пор, пока не происходит возврат из обработчика сигнала, после чего они автоматически удаляются. Поле sa_mask позволяет указать набор сигналов, которым не разрешено прерывать выполнение данного обработчика. Кроме того, сигнал, активировавший обработчик, также автоматически добавляется в маску процесса. Это означает, что обработчик сигнала рекурсивно не прервет сам себя в случае доставки второго экземпляра того же сигнала во время выполнения кода обработчика. Так как заблокированные сигналы не ставятся в очередь, если любой из них генерируется повторно во время выполнения обработчика, то этот сигнал будет (позже) доставлен в процесс только однажды.

Поле sa_flags — битовая маска, устанавливающая разные параметры, контролирующие обработку сигнала. Следующие биты могут быть объединены в этом поле битовой операцией ИЛИ (|).

- SA_NOCLDSTOP — если аргумент sig равен SIGCHILD, не генерировать сигнал, если дочерний процесс остановлен или возобновлен в результате получения сигнала.

- SA_NOCLDWAIT — (начиная с Linux 2.6) если аргумент sig равен SIGCHILD, не превращать дочерние процессы в «зомби» при завершении. Для получения более подробной информации см. подраздел 26.3.2.
- SA_NODEFER — при перехвате этого сигнала не добавлять его автоматически в сигнальную маску процесса на время выполнения обработчика. Имя SA_NOMASK является историческим синонимом для SA_NODEFER, однако последнее считается предпочтительным, так как оно стандартизировано в SUSv3.
- SA_ONSTACK — активировать обработчик для этого сигнала с использованием альтернативного стека, установленного функцией sigaltstack(). См. раздел 21.3.
- SA_RESETHAND — при перехвате сигнала сбросить его диспозицию до значения по умолчанию (то есть SIG_DFL) перед активацией обработчика. (По умолчанию обработчик остается установленным до тех пор, пока не будет явно отключен следующим вызовом функции sigaction().) Имя SA_ONESHOT является историческим синонимом для SA_RESETHAND, однако последнее считается предпочтительным, так как оно стандартизировано в SUSv3.
- SA_RESTART — автоматически перезапустить системный вызов, прерванный обработчиком сигнала. См. раздел 21.5.
- SA_SIGINFO — активировать обработчик сигнала с дополнительными аргументами, предоставляющими дополнительную информацию о сигнале (см. раздел 21.4).

Все вышеперечисленные параметры определены в стандарте SUSv3. Пример использования функции sigaction() приведен в листинге 21.1.

20.14. Ожидание сигнала: pause()

Вызов функции pause() приостанавливает выполнение процесса до тех пор, пока вызов не будет прерван обработчиком (или до тех пор, пока необрабатываемый сигнал не завершит процесс).

```
#include <unistd.h>

int pause(void);
```

Всегда возвращает -1 с установкой errno в EINTR

При получении обрабатываемого сигнала функция pause() всегда прерывается и возвращает -1 с установкой errno в EINTR. (Более подробно об ошибке EINTR мы поговорим в разделе 21.5.) Пример использования функции pause() приведен в листинге 20.2.

В разделах 22.9–22.11 мы рассматриваем различные способы приостановки работы программы на время ожидания сигнала.

20.15. Резюме

Сигнал — это оповещение о том, что произошло некое событие. Сигнал может быть отпроцессу из ядра, из другого процесса или из самого себя. Существует набор стандартных сигналов, каждый из которых имеет уникальный номер и предназначение.

Доставка сигнала, как правило, асинхронна, а это означает, что невозможно предсказать, в какой точке сигнал прерывает выполнение процесса. В некоторых случаях (например, в случае с аппаратно-генерируемыми сигналами) сигналы доставляются синхронно, а это значит, что доставка случается предсказуемо и повторимо в определенной точке

выполнения программы. По умолчанию сигнал может быть либо проигнорирован, либо приводит к завершению процесса (с дампом ядра или без), останавливает запущенный процесс или повторно запускает остановленный. Какое из действий будет выполнено, зависит от типа сигнала. Кроме того, в программе могут использоваться функции `signal()` или `sigaction()` для явного игнорирования сигнала или установки функции обработчика сигнала, определяемой программистом и активируемой при получении сигнала. Для обеспечения лучшей переносимости установку обработчика рекомендуется осуществлять с помощью функции `sigaction()`.

Процесс (с необходимыми разрешениями) может отправить сигнал в другой процесс с помощью функции `kill()`. Отправка нулевого сигнала (`0`) — это один из способов определения существования процесса с заданным ID. У каждого процесса есть сигнальная маска, представляющая собой набор сигналов, доставка которых временно заблокирована. Они могут быть добавлены и удалены из сигнальной маски с помощью функции `sigprocmask()`.

При получении заблокированного сигнала он остается в режиме ожидания до разблокирования. Стандартные сигналы не могут быть поставлены в очередь. Иными словами, сигнал может быть отмечен в качестве ожидающего (и, следовательно, доставлен позже) только однажды. Для получения набора ожидающих сигналов (структуры данных, используемой для представления нескольких различных сигналов и идентифицирующей сигналы, находящиеся в режиме ожидания), процесс может задействовать системный вызов `sigpending()`.

При установке диспозиции сигнала системный вызов `sigaction()` предоставляет больше контроля и гибкости по сравнению с функцией `signal()`. В первую очередь мы можем указать набор дополнительных сигналов, блокируемых при активации обработчика. Вдобавок к этому мы можем использовать различные флаги для управления действиями, выполняемыми при активации обработчика сигнала. Например, существуют флаги, задействующие старую ненадежную семантику сигнала (сигнал, активировавший обработчик, не блокируется, а диспозиция сигнала сбрасывается к значению по умолчанию до вызова обработчика).

С помощью функции `pause()` процесс может прервать выполнение до получения сигнала.

Дополнительная информация

[Bovet & Cesati, 2005] и [Maxwell, 1999] раскрывают историю реализации сигналов в Linux. [Goodheart & Cox, 1994] приводят детальную информацию о реализации сигналов в ОС System V Release 4. Справочник GNU библиотеки C (доступен онлайн по адресу www.gnu.org) содержит обширное описание сигналов.

20.16. Упражнения

20.1. Как отмечается в разделе 20.3, функция `sigaction()` обладает большей переносимостью, чем `signal()`, если речь идет об установке обработчика сигнала. Замените `signal()` на `sigaction()` в программе из листинга 20.7 (`sig_receiver.c`).

20.2. Напишите программу, демонстрирующую следующее поведение: если установить диспозицию сигнала `SIG_IGN`, то программа никогда не перехватит (и вообще не заметит) этот сигнал.

20.3. Напишите программу, демонстрирующую действие флагов `SA_RESETHAND` и `SA_NODEFER` при установке обработчика сигнала с помощью функции `sigaction()`.

20.4. Реализуйте использование функции `siginterrupt()` с помощью функции `sigaction()`.

21 Сигналы: обработчики сигналов

В этой главе мы продолжим рассматривать сигналы, но сделаем основной акцент на обработчиках сигналов и расширим обсуждение, начатое в разделе 20.4. Среди рассматриваемых в этой главе тем можно выделить следующие.

1. Как спроектировать обработчик сигнала. Это вовлекает нас в дискуссию о реентерабельности и функциях, безопасных для асинхронных сигналов.
2. Альтернативы выполнению обычного возврата из обработчика сигнала, в частности применение для этих целей нелокального перехода `goto`.
3. Обработка сигналов в альтернативном стеке.
4. Использование флага `SA_SIGINFO` функции `sigaction()` для передачи обработчику сигнала возможности получить более детализированную информацию о сигнале, вызвавшем его активацию.
5. Как блокирующий системный вызов может быть прерван обработчиком сигнала и каким образом при необходимости данный системный вызов может быть перезапущен.

21.1. Проектирование обработчиков сигналов

Предпочтительнее писать простые обработчики сигналов. Одна из важных причин для этого — уменьшение вероятности создания состояний гонки. Чаще всего встречаются следующие две схемы обработчиков сигналов.

- Обработчик сигнала устанавливает глобальный флаг и завершается. Основная программа периодически проверяет флаг — и, если он установлен, выполняет определенное действие. (Если основная программа не может проводить такую проверку, так как ей необходимо осуществлять мониторинг одного или нескольких файловых дескрипторов на предмет возможности ввода-вывода, то обработчик сигнала также может записать один байт в выделенный канал, считываемый конец которого включен в список наблюдаемых программой файловых дескрипторов. Пример использования этого метода приводится в подразделе 59.5.2.)
- Обработчик сигнала производит некоего рода чистку, а затем либо прекращает процесс, либо выполняет нелокальный переход (см. подраздел 21.2.1) для раскрутки стека и возврата управления в предустановленный участок кода основной программы.

Дальше мы рассмотрим эти схемы, а также другие концепции, важные для проектирования обработчиков сигналов.

21.1.1. Сигналы не ставятся в очередь (еще раз о...)

В разделе 20.10 отмечалось, что доставка сигнала блокируется на время выполнения его обработчика (если не установлен флаг `SA_NODEFER` для функции `sigaction()`). Если сигнал (повторно) генерируется во время выполнения обработчика, то он помечается как ожидающий и доставляется по возвращении из обработчика. Еще отмечалось, что сигналы не ставятся в очередь. Если за время выполнения обработчика сигнал

сгенерирован более одного раза, то он также помечается как ожидающий и будет позже доставлен только один раз.

Тот факт, что сигналы могут «исчезать», определенным образом влияет на проектирование обработчиков сигналов. Начать следует с того, что возможности точно подсчитать количество раз, когда сигнал был сгенерирован, нет. Более того, нам, возможно, потребуется создать код обработчика таким образом, чтобы у него была возможность отвечать на совершение сразу нескольких событий типа, соответствующего сигналу. Пример этого мы увидим при рассмотрении сигнала `SIGCHLD` в подразделе 26.3.1.

21.1.2. Реентерабельные функции и функции, безопасные для асинхронных сигналов

Не все системные вызовы и библиотечные функции могут быть безопасно вызваны из обработчика сигнала. Чтобы понять почему, необходимо объяснить две концепции: реентерабельные функции и функции, безопасные для асинхронных сигналов.

Реентерабельные и нереентерабельные функции

Чтобы разобраться, что такое реентерабельные функции, сначала рассмотрим различие между однопоточными и многопоточными программами. Классические программы UNIX имеют только *один поток выполнения*: ЦПУ обрабатывает инструкции одного логического потока выполнения программы. В многопоточной программе присутствуют несколько независимых и параллельных (concurrent) логических потоков в рамках одного процесса.

В главе 29 мы рассмотрим, каким образом можно явно создавать программы, содержащие несколько потоков выполнения. Однако понятие многопоточного выполнения также имеет отношение к программам, в которых используются обработчики сигналов. Поскольку обработчик сигнала может асинхронно прервать выполнение программы в любой момент времени, основная программа и обработчик сигнала, по сути, формируют два независимых (хотя и не параллельных) потока в рамках одного процесса.

Функция называется *реентерабельной* в том случае, если она может одновременно безопасно выполняться несколькими потоками в рамках одного процесса. В данном контексте слово «безопасно» означает, что функция достигает ожидаемого результата вне зависимости от текущего состояния любого другого потока выполнения.

> В стандарте SUSv3 реентерабельные функции определены как функции, «успешное выполнение которых гарантируется при вызове двумя или более потоками так, как если бы эти потоки вызывали данную функцию поочередно в неопределенном порядке, даже если в действительности выполнение функций накладывается».

Функция может быть нереентерабельной, если она обновляет глобальные или статические структуры данных. (Функция, которая использует только локальные переменные, гарантированно является реентерабельной.) Если две инициации (то есть два потока) функции одновременно пытаются обновить одну и ту же глобальную переменную или структуру данных, то очень вероятно, что эти обновления вступят в конфликт и приведут к выдаче неверных результатов. Для примера представим ситуацию, когда один поток выполнения находится в процессе обновления связанного списка с целью добавить в него новый элемент, но одновременно другой поток пытается обновить тот же список. Так как добавление нового элемента списка требует обновления нескольких указателей, если другой поток прерывает эти шаги и обновляет те же указатели, в результате получается хаос.

Такие случаи на самом деле далеко не редкость при работе со стандартной библиотекой C. Например, в подразделе 7.1.3 мы уже отмечали, что функции `malloc()` и `free()` работают со связанными списками освобожденной памяти, доступной для перераспределения из динамической области. Если вызов функции `malloc()`, осуществленный основной программой, прерывается обработчиком сигнала, который также вызывает функцию `malloc()`, то такой связанный список может быть поврежден. По этой причине семейство функций `malloc()` и другие библиотечные функции, которые применяют их, являются нереентерабельными.

Другие библиотечные функции являются нереентерабельными, потому что они возвращают информацию, используя статически выделенную память. Примерами таких функций (описываются по тексту книги) могут быть `crypt()`, `getpwnam()`, `gethostbyname()` и `getservbyname()`. Если обработчик сигнала вызывает одну из этих функций, то он перезапишет информацию, возвращенную любым предыдущим вызовом той же функции из основной программы (или наоборот).

Функции могут быть нереентерабельными, если они используют для внутренних операций статические структуры данных. Самыми очевидными примерами могут быть члены библиотеки `stdio`: `printf()`, `scanf()` и т. д, обновляющие внутренние структуры данных для буферизированного ввода-вывода. Таким образом, задействуя в обработчике сигнала функцию `printf()`, мы иногда можем увидеть странный вывод или даже аварийное завершение программы, или повреждение данных, если обработчик события прерывает программу во время выполнения функции `printf()` или другой функции из библиотеки `stdio`.

Даже если мы не используем нереентерабельные библиотечные функции, мы все равно можем столкнуться с проблемами реентерабельности. Если обработчик событий обновляет глобальные структуры данных, определенные программистом, обновляемые также из основной программы, то мы можем сказать, что обработчик событий является нереентерабельным по отношению к основной программе.

Если функция является нереентерабельной, то страница справочника будет, как правило, содержать явное или неявное указание на это. В частности, обращайте внимание на утверждения, что функция задействует или возвращает данные в статически выделенных переменных.

Пример программы

В листинге 21.1 продемонстрирована нереентерабельная природа функции `crypt()` (см. раздел 8.5). В качестве аргументов командной строки данная программа принимает две строки текста. Программа выполняет следующие шаги.

1. Вызов функции `crypt()` для зашифровки первой строки текста — аргумента и копирование этой строки текста в отдельный буфер с помощью функции `strdup()`.
2. Установка обработчика сигнала `SIGINT` (генерируется при нажатии Ctrl+C). Обработчик вызывает функцию `crypt()` для зашифровки строки текста, предоставленной в качестве второго аргумента.
3. Вход в бесконечный цикл `for`, в котором используется функция `crypt()` для зашифровки строки текста в первом аргументе командной строки и осуществление проверки того, что возвращенная строка совпадает со строкой, сохраненной при выполнении шага 1.

Строки в шаге 3 всегда будут совпадать при отсутствии сигнала. Другое дело, если поступает сигнал `SIGINT` и выполнение обработчика прерывает выполнение основной программы сразу же после вызова функции `crypt()` в цикле `for` и перед осуществлением проверки совпадения строк. В этом случае программа сообщит о несовпадении. При запуске программы мы увидим следующее:

```
$ ./non_reentrant abc def
Несколько раз нажимать Ctrl-C для генерации SIGINT
Mismatch on call 109871 (mismatch=1 handled=1)
Mismatch on call 128061 (mismatch=2 handled=2)
Многие строки вывода были удалены
Mismatch on call 727935 (mismatch=149 handled=156)
Mismatch on call 729547 (mismatch=150 handled=157)
Нажать Ctrl-\ для генерации SIGQUIT
Quit (core dumped)
```

Если мы сравним значения `mismatch` и `handled` в вышеприведенном выводе, то увидим, что в большинстве случаев при инициализации сигнала происходит перезапись статически выделенного буфера между вызовом функции `crypt()` и сравнением строк в `main()`.

Листинг 21.1. Вызов нереентерабельной функции из main() и обработчика сигнала

signals/nonreentrant.c

```
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include "tlpi_hdr.h"

static char *str2;              /* Устанавливается из argv[2] */
static int handled = 0;         /* Счетчик вызовов обработчика */

static void
handler(int sig)
{
    crypt(str2, "xx");
    handled++;
}

int
main(int argc, char *argv[])
{
    char *cr1;
    int callNum, mismatch;
    struct sigaction sa;

    if (argc != 3)
        usageErr("%s str1 str2\n", argv[0]);
    str2 = argv[2];            /* Сделать argv[2] доступным обработчику */
    cr1 = strdup(crypt(argv[1], "xx")); /* Скопировать статически
                                           выделенную строку в иной буфер */

    if (cr1 == NULL)
        errExit("strdup");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    /* Многократно вызывать crypt() с использованием argv[1]. Если прервано
       обработчиком сигнала, то статическое хранилище, возвращенное crypt(),
```

```
        будет перезаписано результатами зашифровки argv[2], а strcmp()
        обнаружит несовпадение со значением 'cr1'. */
    for (callNum = 1, mismatch = 0; ; callNum++) {
        if (strcmp(crypt(argv[1], "xx"), cr1) != 0) {
            mismatch++;
            printf("Mismatch on call %d (mismatch=%d handled=%d)\n",
                    callNum, mismatch, handled);
        }
    }
}
```

signals/nonreentrant.c

Стандартные функции, безопасные для асинхронных сигналов

Функция, безопасная для асинхронных сигналов, — это такая функция, реализация которой гарантирует безопасность при вызове из обработчика. Функция может быть безопасной для асинхронного сигнала либо благодаря тому, что она реентерабельна, либо потому, что она непрерываема обработчиком.

Безопасность перечисленных в табл. 21.1 функций для асинхронных сигналов требуется различными стандартами. Если после имени в таблице не указано *v2* или *v3*, то требование безопасности функции для асинхронных сигналов установлено стандартом POSIX.1-1990. В стандарт SUSv2 были добавлены функции, помеченные в таблице *v2*, а функции с отметкой *v3* были включены в стандарт SUSv3. В отдельных реализациях UNIX могут встречаться и другие функции, безопасные для асинхронных сигналов, однако все реализации UNIX, соответствующие стандарту, должны гарантировать, что как минимум функции, приведенные в таблице, являются безопасными для асинхронных сигналов (если присутствуют в реализации: не все из перечисленных функций реализованы в Linux).

Таблица 21.1. Функции, которые должны быть безопасными для асинхронных сигналов согласно стандартам POSIX.1-1990, SUSv2 и SUSv3

_Exit() (v3)	getpid()	sigdelset()
_exit()	getppid()	sigemptyset()
abort() (v3)	getsockname() (v3)	sigfillset()
accept() (v3)	getsockopt() (v3)	sigismember()
access()	getuid()	signal() (v2)
aio_error() (v2)	kill()	sigpause() (v2)
aio_return() (v2)	link()	sigpending()
aio_suspend() (v2)	listen() (v3)	sigprocmask()
alarm()	lseek()	sigqueue() (v2)
bind() (v3)	lstat() (v3)	sigset() (v2)
cfgetispeed()	mkdir()	sigsuspend()
cfgetospeed()	mkfifo()	sleep()
cfsetispeed()	open()	socket() (v3)
cfsetospeed()	pathconf()	sockatmark() (v3)
chdir()	pause()	socketpair() (v3)
chmod()	pipe()	stat()
chown()	poll() (v3)	symlink() (v3)
clock_gettime() (v2)	posix_trace_event() (v3)	sysconf()
close()	pselect() (v3)	tcdrain()
connect() (v3)	raise() (v2)	tcflow()

Продолжение ⇗

Таблица 21.1 *(продолжение)*

creat()	read()	tcflush()
dup()	readlink() (v3)	tcgetattr()
dup2()	recv() (v3)	tcgetpgrp()
execle()	recvfrom() (v3)	tcsendbreak()
execve()	recvmsg() (v3)	tcsetattr()
fchmod() (v3)	rename()	tcsetpgrp()
fchown() (v3)	rmdir()	time()
fcntl()	select() (v3)	timer_getoverrun() (v2)
fdatasync() (v2)	sem_post() (v2)	timer_gettime() (v2)
fork()	send() (v3)	timer_settime() (v2)
fpathconf() (v2)	sendmsg() (v3)	times()
fstat()	sendto() (v3)	umask()
fsync() (v2)	setgid()	uname()
ftruncate() (v3)	setpgid()	unlink()
getegid()	setsid()	utime()
geteuid()	setsockopt() (v3)	wait()
getgid()	setuid()	waitpid()
getgroups()	shutdown() (v3)	write()
getpeername() (v3)	sigaction()	
getpgrp()	sigaddset()	

Стандарт SUSv4 вносит следующие изменения в табл. 21.1.

- Удалены функции `fpathconf()`, `pathconf()` и `sysconf()`.
- Добавлены функции `execl()`, `execv()`, `faccessat()`, `fchmodat()`, `fchownat()`, `fexecve()`, `fstatat()`, `futimens()`, `linkat()`, `mkdirat()`, `mkfifoat()`, `mknod()`, `mknodat()`, `openat()`, `readlinkat()`, `renameat()`, `syslinkat()`, `unlinkat()`, `utimensat()` и `utimes()`.

В стандарте SUSv3 отмечается, что все функции, не перечисленные в табл. 21.1, считаются небезопасными по отношению к сигналам. Однако говорится также, что функция является небезопасной, только когда инициация обработчика сигнала прерывает выполнение небезопасной функции и если сам обработчик также вызывает небезопасную функцию. Иными словами, при написании обработчиков сигналов у нас есть две альтернативы.

- Убедиться, что сам код обработчика является реентерабельным и что из него вызываются только функции, безопасные для асинхронных сигналов.
- Блокировать доставку сигналов во время выполнения кода основной программы, вызывающей небезопасную функцию или работающую с глобальной структурой данных, также обновляемой обработчиком сигнала.

Проблема второго подхода в том, что в сложной программе бывает непросто гарантировать то, что обработчик сигнала никогда не прервет основную программу во время вызова небезопасной функции. По этой причине вышеприведенные правила обычно упрощаются до того, что не нужно вызывать небезопасные функции из обработчиков сигналов.

> Если мы установим одну и ту же функцию обработчика для работы с несколькими различными сигналами или воспользуемся флагом SA_NODEFER функции sigaction(), то обработчик может прервать сам себя. Как следствие этого, обработчик может быть нереентерабельным, если он обновляет глобальные (или статичные) переменные, даже если эти переменные не задействуются основной программой.

Использование errno в обработчиках сигналов

Поскольку функции, перечисленные в табл. 21.1, могут обновлять переменную `errno`, их применение может все-таки превратить обработчик сигнала в нереентерабельный, так как данные функции могут перезаписать значение переменной `errno`, установленное основной программой. Обойти проблему можно путем сохранения текущего значения переменной `errno` на входе в обработчик сигнала, задействующий одну из приведенных в табл. 21.1 функций, и восстановления значения на выходе из обработчика, как показано в примере:

```
void
handler(int sig)
{
    int savedErrno;
    /* Теперь мы можем выполнить функцию, потенциально изменяющую errno */
    errno = savedErrno;
}
```

Использование небезопасных функций в примерах программ из книги

Несмотря на то что функция `printf()` не является безопасной для асинхронных сигналов, мы вызываем ее в обработчиках в различных примерах программ из этой книги. Это делается потому, что функция `printf()` предоставляет простой и понятный способ продемонстрировать, что был вызван обработчик события, а также отобразить содержимое нужных переменных внутри обработчика. По аналогичным причинам мы также используем в обработчиках событий и другие небезопасные функции, в том числе другие функции библиотеки `stdio` и `strsignal()`.

В приложениях, применяемых на практике, следует избегать вызова небезопасных функций из тела обработчиков сигналов. Для того чтобы избежать недопониманий, в примерах программ, содержащих обработчики сигналов, при задействовании небезопасных функций мы сопровождаем каждый вызов такой функции примечанием о небезопасности:

```
printf("Some message\n");     /* НЕБЕЗОПАСНО */
```

21.1.3. Глобальные переменные и тип данных sig_atomic_t

Несмотря на проблемы реентерабельности, совместное использование глобальных переменных основной программой и обработчиком сигнала может оказаться полезным. Это безопасно до тех пор, пока основная программа корректно воспринимает и обрабатывает вероятность, что обработчик может в любое время изменить значение глобальной переменной. Например, распространенной практикой является применение обработчика сигнала для выполнения единственного действия: установки глобального флага. Флаг периодически проверяется основной программой, выполняющей соответствующее действие в ответ на получение сигнала (и снимающей флаг). Если обработчик сигнала получает доступ к глобальным переменным таким образом, то мы всегда должны объявлять эти переменные с использованием ключевого слова `volatile` (см. раздел 6.8), дабы запретить компилятору выполнять оптимизацию, результатом которой может стать хранение переменной в регистре.

Для осуществления чтения и записи глобальной переменной может потребоваться несколько машинных кодов, а обработчик сигнала может прервать основную программу во время выполнения последовательности таких машинных кодов. (Мы говорим, что доступ к переменным является *неатомарным*.) По этой причине стандарты языка C, а также

стандарт SUSv3 устанавливают целочисленный тип данных `sig_atomic_t`, чтение и запись экземпляров которого гарантированно являются атомарным. Таким образом, глобальную переменную флага, совместно используемую основной программой и обработчиком сигнала, необходимо объявлять следующим образом:

```
volatile sig_atomic_t flag;
```

Мы приводим пример использования типа данных `sig_atomic_t` в листинге 22.5.

Обратите внимание, что операторы C инкремента (`++`) и декремента (`--`) не попадают в гарантию, предоставляемую типом данных `sig_atomic_t`. В некоторых аппаратных архитектурах эти операции могут не быть атомарным (см. раздел 30.1 для получения более подробной информации). В гарантию безопасности, предоставляемую типом данных `sig_atomic_t`, попадает только то, что мы можем установить переменную этого типа в обработчике сигнала и проверить ее значение в основной программе (или наоборот).

Стандарты C99 и SUSv3 устанавливают, что в каждой реализации операционной системы должны определяться две константы (в `<stdint.h>`): `SIG_ATOMIC_MIN` и `SIG_ATOMIC_MAX`, устанавливающие диапазон значений, которые могут быть присвоены переменным типа `sig_atomic_t`. Согласно требованиям стандарта значения должны варьироваться в диапазоне как минимум от –127 до 127, если тип `sig_atomic_t` представлен как величина со знаком или как минимум от 0 до 255, если величина без знака. В Linux эти константы равняются отрицательному и положительному пределу для 32-битных целых чисел со знаком.

21.2. Другие методы завершения работы обработчика сигнала

Все рассмотренные до этого момента обработчики сигналов завершаются возвратом в основную программу. Однако простой возврат из обработчика не всегда является желательным, а в некоторых случаях может быть и бесполезным. (Такой случай мы рассмотрим при изучении аппаратно генерируемых сигналов в разделе 22.4.)

Существуют другие различные методы завершения работы обработчика сигнала.

- Вызов функции `_exit()` для завершения процесса. Перед этим обработчик может выполнить несколько действий для очистки. Обратите внимание, что мы не можем использовать `exit()` для завершения обработчика, так как эта функция не является одной из безопасных, приведенных в табл. 21.1. Эта функция небезопасна, так как сбрасывает буферы stdio перед осуществлением вызова функции `_exit()`, как описано в разделе 25.1.
- Вызов функции `kill()` или `raise()` для отправки сигнала, аварийно завершающего процесс (то есть сигнала, действие по умолчанию для которого — завершение процесса).
- Выполнение нелокального перехода из обработчика.
- Вызов функции `abort()`.

Последние два метода более подробно рассматриваются в следующих подразделах.

21.2.1. Выполнение нелокального перехода из обработчика сигнала

В разделе 6.8 описаны функции `setjmp()` и `longjmp()`, позволяющие выполнять нелокальный переход из функции в код одной из процедур, вызвавших эту функцию. Мы можем это сделать и из обработчика. Это позволит восстановить работу программы после получения сигнала, сгенерированного аппаратным исключением (например, при

ошибке доступа к памяти). Кроме того, этот метод позволяет перехватить сигнал и вернуть управление в конкретный участок программы. Например, оболочка по получении сигнала SIGINT (генерируемого, как правило, нажатием Ctrl+C) выполняет нелокальный переход для возвращения управления в основной цикл ввода (и, таким образом, чтения новой команды).

Однако при использовании стандартной функции longjmp() для выхода из обработчика сигнала возникает одна проблема. Ранее уже было отмечено, что после входа в обработчик ядро автоматически заносит активирующий его сигнал, так же как и любой другой указанный в поле act.sa_mask, в сигнальную маску процесса, а затем удаляет эти сигналы из маски, когда обработчик выполняет нормальный возврат.

Но что случается с сигнальной маской, когда мы выходим из обработчика с помощью функции longjmp()? Ответ зависит от происхождения конкретной реализации UNIX. В System V функция longjmp() не восстанавливает сигнальную маску, таким образом, блокированные сигналы так и остаются заблокированными до выхода из обработчика. Linux унаследовала модель поведения System V. (И, как правило, нам это мешает, ведь такое поведение оставляет сигнал, активировавший обработчик, заблокированным). В реализациях, восходящих к BSD, функция setjmp() сохраняет сигнальную маску в аргументе env, а функция longjmp() восстанавливает сохраненную маску. Иными словами, мы, возможно, не сможем использовать функцию longjmp() для выхода из обработчика сигнала.

В случае если при компиляции программы мы определяем макрос тестирования возможности _BSD_SOURCE, то функция setjmp() (glibc) следует семантике BSD.

Из-за вышеописанной разницы между двумя основными вариантами UNIX, комитет по подготовке стандарта POSIX.1-1990 решил не включать в него способ обработки сигнальной маски функциями setjump() и longjump(). Вместо этого комитет определил две новые функции — sigsetjump() и siglongjump(), предоставляющие явный контроль над сигнальной маской при выполнении нелокального перехода.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
                                    Возвращает 0 при первом вызове, ненулевое значение
                                                при возврате через siglongjmp()
void siglongjmp(sigjmp_buf env, int val);
```

Функции sigsetjump() и siglongjump() работают аналогично функциям setjump() и longjump(). Единственное отличие заключается в типе аргумента env (sigjmp_buf вместо jmp_buf), а также в дополнительном аргументе savings функции sigsetjump(). Если значение savings не нуль, значит, сигнальная маска процесса, актуальная на момент вызова функции sigsetjump(), сохранена в переменной env и восстановлена последующим вызовом siglongjump() с указанным именем аргумента env. Если же значение аргумента savings равно 0, значит, сигнальная маска процесса не была ни сохранена, ни восстановлена.

Функции longjump() и siglongjump() не включены в перечень безопасных для асинхронных сигналов (см. табл. 21.1), так как вызов небезопасной для асинхронных сигналов функции после выполнения нелокального перехода несет в себе такие же риски, как и вызов этой функции из обработчика сигнала. Более того, если обработчик прерывает основную программу во время обновления структуры данных, после чего происходит

выход из обработчика с помощью нелокального перехода, это может оставить структуру в недозаполненном состоянии. Применение функции `sigprocmask()`, временно блокирующей сигнал во время выполнения важных обновлений, — один из способов избежать проблем в такой ситуации.

Пример программы

В листинге 21.2 демонстрируется различие в методах обработки сигнальной маски для нелокальных переходов двух типов. Эта программа устанавливает обработчик для сигнала `SIGINT`. Она создана таким образом, чтобы разрешать использование сочетания функций `setjmp()` плюс `longjmp()` или `sigsetjmp()` плюс `siglongjmp()` для выхода из обработчика сигнала в зависимости от того, был ли определен макрос компиляции `USE_SIGSETJMP`. Программа отображает текущие установки сигнальной маски как на момент входа в обработчик сигнала, так и после выполнения нелокального перехода, переносящего управление из обработчика сигнала обратно в основную программу.

При создании программы с тем расчетом, что для выхода из обработчика сигнала применяется функция `longjump()`, на экране мы увидим примерно следующее:

```
$ make -s sigmask_longjmp      Компиляция по умолчанию с использованием setjmp()
$ ./sigmask_longjmp
Signal mask at startup:
            <empty signal set>
Calling setjmp()
Нажмите Ctrl-C для генерации SIGINT
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                2 (Interrupt)
(На данном этапе сочетание Ctrl-C не сработает, так как SIGINT заблокирован)
Нажмите Ctrl-\ для завершения программы
Quit
```

Из вывода программы мы можем видеть, что после вызова функции `longjump()` из обработчика сигнала значение сигнальной маски остается таким же, каким оно было при входе в обработчик.

> В вышеприведенной сессии оболочки мы строим программу с помощью сборочного файла, поставляемого вместе с дистрибутивом исходного кода этой книги (http://www.man7.org/tlpi). Параметр -s сообщает программе make не распечатывать выполняемые команды. Мы используем этот параметр, чтобы не засорять журнал сессии. (В [Mecklenburg, 2005] содержится описание программы make проекта GNU.)

При компиляции того же исходного кода для построения исполняемого файла, использующего функцию `siglongjump()`, для выхода из обработчика сигнала мы увидим на экране следующее:

```
$ make -s sigmask_siglongjmp     Компиляция с использованием cc –DUSE_SIGSETJMP
$ ./sigmask_siglongjmp
Signal mask at startup:
            <empty signal set>
Calling sigsetjmp()
Нажмите Ctrl-C
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
               <empty signal set>
```

На данный момент сигнал SIGINT не блокируется, так как функция siglongjump() восстанавливает значение сигнальной маски до состояния на момент вызова функции sigsetjump() (иными словами, к пустому набору сигналов).

В листинге 21.2 также демонстрируется полезная техника с использованием обработчика сигнала, в котором выполняется нелокальный переход. Поскольку сигнал может быть сгенерирован в любое время, значит, он может быть сгенерирован прежде, чем цель перехода будет установлена функцией sigsetjump() (или setjump()). Для пресечения этой вероятности (из-за которой обработчик сигнала выполнит нелокальный переход с не инициализированным буфером env) мы применим защитную переменную canJump для обозначения того, был ли инициализирован буфер env. Если значение переменной canJump равно false, то вместо выполнения нелокального перехода обработчик просто осуществляет возврат. Альтернативный подход заключается в организации программного кода таким образом, что вызов функции sigsetjump() (или setjump()) производится перед установкой обработчика сигнала. Однако в сложных программах трудно гарантировать, что вышеуказанные шаги будут выполняться именно в нужном порядке, и применение защитной переменной может быть более простым выходом.

Обратите внимание, что использование директивы препроцессора #ifndef было самым элементарным способом написания программы в листинге 21.2 в соответствии со стандартом. В частности, мы не смогли бы заменить директиву #ifndef следующей проверкой времени выполнения:

```
if (useSiglongjmp)
    s = sigsetjmp(senv, 1);
else
    s = setjmp(env);
if (s == 0)
    …
```

Такая проверка недопустима, поскольку стандарт SUSv3 не разрешает использование функций setjmp() и sigsetjmp() в выражениях присваивания (см. раздел 6.8).

Листинг 21.2. Выполнение нелокального перехода из обработчика сигнала

signals/sigmask_longjmp.c

```
#define _GNU_SOURCE     /* Получить объявление strsignal() из <string.h> */
#include <string.h>
#include <setjmp.h>
#include <signal.h>
#include "signal_functions.h"    /* Объявление printSigMask() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t canJump = 0;
/* Присваивается значение 1, если буфер "env" был инициализирован [sig]setjmp() */
#ifdef USE_SIGSETJMP
static sigjmp_buf senv;
#else
static jmp_buf env;
#endif

static void
handler(int sig)
{
    /* НЕБЕЗОПАСНО: В этом обработчике используются функции, небезопасные
       для асинхронных сигналов (printf(), strsignal(), printSigMask();
       см. подраздел 21.1.2) */
```

```
    printf("Received signal %d (%s), signal mask is:\n", sig,
            strsignal(sig));
    printSigMask(stdout, NULL);

    if (!canJump) {
        printf("'env' buffer not yet set, doing a simple return\n");
        return;
    }

#ifdef USE_SIGSETJMP
    siglongjmp(senv, 1);
#else
    longjmp(env, 1);
#endif
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;

    printSigMask(stdout, "Signal mask at startup:\n");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

#ifdef USE_SIGSETJMP
    printf("Calling sigsetjmp()\n");
    if (sigsetjmp(senv, 1) == 0)
#else
    printf("Calling setjmp()\n");
    if (setjmp(env) == 0)
#endif
        canJump = 1;      /* Выполняется после [sig]setjmp() */
    else                  /* Выполняется после [sig]longjmp() */
        printSigMask(stdout, "After jump from handler, signal mask is:\n" );
    for (;;)              /* Ожидание сигналов до завершения */
        pause();
}
```

signals/sigmask_longjmp.c

21.2.2. Аварийное завершение процесса: abort()

Функция `abort()` завершает процесс и заставляет его создать файл дампа ядра.

```
#include <stdlib.h>

void abort(void);
```

Функция `abort()` завершает вызывающий процесс путем подачи сигнала `SIGABRT`. Действие по умолчанию для сигнала `SIGABRT` — создание файла дампа ядра и завершение процесса. Файл дампа впоследствии может быть использован отладчиком для изучения состояния программы на момент вызова функции `abort()`.

Стандарт SUSv3 требует, чтобы функция `abort()` переопределяла эффект блокирования сигнала `SIGABRT`. Более того, стандарт SUSv3 устанавливает, что `abort()` должна завершать процесс в том случае, если он не перехватывает сигнал, обработчик которого не выполняет возврат. Последнее предложение требует осмысления. Среди методов завершения процессов, описываемых в разделе 21.2, только метод, предполагающий использование нелокального перехода, может рассматриваться как имеющий отношение к этому предложению. Если применяется этот метод, то эффект функции `abort()` будет обнулен; во всех остальных случаях функция `abort()` всегда завершает процесс.

В большинстве реализаций завершение процесса гарантируется следующим образом: если процесс не завершен после однократной подачи сигнала `SIGABRT` (то есть обработчик перехватывает сигнал и выполняет возврат таким образом, что выполнение функции `abort()` возобновляется), то функция `abort()` восстанавливает обработку сигнала `SIGABRT` до `SIG_DFL` и повторно подает сигнал `SIGABRT`, который гарантированно завершит процесс.

Если функция `abort()` успешно завершает процесс, то она также сбрасывает и закрывает потоки `stdio`.

Пример использования функции `abort()` приводится при рассмотрении функций обработки ошибок в листинге 3.3.

21.3. Обработка сигнала на альтернативном стеке: signalstack()

Как правило, при активации обработчика сигнала ядро выделяет для него участок на стеке процесса. Однако это может быть невозможно, если процесс попытается расширить стек за пределы максимально возможного размера. Например, такое может произойти, если стек становится настолько большим, что сталкивается с участком отображенной памяти или увеличивающейся областью динамически распределяемой памяти (кучей) либо достигает ресурсного ограничения `RLIMIT_STACK` (см. раздел 36.3).

Когда процесс пытается увеличить стек за пределы установленного максимально возможного размера, ядро направляет в процесс сигнал `SIGSEGV`. Однако по причине того, что пространство стека израсходовано, ядро не сможет выделить участок для обработчика сигнала `SIGSEGV`, возможно установленного в программе. Следовательно, инициализации обработчика не происходит и процесс завершается (действие по умолчанию для сигнала `SIGSEGV`).

Если же нам нужно, чтобы в вышеописанной ситуации происходила обработка сигнала `SIGSEGV`, мы можем поступить следующим образом.

1. Выделить участок памяти, называемый *альтернативным сигнальным стеком*, который может использоваться в качестве кадра стека для обработчика сигнала.
2. Применить системный вызов `signalstack()` для информирования ядра о наличии альтернативного сигнального стека.
3. При установке обработчика сигнала указать флаг `SA_ONSTACK`, чтобы информировать ядро о том, что кадр для данного обработчика должен быть создан на альтернативном стеке.

Системный вызов `signalstack()` не только создает альтернативный сигнальный стек, но также возвращает информацию о любом уже созданном альтернативном сигнальном стеке.

```
#include <signal.h>

int signalstack(const stack_t *sigstack, stack_t *old_sigstack);
```

Возвращает 0 при успешном завершении или –1 при ошибке

Аргумент sigstack указывает на структуру, в которой приводятся местоположение и атрибуты нового альтернативного сигнального стека. Аргумент old_sigstack указывает на структуру, используемую для возврата информации о ранее созданном альтернативном сигнальном стеке (если таковой был создан). Любой из этих аргументов может быть указан как NULL. Например, мы можем получить информацию о существующем альтернативном сигнальном стеке, не изменяя его, просто указав аргумент sigstack как NULL. В противном случае каждый аргумент будет указывать на структуру следующего типа:

```
typedef struct {
    void     *ss_sp;     /* Начальный адрес альтернативного стека*/
    int       ss_flags;  /* Флаги: SS_ONSTACK, SS_DISABLE */
    size_t    ss_size;   /* Размер альтернативного стека */
} stack_t;
```

В полях ss_sp и ss_size указываются местоположение и размер альтернативного сигнального стека. Когда мы действительно им воспользуемся, ядро возьмет на себя заботу о выравнивании указанного значения поля ss_sp с границей адресов, совместимой с применяемой аппаратной архитектурой.

Как правило, место для альтернативного сигнального стека выделяется либо статически, либо динамически на куче. Стандартом SUSv3 устанавливается использование константы SIGSTKSZ для указания типичного значения размера альтернативного стека и константы MINSIGSTKSZ для указания минимального размера, требующегося для активации обработчика сигнала. В Linux/x86-32 эти константы определены значениями 8192 и 2048 соответственно.

Ядро не изменяет размер альтернативного сигнального стека. Если размер стека превосходит количество выделенного для него места, то в системе случается хаос (например, происходит перезапись значений переменных, находящихся вне пределов стека). Однако с такой проблемой можно столкнуться крайне редко, так как альтернативный сигнальный стек, как правило, используется для обработки частных случаев переполнения стандартного стека. На стеке, как правило, выделяется только один или несколько кадров. Работа сигнала SIGSEGV заключается либо в проведении чистки и завершении процесса, либо в раскрутке стандартного стека с помощью нелокального перехода.

Поле ss_flags может содержать одни из следующих значений.

- SS_ONSTACK — если при получении информации о созданном в настоящий момент альтернативном стеке (old_sigstack) установлен этот флаг, это означает, что процесс сейчас выполняется на альтернативном стеке. Попытки создать новый альтернативный стек при выполнении процесса уже на альтернативном стеке приведут к ошибке (EPERM) в функции signalstack().
- SS_DISABLE — возвращаемый в аргументе old_sigstack, данный флаг означает, что сейчас отсутствуют созданные альтернативные стеки. При указании в аргументе sigstack флаг приводит к отключению уже созданного альтернативного стека.

В листинге 21.3 демонстрируется создание и использование альтернативного стека. После создания альтернативного сигнального стека и установки обработчика сигнала SIGSEGV данная программа вызывает функцию, выполняющую бесконечный рекурсив-

ный вызов самой себя таким образом, что происходит переполнение стека — и процессу направляется сигнал SIGSEGV. При запуске программы на экране мы увидим следующее:

```
$ ulimit -s unlimited
$ ./t_sigaltstack
Top of standard stack is near 0xbffff6b8
Alternate stack is at 0x804a948-0x804cfff
Call    1 - top of stack near 0xbff0b3ac
Call    2 - top of stack near 0xbfe1714c
Мы опустили много строк вывода для экономии места
Call 2144 - top of stack near 0x4034120c
Call 2145 - top of stack near 0x4024cfac
Caught signal 11 (Segmentation fault)
Top of handler stack near     0x804c860
```

В этой сессии оболочки мы использовали команду ulimit для удаления любого ограничения ресурса RLIMIT_STACK, которое, возможно, могло быть установлено в оболочке. Мы объясним понятие ресурсного ограничения в разделе 36.3.

Листинг 21.3. Использование функции signalstack()

signals/t_sigaltstack.c

```
#define _GNU_SOURCE     /* Получить объявление strsignal() из <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigsegvHandler(int sig)
{
    int x;

    /* НЕБЕЗОПАСНО: В этом обработчике используются функции, небезопасные для
       асинхронных сигналов (printf(), strsignal(), fflush(); см. подраздел 21.1.2) */

    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
    printf("Top of handler stack near     %10p\n", (void *) &x);
    fflush(NULL);

    _exit(EXIT_FAILURE);      /* После SIGSEGV  возврат невозможен */
}

static void                   /* Рекурсивная функция, переполняющая стек */
overflowStack(int callNum)
{
    char a[100000];           /* Увеличиваем размер этого кадра стека */

    printf("Call %4d - top of stack near %10p\n", callNum, &a[0]);
    overflowStack(callNum+1);
}

int
main(int argc, char *argv[])
{
    stack_t sigstack;
    struct sigaction sa;
    int j;
```

```
    printf("Top of standard stack is near %10p\n", (void *) &j);

    /* Выделить альтернативный стек и оповестить ядро о его существовании */

    sigstack.ss_sp = malloc(SIGSTKSZ);
    if (sigstack.ss_sp == NULL)
        errExit("malloc");
    sigstack.ss_size = SIGSTKSZ;
    sigstack.ss_flags = 0;
    if (sigaltstack(&sigstack, NULL) == -1)
        errExit("sigaltstack");
    printf("Alternate stack is at         %10p-%p\n",
            sigstack.ss_sp, (char *) sbrk(0) - 1);

    sa.sa_handler = sigsegvHandler;   /* Установить обработчик SIGSEGV */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_ONSTACK;         /* Обработчик использует альтернативный стек */
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        errExit("sigaction");

    overflowStack(1);
}
```

signals/t_sigaltstack.c

21.4. Флаг SA_SIGINFO

Флаг `SA_SIGINFO` при установке обработчика с помощью функции `sigaction()` позволяет обработчику получать дополнительную информацию о полученном сигнале. Для этого мы должны объявить обработчик следующим образом:

```
void handler(int sig, siginfo_t *siginfo, void *ucontext);
```

Первый аргумент, `sig`, как и в случае со стандартным обработчиком сигнала, — это номер сигнала. Второй аргумент, `siginfo`, — это структура, используемая для предоставления дополнительной информации о сигнале. Мы рассмотрим эту структуру ниже. Последний аргумент, `ucontext`, также описан ниже.

Так как прототип вышеприведенного обработчика сигнала отличается от прототипа стандартного обработчика, правила типов языка C не позволяют нам задействовать поле `sa_handler` структуры `sigaction` для указания адреса этого обработчика. По этой причине мы вынуждены использовать альтернативное поле: `sa_sigaction`. Иными словами, определение структуры `sigaction` несколько более сложно, чем было показано в разделе 20.13. Полностью определение этой структуры выглядит следующим образом:

```
struct sigaction {
    union {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    } __sigaction_handler;
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
/* Благодаря следующим строкам define поля union выглядят
   как простые поля родительской структуры */
```

```
#define sa_handler __sigaction_handler.sa_handler
#define sa_sigaction __sigaction_handler.sa_sigaction
```

В структуре `sigaction` используется объединение для сочленения полей `sa_sigaction` и `sa_handler`. (В большинстве реализаций UNIX объединение применяется для аналогичных целей.) Объединение возможно, потому что при осуществлении каждого конкретного вызова функции `sigaction()` требуется только одно из этих полей. (Попытка независимо друг от друга установить значения полей `sa_sigaction` и `sa_handler` может привести к ошибкам, возможно, из-за того, что мы повторно используем одну и ту же структуру `sigaction` для нескольких вызовов функции `sigaction()`, чтобы установить обработчики различных сигналов.)

Далее приведен пример задействования флага `SA_SIGINFO` для установки одного обработчика:

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) == -1)
    errExit("sigaction");
```

Для ознакомления с завершенными примерами использования флага `SA_SIGINFO` см. листинги 22.3 и 23.5.

Структура siginfo_t

Структура `siginfo_t`, которая передается в качестве второго аргумента обработчику сигнала, установленного с помощью флага `SA_SIGINFO`, имеет такой вид:

```
typedef struct {
    int     si_signo;         /* Номер сигнала */
    int     si_code;          /* Код сигнала */
    int     si_trapno;        /* Номер ловушки для аппаратно генерируемого сигнала
                                 (не используется в большинстве архитектур) */
    union sigval si_value;    /* Дополнительные данные из sigqueue() */
    pid_t si_pid;             /* ID посылающего процесса */
    uid_t si_uid;             /* Реальный ID пользователя
                                 посылающего процесса */
    int      si_errno;        /* Номер ошибки (обычно не используется) */
    void  *si_addr;           /* Адрес, сгенерировавший сигнал (только
                                 для аппаратно-генерируемых сигналов ) */
    int     si_overrun;       /* Счетчик переполнений таймера
                                 (Linux 2.6, таймеры POSIX) */
    int   si_timerid;         /* (Внутриядерный) ID таймера
                                 (Linux 2.6, таймеры POSIX) */
    long  si_band;            /* Связывающее событие (SIGPOLL/SIGIO) */
    int      si_fd;           /* Файловый дескриптор (SIGPOLL/SIGIO) */
    int      si_status;       /* Код завершения или сигнал (SIGCHLD) */
    clock_t si_utime;         /* Пользовательское время ЦП (SIGCHLD) */
    clock_t si_stime;         /* Системное время ЦП (SIGCHLD) */
} siginfo_t;
```

Макрос тестирования возможности `_POSIX_C_SOURCE` должен быть определен со значением, большим или равным `199309`, чтобы объявление структуры `siginfo_t` было доступно из файла `<signal.h>`.

В Linux, как и в большинстве реализаций UNIX, многие поля структуры `siginfo_t` объединены, так как не все поля нужны для каждого сигнала. (См. `<bits/siginfo.h>` для получения более подробной информации.)

На входе в обработчик сигнала поля структуры `siginfo_t` установлены следующим образом.

- `si_signo` — устанавливается для всех сигналов. Поле содержит номер сигнала, вызвавшего активацию обработчика, то есть то же самое значение, что и аргумент `sig` обработчика.
- `si_code` — устанавливается для всех сигналов. Содержит код, предоставляющий дополнительную информацию об источнике сигнала, как показано в табл. 21.2.
- `si_value` — содержит дополнительные данные, отправляемые в сигнал с помощью функции `sigqueue()`. Функция `sigqueue()` описывается в подразделе 22.8.1.
- `si_pid` — для сигналов, отправляемых через функцию `kill()` или `siqueue()`, это поле содержит идентификатор пославшего сигнал процесса.
- `si_uid` — для сигналов, отправляемых через функцию `kill()` или `siqueue()`, это поле содержит реальный ID пользователя процесса, пославшего сигнал. Система предоставляет реальный ID пользователя процесса, так как он является более информативным по сравнению с действующим ID пользователя. Рассмотрим права доступа для отправки сигналов, описанные в разделе 20.5: если действующий ID пользователя дает отправителю право послать сигнал, значит, он должно быть равен `0` (то есть привилегированный процесс) или совпадать с реальным или сохраненным установленным ID пользователя процесса, получающего сигнал. В этом случае получателю было бы полезно знать реальный ID пользователя отправителя, которое может отличаться от действующего ID пользователя (если отправитель — это программа с установленным ID пользователя).
- `si_errno` — если значение этого поля не равно `0`, значит, оно содержит номер ошибки (как `errno`), идентифицирующий причину отправки сигнала. Как правило, это поле в Linux не задействуется.
- `si_addr` — устанавливается только для аппаратно генерируемых сигналов `SIGBUS`, `SIGSEGV`, `SIGILL` и `SIGFPE`. Для сигналов `SIGBUS` и `SIGSEGV` оно содержит адрес, вызвавший возникновение ошибки неверной ссылки на участок памяти. Для сигналов `SIGILL` и `SIGFPE` это поле содержит адрес программной инструкции, вызвавшей подачу сигнала.

Следующие поля, не являющиеся стандартными расширениями Linux, устанавливаются только по получении сигнала, генерируемого по истечении времени таймера POSIX (см. раздел 23.6).

- `si_timerid` — содержит идентификатор, используемый внутри ядра для обозначения таймера.
- `si_overrun` — устанавливается равным счетчику переполнения таймера.

Следующие поля устанавливаются лишь по получении сигнала `SIGIO` (см. раздел 59.3).

- `si_band` — содержит значение «связывающего события», ассоциированного с событием ввода-вывода. (В версиях glibc вплоть до 2.3.2 поле `si_band` имело тип `int`.)
- `si_fd` — хранит номер файлового дескриптора, ассоциированного с событием ввода-вывода. Это поле не указано в стандарте SUSv3, но есть во многих реализациях.

Следующие поля устанавливаются только по получении сигнала `SIGCHLD` (см. раздел 26.3).

- `si_status` — может содержать либо код завершения дочернего процесса (если значение поля `si_code` установлено как `CLD_EXITED`), либо номер сигнала, отправленного в дочерний процесс (иными словами, номер сигнала, завершившего или остановившего дочерний процесс, как описано в подразделе 26.1.3).

- `si_utime` — содержит пользовательское время ЦП, затраченное дочерним процессом. В версиях ядра 2.6 старше 2.6.27 это время измеряется тактами системных часов (деление на `sysconf(_SC_CLK_TCK)`). В версиях ядра 2.6 младше 2.6.27 из-за допущенной ошибки это поле отражало время, измеряемое в тактах, частота которых могла быть изменена пользователем (см. раздел 10.6). Это поле не установлено стандартом SUSv3, однако присутствует во многих реализациях.
- `si_stime` — хранит системное время ЦП, затраченное дочерним процессом. См. описание поля `si_utime`. Не установлено стандартом SUSv3, однако присутствует во многих реализациях.

Поле `si_code` предоставляет дополнительную информацию об источнике сигнала с помощью значений, приведенных в табл. 21.2. Не все значения, зависящие от конкретного сигнала и приведенные в таблице, свойственны для всех реализаций UNIX и аппаратных архитектур (особенно если речь идет об аппаратно генерируемых сигналах: `SIGBUS`, `SIGSEGV`, `SIGILL` и `SIGFPE`). Однако все эти константы определены в Linux, и большинство из них установлено стандартом SUSv3.

Обратите внимание на следующие аспекты значений, приведенных в табл. 21.2.

- Значения `SI_KERNEL` и `SI_SIGIO` зависят от версии Linux. Они не установлены стандартом SUSv3 и не свойственны абсолютно всем реализациям UNIX.
- Константа `SI_SIGIO` используется только в Linux версии 2.2. Начиная с версии ядра 2.4, эта константа в Linux была заменена на `POLL_*`, также приведенные в таблице.

Таблица 21.2. Значения, возвращаемые полем si_code структуры siginfo_t

Сигнал	Значение поля si_code	Источник сигнала
Любой	SI_ASYNCIO	Завершение асинхронной операции ввода-вывода (AIO)
	SI_KERNEL	Отправлен ядром (например, сигнал драйвера терминала)
	SI_MESGQ	Прибытие сообщения в очередь сообщений POSIX (начиная с Linux 2.6)
	SI_QUEUE	Сигнал реального времени от пользовательского процесса через sigqueue()
	SI_SIGIO	Сигнал SIGIO (Только Linux 2.2)
	SI_TIMER	Истечение времени таймера POSIX (реальное время)
	SI_TKILL	Пользовательский процесс через tkill() или tgkill() (начиная с Linux 2.4.19)
	SI_USER	Пользовательский процесс через kill()
SIGBUS	BUS_ADRALN	Неверное выравнивание адреса
	BUS_ADREER	Несуществующий физический адрес
	BUS_MCEERR_AO	Аппаратная ошибка памяти, возможно действие (начиная с Linux 2.6.32)
	BUS_MCEERR_AR	Аппаратная ошибка памяти, требуется действие (начиная с Linux 2.6.32)
	BUS_OBJERR	Объектная аппаратная ошибка (зависит от объекта)

Продолжение ⇙

Таблица 21.2 *(продолжение)*

Сигнал	Значение поля si_code	Источник сигнала
SIGCHLD	CLD_CONTINUED	Дочерний процесс продолжен по сигналу SIGCONT (начиная с Linux 2.6.9)
	CLD_DUMPED	Дочерний процесс завершен аварийно с дампом ядра
	CLD_EXITED	Из дочернего процесса осуществлен выход
	CLD_KILLED	Дочерний процесс завершен аварийно без дампа ядра
	CLD_STOPPED	Дочерний процесс остановлен
	CLD_TRAPPED	Отслеживаемый дочерний процесс остановлен
SIGFPE	FPE_FLTDIV	Деление на ноль числа с плавающей точкой
	FPE_FLTINV	Неверная операция с числом с плавающей точкой
	FPE_FLTOVF	Переполнение числа с плавающей точкой
	FPE_FLTRES	Неточный результат операции с числом с плавающей точкой
	FPE_FLTUND	Исчезновение значащих разрядов числа с плавающей точкой
	FPE_INTDIV	Целочисленное деление на ноль
	FPE_INTOVF	Переполнение целого числа
	FPE_SUB	Основание числа вне допустимого диапазона
SIGILL	ILL_BADSTK	Внутренняя ошибка стека
	ILL_COPROC	Ошибка сопроцессора
	ILL_ILLADR	Недопустимый режим адресации
	ILL_ILLOPC	Недопустимый код операции
	ILL_ILLOPN	Недопустимый операнд
	ILL_ILLTRP	Недопустимая ловушка
	ILL_PRVOPC	Привилегированный код операции
	ILL_PRVREG	Привилегированный регистр
SIGPOLL/ SIGIO	POLL_ERR	Ошибка ввода-вывода
	POLL_HUP	Устройство отключено
	POLL_IN	Доступны данные ввода
	POLL_MSG	Доступно сообщение ввода
	POLL_OUT	Доступны буферы вывода
	POLL_PRI	Доступен высокоприоритетный ввод
SIGSEGV	SEGV_ACCERR	Нарушение прав доступа к странице памяти
	SEGV_MAPERR	Искомый адрес незадействован

Сигнал	Значение поля si_code	Источник сигнала
SIGTRAP	TRAP_BRANCH	Ловушка ветвления процессов
	TRAP_BRKPT	Точка останова процесса
	TRAP_HWBKPT	Аппаратная точка остановки/наблюдения процесса
	TRAP_TRACE	Ловушка трассировки процесса

Стандартом SUSv4 определяется функция psiginfo(), предназначение которой аналогично функции psignal() (раздел 20.8). Функция psiginfo() принимает два аргумента: указатель на структуру siginfo_t и строку сообщения. Функция распечатывает строку сообщения при возникновении стандартной ошибки, а также дополнительную информацию о сигнале, описываемом в структуре siginfo_t. Функция psiginfo() предоставляется в библиотеке glibc, начиная с версии 2.10. Реализация из библиотеки glibc распечатывает описание сигнала, место происхождения сигнала (в соответствии со значением поля si_code), а также, для некоторых сигналов, содержимое других полей структуры siginfo_t. Функция psiginfo() — это нововведение стандарта SUSv4, поэтому она может доступной не во всех системах.

Аргумент ucontext

Последний аргумент, передаваемый обработчику, установленному с флагом `SA_SIGINFO`, `ucontext`, — это указатель на структуру типа `ucontext_t` (определена в файле `<ucontext.h>`). (В стандарте SUSv3 для этого аргумента применяется указатель типа `void`, так как данный стандарт не устанавливает каких-либо деталей этого аргумента.) Структура предоставляет так называемую информацию о пользовательском контексте, описывающую состояние процесса перед активацией обработчика сигнала, в том числе и предшествующую сигнальную маску и сохраненные значения регистра (например, программный счетчик и указатель стека). Эта информация редко применяется в обработчиках сигналов, поэтому мы не будем вдаваться в подробности.

Еще один вариант применения структур ucontext_t — это функции getcontext(), makecontext(), setcontext(), и swapcontext(), позволяющие процессу соответственно получать, создавать, изменять или заменять контексты выполнения. (Эти операции аналогичны функциям setjump() и longjump(), но менее специфичны.) Эти функции могут применяться для реализации сопрограмм (coroutine), когда поток выполнения переключается между двумя (или более) функциями. В SUSv3 эти функции указаны, но отмечены как устаревшие. Из стандарта SUSv4 эти функции изъяты с указанием, что приложения должны быть переписаны с использованием потоков POSIX. Более подробная информация об этих функциях содержится в справочнике glibc.

21.5. Прерывание и повторный запуск системных вызовов

Рассмотрим следующий сценарий.

1. Мы устанавливаем обработчик некоего сигнала.
2. С устройства терминала мы совершаем блокирующий системный вызов, например, `read()`, который блокируется до момента предоставления ввода.
3. Пока системный вызов заблокирован, осуществляется доставка сигнала, для которого мы установили обработчик, и, соответственно, активация обработчика этого сигнала.

Что случится после возврата из обработчика сигнала? По умолчанию, системный вызов завершается неудачно с ошибкой EINTR («Функция прервана»). Это может быть полезной особенностью. В разделе 23.3 мы рассмотрим, как использовать таймер (результатом чего будет доставка сигнала SIGALRM) для установки тайм-аута блокирующих системных вызовов, таких как read().

Однако зачастую мы предпочли бы продолжить выполнение прерванного системного вызова. Для этого мы можем воспользоваться кодом, например приведенным ниже, позволяющим вручную перезапустить системный вызов, если он был прерван обработчиком сигнала:

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue;   /* Тело цикла без действий */

if (cnt == -1)  /* read() завершился с ошибкой, не равной EINTR */
    errExit("read");
```

Если мы часто пишем такой код, то может быть удобно определить некий макрос, например, так:

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);
```

С его помощью мы можем переписать ранее упомянутый вызов read():

```
NO_EINTR(cnt = read(fd, buf, BUF_SIZE));

if (cnt == -1) /* read() завершился с ошибкой, не равной EINTR */
    errExit("read");
```

GNU библиотека C предоставляет (нестандартный) макрос со схожим предназначением NO_EINTR в файле <unistd.h>. Этот макрос называется TEMP_FAILURE_RETRY() и становится доступным при определении макроса тестирования возможности _GNU_SOURCE.

Даже если мы воспользуемся макросом, подобным NO_EINTR, настройка прерывания обработчиками сигналов системных вызовов может быть затруднительна, так как мы должны будем добавить программный код перед каждым блокирующим системным вызовом (в том случае, если мы хотим повторно запускать вызов в каждом из таких случаев). Вместо этого мы можем указать флаг SA_RESTART при установке обработчика с помощью функции sigaction(). Такой системный вызов будет автоматически перезапущен ядром от имени процесса. Это значит, что нам не нужно самим обрабатывать возможный возврат с ошибкой EINTR для данных системных вызовов.

Флаг SA_RESTART устанавливается для каждого сигнала отдельно. Иными словами, мы можем самостоятельно разрешать некоторым обработчикам сигналов прерывать блокирующий системный вызов, тем временем позволяя другим обработчикам автоматически перезапускать системные вызовы.

Системные вызовы (и библиотечные функции), на которые действует флаг SA_RESTART

К сожалению, не все блокирующие системные вызовы автоматически перезапускаются в результате установки флага SA_RESTART. Отчасти причины этому исторические.

- Перезапуск системных вызовов был представлен в ОС 4.2BSD и использовался для прерванных вызовов функций wait() и waitpid(), а также следующих системных вызовов ввода-вывода: read(), readv(), write(), writev() и блокирующих операций ioctl(). Системные вызовы ввода-вывода прерываемы и, следовательно, автоматически переза-

пускаются с помощью флага SA_RESTART только при работе на «медленных» устройствах. К медленным устройствам относятся терминалы, конвейеры, именованные каналы FIFO и сокеты. Для файлов этих типов различные операции ввода-вывода могут блокироваться. (Напротив: дисковые файлы не попадают в эту категорию медленных устройств, так как операции ввода-вывода дисков обычно могут быть незамедлительно выполнены через буферный кэш. Если требуется дисковый ввод/вывод, то ядро переводит процесс в состояние сна до тех пор, пока не завершится операция ввода-вывода).

- Целый массив других блокирующих системных вызовов унаследован от System V, которая изначально не предоставляла средств для перезапуска системных вызовов.

В Linux следующие блокирующие системные вызовы (и библиотечные функции, реализованные поверх системных вызовов) автоматически перезапускаются в том случае, если они были прерваны обработчиком сигнала, установленным с флагом SA_RESTART.

- Системные вызовы, используемые для ожидания дочернего процесса (см. раздел 26.1): wait(), waitpid(), wait3(), wait4() и waitid().
- Системные вызовы ввода-вывода read(), readv(), write(), writev() и ioctl() при использовании с «медленными» устройствами. В случаях, когда данные уже были частично переданы на момент доставки сигнала, системные вызовы ввода-вывода будут прерваны, но возвратят код успешного завершения: целое число, показывающее количество успешно переданных байт.
- Системный вызов open() в случаях, когда он может блокироваться (например, при открытии канала FIFO, как описано в разделе 44.7).
- Различные системные вызовы, используемые с сокетами: accept(), accept4(), connect(), send(), sendmsg(), sendto(), recv(), recvfrom() и recvmsg(). (В Linux эти системные вызовы не перезапускаются автоматически, если на сокете был установлен тайм-аут с помощью функции setsockopt(). См. страницу справочника signal(7) для получения дополнительной информации.)
- Системные вызовы ввода-вывода, используемые с очередями сообщений POSIX: mq_receive(), mq_timedreceive(), mq_send() и mq_timedsend().
- Системные вызовы и библиотечные функции, используемые для установки блокировки на файл: flock(), fcntl() и lockf().
- Операция FUTEX_WAIT для специфичного системного вызова Linux futex().
- Функции sem_wait() и sem_timedwait(), используемые для декремента семафора POSIX. (В некоторых реализациях UNIX функция sem_wait() перезапускается, если установлен флаг SA_RESTART.)
- Функции, используемые для синхронизации потоков POSIX: pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_timedlock(), pthread_cond_wait(), а также pthread_cond_timedwait().

В версиях ядра до 2.6.22 функции futex(), sem_wait() и sem_timedwait() всегда завершались с ошибкой EINTR в случае прерывания вне зависимости от установки флага SA_RESTART.

Следующие блокирующие системные вызовы (и библиотечные функции, реализованные поверх системных вызовов) никогда не перезапускаются (даже если установлен флаг SA_RESTART.)

- Мультиплицирующие вызовы ввода-вывода poll(), ppoll(), select() и pselect(). (В стандарте SUSv3 явно указывается, что поведение вызовов select() и pselect() в случае прерывания обработчиком сигнала не устанавливается вне зависимости от флага SA_RESTART).
- Специфичные системные вызовы Linux epoll_wait() и epoll_pwait().
- Специфичный системный вызов Linux io_getevents().

- Блокирующие системные вызовы, использовавшиеся с очередями сообщений и семафорами System V: `semop()`, `semtimedop()`, `msgrcv()` и `msgsnd()`. (Несмотря на то что изначально в System V не был реализован автоматический перезапуск системных вызовов, в некоторых реализациях UNIX эти системные вызовы *перезапускаются* при установке флага `SA_RESTART`.)
- Вызов `read()` из файлового дескриптора `inotify`.
- Системные вызовы и библиотечные функции, созданные специально для ожидания доставки сигнала: `pause()`, `sigsuspend()`, `sigtimedwait()` и `sigwaitinfo()`.

Изменение флага SA_RESTART сигнала

Функция `siginterrupt()` изменяет установку `SA_RESTART` конкретного сигнала.

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

Возвращает `0` при успешном завершении и `-1` при ошибке

Если параметр `flag` истинен (`1`), значит, обработчик сигнала `sig` будет прерывать блокирующие системные вызовы. Если `flag` ложен (`0`), то блокирующие системные вызовы будут перезапущены после выполнения обработчика сигнала `sig`.

Функция `siginterrupt()` в своей работе использует функцию `sigaction()` для получения копии текущей диспозиции сигнала, изменяет значение флага `SA_RESTART`, возвращаемого структурой `oldact`, а затем вызывает функцию `sigaction()` еще раз, чтобы обновить диспозицию сигнала.

В стандарте SUSv4 функция `siginterrupt()` помечена как устаревшая с рекомендацией использовать для этих целей функцию `sigaction()`.

Необрабатываемые сигналы остановки могут сгенерировать ошибку EINTR для некоторых системных вызовов Linux

В Linux некоторые блокирующие системные вызовы могут возвращать ошибку `EINTR` даже при отсутствии обработчика сигнала. Это может случиться, если системный вызов заблокирован и процесс сначала остановлен сигналом (`SIGSTOP`, `SIGTSTP`, `SIGTTIN` и `SIGTTOU`), а затем возобновлен сигналом `SIGCONT`.

Так ведут себя следующие системные вызовы: `epoll_pwait()`, `epoll_wait()`, `read()` из файлового дескриптора `inotify`, `semop()`, `semtimedop()`, `sigtimedwait()` и `sigwaitinfo()`.

В ядрах, предшествовавших версии 2.6.24, так вел себя вызов `poll()`, в ядрах старше версии 2.6.22 — вызовы `sem_wait()`, `sem_timedwait()`, `futex(FUTEX_WAIT)`, вызовы `msgrcv()`, `msgsnd()` в ядрах до 2.6.9 и `nanosleep()` в Linux 2.4 и более ранних.

В Linux 2.4 и более ранних версиях вызов `sleep()` также мог быть прерван подобным образом, но вместо возврата ошибки он возвращал количество оставшихся секунд сна.

Результатом такого поведения является то, что если существует вероятность того, что наша программа может быть остановлена и перезапущена сигналами, значит, нам может потребоваться включить в нее программный код для перезапуска вышеперечисленных системных вызовов, даже если в программе не устанавливаются обработчики сигналов остановки.

21.6. Резюме

В этой главе мы рассмотрели целый набор факторов, влияющих на работу и проектирование обработчиков сигналов.

Так как сигналы не ставятся в очередь, обработчики сигналов должны иногда быть запрограммированы таким образом, чтобы они могли справляться с вероятностью одновременного свершения нескольких событий одного конкретного типа, даже если получателю доставлен только один сигнал. Проблемы реентерабельности влияют на то, каким образом мы обновляем глобальные переменные, и ограничивают набор функций, которые мы можем безопасно вызывать из обработчиков сигналов.

Кроме выполнения возврата управления, обработчик сигнала может быть завершен одним из нескольких способов, в том числе вызов функции `_exit()`, путем отправки сигнала (`kill()`, `raise()` или `abort()`) либо выполнения нелокального перехода. Использование функций `sigsetjmp()` и `siglongjmp()` предоставляет программе явный контроль над обработкой сигнальной маски процесса при выполнении нелокального перехода.

Мы можем использовать функцию `signalstack()` для определения альтернативного сигнального стека процесса. Это участок памяти, который будет задействоваться вместо стандартного стека процесса при активации обработчика сигнала. Альтернативный стек может потребоваться в ситуациях, когда ресурсы стандартного разросшегося стека израсходованы (в этом случае ядро направляет процессу сигнал `SIGSEGV`).

Флаг `SA_SIGINFO` функции `sigaction()` позволяет установить обработчик сигнала, получающий дополнительную информацию о сигнале. Эта информация предоставляется через структуру `siginfo_t`, адрес которой передается обработчику сигнала в качестве аргумента.

Когда обработчик сигнала прерывает заблокированный системный вызов, этот системный вызов завершается с ошибкой `EINTR`. Мы можем воспользоваться этой моделью поведения для, например, установки таймера на блокирующий системный вызов. При необходимости прерванные системные вызовы могут быть перезапущены вручную. Кроме того, установка обработчика сигнала с помощью функции `sigaction()` с флагом `SA_RESTART` приводит к автоматическому перезапуску многих (но не всех) системных вызовов.

Дополнительная информация

См. источники, перечисленные в разделе 20.15.

21.7. Упражнение

21.1. Реализуйте функцию `abort()`.

22 Сигналы: дополнительные возможности

В этой главе мы завершим рассматривать сигналы. Здесь мы обсудим некоторые более сложные темы, включая такие, как:

- файлы дампа ядра;
- особые случаи доставки сигнала, его диспозиции и обработки;
- синхронная и асинхронная генерация сигналов;
- когда и в какой очередности доставляются сигналы;
- сигналы реального времени;
- использование функции `sigsuspend()` для настройки сигнальной маски процесса и ожидания прибытия сигнала;
- использование функции `sigwaitinfo()` (и `sigtimedwait()`) для синхронного ожидания доставки сигнала;
- использование функции `signalfd()` для получения сигнала через файловый дескриптор;
- старые сигнальные API операционной системы BSD.

22.1. Файлы дампа ядра

Некоторые сигналы заставляют процесс создать файл дампа ядра и завершиться (см. табл. 20.1). Дамп ядра — это файл, содержащий образ памяти процесса на момент его завершения. (Термин *«ядро»* (core) восходит к старой технологии памяти.) Такой образ может быть загружен в отладчик для изучения состояния программного кода и данных в момент получения сигнала.

Одним из способов заставить программу создать файл дампа ядра является ввод символа *«выход»* (обычно Ctrl+\), вызывающего генерацию сигнала `SIGQUIT`.

```
$ ulimit -c unlimited          Объяснено в тексте
$ sleep 30
Нажатие Ctrl+\
Quit (core dumped)
$ ls -l core                  Показывает файл дампа ядра для sleep(1)
-rw-------   1 mtk   users     57344  Nov 30 13:39 core
```

В данном примере оболочка распечатывает сообщение `Quit (core dumped)` после обнаружения того, что дочерний процесс (запустивший программу `sleep`) был завершен сигналом `SIGQUIT` с созданием файла дампа ядра.

Файл дампа ядра был создан в рабочем каталоге процесса и назван `core`. В скором времени будет рассказано, как можно изменить эти настройки по умолчанию.

> Во многих реализациях есть инструмент (например, gcore во FreeBSD и Solaris), позволяющий получить дамп ядра запущенного процесса. Аналогичная функциональность доступна и на Linux за счет подключения процесса с помощью gdb и последующего запуска команды gcore.

Обстоятельства, при которых файлы дампа ядра не создаются

Файл дампа ядра не создается в следующих случаях.

- У процесса нет права записи файла дампа ядра. Это может произойти, если у процесса отсутствуют права записи в каталоге, в котором создается файл дампа, или потому, что файл с таким именем уже существует в каталоге и либо недоступен для записи, либо не является обычным файлом (например, это может быть каталог или символьная ссылка).
- Обычный файл с таким именем уже существует и доступен для записи, но на него уже создано несколько (жестких) ссылок.
- Каталог, в котором планируется создание файла дампа, не существует.
- Ограничение ресурсов процесса для размера файла дампа памяти ядра установлено на `0`. Это ограничение, `RLIMIT_CORE`, более подробно рассматривается в разделе 36.3. В вышеприведенном примере мы использовали команду `ulimit` (`limit` в оболочке C shell) для гарантии того, что для файлов `core` никакие ограничения не устанавливаются.
- Ограничение ресурсов процесса для размера файла, которое может быть создано процессом, установлено на `0`. Это ограничение, `RLIMIT_FSIZE`, обсуждается более подробно в разделе 36.3.
- Бинарный исполняемый файл недоступен для чтения. Это не позволяет пользователям обратиться к дампу ядра для получения кода программы, недоступного никаким иным способом.
- Файловая система, в которой находится текущий рабочий каталог, монтирована в режиме «только для чтения», не располагает свободным местом или свободными индексными дескрипторами. Возможно, пользователь также достиг ограничения выделенной ему квоты пространства файловой системы.
- Выполняемая программа с установленным ID пользователя (или с установленным ID группы) не генерирует дамп ядра, если выполняется не владельцем файла (или из группы владельца). Это предотвращает ситуации, когда злоумышленники могут воспользоваться дампом памяти для получения и изучения конфиденциальной информации из программы, например паролей.

  Применяя операцию PB_SET_DUMPABLE в характерном для Linux системном вызове prctl(), мы можем установить для процесса флаг dumpable. Таким образом, когда программа с установленным ID пользователя (группы) запускается пользователем, не являющимся ее владельцем (из группы владельца), может быть создан дамп ядра. Операция PB_SET_DUMPABLE доступна в Linux, начиная с версии 2.4. Для получения дополнительной информации см. страницу справочника prctl(2). Кроме того, начиная с версии ядра 2.6.13, файл /proc/sys/fs/suid_dumpable предоставляет контроль уровня системы за тем, создают ли процессы, запущенные с установленным ID пользователя (группы) файлы дампа ядра. Для получения дополнительной информации см. страницу справочника prctl(5).

Начиная с версии 2.6.23 характерный для Linux файл `/proc/`*`PID`*`/coredump_filter` может для каждого процесса определить, какого рода отображения памяти записываются в файл дампа ядра. (Виды отображений памяти будут рассмотрены в главе 45.) Значение, содержащееся в этом файле — это маска, состоящая из четырех битов, соответствующих четырем типам отображений памяти: приватные анонимные отображения, приватные отображения файлов, анонимные отображения с общим доступом и отображения файлов с общим доступом. Значение файла по умолчанию представляет собой традиционное поведение Linux: создается дамп только частных анонимных отображений и анонимных отображений с общим доступом. Для получения дополнительной информации см. страницу справочника `core(5)`.

Имена файлов дампа ядра: /proc/sys/kernel/core_pattern

Начиная с Linux 2.6, строка формата, содержащаяся в характерном для Linux файле `/proc/sys/kernel/core_pattern`, контролирует присвоение имен всем файлам дампа ядра, создаваемым в системе. По умолчанию этот файл содержит строку `core`. Привилегированный пользователь может настроить процесс присвоения имен так, чтобы имя файлов дампа содержало любой из перечисленных в табл. 22.1 спецификаторов формата. Они заменяются значением, приведенным в правом столбце таблицы. Кроме того, строка может содержать косые черты (`/`). Иными словами, можно контролировать не только имя файла дампа, но также и каталог (абсолютный и относительный), в котором файл создается. После замены всех спецификаторов формата получившаяся строка путевого имени уменьшается до 128 символов максимально (или 64 в версиях Linux до 2.6.19).

Начиная с версии ядра 2.6.19, Linux поддерживает дополнительный синтаксис в файле `core_pattern`. Если этот файл содержит строку, начинающуюся с символа конвейера (`|`), значит, все последующие символы файла воспринимаются как программа с необязательными аргументами (которые могут также включать классификаторы с символом `%`, перечисленные в табл. 22.1). Эта команда будет выполнена при создании файла дампа ядра. Дамп ядра записывается в стандартный ввод программы, а не в файл. Для получения дополнительной информации см. страницу справочника `core(5)`.

> Некоторые другие реализации UNIX предоставляют средства, аналогичные core_pattern. Например, в системах, производных от BSD, имя программы приставляется к имени файла, например core.progname. В Solaris предоставляется специальный инструмент (coreadm), который позволяет пользователю выбрать имя файла и каталог, в который будет сохранен файл дампа ядра.

Таблица 22.1. Спецификаторы формата для /proc/sys/kernel/core_pattern

Спецификатор	Заменяется на
%c	Мягкое ресурсное ограничение размера файла дампа (в байтах, начиная с Linux 2.6.24)
%e	Имя исполняемого файла (без путевого префикса)
%g	Реальный групповой идентификатор процесса завершенного с дампом
%h	Имя хоста системы
%p	Идентификатор процесса, завершенного с дампом
%s	Номер сигнала, завершившего процесс
%t	Время создания дампа, в секундах с начала отсчета
%u	Реальный ID пользователя процесса, завершенного с дампом
%%	Единичный символ %

22.2. Частные случаи доставки, диспозиции и обработки

Для некоторых сигналов применяются особые правила доставки, задания диспозиции и обработки, как описано в этом разделе.

SIGKILL и SIGSTOP

Невозможно изменить действие по умолчанию сигнала SIGKILL, который всегда завершает процесс и SIGSTOP, который всегда останавливает процесс. Обе функции, signal() и sigaction(), возвращают ошибку при попытке изменить диспозицию этих сигналов. Эти сигналы не могут быть заблокированы, что характерно именно для них. Запрет изменения действий этих сигналов по умолчанию означает, что они всегда могут быть использованы для завершения или остановки процесса, над которым был потерян контроль.

SIGCONT и стоп-сигналы

Как уже отмечалось ранее, сигнал SIGCONT используется для возобновления процесса, ранее остановленного одним из стоп-сигналов (SIGSTOP, SIGTSTP, SIGTTIN и SIGTTOU). Из-за уникального предназначения в некоторых ситуациях ядро обрабатывает эти сигналы отлично от других.

Если процесс в данный момент остановлен, то получение сигнала SIGCONT всегда вызывает его возобновление, даже если процесс временно блокирует или игнорирует сигналы SIGCONT. Эта особенность является необходимостью, так как в противном случае возобновить такие остановленные процессы было бы невозможно. (Если остановленный процесс блокировал сигналы SIGCONT, и для него был установлен обработчик сигнала SIGCONT, то данный обработчик инициируется, только когда сигналы SIGCONT будут разблокированы в дальнейшем).

> Если в остановленный процесс посылается любой другой сигнал, то он на самом деле не будет доставлен до тех пор, пока процесс не будет возобновлен через получение сигнала SIGCONT. Единственным исключением из этого правила является сигнал SIGKILL, который всегда завершает процесс, даже если этот процесс в настоящее время остановлен.

При получении сигнала SIGCONT происходит удаление всех ожидающих процесса стоп-сигналов (иными словами, процесс их даже не видит). Напротив, при доставке любого стоп-сигнала происходит автоматическое удаление ожидающего сигнала SIGCONT. Эти шаги предпринимаются для предотвращения отмены действия SIGCONT последующей доставкой отправленных ранее, но находящихся в режиме ожидания стоп-сигналов и наоборот.

Не изменяйте диспозицию игнорируемых сигналов, генерируемых терминалом

Если на момент запуска программа обнаруживает, что диспозиция сигнала, генерируемого терминалом, была установлена равной SIG_IGN (игнорировать), то в большинстве случаев программа не должна изменять диспозицию такого сигнала. Это правило не является системным требованием, но, скорее, есть соглашение, которого следует придерживаться при написании приложений. Я объясню причины этого в подразделе 34.7.3. Сигналы, для которых действительно это соглашение: SIGHUP, SIGINT, SIGQUIT, SIGTTIN, SIGTTOU и SIGTSTP.

22.3. Прерываемые и непрерываемые состояния сна процесса

Нам необходимо сделать оговорку к ранее сказанному утверждению, что сигналы SIGKILL и SIGSTOP всегда оказывают немедленное действие на процесс. В различные моменты времени ядро может перевести процесс в режим сна, при этом различают два состояния сна.

- TASK_INTERRUPTABLE. Процесс ожидает свершения некоего события. Например, это может быть ожидание ввода с терминала, запись данных в пустой на данный момент

канал или увеличение значения семафора в System V. Процесс может провести в этом состоянии очень длительное время. Если сигнал генерируется для процесса, находящегося в этом состоянии, то операция прерывается — и процесс пробуждается доставкой сигнала. При выводе списка процессов с помощью команды `ps(1)`, процессы, находящиеся в состоянии `TASK_INTERRUPTABLE`, помечены буквой `S` в поле `STAT` (статус процесса).

- `TASK_UNINTERRUPTABLE`. Процесс ожидает свершения события определенного специального класса, например завершения операции дискового ввода-вывода. Если сигнал генерируется для процесса в этом состоянии, то сигнал не будет доставлен до тех пор, пока процесс не выйдет из сна. Процессы в состоянии `TASK_UNINTERRUPTABLE` при перечислении командой `ps(1)` помечаются буквой `D` в поле `STAT`.
  Поскольку процесс находится в состоянии `TASK_UNINTERRUPTABLE` лишь непродолжительное время, то факт того, что сигнал доставляется только в момент выхода процесса из сна, практически незаметен. Однако в некоторых ситуациях процесс может зависнуть в этом состоянии, возможно, из-за неполадок с оборудованием, проблем NFS или ошибки в коде ядра. В таких случаях сигнал `SIGKILL` не завершит зависший процесс. Если причина этого состояния не может быть решена иным образом, то для завершения процесса нам потребуется перезагрузить систему.
  Состояния `TASK_INTERRUPTABLE` и `TASK_UNINTERRUPTABLE` предусмотрены в большинстве реализаций UNIX. Начиная с версии ядра 2.6.25, в Linux появилось также третье состояние, которое призвано разрешить вышеописанную проблему зависшего процесса.
- `TASK_KILLABLE`. Это состояние аналогично `TASK_UNINTERRUPTABLE`, с той лишь разницей, что процесс пробуждается из него в случае получения фатального сигнала (то есть сигнала, завершающего процесс). Переписав соответствующие участки кода ядра с учетом этого состояния, можно избежать различных ситуаций, при которых зависший процесс требует перезагрузки системы. Первым участком кода ядра, конвертированным на использование `TASK_KILLABLE`, была сетевая файловая система NFS.

22.4. Аппаратно генерируемые сигналы

Сигналы `SIGBUS`, `SIGFPE`, `SIGILL` и `SIGSEGV` могут быть сгенерированы вследствие аппаратного исключения или, что реже, путем отправки через функцию `kill()`. В случае аппаратного исключения SUSv3 устанавливает поведение процесса как неопределенное, если выполняется возврат из обработчика сигнала или если он блокирует или игнорирует сигнал. Для этого есть следующие причины.

- *Возврат из обработчика сигнала.* Предположим, что некоторый машинный код генерирует один из перечисленных сигналов, следовательно, инициируется обработчик. При нормальном возврате из обработчика программа пытается возобновить выполнение с той точки, в которой она была прервана. Однако это и есть та самая инструкция, которая сгенерировала сигнал, следовательно, сигнал генерируется повторно. Последствием такого поведения обычно является то, что программа уходит в бесконечный цикл, вновь и вновь вызывая обработчик сигнала.
- *Игнорирование сигнала.* В игнорировании аппаратно генерируемого сигнала очень мало смысла, так как непонятно, каким образом программа должна продолжать выполнение в случае, например, арифметического исключения. При генерации одного из вышеперечисленных сигналов в результате аппаратного исключения Linux

доставляет этот сигнал в программу, даже несмотря на инструкцию игнорировать такие сигналы.

- *Блокирование сигнала.* Как и в предыдущем случае, в блокировании сигнала очень мало смысла, так как непонятно, каким образом программа должна продолжать выполнение. В Linux 2.4 и более ранних версиях ядро просто игнорирует попытки заблокировать аппаратно генерируемый сигнал. Он доставляется в процесс в любом случае, а затем либо завершает процесс, либо перехватывается обработчиком, если таковой был установлен. Начиная с Linux 2.6, если сигнал заблокирован, то процесс всегда незамедлительно аварийно завершается этим сигналом, даже если для процесса установлен обработчик данного сигнала. (Причина такого кардинального изменения в Linux 2.6 по части обработки заблокированных аппаратно генерируемых сигналов в скрытых ошибках поведения Linux 2.4, которые могли приводить к полному зависанию распоточенных программ.)

  Программа signals/demo_SIGFPE.c, поставляемая вместе с файлами исходного кода к этой книге, может быть использована для демонстрации результатов игнорирования или блокирования сигнала SIGFPE либо его перехвата обработчиком, который выполняет нормальный возврат.

Правильным способом работы с аппаратно генерируемыми сигналами является либо принятие их действия по умолчанию (завершение процесса), либо написание обработчиков, которые не выполняют нормальный возврат. Вместо выполнения нормального возврата обработчик может завершить выполнение вызовом функции `_exit()` для завершения процесса либо вызовом функции `siglongjump()` (см. подраздел 21.2.1) для гарантии того, что управление передается в некую точку программы, отличную от инструкции, вызвавшей генерацию сигнала.

22.5. Синхронная и асинхронная генерация сигнала

Мы уже увидели, что процесс, как правило, не может предсказать, когда он получит сигнал. Теперь нам необходимо уточнить это наблюдение, рассмотрев различия между *синхронной* и *асинхронной* генерацией сигналов.

Модель, которую мы до сих пор неявно подразумевали, называется *асинхронной* генерацией сигналов. В рамках ее сигнал посылается либо другим процессом, либо генерируется ядром при свершении события, не зависящего от выполняемого процесса (например, когда пользователь вводит символ *прерывания* или завершается дочерний по отношению к рассматриваемому процесс). Для асинхронно генерируемых сигналов утверждение, что процесс не может предсказать, когда он получит сигнал, истинно.

Однако в некоторых случаях сигнал генерируется во время выполнения процесса. Мы уже рассмотрели два примера таких ситуаций.

- Аппаратно генерируемые сигналы (`SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` и `SIGEMT`), описываемые в разделе 22.4, генерируются в результате выполнения конкретного машинного кода, результатом выполнения которого является аппаратное исключение.
- Процесс может использовать функции `raise()`, `kill()` или `killpg()` для отправки сигналов самому себе.

В вышеописанных случаях генерация сигнала *синхронная* — сигнал доставляется моментально (если этот сигнал не заблокирован, однако см. раздел 22.4 для получения информации о том, что происходит при блокировании аппаратно генерируемых сигналов).

Иными словами, утверждение о непредсказуемости доставки сигналов в данном случае неприменимо. Для синхронно генерируемых сигналов доставка предсказуема и воспроизводима.

Обратите внимание, что синхронность — это описание того, каким образом сигнал генерируется, а не самого сигнала. Все сигналы могут быть сгенерированы синхронно (например, когда процесс отправляет самому себе сигнал `kill()`) или асинхронно (например, когда сигнал `kill()` отправлен другим процессом).

22.6. Тайминг и порядок доставки сигнала

В качестве первой темы этого раздела мы рассмотрим, когда именно доставляется ожидающий сигнал. А затем — что происходит при одновременном разблокировании нескольких ожидающих сигналов.

Когда доставляется сигнал

Как отмечалось в разделе 22.5, синхронно генерируемые сигналы доставляются незамедлительно. Например, аппаратное исключение приводит к немедленной генерации сигнала, а когда процесс посылает самому себе сигнал с помощью функции `raise()`, то сигнал доставляется перед тем, как происходит возврат из вызова `raise()`.

При асинхронной генерации сигнала может наблюдаться (небольшая) задержка во время нахождения его в режиме ожидания после генерации и перед фактической доставкой, даже если этот сигнал не заблокирован. Причиной этого является тот факт, что ядро доставляет в процесс ожидающий сигнал только при следующем переключении из режима ядра в режим пользователя во время выполнения данного процесса. На практике это означает, что процесс доставляется в один из следующих моментов:

- когда процесс заново запланирован после предшествующего тайм-аута (то есть в начале тайм-слота);
- по завершении системного вызова (доставка сигнала может привести к преждевременному завершению блокирующего системного вызова).

Очередность доставки нескольких разблокированных сигналов

Если несколько сигналов ожидают доставки в процесс и были одновременно разблокированы функцией `sigprocmask()`, то все эти сигналы будут незамедлительно доставлены в процесс.

В текущих версиях реализаций ядро Linux доставляет сигналы в порядке возрастания. Например, при одновременном разблокировании сигналов `SIGINT` (сигнал 2) и `SIGQUIT` (сигнал 3) сигнал `SIGINT` будет доставлен раньше, чем сигнал `SIGQUIT` вне зависимости от того, в какой последовательности эти сигналы были сгенерированы.

Впрочем, мы не можем всецело полагаться на то, что (стандартные) сигналы будут доставляться в какой-то конкретной очередности, так как согласно стандарту SUSv3 очередность доставки нескольких сигналов зависит от реализации системы. (Это высказывание действительно только для стандартных сигналов. Как мы увидим в разделе 22.8, стандарты, специфицирующие сигналы реального времени, гарантируют очередность доставки нескольких разблокированных сигналов реального времени.)

В случаях, когда несколько разблокированных сигналов ожидают доставки, если переключение между режимами ядра и пользователя происходит во время выполнения обработчика, то выполнение этого обработчика будет прервано активацией второго обработчика сигнала (и т. д.), как показано на рис. 22.1.

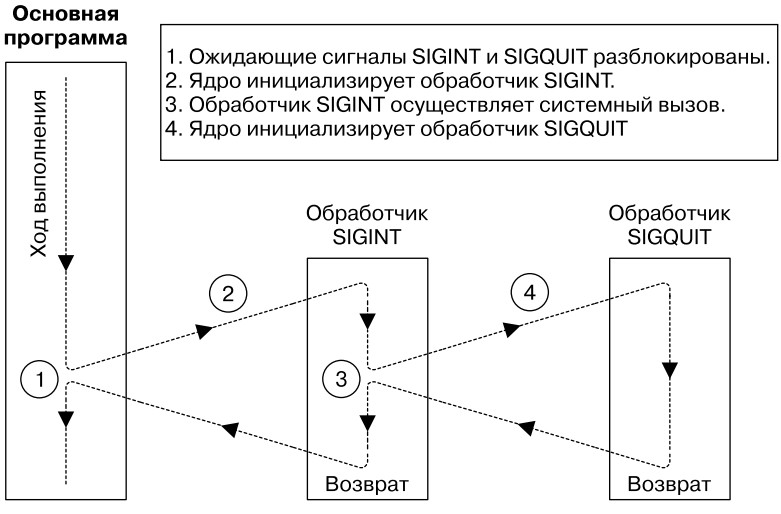


Рис. 22.1. Доставка нескольких разблокированных сигналов

22.7. Реализация и переносимость функции signal()

В этом разделе будет показано, как реализовать функцию `signal()` через функцию `sigaction()`. Реализация прямолинейна, однако следует учесть тот факт, что исторически и в разных реализациях UNIX семантика функции `signal()` была различной. В частности, ранние реализации сигналов были ненадежными, а это означало следующее.

- По входу в обработчик сигнала диспозиция этого сигнала сбрасывалась до значения по умолчанию. (Это соответствует флагу `SA_RESETHAND`, описанному в разделе 20.13.) Чтобы при повторной доставке этого же сигнала был активирован обработчик сигнала, программист должен вызывать `signal()` внутри обработчика, и тогда обработчик явным образом переустановится. Проблема такого подхода заключается в наличии небольшого временного промежутка между входом в обработчик и переустановкой обработчика. Если сигнал прибывает как раз в этот промежуток, то он будет обработан в соответствии с диспозицией по умолчанию.
- Доставка последующих копий сигнала не блокировалась во время выполнения обработчика сигнала. (Соответствует флагу `SA_NODEFER`, описанному в разделе 20.13.) Это означало, что если сигнал был повторно доставлен во время выполнения обработчика, то происходила рекурсивная активация обработчика. При достаточно интенсивном потоке сигналов происходящие рекурсивные активации могли переполнять стек.

В добавок к ненадежности ранние версии UNIX не предоставляли автоматический перезапуск системных вызовов (иными словами, поведение, описанное для флага `SA_RESTART` в разделе 21.5).

Надежные сигналы 4.2BSD сняли эти ограничения, чему также последовали некоторые другие реализации UNIX. Однако старая семантика сохраняется и сегодня в реализации функции `signal()` в System V, более того, в современных стандартах, таких как SUSv3 и C99, эти аспекты намеренно не специфицированы.

Соединяя все вышесказанное, мы реализуем функцию `signal()` как показано в листинге 22.1. По умолчанию эта реализация предоставляет современную семантику сигналов.

При компиляции с опцией `-DOLD_SIGNAL` эта реализация предоставит раннюю ненадежную семантику сигналов, автоматический перезапуск системных вызовов также будет отключен.

Листинг 22.1. Реализация функции signal()

signals/signal.c

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t
signal(int sig, sighandler_t handler)
{
    struct sigaction newDisp, prevDisp;

    newDisp.sa_handler = handler;
    sigemptyset(&newDisp.sa_mask);
#ifdef OLD_SIGNAL
    newDisp.sa_flags = SA_RESETHAND | SA_NODEFER;
#else
    newDisp.sa_flags = SA_RESTART;
#endif

    if (sigaction(sig, &newDisp, &prevDisp) == -1)
        return SIG_ERR;
    else
        return prevDisp.sa_handler;
}
```

signals/signal.c

Некоторые детали glibc

Реализация функции `signal()` в библиотеке glibc изменялась с течением времени несколько раз. В современных версиях библиотеки (glibc 2 и новее) по умолчанию предоставляется новая семантика. В старых версиях библиотеки дается более ранняя ненадежная (совместимая с System V) семантика.

> Ядро Linux содержит реализацию функции signal() как системного вызова. Эта реализация предоставляет старую ненадежную семантику. Однако библиотека glibc обходит системный вызов, предоставляя библиотечную функцию signal(), осуществляющую вызов sigaction().

Если мы хотим получить ненадежные семантики сигналов в современных версиях библиотеки `glibc`, то мы можем явно заменить наши вызовы функции `signal()` вызовами (нестандартной) функции `sysv_signal()`.

```
#define _GNU_SOURCE
#include <signalh.>

void ( *sysv_signal(int sig, void (*handler)(int)) ) (int);
```

Возвращает предыдущую диспозицию сигнала
при успешном завершении или `SIG_ERR` при ошибке

Функция sysv_signal() принимает те же самые аргументы, что и функция signal().

Если макрос тестирования возможности _BSD_SOURCE не определен при компиляции программы, то библиотека glibc неявно переопределяет все вызовы функции signal() на вызовы sysv_signal(), а это значит, что signal() получает ненадежную семантику сигналов. По умолчанию макрос тестирования возможности _BSD_SOURCE *определен*, однако он отключается (за исключением случая, когда он определен явным образом) в случае обнаружения при компиляции программы других макросов тестирования возможности, например SVID_SOURCE или _XOPEN_SOURCE.

sigaction() — предпочтительный API для установки обработчика сигнала

Из-за вышеописанных проблем переносимости между ОС System V и BSD (старые и современные версии библиотеки glibc) лучшим подходом всегда является использование функции sigaction(), а не signal() для установки обработчиков сигналов. Мы будем делать именно так в оставшейся части книги. (Альтернатива — написание собственной версии функции signal(), возможно аналогичной листингу 22.1, которая позволит указать собственные флаги по необходимости, и применение этой версии в наших приложениях.) Стоит заметить, что код будет переносимее (и короче) при использовании функции signal() для установки диспозиции по умолчанию сигналов SIG_IGN и SIG_DFL; мы зачастую будем использовать функцию signal() для этих целей.

22.8. Сигналы реального времени

Сигналы реального времени были определены в стандарте POSIX.1b для устранения ограничений, накладываемых стандартными сигналами. По сравнению с ними сигналы реального времени обладают следующими преимуществами.

- Предоставляют дополнительный диапазон сигналов, которые могут применяться для целей, определяемых приложением. Для этого доступны только два стандартных сигнала — SIGUSR1 и SIGUSR2.
- Ставятся в очередь. Если некий сигнал реального времени посылается в процесс несколько раз, значит, процесс и получит его несколько раз. Напротив, при отправке нескольких экземпляров стандартного сигнала, уже находящегося в режиме ожидания процесса, такой сигнал будет доставлен лишь единожды.
- При отправке сигнала реального времени есть возможность указать данные (в виде целого числа или числа с плавающей точкой), которые будут сопровождать сигнал. Обработчик сигнала в целевом процессе может получить эти данные.
- Гарантируется очередность доставки различных сигналов реального времени. Если несколько сигналов реального времени находятся в режиме ожидания, то первым будет доставлен сигнал с наименьшим номером. Иными словами, существует определенная приоритетность сигналов, при которой сигнал с меньшим номером имеет больший приоритет. Если несколько сигналов одного типа находятся в очереди доставки (вместе с сопровождающими их данными), то эти сигналы доставляются в той очередности, в которой они были отправлены.

Согласно требованиям стандарта SUSv3 реализация должна предоставлять как минимум _POSIX_RTSIG_MAX (константа определена со значением 8) различных сигналов реального времени. Ядро Linux определяет 33 различных сигнала реального времени, пронумерованных от 32 до 64. В заголовочном файле <limits.h> определена константа RTSIG_MAX, указывающая количество доступных сигналов реального времени. В файле

также определены константы SIGRTMIN и SIGRTMAX, обозначающие наименьший и наибольший доступные номера сигналов реального времени.

> В системах, в которых задействуется реализация потоков LinuxThreads, константа SIGRTMIN определена со значением 35 (а не 32), потому что в технологии LinuxThreads первые три сигнала реального времени применяются для внутренних целей. В системах, в которых задействуется реализация потоков NPTL, константа SIGRTMIN определена со значением 34, так как в технологии NPTL первые два сигнала реального времени применяются для внутренних целей.

В отличие от стандартных сигналы реального времени не определяются отдельными константами. Однако не стоит использовать в коде программы непосредственно целочисленные значения номеров сигналов, так как диапазон номеров в различных реализациях UNIX отличается. Вместо этого сигнал реального времени может быть обозначен с помощью прибавления значения к константе SIGRTMIN, так выражение (SIGRTMIN+1) ссылается на второй сигнал реального времени.

Обратите внимание, что стандарт SUSv3 не требует, чтобы значения констант SIGRTMIN и SIGRTMAX были представлены обычными целыми числами. Значения этих констант могут быть определены как функции (как это происходит в Linux). А это значит, что мы не можем написать для препроцессора код, аналогичный следующему:

```
#if SIGRTMIN+100 > SIGRTMAX        /* НЕПРАВИЛЬНО! */
#error "Not enough realtime signals"
#endif
```

Вместо этого мы должны осуществить эквивалентные проверки во время выполнения.

Ограничения количества сигналов реального времени в очереди

Создание очереди из сигналов реального времени означает, что ядру необходимо хранить структуры данных, в которых перечисляются сигналы в очереди для каждого процесса. Так как эти структуры данных потребляют память ядра, ядро устанавливает ограничения на количество сигналов реального времени, которые могут быть поставлены в очередь.

В этом случае стандарт SUSv3 позволяет реализации устанавливать верхний предел количества сигналов в очереди, а также требует, чтобы это количество как минимум равнялось _POSIX_SIGQUEUE_MAX (константа определена со значением 32). В реализации может быть определена константа SIGQUEUE_MAX, показывающая наибольшее количество сигналов реального времени, которые могут быть поставлены в очередь. Данная информация может быть доступна посредством осуществления следующего вызова:

```
lim = sysconf(_SC_SIGQUEUE_MAX);
```

В Linux данный вызов возвращает значение -1. Причина кроется в том, что в Linux реализована иная модель ограничения количества сигналов реального времени, которые могут быть поставлены в очередь процесса. В Linux версий до 2.6.7 включительно ядро требовало установки ограничения уровня системы для общего количества сигналов реального времени, которые могут быть поставлены в очереди всех процессов. Это ограничение может быть просмотрено и (при наличии привилегий) изменено в специфичном файле Linux /proc/sys/kernel/rtsig-max. По умолчанию значение в этом файле равно 1024. Количество сигналов, находящихся в очереди в данный момент, можно просмотреть в характерном для Linux файле /proc/sys/kernel/rtsig-nr.

Начиная с Linux 2.6.8, эта модель была изменена, а вышеупомянутые интерфейсы /proc — удалены. По новой модели константа RLIMIT_SIGPENDING устанавливает мягкое ресурсное ограничение на количество сигналов, которые могут быть поставлены в оче-

реди всех процессов конкретного реального ID пользователя. Более подробное описание этого ограничения приводятся в разделе 36.3.

Использование сигналов реального времени

Для того чтобы пара процессов могла отправить и получить сигналы реального времени, стандарт SUSv3 требует следующее.

- Процесс, посылающий сигнал, осуществляет отправку сигнала и сопровождающих этот сигнал данных с помощью системного вызова `sigqueue()`.

  Сигнал реального времени также может быть послан с помощью функций kill(), killpg() и raise(). Однако стандарт SUSv3 оставляет авторам реализации решать, возможна ли постановка в очередь сигналов реального времени, отправляемых с помощью вышеперечисленных интерфейсов. Так, в Linux эти интерфейсы могут использоваться для постановки сигналов в очередь, однако во многих других реализациях UNIX — нет.

- Процесс, получающий сигнал, устанавливает обработчик для этого сигнала с помощью вызова функции `sigaction()`, в которой указывается флаг `SA_SIGINFO`. Этот флаг приводит к инициализации обработчика сигнала с дополнительными аргументами, один из которых включает данные, сопровождающие сигнал.

  В Linux есть возможность поставить в очередь сигнал реального времени, даже если процесс, получающий сигнал, не указал флаг SA_SIGINFO при установке обработчика (однако в данном случае будет невозможно получить данные, связанные с сигналом). Однако стандарт SUSv3 не обязывает реализации гарантировать такое поведение, следовательно, мы не можем полагаться на наличие такой возможности.

22.8.1. Отправка сигналов реального времени

Системный вызов `sigqueue()` посылает сигнал реального времени, указанный в атрибуте `sig`, в процесс, указанный в атрибуте `pid`.

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);
```

Возвращает `0` при успешном завершении, `-1` при ошибке

Для отправки сигнала с помощью функции `sigqueue()` требуются те же разрешения, что и при использовании функции `kill()` (см. раздел 20.5). Можно отправить нулевой сигнал (то есть сигнал `0`) — такая отправка несет тот же смысл, что и вызов `kill()`. (В отличие от функции `kill()` мы не можем вызвать функцию `sigqueue()` для отправки сигнала всей группе процессов, указав отрицательное значение параметра `pid`) (листинг 22.2).

Листинг 22.2. Использование функции sigqueue() для отправки сигнала реального времени
signals/t_sigqueue.c

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    int sig, numSigs, j, sigData;

    union sigval sv;
    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid sig-num data [num-sigs]\n", argv[0]);

    /* Отобразить наши PID и UID, чтобы их можно было сравнить с соответствующими
       полями аргумента siginfo_t, передаваемого обработчику получающего процесса */

    printf("%s: PID is %ld, UID is %ld\n", argv[0],
            (long) getpid(), (long) getuid());

    sig = getInt(argv[2], 0, "sig-num");
    sigData = getInt(argv[3], GN_ANY_BASE, "data");
    numSigs = (argc > 4) ? getInt(argv[4], GN_GT_0, "num-sigs") : 1;

    for (j = 0; j < numSigs; j++) {
        sv.sival_int = sigData + j;
        if (sigqueue(getLong(argv[1], 0, "pid"), sig, sv) == -1)
            errExit("sigqueue %d", j);
    }

    exit(EXIT_SUCCESS);
}
```

signals/t_sigqueue.c

В аргументе `value` указываются данные, сопровождающие сигнал. Аргумент имеет такой вид:

```
union sigval {
    int     sival_int;    /* Целочисленное значение для сопровождающих данных */
    void    *sival_ptr;   /* Указатель для сопровождающих данных */
};
```

Интерпретация этого аргумента зависит от приложения, равно как и выбор, устанавливать ли значение поля `sival_int` или `sival_ptr`. Поле `sival_ptr` редко используется вместе с функцией `sigqueue()`, так как указатель, полученный в одном процессе, редко имеет какой-то смысл в другом. Однако это поле задействуется в других функциях, в которых применяются объединения `sigval`, как мы увидим при рассмотрении таймеров POSIX в разделе 23.6 и очередей сообщений POSIX в разделе 48.6.

> В некоторых реализациях, в том числе Linux, в качестве синонима объединения sigval определяется тип данных sigval_t. Однако этот тип данных не указан в стандарте SUSv3 и может быть недоступен в некоторых реализациях. При создании переносимых приложений следует избегать использования этого типа данных.

Вызов функции `sigqueue()` может завершиться неудачей, если был достигнут предел количества сигналов реального времени в очереди. В этом случае `errno` присваивается значение `EAGAIN`, показывающее, что требуется повторная отправка сигнала (позже, когда хотя бы один из находящихся в очереди сигналов будет доставлен).

Пример использования функции `sigqueue()` приведен в листинге 22.2. Эта программа принимает до четырех аргументов, три первых из которых являются обязательными: идентификатор целевого процесса, номер сигнала, а также целое число, сопровождающее сигнал

реального времени. Если требуется отправка нескольких экземпляров указанного сигнала, то можно воспользоваться четвертым, необязательным, аргументом для указания их количества. В этом случае сопровождающее целое число увеличивается на единицу для каждого последующего сигнала. Мы продемонстрируем выполнение этой программы в подразделе 22.8.2.

22.8.2. Обработка сигналов реального времени

Мы можем обрабатывать сигналы реального времени так же, как и стандартные сигналы: с помощью нормального обработчика с одним аргументом. Кроме того, мы также можем обрабатывать сигналы реального времени с помощью обработчика с тремя аргументами, устанавливаемого посредством флага `SA_SIGINFO` (см. раздел 21.4). Далее приведен пример использования флага `SA_SIGINFO` для установки обработчика шестого сигнала реального времени:

```
struct sigaction act;
sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_RESTART | SA_SIGINFO;
if (sigaction(SIGRTMIN + 5, &act, NULL) == -1)
        errExit("sigaction");
```

При использовании флага `SA_SIGINFO` второй аргумент, передаваемый обработчику сигнала, — это структура `siginfo_t`, содержащая дополнительную информацию о сигнале реального времени. Детальное описание этой структуры приводится в разделе 21.4. Для сигнала реального времени в структуре `siginfo_t` устанавливаются значения следующих полей.

- Поле `si_signo` содержит значение, переданное в первом аргументе обработчика.
- Поле `si_code` обозначает источник сигнала и содержит одно из значений, перечисленных в табл. 21.2. Для сигнала реального времени, отправленного через функцию `sigqueue()`, это поле всегда имеет значение `SI_QUEUE`.
- Поле `si_value` содержит данные, указанные в аргументе `value` (объединение `sigval`) процессом, пославшим сигнал с помощью функции `sigqueue()`.
- Поля `si_pid` и `si_uid` содержат соответственно идентификатор процесса и реальный идентификатор пользователя процесса, отправившего сигнал.

В листинге 22.3 приводится пример обработки сигналов реального времени. Программа перехватывает сигналы и выводит на экран значения различных полей структуры `siginfo_t`, передаваемой ей обработчиком сигнала. Программа принимает два необязательных целочисленных аргумента командной строки.

Если указан первый аргумент, то программа блокирует все сигналы и затем переходит в режим сна на количество секунд, указанное в этом аргументе. В это время мы можем поставить в очередь к процессу несколько сигналов реального времени и пронаблюдать, что произойдет, когда сигналы будут разблокированы. Второй аргумент задает количество секунд, на протяжении которых обработчик сигнала должен оставаться в режиме сна перед возвратом управления. Указание ненулевого значения (значение по умолчанию `1`) может быть полезным для замедления работы программы с тем, чтобы мы могли более четко увидеть, что происходит, когда обрабатываются несколько сигналов.

Мы можем использовать программу из листинга 22.3 вместе с программой из листинга 22.2 (`t_sigqueue.c`) для изучения поведения сигналов реального времени, как показано в следующем журнале сессии оболочки:

```
$ ./catch_rtsigs 60 &
[1] 12842
```

```
$ ./catch_rtsigs: PID is 12842      Приглашение оболочки и вывод программы смешались
./catch_rtsigs: signals blocked - sleeping 60 seconds
Нажмите Enter, чтобы увидеть следующее приглашение оболочки
$ ./t_sigqueue 12842 54 100 3       Отправить сигнал трижды
./t_sigqueue: PID is 12843, UID is 1000
$ ./t_sigqueue 12842 43 200
./t_sigqueue: PID is 12844, UID is 1000
$ ./t_sigqueue 12842 40 300
./t_sigqueue: PID is 12845, UID is 1000
```

Со временем программа `catch_rtsigs` выходит из режима сна и начинает выводить на печать сообщения по мере того, как обработчик перехватывает различные сигналы. (На экране мы видим, что приглашение оболочки смешивается с программным выводом. Это происходит потому, что программа `catch_rtsigs` распечатывает информацию из фонового режима.) Мы видим, что сигналы реального времени с меньшим номером доставляются первыми, а также что структура `siginfo_t`, передаваемая обработчику, содержит PID и UID процесса, пославшего сигнал:

```
$ ./catch_rtsigs: sleep complete
caught signal 40
    si_signo=40,  si_code=-1 (SI_QUEUE), si_value=300
    si_pid=12845, si_uid=1000
caught signal 43
    si_signo=43,  si_code=-1 (SI_QUEUE), si_value=200
    si_pid=12844, si_uid=1000
```

Оставшийся вывод производится тремя экземплярами одного и того же сигнала реального времени. Просмотрев значения поля `si_value`, мы можем увидеть, что эти сигналы были доставлены в том же порядке, в котором они были отправлены:

```
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=101
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=102
    si_pid=12843, si_uid=1000
```

Далее мы воспользуемся командой оболочки `kill` для отправки сигнала в программу `catch_rtsigs`. Как и раньше, мы увидим, что структура `siginfo_t`, получаемая обработчиком, включает идентификатор процесса и идентификатор (имя) пользователя процесса, пославшего сигнал, однако в этом случае значение поля `si_code` — константа `SI_USER`:

```
Нажмите Enter, чтобы увидеть следующее приглашение оболочки
$ echo $$               Отобразить PID оболочки
12780
$ kill -40 12842        Использует kill(2) для отправки сигнала
$ caught signal 40
    si_signo=40,  si_code=0 (SI_USER), si_value=0
    si_pid=12780, si_uid=1000          PID совпадает с идентификатором оболочки
Нажмите Enter, чтобы увидеть следующее приглашение оболочки
$ kill 12842            Завершить catch_rtsigs отправкой SIGTERM
Caught 6 signals
Нажмите Enter, чтобы увидеть оповещение оболочки о завершении фонового задания
[1]+ Done                ./catch_rtsigs 60
```

Листинг 22.3. Обработка сигналов реального времени

signals/catch_rtsigs.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static volatile int handlerSleepTime;
static volatile int sigCnt = 0;      /* Количество полученных сигналов */
static volatile int allDone = 0;
static void            /* Обработчик для сигналов, установленных с SA_SIGINFO */
siginfoHandler(int sig, siginfo_t *si, void *ucontext)
{
    /* НЕБЕЗОПАСНО: В этом обработчике используются функции, небезопасные
       для асинхронных сигналов (printf()); см. подраздел 21.1.2) */

    /* Можно завершить программу с помощью SIGINT или SIGTERM */

    if (sig == SIGINT || sig == SIGTERM) {
        allDone = 1;
        return;
    }

    sigCnt++;
    printf("caught signal %d\n", sig);

    printf("    si_signo=%d, si_code=%d (%s), ", si->si_signo, si->si_code,
           (si->si_code == SI_USER) ? "SI_USER" :
           (si->si_code == SI_QUEUE) ? "SI_QUEUE" : "other");
    printf("si_value=%d\n", si->si_value.sival_int);
    printf("    si_pid=%ld, si_uid=%ld\n",
           (long) si->si_pid, (long) si->si_uid);

    sleep(handlerSleepTime);
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    int sig;
    sigset_t prevMask, blockMask;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [block-time [handler-sleep-time]]\n", argv[0]);

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    handlerSleepTime = (argc > 2) ?
        getInt(argv[2], GN_NONNEG, "handler-sleep-time") : 1;

    /* Установить обработчик для большинства сигналов. Во время выполнения
       обработчика замаскировать все прочие сигналы для предотвращения рекурсивного
       прерывания обработчиков (что сделает вывод неудобочитаемым). */

    sa.sa_sigaction = siginfoHandler;
    sa.sa_flags = SA_SIGINFO;
    sigfillset(&sa.sa_mask);
```

```
    for (sig = 1; sig < NSIG; sig++)
        if (sig != SIGTSTP && sig != SIGQUIT)
            sigaction(sig, &sa, NULL);

    /* Опционально блокировать сигналы и переходить в режим сна, что позволит отправить
       сигналы перед тем, как они будут разблокированы и обработаны */

    if (argc > 1) {
        sigfillset(&blockMask);
        sigdelset(&blockMask, SIGINT);
        sigdelset(&blockMask, SIGTERM);

        if (sigprocmask(SIG_SETMASK, &blockMask, &prevMask) == -1)
errExit("sigprocmask");

        printf("%s: signals blocked - sleeping %s seconds\n",
                    argv[0], argv[1]);
        sleep(getInt(argv[1], GN_GT_0, "block-time"));
        printf("%s: sleep complete\n", argv[0]);

        if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
            errExit("sigprocmask");
        }

        while (!allDone)        /* Ожидать входящих сигналов */
            pause();
        printf("Caught %d signals\n", sigCnt);
        exit(EXIT_SUCCESS);
}
```

signals/catch_rtsigs.c

22.9. Ожидание сигнала с использованием маски: sigsuspend()

Прежде чем перейти к объяснению того, что делает функция `sigsuspend()`, мы разберем ситуацию, при которой нам потребуется воспользоваться этой функцией. Рассмотрим следующий сценарий, с которым разработчики иногда сталкиваются при программировании сигналов.

- Мы временно блокируем сигнал, чтобы его обработчик не прерывал выполнение какого-либо важного участка кода.
- Мы разблокируем сигнал и затем приостанавливаем выполнение программы до тех пор, пока сигнал не будет доставлен.

Чтобы выполнить вышеописанный алгоритм, мы можем попробовать воспользоваться кодом, показанным в листинге 22.4.

Листинг 22.4. Некорректное разблокирование и ожидание сигнала

```
sigset_t prevMask, intMask;
struct sigaction sa;

sigemptyset(&intMask);
sigaddset(&intMask, SIGINT);
```

```
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");

/* Заблокировать SIGINT перед выполнением важного участка.
   (На этом этапе мы предполагаем, что SIGINT еще не блокирован.) */

if (sigprocmask(SIG_BLOCK, &intMask, &prevMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

/* Критический участок: выполнить работу, которая не должна быть
   прервана обработчиком SIGINT */

/* Конец критического участка – восстановить старую маску и разблокировать SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

/* ОШИБКА: что, если SIGINT будет доставлен сейчас... */

pause();         /* Ожидание SIGINT */
```

В коде из листинга 22.4 есть одна проблема. Предположим, что сигнал `SIGINT` доставляется после выполнения второго вызова функции `sigprocmask()`, но перед вызовом функции `pause()`. (На самом деле сигнал мог быть сгенерирован в любой момент времени, когда программа выполняла критический участок кода, но доставлен он будет только после разблокирования). Доставка сигнала `SIGINT` приведет к активации обработчика, но после возврата управления из обработчика и возобновления выполнения основной программы вызов функции `pause()` заблокирует программу до того момента, как будет доставлен *второй* экземпляр сигнала `SIGINT`. А это, в свою очередь, сделает код бессмысленным, так как его предназначение было в том, чтобы разблокировать сигнал `SIGINT` и ждать доставки его *первого* экземпляра.

Хотя вероятность генерации сигнала `SIGINT` в период между началом выполнения критического участка кода (то есть первый вызов функции `sigprocmask()`) и вызовом функции `pause()` мала, в вышеприведенном коде это является программной ошибкой. Эта зависящая от времени ошибка может быть примером состояния гонки (см. раздел 5.1). Как правило, состояние гонки (состязательная ситуация) возникает, если два процесса или потока задействуют общие ресурсы. Однако в данном случае основная программа состязается с собственным обработчиком сигнала.

Чтобы избежать возникновения этой проблемы, от нас требуется использование средств *автоматического* разблокирования сигнала и приостановки процесса. В этом и есть предназначение системного вызова `sigsuspend()`.

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

(Обычно) возвращает `-1` с установкой `errno` значения `EINTR`

Системный вызов `sigsuspend()` заменяет сигнальную маску процесса набором сигналов, на который указывает аргумент `mask`, а затем приостанавливает выполнение процесса

до тех пор, пока не будет перехвачен сигнал и не будет выполнен возврат из обработчика. После того как обработчик возвращает управление в программу, функция `sigsuspend()` восстанавливает сигнальную маску процесса к значению до вызова этой функции.

Вызов функции `sigsuspend()` эквивалентен автоматическому выполнению следующих операций:

```
sigprocmask(SIG_SETMASK, &mask, &prevMask); /* Назначить новую маску */
pause();
sigprocmask(SIG_SETMASK, &prevMask, NULL);  /* Восстановить старую маску */
```

Несмотря на то что восстановление старой маски (то есть последний шаг в вышеприведенной последовательности) может казаться неуместным, он необходим для избегания состояния гонки в ситуациях, когда нам требуется раз за разом ожидать сигнал. В таких ситуациях сигналы должны оставаться заблокированными, кроме промежутков времени, когда выполняется вызов `sigsuspend()`. Если впоследствии нам потребуется разблокировать сигналы, которые мы заблокировали перед вызовом функции `sigsuspend()`, то мы можем воспользоваться еще одним вызовом функции `sigprocmask()`.

Если функция `sigsuspend()` прерывается доставкой сигнала, она возвращает `-1`, а переменной `errno` устанавливается значение `EINTR`. Если же аргумент `mask` указывает на недействительный адрес, то вызов `sigsuspend()` завершается неудачей с ошибкой `EFAULT`.

Пример программы

В листинге 22.5 продемонстрировано использование функции `sigsuspend()`. Эта программа выполняет следующие шаги.

1. Выводит изначальное значение сигнальной маски процесса с помощью функции `printSigMask()` (см. листинг 20.4) ❶.
2. Блокирует сигналы `SIGINT` и `SIGQUIT` и сохраняет исходную сигнальную маску процесса ❷.
3. Устанавливает один обработчик для двух сигналов — `SIGINT` и `SIGQUIT` ❸. Этот обработчик выводит на экран сообщение, и, если обработчик был инициализирован доставкой сигнала `SIGQUIT`, происходит установка значения глобальной переменной `gotSigquit`.
4. Выполняет цикл до тех пор, пока не будет установлено значение переменной `gotSigquit` ❹. Каждая итерация цикла включает следующие шаги:
   1) вывести текущее значение сигнальной маски с помощью нашей функции `printSigMask()`;
   2) симулировать на протяжении нескольких секунд критический участок кода выполнением цикла, потребляющим время ЦП;
   3) вывести маску ожидающих сигналов с помощью нашей функции `printPendingSigs()` (см. листинг 20.4);
   4) использовать функцию `sigsuspend()` для разблокирования сигналов `SIGINT` и `SIGQUIT` и ожидать сигнал (если таковой уже находится в режиме ожидания).
5. Вызывает функцию `sigprocmask()` для восстановления исходного значения сигнальной маски процесса ❺, а затем вывести сигнальную маску с помощью функции `printSigMask()` ❻.

Листинг 22.5. Использование функции sigsuspend()

signals/t_sigsuspend.c

```
#define _GNU_SOURCE       /* Взять объявление strsignal() из <string.h> */
#include <string.h>
#include <signal.h>
```

```
#include <time.h>
#include "signal_functions.h"   /* Объявления printSigMask() и printPendingSigs() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigquit = 0;
static void
handler(int sig)
{
    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
    /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */

    if (sig == SIGQUIT)
        gotSigquit = 1;
}

int
main(int argc, char *argv[])
{
    int loopNum;
    time_t startTime;
    sigset_t origMask, blockMask;
    struct sigaction sa;

❶   printSigMask(stdout, "Initial signal mask is:\n");

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SIGINT);
    sigaddset(&blockMask, SIGQUIT);
❷   if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
        errExit("sigprocmask - SIG_BLOCK");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;

❸   if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGQUIT, &sa, NULL) == -1)
        errExit("sigaction");

❹   for (loopNum = 1; !gotSigquit; loopNum++) {
     printf("=== LOOP %d\n", loopNum);

    /* Симуляция критического участка с помощью задержки в несколько секунд */

    printSigMask(stdout, "Starting critical section, signal mask is:\n");
    for (startTime = time(NULL); time(NULL) < startTime + 4; )
        continue;        /* Запуск в течение нескольких секунд */

    printPendingSigs(stdout,
        "Before sigsuspend() - pending signals:\n");
    if (sigsuspend(&origMask) == -1 && errno != EINTR)
        errExit("sigsuspend");
}

❺   if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
        errExit("sigprocmask - SIG_SETMASK");
```

```
❻ printSigMask(stdout, "=== Exited loop\nRestored signal mask to:\n");

    /* Выполнение других команд... */

    exit(EXIT_SUCCESS);
}
```

signals/t_sigsuspend.c

Следующий журнал сессии оболочки показывает пример того, что мы можем увидеть на экране при запуске программы из листинга 22.5:

```
$ ./t_sigsuspend
Initial signal mask is:
               <empty signal set>
=== LOOP 1
Starting critical section, signal mask is:
               2 (Interrupt)
               3 (Quit)
Нажмите Ctrl+C; SIGINT сгенерирован, но остается в режиме ожидания, так как заблокирован
Before sigsuspend() - pending signals:
               2 (Interrupt)
Caught signal 2 (Interrupt)    Вызов sigsuspend(), сигналы разблокированы
```

Последняя строка вывода появилась на экране, когда программа вызвала функцию `sigsuspend()`, разблокировавшую сигнал `SIGINT`. В этот момент был вызван обработчик, отобразивший эту строку.

Основная программа продолжает цикл:

```
=== LOOP 2
Starting critical section, signal mask is:
               2 (Interrupt)
               3 (Quit)
Нажмите Ctrl+\ для генерации SIGQUIT
Before sigsuspend() - pending signals:
               3 (Quit)
Caught signal 3 (Quit)         Вызов sigsuspend(), сигналы разблокированы
=== Exited loop                Обработчик сигнала установил gotSigquit
Restored signal mask to:
        <empty signal set>
```

В этот раз мы нажали сочетание выхода Ctrl+\, заставившее обработчик сигнала установить флаг `gotSigquit`, который, в свою очередь, заставил программу завершить цикл.

22.10. Синхронное ожидание сигнала

В разделе 22.9 мы увидели, как использовать обработчик сигнала вместе с функцией `sigsuspend()` для приостановки выполнения процесса до того, как будет доставлен сигнал. Однако необходимость написания обработчика сигнала и обработки сложностей асинхронной доставки делает некоторые приложения громоздкими. Вместо этого мы можем использовать системный вызов `sigwaitinfo()` для реализации синхронного *приема* сигнала.

Функция `sigwaitinfo()` приостанавливает выполнение процесса до тех пор, пока из набора, указываемого аргументом `set`, не будет ожидать по крайней мере один сигнал. Если в наборе `set` один сигнал уже находится в режиме ожидания во время вызова, тогда функция `sigwaitinfo()` выполняет возврат незамедлительно. Этот сигнал удаляется из списка ожидания, а его номер возвращается как результат выполнения функции. Если

значение `info` не равно `NULL`, то данный аргумент указывает на структуру `siginfo_t`, инициализацируемую для хранения той же самой информации, что и предоставляется обработчику сигнала через аргумент `siginfo_t` (см. раздел 21.4).

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

Возвращает номер доставленного сигнала при успешном завершении и `-1` при ошибке

Очередность доставки и характеристики очереди сигналов, принимаемых функцией `sigwaitinfo()`, аналогичны свойствам сигналов, перехватываемых обработчиками. Иными словами, стандартные сигналы не ставятся в очередь, тогда как сигналы реального времени ставятся в очередь и доставляются по возрастанию номера сигнала.

Ожидание сигналов с использованием функции `sigwaitinfo()` не только избавляет от лишнего кода при написании обработчиков сигналов, но и является несколько более быстрым решением по сравнению с сочетанием «обработчик сигнала + функция `sigsuspend()`» (см. упражнение 22.3).

Обычно имеет смысл использовать функцию `sigwaitinfo()` только в сочетании с блокированием сигналов, в ожидании которых мы заинтересованы. (Мы можем получить ожидающий сигнал с помощью функции `sigwaitinfo()` даже в то время, когда он заблокирован). Если мы не сможем этого сделать и сигнал будет доставлен в процесс перед первым или между двумя успешными вызовами функции `sigwaitinfo()`, то такой сигнал будет обработан в соответствии с его текущей диспозицией.

Согласно стандарту SUSv3 вызов `sigwaitinfo()` без блокировки сигналов, указанных в аргументе `set`, приводит к неопределенному поведению. Пример использования функции `sigwaitinfo()` приведен в листинге 22.6. Эта программа сначала блокирует все сигналы, а затем делает отсрочку на количество секунд, указанное в необязательном аргументе командной строки. Это позволяет отправлять сигналы в программу перед вызовом `sigwaitinfo()`. Затем программа выполняет цикл с `sigwaitinfo()` до тех пор, пока не получит сигнал `SIGINT` или `SIGTERM`.

Следующий журнал сессии оболочки демонстрирует выполнение программы из листинга 22.6. Мы запускаем программу в фоновом режиме, указываем, что она должна сделать отсрочку на 60 секунд перед вызовом `sigwaitinfo()`, а затем отправляем в нее два сигнала:

```
$ ./t_sigwaitinfo 60 &
./t_sigwaitinfo: PID is 3837
./t_sigwaitinfo: signals blocked
./t_sigwaitinfo: about to delay 60 seconds
[1] 3837
$ ./t_sigqueue 3837 43 100                Отправить сигнал 43
./t_sigqueue: PID is 3839, UID is 1000
$ ./t_sigqueue 3837 42 200                Отправить сигнал 42
./t_sigqueue: PID is 3840, UID is 1000
```

Со временем программа завершает период сна — и цикл с функцией `sigwaitinfo()` принимает поставленные в очередь сигналы. (На экране приглашения оболочки перемешаны с выводом программы, так как программа `t_sigwaitinfo` распечатывает вывод из фонового режима). Как и в случае с сигналами реального времени, перехваченными обработчиком, мы видим, что сигналы доставляются в порядке увеличения их номера, а также

что структура `siginfo_t`, передаваемая обработчику сигнала, позволяет нам получить идентификатор процесса и идентификатор пользователя процесса, пославшего сигнал:

```
$ ./t_sigwaitinfo: finished delay
got signal: 42
    si_signo=42, si_code=-1 (SI_QUEUE), si_value=200
    si_pid=3840, si_uid=1000
got signal: 43
    si_signo=43, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=3839, si_uid=1000
```

Мы продолжаем, используя команду оболочки `kill` для отправки сигнала в процесс. В этот раз мы увидим, что полю `si_code` присвоено значение `SI_USER` (вместо `SI_QUEUE`):

```
Нажмите Enter, чтобы увидеть следующее приглашение оболочки
$ echo $$                       Отобразить PID оболочки
3744
$ kill -USR1 3837               Оболочка посылает SIGUSR1 с помощью kill()
$ got signal: 10                Доставка SIGUSR1
    si_signo=10, si_code=0 (SI_USER), si_value=100
    si_pid=3744, si_uid=1000    3744 – это PID оболочки
Нажмите Enter, чтобы увидеть следующее приглашение оболочки
$ kill %1                       Завершить программу с помощью SIGTERM
$
Нажмите Enter, чтобы увидеть уведомление о завершении фоновой задачи
[1]+ Done               ./t_sigwaitinfo 60
```

В выводе для принятого сигнала `SIGUSR1` мы видим, что полю `si_value` присвоено значение `100`. Это то значение, с каким данное поле было инициализировано предшествующим сигналом, отправленным с помощью функции `sigqueue()`. Ранее отмечалось, что поле `si_value` содержит валидную информацию только для сигналов, посланных с помощью функции `sigqueue()`.

Листинг 22.6. Синхронное ожидание сигналов с sigwaitinfo()

signals/t_sigwaitinfo.c

```
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include <time.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int sig;
    siginfo_t si;
    sigset_t allSigs;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [delay-secs]\n", argv[0]);
    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    /* Блокировать все сигналы (исключая SIGKILL и SIGSTOP) */
    sigfillset(&allSigs);
    if (sigprocmask(SIG_SETMASK, &allSigs, NULL) == -1)
        errExit("sigprocmask");
    printf("%s: signals blocked\n", argv[0]);
    if (argc > 1) {
```

```
        /* Пауза для того, чтобы сигналы могли быть отправлены нам */
        printf("%s: about to delay %s seconds\n", argv[0], argv[1]);
        sleep(getInt(argv[1], GN_GT_0, "delay-secs"));
        printf("%s: finished delay\n", argv[0]);
    }

    for (;;) {    /* Получать сигналы до SIGINT (^C) или SIGTERM */
        sig = sigwaitinfo(&allSigs, &si);
    if (sig == -1)
        errExit("sigwaitinfo");
    if (sig == SIGINT || sig == SIGTERM)
        exit(EXIT_SUCCESS);

    printf("got signal: %d (%s)\n", sig, strsignal(sig));
    printf("    si_signo=%d, si_code=%d (%s), si_value=%d\n",
            si.si_signo, si.si_code,
            (si.si_code == SI_USER) ? "SI_USER" :
              (si.si_code == SI_QUEUE) ? "SI_QUEUE" : "other",
            si.si_value.sival_int);
    printf("    si_pid=%ld, si_uid=%ld\n",
           (long) si.si_pid, (long) si.si_uid);
    }
}
```

signals/t_sigwaitinfo.c

Системный вызов `sigtimedwait()` — это вариация функции `sigwaitinfo()`. Единственное отличие заключается в том, что функция `sigtimedwait()` позволяет ограничить время ожидания.

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
```

Возвращает номер доставленного сигнала при успешном завершении или `-1` при ошибке или тайм-ауте (`EAGAIN`)

Аргумент `timeout` устанавливает максимальное количество времени, на протяжении которого функция `sigtimedwait()` будет ожидать сигнал. Этот аргумент — это указатель на структуру следующего типа:

```
struct timespec {
time_t tv_sec;        /* Секунды ('time_t' целочисленный тип) */
    long tv_nsec;     /* Наносекунды */
};
```

Поля структуры `timespec` заполняются для указания максимального количества секунд и наносекунд, на протяжение которых функция `sigtimedwait()` будет ожидать сигнал. Присвоение обоим полям структуры значения `0` приводит к незамедлительному тайм-ауту, иными словами — к опросу на предмет того, что хотя бы один из указанных сигналов находится в режиме ожидания. Если вызов завершается с тайм-аутом, при этом ни один сигнал так и не был доставлен, то функция `sigtimedwait()` возвращает ошибку `EAGAIN`.

Если аргумент `timeout` равен `NULL`, функция `sigtimedwait()` является абсолютным эквивалентом `sigwaitinfo()`. В SUSv3 значение `NULL` аргумента `timeout` не прописано, и в некоторых реализациях UNIX данное значение интерпретируется как запрос на проведение опроса, незамедлительно возвращающего результат.

22.11. Получение сигналов через файловый дескриптор

Начиная с версии ядра 2.6.22, в Linux предоставляется (нестандартный) системный вызов `signalfd()`, создающий специальный файловый дескриптор, из которого можно прочитать сигналы, направляемые в вызывающий участок кода. Механизм `signalfd` предоставляет альтернативу функции `sigwaitinfo()` для синхронного приема сигналов.

```
#include <sys/signalfd.h>

int signalfd(int fd, const sigset_t *mask, int flags);
```

Возвращает файловый дескриптор при успешном завершении или `-1` при ошибке

Аргумент `mask` — это набор сигналов, в котором перечислены те сигналы, которые мы хотим иметь возможность прочитать через файловый дескриптор `signalfd`. Как и в случае с функцией `sigwaitinfo()`, как правило, нам следует также заблокировать все сигналы, перечисленные в `mask`, с помощью функции `sigprocmask()`, чтобы они не обрабатывались в соответствии с их диспозициями по умолчанию прежде, чем мы сможем их прочитать.

Если аргумент `fd` указан как `-1`, то функция `signalfd()` создает новый файловый дескриптор, который может использоваться для чтения сигналов в `mask`. В противном случае она изменяет маску, связанную с `fd`, значением которого должен быть файловый дескриптор, созданный предшествующим вызовом `signalfd()`.

В изначальной реализации аргумент `flags` был зарезервирован для будущего и его необходимо было указывать как `0`. Однако, начиная с версии Linux 2.6.27, функция стала поддерживать два флага:

- `SFD_CLOEXEC` — установить флаг «закрыть при выходе» (`FD_CLOEXEC`) для нового файлового дескриптора. Этот флаг полезен по тем же причинам, что и флаг `O_CLOEXEC` вызова `open()`, описанный в разделе 4.3.1;
- `SFD_NONBLOCK` — установить флаг `O_NONBLOCK` для соответствующего файлового дескриптора. Таким образом, дальнейшие чтения будут неблокирующими.

Создав файловый дескриптор, мы можем затем считывать из него сигналы с помощью функции `read()`. Буфер, переданный функции `read()`, должен быть достаточно большим для хранения хотя бы одной структуры `signalfd_siginfo`, определенной следующим образом в файле `<sys/signalfd.h>`:

```
struct signalfd_siginfo {
    uint32_t ssi_signo;    /* Номер сигнала */
    int32_t ssi_errno;     /* Номер ошибки (обычно не используется) */
    int32_t ssi_code;      /* Код сигнала */
    uint32_t ssi_pid;      /* ID процесса-отправителя */
    uint32_t ssi_uid;      /* Реальный ID пользователя процесса-отправителя */
    int32_t ssi_fd;        /* Файловый дескриптор (SIGPOLL/SIGIO) */
    uint32_t ssi_tid;      /* ID внутреннего таймера ядра (таймеры POSIX) */
    uint32_t ssi_band;     /* Связывающее событие (SIGPOLL/SIGIO) */
    uint32_t ssi_overrun;  /* Счетчик превышений таймера (таймеры POSIX)*/
    uint32_t ssi_trapno;   /* Номер ловушки */
    int32_t ssi_status;    /* Код завершения или сигнал (SIGCHILD) */
    int32_t ssi_int;       /* Целое число, отправленное функцией sigqueue() */
```

```
    uint64_t ssi_ptr;      /* Указатель, отправленный функцией sigqueue() */
    uint64_t ssi_utime;    /* Пользовательское время ЦП (SIGCHLD) */
    uint64_t ssi_stime;    /* Системное время ЦП (SIGCHLD) */
    uint64_t ssi_addr;     /* Адрес, сгенерировавший сигнал
                              (только аппаратно генерируемые сигналы) */
};
```

Поля этой структуры возвращают ту же информацию, что и поля традиционной структуры `siginfo_t` с похожими именами (см. раздел 21.4).

Каждый вызов функции `read()` возвращает столько экземпляров структуры `signalfd_siginfo`, сколько ожидающих сигналов поместится в предоставленный буфер. Если во время вызова функции нет ожидающих сигналов, функция `read()` блокируется до поступления сигнала. Мы также можем воспользоваться операцией `fcntl() F_SETFL` (см. раздел 5.3), чтобы установить флаг `O_NONBLOCK` для файлового дескриптора. Таким образом, операции чтения будут неблокирующими и будут возвращать ошибку `EAGAIN` при отсутствии сигналов в режиме ожидания.

При чтении сигнала из файлового дескриптора `signalfld`, он считается потребленным и выходит из режима ожидания процесса.

Листинг 22.7. Использование signalfld() для чтения сигналов

signals/singalfd_sigval.c

```
#include <sys/signalfd.h>
#include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    sigset_t mask;
    int sfd, j;
    struct signalfd_siginfo fdsi;
    ssize_t s;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sig-num...\n", argv[0]);

    printf("%s: PID = %ld\n", argv[0], (long) getpid());

    sigemptyset(&mask);
    for (j = 1; j < argc; j++)
        sigaddset(&mask, atoi(argv[j]));

    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        errExit("sigprocmask");

    sfd = signalfd(-1, &mask, 0);
    if (sfd == -1)
        errExit("signalfd");

    for (;;) {
        s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
        if (s != sizeof(struct signalfd_siginfo))
            errExit("read");

        printf("%s: got signal %d", argv[0], fdsi.ssi_signo);
        if (fdsi.ssi_code == SI_QUEUE) {
```

```
            printf("; ssi_pid = %d; ", fdsi.ssi_pid);
            printf("ssi_int = %d", fdsi.ssi_int);
        }
        printf("\n");
    }
}
```

signals/singalfd_sigval.c

За файловым дескриптором `signalfd` можно осуществлять мониторинг так же, как и за остальными дескрипторами, с помощью функций `select()`, `poll()` и `epoll()` (описаны в главе 59). Кроме прочего, этот механизм также предоставляет альтернативу трюку с зацикленным каналом, описанному в подразделе 59.5.2. Если сигналы находятся в режиме ожидания, то эти техники помечают файловый дескриптор как доступный для чтения.

Когда дескриптор `signalfd` больше не нужен, следует закрыть его, чтобы освободить соответствующий ресурс ядра.

Использование функции `signalfd()` продемонстрировано в листинге 22.7. Эта программа создает маску из номеров сигналов, указанных в качестве аргументов командой строки, блокирует эти сигналы, а затем создает файловый дескриптор `signalfd` для чтения этих сигналов. Затем программа переходит в цикл, считывая сигналы из файлового дескриптора, и выводит некоторую информацию из возвращенной структуры `signalfd_siginfo`. В следующей сессии оболочки мы запустили программу из листинга 22.7 в фоновом режиме, а затем отправили ей сигнал реального времени с сопровождающими данными с помощью программы из листинга 22.2 (`t_sigqueue.c`):

```
$ ./signalfd_sigval 44 &
./signalfd_sigval: PID = 6267
[1] 6267
$ ./t_sigqueue 6267 44 123     Отправить сигнал 44 с данными 123 в PID 6267
./t_sigqueue: PID is 6269, UID is 1000
./signalfd_sigval: got signal 44; ssi_pid=6269; ssi_int=123
$ kill %1                      Завершить программу, запущенную в фоновом режиме
```

22.12. Межпроцессное взаимодействие посредством сигналов

С одной точки зрения мы можем рассматривать сигналы как форму межпроцессного взаимодействия (IPC). Однако в таком виде сигналы страдают от большого количества ограничений. Во-первых, по сравнению с другими методами взаимодействия процессов, которые мы рассматриваем в следующих главах, программирование на сигналах сложно и громоздко. Этому есть следующие причины.

- Асинхронная природа сигналов означает, что появляется вероятность столкновения с проблемами, в том числе с требованиями реентерабельности, состоянием гонки, а также с проблемой корректной обработки глобальных переменных из обработчиков сигналов. (Вероятность возникновения большинства этих проблем снимается при использовании функций `sigwaitinfo()` и `signalfd()` для синхронного получения сигналов.)
- Стандартные сигналы не ставятся в очередь. Даже для сигналов реального времени существуют пределы по количеству сигналов, которые могут быть поставлены в очередь. Это значит, что во избежание потери информации процесс, получающий сигналы, должен иметь метод информирования отправителя о том, что он (получатель) готов

к приему следующего сигнала. Самый очевидный способ решения этой проблемы — отправка получателем сигнала отправителю.

Еще одна, более сложная, проблема заключается в том, что сигналы могут передавать только ограниченный объем информации: номер сигнала и — в случае с сигналами реального времени — слово (целое число или указатель) дополнительных данных. Эта низкая пропускная способность делает сигналы медленными по сравнению с другими методами IPC, например такими, как каналы.

По причине вышеперечисленных ограничений сигналы редко используются для реализации межпроцессного взаимодействия.

22.13. Ранние API сигналов

В нашем обсуждении сигналов мы сосредоточились на API сигналов POSIX. Сейчас настало время коротко обсудить исторические API, предоставлявшиеся в системе BSD. Несмотря на то что во всех современных приложениях должны использоваться POSIX-интерфейсы, мы можем встретиться с этими устаревшими версиями при переносе (как правило, старых) приложений из других реализаций UNIX. Так как Linux предоставляет (как и многие другие реализации UNIX) совместимость с BSD, зачастую все, что нам нужно для переноса старых интерфейсов, — это перекомпилировать их под Linux.

В основе API сигналов POSIX лежат интерфейсы 4.2BSD, поэтому функции POSIX — это прямые аналоги функций BSD.

Я приведу прототипы функций в интерфейсах BSD, а затем вкратце объясню работу каждой функции. Более подробная информация представлена на страницах справочника.

```
#define _BSD_SOURCE
#include <signal.h>

int sigvec(int sig, const struct sigvec *vec, struct sigvec *ovec);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `sigvec()` — аналог функции `sigaction()`. Аргументы `vec` и `ovec` — это указатели на структуры следующего типа:

```
struct sigvec {
    void    (*sv_handler)(int);
    int     sv_mask;
    int     sv_flags;
};
```

Поля структуры `sigvec` соответствуют полям структуры `sigaction`. Первое заметное отличие в том, что поле `sv_mask` раньше было целочисленным, а не типа `sigset_t`. Это означает, что на 32-битных архитектурах максимально можно было реализовать 31 сигнал. Еще одно отличие — в использовании флага `SV_INTERRUPT` в поле `sv_flags` (аналог `sa_flags`). Поскольку перезапуск системных вызовов был настройкой по умолчанию в 4.2BSD, установка этого флага означала, что медленные системные вызовы должны прерываться обработчиками сигналов. (Это кардинальным образом отличается от функций POSIX API, в случае использования которых мы должны явно устанавливать флаг `SA_RESTART` для включения перезапуска системных вызовов при установке обработчика сигнала с помощью функции `sigaction()`.)

```
#define _BSD_SOURCE
#include <signal.h>

int sigblock(int mask);
int sigsetmask(int mask);
```

Обе функции возвращают предыдущую сигнальную маску

```
int sigpause(int sigmask);
```

Всегда возвращает -1 с установкой errno значения EINTR

```
int sigmask(int sig);
```

Возвращает значение сигнальной маски с битовым набором *sig*

Функция `sigblock()` добавляет набор сигналов в сигнальную маску процесса. Эта функция аналогична операции `sigprocmask() SIG_BLOCK`. Вызов `sigsetmask()` устанавливает абсолютное значение сигнальной маски. Эта функция аналогична операции `sigprocmask() SIG_SETMASK`.

Функция `sigpause()` аналогична функции `sigsuspend()`. В GNU библиотеке C по умолчанию предоставляется версия функции из системы System V, если мы не укажем макрос проверки возможностей `_BSD_SOURCE` при компиляции программы.

Макрос `sigmask()` преобразует номер сигнала в соответствующее значение 32-битной маски. Такие битовые маски могут быть объединены вместе с помощью логического оператора ИЛИ для создания набора сигналов, как показано в следующем примере:

```
sigblock(sigmask(SIGINT) | sigmask(SIGQUIT));
```

22.14. Резюме

Некоторые сигналы приводят к завершению процесса с созданием файла дампа ядра. Этот файл содержит информацию о состоянии процесса на момент завершения, которая может использоваться отладчиком. По умолчанию имя файла дампа ядра — `core`, но в Linux предоставляется файл `/proc/sys/kernel/core_pattern` для управления именованием файлов дампа.

Сигнал может быть сгенерирован синхронно и асинхронно. Асинхронная генерация происходит, когда сигнал отправляется процессу ядром или другим процессом. Процесс не может точно предсказать, когда в него будет доставлен сигнал, сгенерированный асинхронно. (Уже отмечалось, что, как правило, асинхронные сигналы доставляются в следующий раз, когда процесс-получатель переключается из режима ядра в режим пользователя.) Синхронная генерация происходит, когда сам процесс выполняет код, напрямую генерирующий сигнал, например, при выполнении кода, вызывающего аппаратное исключение или при вызове функции `raise()`. Доставку сигнала, генерируемого синхронно, можно предсказать с высокой точностью (она происходит мгновенно).

Сигналы реального времени — надстройка POSIX на исходную модель сигналов. Они отличаются от стандартных тем, что могут быть поставлены в очередь, имеют определенную очередность доставки, а также тем, что могут быть отправлены с сопровождающими данными. Сигналы реального времени предназначены для использования в целях, определяемых приложением. Отправка сигнала реального времени происходит с помощью системного вызова `sigqueue()`, а через дополнительный аргумент (структура `siginfo_t`), переданный обработчику сигнала, можно получить данные, сопровождающие сигнал, а также идентификатор процесса и реальный идентификатор пользователя процесса-отправителя.

Системный вызов `sigsuspend()` позволяет программе автоматически изменять сигнальную маску процесса и приостанавливать выполнение до прибытия сигнала. Атомарность функции `sigsuspend()` является ее неотъемлемой характеристикой, позволяющей избежать возникновения состояний гонки при разблокировании сигнала и последующей приостановки выполнения до прибытия сигнала.

Как и функции `sigwaitinfo()` и `sigtimedwait()`, специфичный системный вызов Linux `signalfd()` может использоваться для синхронного ожидания сигнала. Отличительной особенностью этого интерфейса является то, что сигналы могут быть прочитаны через файловый дескриптор. Над файловым дескриптором можно осуществлять мониторинг с помощью функций `select()`, `poll()` и `epoll()`.

Хотя сигналы могут рассматриваться в качестве метода взаимодействия процессов (IPC), по причине многих факторов сигналы не подходят для этих целей: из-за их асинхронной природы, из-за того, что сигналы не ставятся в очередь, и из-за их низкой пропускной способности. Чаще всего, сигналы используются для синхронизации процессов и для других целей (например, оповещение о событии, управление заданиями, истечение таймера).

Сигналы играют важную роль в различных частях API системных вызовов, и мы еще вернемся к ним в последующих главах. Кроме того, некоторые функции, связанные с сигналами, также характерны для потоков (например, `pthread_kill()` и `pthread_sigmask()`), но мы отложим обсуждение этих функций до раздела 33.2.

Дополнительная информация

См. ресурсы, перечисленные в разделе 20.15.

22.15. Упражнения

22.1. В разделе 22.2 отмечалось, что если остановленный процесс, установивший обработчик для заблокированного сигнала `SIGCONT`, в дальнейшем возобновляется в результате получения сигнала `SIGCONT`, то обработчик активируется только после разблокирования сигнала `SIGCONT`. Напишите программу, демонстрирующую это поведение. Не забудьте, что процесс может быть остановлен путем ввода в терминал символа *приостановки* (обычно Ctrl+Z), а сигнал `SIGCONT` может быть отправлен в процесс с помощью команды `kill -CONT` (или неявно, с помощью команды оболочки `fg`).

22.2. Если сигнал реального времени и стандартный сигнал ожидают процесс, стандарт SUSv3 не устанавливает, который из сигналов должен быть доставлен первым. Напишите программу, демонстрирующую, что делает Linux в этом случае. (С помощью программы установите обработчик для всех сигналов, заблокируйте сигналы на определенное время, чтобы вы могли отправить в программу несколько сигналов, а затем разблокируйте все сигналы.)

22.3. В разделе 22.10 сказано, что прием сигналов с помощью функции `sigwaitinfo()` осуществляется быстрее по сравнению с использованием обработчика сигнала с функцией `sigsuspend()`. В программе `signals/sig_speed_sigsuspend.c`, поставляемой с исходным кодом для этой книги, задействуется функция `sigsuspend()` для реализации обмена сигналами между родительским и дочерним процессами. Засеките время выполнения этой программы при обмене одним миллионом сигналов между двумя процессами. (Количество сигналов, которыми необходимо обменяться, указывается в аргументе командной строки программы.) Создайте модифицированную версию программы, в которой будет использоваться функция `sigwaitinfo()`, и засеките время выполнения этой версии программы. Какова разница в скоростях выполнения этих программ?

23 Таймеры и переход в режим сна

Таймер позволяет процессу планировать появление уведомлений, которые должны поступать в будущем. Переход в режим сна дает возможность процессу (или потоку) приостанавливать выполнение на определенный период времени. В данной главе описываются интерфейсы для этих механизмов. Мы рассмотрим следующие темы:

- классические программные интерфейсы UNIX для установки интервальных таймеров (`setitimer()` и `alarm()`), срабатывание которых приводит к уведомлению процесса;
- программные интерфейсы, которые позволяют процессу приостанавливать выполнение на определенном отрезке времени;
- программные интерфейсы системных часов и таймеров, входящие в стандарт POSIX.1b;
- механизм `timerfd`, доступный только в Linux, который позволяет создавать таймеры, чье срабатывание можно определить по файловому дескриптору.

23.1. Интервальные таймеры

Системный вызов `setitimer()` устанавливает *интервальный таймер*, который срабатывает в какой-то момент времени (возможно, многократно и с определенным интервалом).

```
#include <sys/time.h>

int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

С помощью вызова `setitimer()` процесс может устанавливать таймеры трех разных видов, в зависимости от выбранного значения `which`:

- `ITIMER_REAL` — создает таймер, который ведет обратный отсчет в реальном времени (то есть так, как на обычных настенных часах). По истечении времени процессу передается сигнал `SIGALRM`;
- `ITIMER_VIRTUAL` — создает таймер, который ведет обратный отсчет в виртуальном времени процесса (то есть учитывается процессорное время в пользовательском режиме). По истечении времени процессу передается сигнал `SIGVTALRM`;
- `ITIMER_PROF` — создает профилирующий таймер, который ведет обратный отсчет с учетом времени процесса (суммируя процессорное время в пользовательском режиме и в режиме ядра). По истечении времени процессу передается сигнал `SIGPROF`.

Стандартное действие всех сигналов, которые генерируются таймерами, состоит в завершении процесса. Если нам нужно сделать что-то другое, мы должны установить собственный обработчик сигнала.

Аргументы `new_value` и `old_value` представляют собой указатели на структуры `itimerval`, определенные следующим образом:

```
struct itimerval {
    struct timeval it_interval;    /* Временной отрезок для интервального таймера */
    struct timeval it_value;       /* Текущее значение (время
                                      до следующего срабатывания) */
};
```

Каждое из полей структуры `itimerval` является, в свою очередь, структурой типа `timeval`, содержащей секунды и микросекунды:

```
struct timeval {
    time_t      tv_sec;            /* Секунды */
    suseconds_t tv_usec;           /* Микросекунды (long int) */
};
```

Тип `it_value` аргумента `new_value` определяет время до срабатывания таймера. Структура `it_interval` определяет, является ли таймер периодическим. Если оба поля структуры `it_interval` равны `0`, таймер срабатывает в момент времени `it_value` и делает это только один раз. Если одно или оба поля `it_interval` содержат ненулевое значение, после срабатывания таймер будет сбрасываться и снова отсчитывать указанный интервал.

Процессу доступно по одному экземпляру каждого из этих трех таймеров. Делая повторный вызов `setitimer()`, мы изменяем характеристики существующего таймера, заданного в аргументе `which`. Если оба поля `new_value.it_value` в вызове `setitimer()` равны `0`, любой имеющийся таймер отключается.

Если аргумент `old_value` не равен `NULL`, он должен указывать на структуру `itimerval`, которая используется для возвращения предыдущего значения таймера. Если оба поля структуры `old_value.it_value` равны `0`, это означает, что перед этим таймер был выключен. Если оба поля структуры `old_value.it_interval` равны `0`, предыдущий таймер был установлен для одиночного срабатывания в момент времени `old_value.it_value`. Предыдущие параметры таймера могут пригодиться, если мы намерены восстановить их после срабатывания нового таймера. Если нам не нужны эти значения, мы можем присвоить `NULL` аргументу `old_value`.

Таймер отсчитывает время в обратном направлении от начального значения (`it_value`) до 0. По истечении времени процессу отправляется соответствующий сигнал; после этого, если указан ненулевой интервал (`it_interval`), значение таймера возвращается к исходному (`it_value`), а отсчет повторяется.

В любой момент времени мы можем воспользоваться вызовом `getitimer()`, чтобы получить текущее состояние таймера и узнать, сколько времени осталось до следующего срабатывания.

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Системный вызов `getitimer()` возвращает текущее состояние таймера типа `which` в виде буфера, на который указывает аргумент `curr_value`. Это точно такая же информация, которая возвращается с помощью аргумента `old_value` вызова `setitimer()`; разница лишь в том, что для ее получения нам не нужно изменять параметры таймера. Вложенная структура `curr_value.it_value` возвращает количество времени, оставшееся до следующего срабатывания таймера. Это значение изменяется по мере обратного отсчета и сбрасывается во время срабатывания таймера, если при его установке структура `it_interval` не была равна `0`. Вложенная структура `curr_value.it_interval` возвращает интервал таймера; это значение остается неизменным до следующего вызова `setitimer()`.

Таймеры, установленные с помощью вызова `setitimer()` (и `alarm()`, который мы обсудим чуть ниже), действуют на протяжении всей работы функции `exec()`, но не наследуются дочерним процессом в результате вызова `fork()`.

В стандарте SUSv4 вызовы getitimer() и setitimer() помечены устаревшими; вместо них рекомендуется использовать программный интерфейс POSIX-таймеров (см. раздел 23.6).

Пример программы

Применение вызовов `getitimer()` и `setitimer()` демонстрируется в листинге 23.1. Данная программа выполняет следующие шаги.

- Устанавливает обработчик сигнала `SIGALRM` (3).
- Инициализирует поля со значением и интервалом для реального таймера (`ITIMER_REAL`), используя параметры, указанные в виде аргументов командной строки (4). Если эти аргументы отсутствуют, программа устанавливает таймер, который срабатывает лишь единожды, после 2 секунд.
- Входит в бесконечный цикл (5), потребляя процессорное время и периодически вызывая функцию `displayTimes()` (1), которая выводит реальное время, прошедшее с момента запуска программы, а также текущее состояние таймера `ITIMER_REAL`.

При каждом срабатывании таймера вызывается обработчик сигнала `SIGALRM`, который устанавливает глобальный флаг `gotAlarm` (2). Каждый раз, когда этот флаг устанавливается, цикл в главной программе вызывает функцию `displayTimes()`, чтобы вывести время вызова обработчика и текущее состояние таймера (6) (обработчик спроектирован таким образом, чтобы избежать вызова функций, несовместимых с асинхронными сигналами; причины описаны в подразделе 21.1.2). Если таймер имеет нулевой интервал, при получении сигнала программа завершается; в противном случае для ее завершения должно быть перехвачено три сигнала (7).

Запустив программу из листинга 23.1, мы увидим следующее:

```
$ ./real_timer 1 800000 1 0       Начальное значение равно 1,8 секунды,
                                  интервал равен 1 секунде
 Elapsed Value Interval
START:    0.00
Main:     0.50    1.30    1.00    Таймер ведет обратный отсчет, пока не достигнет 0
Main:     1.00    0.80    1.00
Main:     1.50    0.30    1.00
ALARM:    1.80    1.00    1.00    В момент срабатывания таймер
                                  засекает время, равное интервалу
Main:     2.00    0.80    1.00
Main:     2.50    0.30    1.00
ALARM:    2.80    1.00    1.00
Main:     3.00    0.80    1.00
Main:     3.50    0.30    1.00
ALARM:    3.80    1.00    1.00
That's all folks
```

Листинг 23.1. Использование таймера реального времени

timers/real_timer.c

```
#include <signal.h>
#include <sys/time.h>
#include <time.h>
#include "tlpi_hdr.h"
static volatile sig_atomic_t gotAlarm = 0;
    /* Устанавливаем ненулевое значение при получении SIGALRM */
```

```
/* Получаем и выводим реальное время, а также (если 'includeTimer'
   равно true) текущее значение и интервал таймера ITIMER_REAL */

static void
1 displayTimes(const char *msg, Boolean includeTimer)
{
    struct itimerval itv;
    static struct timeval start;
    struct timeval curr;
    static int callNum = 0;  /* Количество вызовов данной функции */

    if (callNum == 0)        /* Инициализируем счетчик прошедшего времени */
        if (gettimeofday(&start, NULL) == -1)
            errExit("gettimeofday");

    if (callNum % 20 == 0)    /* Выводим заголовок через каждые 20 строк */
        printf("        Elapsed        Value Interval\n");
    if (gettimeofday(&curr, NULL) == -1)
        errExit("gettimeofday");
    printf("%-7s %6.2f", msg, curr.tv_sec - start.tv_sec +
                    (curr.tv_usec - start.tv_usec) / 1000000.0);

    if (includeTimer) {
        if (getitimer(ITIMER_REAL, &itv) == -1)
            errExit("getitimer");
        printf("    %6.2f    %6.2f",
        itv.it_value.tv_sec + itv.it_value.tv_usec / 1000000.0,
        itv.it_interval.tv_sec + itv.it_interval.tv_usec / 1000000.0);
    }

    printf("\n");
    callNum++;
}

static void
sigalrmHandler(int sig)
{
2 gotAlarm = 1;
}

int
main(int argc, char *argv[])
{
    struct itimerval itv;
    clock_t prevClock;
    int maxSigs;         /* Количество сигналов, которые нужно перехватить до выхода */
    int sigCnt;          /* Количество уже перехваченных сигналов */
    struct sigaction sa;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [secs [usecs [int-secs [int-usecs]]]]\n", argv[0]);

    sigCnt = 0;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigalrmHandler;
3 if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");
```

```
    /* Выходим после трех сигналов или при первом срабатывании, если интервал равен 0 */
    maxSigs = (itv.it_interval.tv_sec == 0 &&
    itv.it_interval.tv_usec == 0) ? 1 : 3;
    displayTimes("START:", FALSE);

    /* Устанавливаем таймер с помощью аргументов командной строки */
    itv.it_value.tv_sec = (argc > 1) ? getLong(argv[1], 0, "secs") : 2;
    itv.it_value.tv_usec = (argc > 2) ? getLong(argv[2], 0, "usecs") : 0;
    itv.it_interval.tv_sec = (argc > 3) ? getLong(argv[3], 0, "int-secs") : 0;
    itv.it_interval.tv_usec = (argc > 4) ? getLong(argv[4], 0, "int-usecs") : 0;

4   if (setitimer(ITIMER_REAL, &itv, NULL) == -1)
        errExit("setitimer");

    prevClock = clock();
    sigCnt = 0;

5   for (;;) {
    /* Внутренний цикл потребляет как минимум 0,5 секунды процессорного времени */
        while (((clock() - prevClock) * 10 / CLOCKS_PER_SEC) < 5) {
6           if (gotAlarm) {             /* Получили ли мы сигнал? */
                gotAlarm = 0;
                displayTimes("ALARM:", TRUE);

                sigCnt++;
7               if (sigCnt >= maxSigs) {
                    printf("That's all folks\n");
                    exit(EXIT_SUCCESS);
                }
            }
        }

        prevClock = clock();
        displayTimes("Main: ", TRUE);
    }
}
```

timers/real_timer.c

Более простой интерфейс таймера: вызов alarm()

Системный вызов `alarm()` предоставляет более простой интерфейс для установки таймеров реального времени, которые срабатывают только один раз, без повторения с регулярным интервалом (на самом деле это первый программный интерфейс UNIX для установки таймера).

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Всегда завершается успешно, возвращая количество секунд, оставшихся до срабатывания предыдущего таймера, или `0`, если таймер ранее не устанавливался

Аргумент `seconds` обозначает количество секунд, по истечении которых произойдет срабатывание таймера. В этот момент вызывающему процессу будет доставлен сигнал `SIGALRM`.

Вызов `alarm()` переопределяет любой ранее установленный таймер. Чтобы выключить имеющийся таймер, можно сделать вызов `alarm(0)`.

В качестве итогового значения `alarm()` возвращает количество секунд, оставшихся до срабатывания предыдущего таймера, или `0`, если такового не обнаружено.

Пример использования вызова `alarm()` показан в разделе 23.3.

В некоторых примерах, которые будут представлены в этой книге, вызов alarm() применяется для запуска таймера без предварительной установки соответствующего обработчика SIGALRM. Это позволяет гарантировать, что, если процесс не завершится самостоятельно, его работа будет остановлена по истечении определенного времени.

Взаимодействие между функциями setitimer() и alarm()

В Linux функции `setitimer()` и `alarm()` используют один и тот же таймер реального времени, который выделяется каждому процессу; это означает, что применение одного из этих вызовов изменяет любой имеющийся таймер, независимо от того, какая функция его установила. Другие реализации UNIX могут вести себя иначе, предоставляя отдельные таймеры для каждого из этих вызовов. Стандарт SUSv3 намеренно оставляет без внимания отношения между функциями `setitimer()` и `alarm()`, равно как и взаимодействие между этими функциями и вызовом `sleep()`, описанным в подразделе 23.4.1. Чтобы обеспечить максимальную переносимость наших приложений, мы должны убедиться в том, что для установки таймеров реального времени используется либо `setitimer()`, либо `alarm()`.

23.2. Планирование и точность таймеров

В зависимости от нагрузки на систему и поведения планировщика реакция процесса на срабатывание таймера может быть запланирована с небольшой задержкой (обычно речь идет о долях секунды). Несмотря на это, срабатывание периодических таймеров, установленных с помощью вызова `setitimer()` или других интерфейсов, описанных в следующих разделах, будет происходить регулярно. Например, если интервал таймера реального времени равен 2 секундам, отдельные события могут наступать с такой задержкой, однако любое последующее срабатывание будет планироваться ровно на 2 секунды вперед. Иными словами, интервальные таймеры не подвержены отклонениям, которые могут накапливаться со временем.

Структура `timeval`, которая применяется в вызове `setitimer()`, позволяет указывать время с точностью до микросекунд, однако точность самого таймера традиционно ограничена частотой программных часов (см. раздел 10.6). Если значение таймера не кратно минимальному отрезку времени в программных часах, оно округляется в большую сторону. Например, если интервал таймера равен 19 100 микросекунд (то есть чуть больше 19 миллисекунд), то при минимальном отрезке 4 миллисекунды получилось бы, что таймер будет срабатывать каждые 20 миллисекунд.

Таймеры высокой точности

В современных ядрах Linux вышеприведенное утверждение о том, что точность таймера ограничена частотой программных часов, больше не действительно. Начиная с версии ядра 2.6.21, Linux предоставляет опциональную поддержку высокоточных таймеров. Если ее включить (с помощью параметра конфигурации ядра `CONFIG_HIGH_RES_TIMERS`), точность различных интерфейсов для перехода в режим сна (будут описаны позже в этой главе) и работы с таймерами больше не будет ограничена размером минимального отрезка времени, который распознается системой. Вместо этого вызовы будут настолько точными, насколько это позволяет аппаратное обеспечение. В современных компьютерах отсчет обычно производится с точностью до микросекунды.

Доступность высокоточных таймеров можно определить на основе показателя точности часов, который возвращается вызовом clock_getres() (см. подраздел 23.5.1).

23.3. Установка времени ожидания для блокирующих операций

Одной из целей применения таймеров реального времени является ограничение максимальной продолжительности блокирования системных вызовов. Например, мы можем отменить чтение из терминала, если пользователь не ввел ни единой строчки на протяжении какого-то времени. Это можно сделать следующим образом.

1. Вызываем `sigaction()`, чтобы установить обработчик сигнала `SIGALRM`. При этом опускаем флаг `SA_RESTART`, чтобы системный вызов не перезапускался (см. раздел. 21.5).
2. Делаем вызов `alarm()` или `setitimer()`, чтобы установить таймер с максимальным интервалом, на протяжении которого может блокироваться наш системный вызов.
3. Выполняем блокирующий системный вызов.
4. По завершении системного вызова еще раз используем функцию `alarm()` или `setitimer()`, чтобы отключить таймер (в случае если системный вызов завершился до того, как истекло время ожидания).
5. Проверяем, не завершился ли блокирующий системный вызов ошибкой `EINTR`, установленной в переменной `errno` (то есть не был ли он прерван).

В листинге 23.2 данный подход демонстрируется на примере операции `read()` и таймера, установленного с помощью вызова `alarm()`.

Листинг 23.2. Выполнение чтения с временем ожидания

timers/timed_read.c

```
#include <signal.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 200

static void        /* Обработчик SIGALRM прерывает блокирующий системный вызов */
handler(int sig)
{
    printf("Caught signal\n");    /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    char buf[BUF_SIZE];
    ssize_t numRead;
    int savedErrno;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-secs [restart-flag]]\n", argv[0]);
    /* Устанавливаем обработчик сигнала SIGALRM. Позволяем прерывать системные
       вызовы, если не был указан второй аргумент командной строки. */
    sa.sa_flags = (argc > 2) ? SA_RESTART : 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = handler;
    if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");
```

```
    alarm((argc > 1) ? getInt(argv[1], GN_NONNEG, "num-secs") : 10);
    numRead = read(STDIN_FILENO, buf, BUF_SIZE);

    savedErrno = errno;       /* На случай, если alarm() изменит errno */
    alarm(0);                 /* Убеждаемся, что таймер выключен */
    errno = savedErrno;

    /* Определяем результат чтения */
    if (numRead == -1) {
        if (errno == EINTR)
            printf("Read timed out\n");
        else
            errMsg("read");
    } else {
        printf("Successful read (%ld bytes): %.*s",
                (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

timers/timed_read.c

Стоит отметить наличие потенциального состояния гонки в программе из листинга 23.2. Если таймер сработает после вызова `alarm()`, но перед началом чтения, операция `read()` не будет прервана обработчиком сигнала. Но, поскольку в таких ситуациях обычно используется довольно продолжительное время ожидания (минимум несколько секунд), эта проблема крайне маловероятна, поэтому на практике данный подход является вполне разумным. В книге [Stevens & Rago, 2005] предлагается альтернативная методика на основе функции `longjmp()`. Еще одним вариантом работы с операциями ввода/вывода является применение системных вызовов `select()` или `poll()` (см. главу 55), которые изначально поддерживают время ожидания и позволяют отслеживать ввод-вывод сразу для нескольких дескрипторов.

23.4. Приостановка выполнения на определенный отрезок времени (переход в режим сна)

Иногда процесс нужно приостановить на определенное время. Этого можно добиться с помощью связки из вызова `sigsuspend()` и интерфейсов для работы с таймерами, с которыми мы только что познакомились, однако проще будет воспользоваться одной из функций перехода в режим сна.

23.4.1. Переход в режим сна (низкая точность): вызов sleep()

Функция `sleep()` приостанавливает выполнение вызывающего процесса на определенное количество секунд, указанное в аргументе `seconds`, или до момента перехвата сигнала (который прерывает данный вызов).

В момент возобновления работы вызов `sleep()` возвращает `0`. Если вызов был прерван сигналом, он возвращает количество секунд, оставшееся до возобновления. Как и в случае с таймерами, установленными с помощью функций `alarm()` и `setitimer()`, нагрузка на систему может привести к тому, что возобновление работы процесса будет запланировано не сразу после вызова `sleep()`, а с некоторой (обычно небольшой) задержкой.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Возвращает 0 при нормальном завершении или количество секунд, остававшихся до возобновления работы, в случае преждевременного прерывания

Стандарт SUSv3 не описывает возможное взаимодействие между операцией `sleep()` и вызовами `alarm()` и `setitimer()`. В Linux функция `sleep()` реализована в виде обертки для вызова `nanosleep()` (см. подраздел 23.4.2), благодаря чему она никак не влияет на функции `alarm()` и `setitimer()`. Однако во многих других системах, особенно старых, операция `sleep()` реализована с помощью вызова `alarm()` и обработчика сигнала `SIGALRM`. При написании переносимого кода ее не следует смешивать с функциями `alarm()` и `setitimer()`.

23.4.2. Переход в режим сна (высокая точность): вызов nanosleep()

Функция `nanosleep()` по принципу своей работы похожа на `sleep()`, но имеет ряд преимуществ, включая повышенную точность при задании интервала.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep(const struct timespec *request, struct timespec *remain);
```

Возвращает 0 в случае успешного завершения или –1, если произошли ошибка или прерывание

Аргумент `request` обозначает продолжительность сна и является указателем на структуру следующего вида:

```
struct timespec {
    time_t tv_sec;            /* Секунды */
    long   tv_nsec;           /* Наносекунды */
};
```

Поле `tv_nsec` содержит количество наносекунд. Это должно быть число в диапазоне от 0 до 999 999 999.

Еще одним преимуществом функции `nanosleep()` является то, что стандарт SUSv3 явно запрещает реализовывать ее с помощью сигналов. Это означает, что, в отличие от `sleep()`, ее можно использовать в сочетании с вызовом `alarm()` или `setitimer()`.

Несмотря на это, функция `nanosleep()` тоже может быть прервана обработчиком сигнала. В этом случае она вернет `–1` и присвоит переменной `errno` ошибку `EINTR`; если же аргумент `remain` не равен `NULL`, буфер, на который он указывает, будет содержать время, остававшееся до возобновления работы. При желании мы можем использовать это значение, чтобы перезапустить системный вызов и продолжить сон. Такой подход продемонстрирован в листинге 23.3. В качестве аргументов командной строки данная программа принимает секунды и наносекунды для вызова `nanosleep()`. Программа циклически вызывает `nanosleep()`, пока не завершится указанный интервал времени. Если

вызов прервется обработчиком сигнала `SIGINT` (сгенерированного с помощью сочетания клавиш Ctrl+C), он будет перезапущен с помощью значения, возвращенного в буфере `remain`. Запустив эту программу, мы увидим следующее:

```
$ ./t_nanosleep 10 0              Сон на протяжении 10 секунд
Нажимаем Ctrl+C
Slept for:    1.853428 secs
Remaining:    8.146617000
Нажимаем Ctrl+C
Slept for:    4.370860 secs
Remaining:    5.629800000
Нажимаем Ctrl+C
Slept for:    6.193325 secs
Remaining:    3.807758000
Slept for:    10.008150 secs
Sleep complete
```

Функция `nanosleep()` позволяет указывать продолжительность сна в наносекундах, одна ее реальная точность ограничена программными часами (см. раздел 10.6). Если указать интервал, который не является кратным программным часам, он будет округлен в большую сторону.

> Как отмечалось ранее, в системах, поддерживающих высокоточные таймеры, точность сна может быть намного лучше, чем у программных часов.

Наличие округления означает, что, если сигналы принимают с высокой частотой, у нас могут возникнуть проблемы с подходом, который используется в программе из листинга 23.3. Дело в том, что при каждом повторном запуске `nanosleep()` будет накапливаться отклонение, связанное с округлением, так как вероятность того, что итоговое значение `remain` окажется кратным наименьшему интервалу программных часов, довольно низкая. Следовательно, каждый следующий вызов `nanosleep()` будет останавливать выполнение на более продолжительное время, чем указано в значении `remain` предыдущего вызова. В случае если сигналы доставляются с высокой частотой (то есть чаще, чем обновляются программные часы), процесс может никогда не выйти из состояния сна. В Linux 2.6 и выше эту проблему можно обойти с помощью вызова `clock_nanosleep()` и параметра `TIMER_ABSTIME`, которые будут рассмотрены в подразделе 23.5.4.

Листинг 23.3. Использование функции nanosleep()

timers/t_nanosleep.c

```
#define _POSIX_C_SOURCE 199309
#include <sys/time.h>
#include <time.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigintHandler(int sig)
{
    return;            /* Просто прерываем nanosleep() */
}

int
main(int argc, char *argv[])
{
    struct timeval start, finish;
    struct timespec request, remain;
```

```
    struct sigaction sa;
    int s;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs nanosecs\n", argv[0]);
    request.tv_sec = getLong(argv[1], 0, "secs");
    request.tv_nsec = getLong(argv[2], 0, "nanosecs");

    /* Позволяем обработчику SIGINT прерывать nanosleep() */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigintHandler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    if (gettimeofday(&start, NULL) == -1)
        errExit("gettimeofday");

    for (;;) {
        s = nanosleep(&request, &remain);
        if (s == -1 && errno != EINTR)
            errExit("nanosleep");

        if (gettimeofday(&finish, NULL) == -1)
            errExit("gettimeofday");
        printf("Slept for: %9.6f secs\n", finish.tv_sec - start.tv_sec +
                        (finish.tv_usec - start.tv_usec) / 1000000.0);

        if (s == 0)
            break;            /* Вызов nanosleep() завершен */

        printf("Remaining: %2ld.%09ld\n", (long) remain.tv_sec, remain.tv_nsec);
        request = remain;      /* Следующий переход в режим сна будет
                                  длиться оставшееся время */
    }

    printf("Sleep complete\n");
    exit(EXIT_SUCCESS);
}
```

timers/t_nanosleep.c

23.5. Часы стандарта POSIX

POSIX-часы (изначально разработанные для стандарта POSIX.1b) предоставляют программный интерфейс для доступа к часам, которые измеряют время в наносекундах. Для представления такого времени используется та же структура `timespec`, которую мы применяли в подразделе 23.4.2 в вызове `nanosleep()`.

В Linux программы, которые работают с этим интерфейсом, должны быть скомпилированы с параметром `-lrt`, иначе их нельзя будет скомпоновать с библиотекой реального времени `librt`.

Главными системными вызовами программного интерфейса POSIX-часов являются `clock_gettime()`, который возвращает текущее значение часов, `clock_getres()`, позволяющий определить их точность, и `clock_settime()`, который обновляет часы.

23.5.1. Получение текущего значения часов: вызов clock_gettime()

Системный вызов `clock_gettime()` возвращает время в соответствии с часами, указанными в аргументе `clockid`.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_getres(clockid_t clockid, struct timespec *res);
```

Оба вызова возвращают `0` при успешном завершении или `-1`, если случилась ошибка

На значение времени, возвращаемое внутри структуры `timespec`, указывает аргумент `tp`. И хотя структура `timespec` поддерживает наносекунды, значение, возвращенное вызовом `clock_gettime()`, может оказаться менее точным. Системный вызов `clock_getres()` возвращает указатель на структуру `timespec` (аргумент `clockid`), в которой содержится точность часов.

Тип данных `clockid_t` предусмотрен стандартом SUSv3 для представления идентификатора часов. Значения, которые можно указывать в аргументе `clockid`, перечислены в первом столбце табл. 23.1.

Таблица 23.1. Типы часов стандарта POSIX.1b

Идентификатор часов	Описание
CLOCK_REALTIME	Общесистемные часы реального времени, доступные для изменения
CLOCK_MONOTONIC	Монотонные часы, доступные только для чтения
CLOCK_PROCESS_CPUTIME_ID	Часы процессорного времени на уровне отдельного процесса (начиная с Linux 2.6.12)
CLOCK_THREAD_CPUTIME_ID	Часы процессорного времени на уровне отдельного потока (начиная с Linux 2.6.12)

Константа `CLOCK_REALTIME` представляет общесистемные часы, которые измеряют обычное время. В отличие от `CLOCK_MONOTONIC` их можно изменять.

В стандарте SUSv3 сказано, что часы `CLOCK_MONOTONIC` измеряют время, начиная с какого-то «неопределенного момента в прошлом», который не меняется после запуска системы. Эти часы могут пригодиться в приложениях, на которые не должны влиять искусственные изменения системных часов (например, когда пользователь редактирует системное время). В Linux этот вид часов измеряет время, прошедшее с момента запуска системы.

Часы `CLOCK_PROCESS_CPUTIME_ID` измеряют пользовательское и системное процессорное время, затраченное вызывающим процессом. Часы `CLOCK_THREAD_CPUTIME_ID` выполняют аналогичное действие, но работают на уровне отдельных потоков внутри процесса.

Все часы, перечисленные в табл. 23.1, входят в стандарт SUSv3, но обязательной и широко распространенной в UNIX-системах является только константа CLOCK_REALTIME.

В Linux 2.6.28 появился новый тип часов, CLOCK_MONOTONIC_RAW, которые доступны только для чтения. Они похожи на CLOCK_MONOTONIC, но предоставляют доступ к «сырому» аппаратному времени, без корректировки со стороны службы NTP. Эти часы являются нестандартными и предназначены для использования в специализированных приложениях для синхронизации времени.

В Linux 2.6.32 появилось еще два типа часов: CLOCK_REALTIME_COARSE и CLOCK_MONOTIC_COARSE. Они похожи на CLOCK_REALTIME и CLOCK_MONOTONIC, но предназначены для приложений, которым нужно получать не самые точные временные метки с минимальной затратой ресурсов. Эти нестандартные константы не предоставляют доступа к аппаратным часам (что в случае с некоторыми источниками времени может оказаться довольно затратным), а точность возвращаемых ими значений соответствует минимальному временному интервалу (см. раздел 10.6).

23.5.2. Изменение значения часов: вызов clock_settime()

Системный вызов `clock_settime()` устанавливает время, хранящееся в буфере, на который указывает аргумент `tp`, для часов, определенных с помощью `clockid`.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_settime(clockid_t clockid, const struct timespec *tp);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Если время, указанное с помощью `tp`, не является кратным минимальному временному интервалу (который возвращается вызовом `clock_getres()`), оно округляется в меньшую сторону.

Изменение часов `CLOCK_REALTIME` доступно привилегированному процессу (`CAP_SYS_TIME`). Их начальное значение обычно равно времени, прошедшему с начала эры UNIX. Из всех часов, перечисленных в табл. 23.1, это единственные, которые можно изменять.

Согласно стандарту SUSv3 система может позволить устанавливать время для часов CLOCK_PROCESS_CPUTIME_ID и CLOCK_THREAD_CPUTIME_ID. На момент написания данной книги в Linux эти часы были доступны только для чтения.

23.5.3. Получение идентификатора часов для определенного процесса или потока

Функции, описанные в этом разделе, позволяют получить идентификатор часов, которые измеряют процессорное время, потребленное конкретным процессом или потоком. Полученное значение можно передать вызову `clock_gettime()`, чтобы определить, сколько всего процессорного времени потребил процесс или поток.

Функция `clock_getcpuclockid()` возвращает идентификатор часов для процесса `pid`; итоговое значение хранится в буфере, на который указывает аргумент `clockid`.

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clockid);
```

Возвращает 0 при успешном завершении или положительный код ошибки

Если аргумент `pid` равен `0`, вызов `clock_getcpuclockid()` возвращает идентификатор часов текущего процесса.

Функция `pthread_getcpuclockid()` аналогична вызову `clock_getcpuclockid()`, но работает с потоками POSIX. Она возвращает идентификатор часов, которые измеряют процессорное время, потребленное определенным потоком в рамках вызывающего процесса.

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread, clockid_t *clockid);
```

Возвращает 0 при успешном завершении или положительный код ошибки

Аргумент `thread` обозначает идентификатор POSIX-потока, часы которого нас интересуют. Идентификатор часов возвращается в буфере, на который указывает аргумент `clockid`.

23.5.4. Улучшенный переход в режим сна (высокая точность): вызов clock_nanosleep()

У системного вызова `nanosleep()` есть аналог, `clock_nanosleep()`, который приостанавливает вызывающий процесс на определенный отрезок времени (либо пока он не будет прерван сигналом). В этом разделе мы перечислим особенности, которые отличают его от `nanosleep()`.

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_nanosleep(clockid_t clockid, int flags,
    const struct timespec *request, struct timespec *remain);
```

Возвращает 0 в случае успешного завершения или положительный код, если вызов завершился ошибкой или был прерван сигналом

Аргументы `request` и `remain` имеют то же назначение, что и в вызове `nanosleep()`.

Продолжительность сна, заданная в аргументе `request`, по умолчанию (то есть когда аргумент `flags` равен `0`) является относительной (как и в случае с `nanosleep()`). Но если среди флагов `flags` указать константу `TIMER_ABSTIME` (см. пример в листинге 23.4), значение `request` станет абсолютным и будет соответствовать времени, измеренному часами `clockid`. Эта возможность востребована приложениями, которым нужно приостанавливать работу на заданное время с высокой точностью. Если вместо этого получить текущее время, вычислив, сколько осталось до заданного момента, и затем выполнить «относительный

переход в режим сна», между любым из этих шагов может возникнуть задержка, которая сделает сон процесса дольше, чем мы того хотели.

Как уже было сказано в подразделе 23.4.2, проблема «затянувшегося сна» особенно остро стоит для процессов, применяющих цикл для возобновления перехода в режим сна, который был прерван обработчиком сигнала. Если сигналы поступают с высокой частотой, относительные значения (которые используются в вызове `nanosleep()`) могут вызывать существенные отклонения во времени, которое уходит на простаивание процесса. Этой проблемы можно избежать, если в самом начале сделать вызов `clock_gettime()`, чтобы получить текущее время и добавить к нему необходимое значение, и затем задействовать вызов `clock_nanosleep()` с флагом `TIMER_ABSTIME` (перезапуская системный вызов, если он был прерван обработчиком сигнала).

Если установить флаг `TIMER_ABSTIME`, аргумент `remain` будет игнорироваться (за ненадобностью). Если вызов `clock_nanosleep()` прерывается обработчиком сигнала, его можно повторить, используя тот же аргумент `request`.

Еще одна отличительная черта вызова `clock_nanosleep()` по сравнению с `nanosleep()` состоит в том, что мы можем выбрать часы, которые применяются для измерения периода сна. Для этого аргументу `clockid` можно передать одно из следующих значений: `CLOCK_REALTIME`, `CLOCK_MONOTONIC` или `CLOCK_PROCESS_CPUTIME_ID`. Описание этих часов приводится в табл. 23.1.

В листинге 23.4 показан пример использования вызова `clock_nanosleep()` для перехода в режим сна на 20 секунд с применением часов `CLOCK_REALTIME` и абсолютного времени.

Листинг 23.4. Использование вызова clock_nanosleep()

```
struct timespec request;

/* Получаем текущее значение часов CLOCK_REALTIME */

if (clock_gettime(CLOCK_REALTIME, &request) == -1)
    errExit("clock_gettime");

request.tv_sec += 20;       /* "Засыпаем" на 20 секунд, начиная с этого момента */

s = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &request, NULL);
if (s != 0) {
    if (s == EINTR)
        printf("Interrupted by signal handler\n");
    else
        errExitEN(s, "clock_nanosleep");
}
```

23.6. Интервальные таймеры POSIX

Классические интервальные таймеры в UNIX, которые устанавливаются с помощью вызова `setitimer()`, имеют целый ряд ограничений.

- Мы можем устанавливать лишь по одному таймеру каждого из трех типов: `ITIMER_REAL`, `ITIMER_VIRTUAL` и `ITIMER_PROF`.
- Единственный способ уведомления о срабатывании таймера заключается в передаче сигнала. Кроме того, мы не можем выбрать, какой именно сигнал должен приходить.
- Если во время блокирования соответствующего сигнала интервальный таймер успеет сработать несколько раз, обработчик будет вызван всего лишь один раз. Иными словами, мы не можем узнать, был ли таймер *просрочен*.

- Таймеры отсчитывают время в микросекундах. Однако некоторые системы поддерживают аппаратные часы с более высокой точностью; в таких случаях желательно иметь программный доступ к точным значениям.

В стандарте POSIX.1b описан программный интерфейс для обхода этих ограничений. В Linux он доступен с версии 2.6.

Жизненный цикл программного интерфейса POSIX-таймеров состоит из следующих этапов.

1. Системный вызов `timer_create()` создает новый таймер и определяет способ, с помощью которого процесс будет оповещен о его срабатывании.
2. Системный вызов `timer_settime()` запускает или останавливает таймер.
3. Системный вызов `timer_delete()` удаляет таймер, который больше не нужен.

POSIX-таймеры не наследуются потомками, которые создаются с помощью вызова `fork()`. Во время выполнения `exec()` или при завершении процесса они останавливаются и удаляются.

В Linux программы, которые работают с программным интерфейсом POSIX-таймеров, должны быть скомпилированы с параметром `-lrt`, иначе их нельзя будет скомпоновать с библиотекой реального времени `librt`.

23.6.1. Создание таймера: вызов timer_create()

Функция `timer_create()` создает новый таймер, который отсчитывает время с помощью часов, указанных в аргументе `clockid`.

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Аргумент `clockid` может содержать любое значение, указанное в табл. 23.1 или возвращаемое вызовами `clock_getcpuclockid()` или `pthread_getcpuclockid()`. Аргумент `timerid` указывает на буфер с идентификатором, который позволяет обращаться к таймеру в последующих вызовах. Этот буфер имеет тип `timer_t`, который предусмотрен стандартом SUSv3 для представления идентификаторов таймера.

Аргумент `evp` определяет то, каким образом программа будет уведомлена о срабатывании таймера. Он указывает на структуру типа `sigevent`, которая имеет следующий вид:

```
union sigval {
    int sival_int;              /* Целочисленное значение для вспомогательных данных */
    void *sival_ptr;            /* Указатель на вспомогательные данные */
};

struct sigevent {
    int sigev_notify;           /* Тип уведомления */
    int sigev_signo;            /* Сигнал о срабатывании таймера */
    union sigval sigev_value;   /* Значение, сопровождающее сигнал
                                   или передаваемое в функцию потока */

    union {
        pid_t _tid;       /* Идентификатор потока, который получит сигнал /
```

```
        struct {
            void (*_function) (union sigval);
            /* Функция уведомления потока */
            void *_attribute;    /* Имеет тип 'pthread_attr_t *' */
        } _sigev_thread;
    } _sigev_un;
};

#define sigev_notify_function    _sigev_un._sigev_thread._function
#define sigev_notify_attributes  _sigev_un._sigev_thread._attribute
#define sigev_notify_thread_id   _sigev_un._tid
```

Поле `sigev_notify` этой структуры может принимать одно из значений, перечисленных в табл. 23.2.

Таблица 23.2. Значения поля sigev_notify структуры sigevent

Значение sigev_notify	Тип уведомления	SUSv3
SIGEV_NONE	Без уведомления; отслеживание таймера с помощью timer_gettime()	*
SIGEV_SIGNAL	Отправляет процессу сигнал sigev_signo	*
SIGEV_THREAD	Начинает выполнение нового потока с функции sigev_notify_function	*
SIGEV_THREAD_ID	Отправляет сигнал sigev_signo потоку sigev_notify_thread_id	

Ниже подробно описываются константы `sigev_notify` и поля структуры `sigval`, связанные с каждой из них.

- `SIGEV_NONE` — не отправляет уведомление о срабатывании таймера. Процесс по-прежнему может отслеживать состояние таймера с помощью вызова `timer_gettime()`.
- `SIGEV_SIGNAL` — при срабатывании таймера шлет процессу сигнал, указанный в поле `sigev_signo`. Если это сигнал реального времени, сопутствующие данные, которые передаются вместе с ним (см. подраздел 22.8.1), должны быть указаны в поле `sigev_value` (это может быть целое число или указатель). Эти данные можно получить из поля `si_value` структуры `siginfo_t`, которая передается обработчику сигнала или возвращается вызовами `sigwaitinfo()` и `sigtimedwait()`.
- `SIGEV_THREAD` — при срабатывании таймера вызывает функцию, указанную в поле `sigev_notify_function`. Эта функция запускается так, *как будто* она является начальной для нового потока. Слова «как будто» используются в самом стандарте SUSv3; это позволяет реализации генерировать уведомление для периодического таймера двумя способами: либо доставляя каждое из них отдельному новому потоку, либо передавая в единый поток все имеющиеся уведомления. Поле `sigev_notify_attributes` может содержать либо значение `NULL`, либо указатель на структуру `pthread_attr_t`, которая определяет атрибуты потока (см. раздел 29.8). Объединение `sigval`, указанное внутри `sigev_value`, передается в функцию в виде единственного аргумента.
- `SIGEV_THREAD_ID` — похоже на `SIGEV_SIGNAL`, но сигнал отправляется потоку, чей идентификатор равен `sigev_notify_thread_id`. Этот поток должен находиться в одном процессе с вызывающим потоком (в случае с `SIGEV_SIGNAL` сигнал попадает в очередь процесса, и если этот процесс состоит из нескольких потоков, сигнал доставляется одному из них в произвольном порядке). Поле `sigev_notify_thread_id` может содержать значение, возвращаемое вызовами `clone()` или `gettid()`.

Флаг `SIGEV_THREAD_ID` предназначен для использования в потоковых библиотеках (он требует, чтобы при реализации многопоточности был задействован параметр `CLONE_THREAD`, описанный в подразделе 28.2.1; этому требованию удовлетворяет современная потоковая библиотека NPTL, в отличие от более старой реализации под названием LinuxThreads).

Все вышеперечисленные константы, кроме `SIGEV_THREAD_ID` (которая доступна только в Linux), входят в стандарт SUSv3.

Аргументу `evp` можно присвоить значение `NULL`, тогда результат будет таким же, как если бы мы указали для полей `sigev_notify`, `sigev_signo` и `sigev_value.sival_int` значения `SIGEV_SIGNAL`, `SIGALRM` (может варьироваться в зависимости от системы, так как в стандарте SUSv3 сказано лишь «номер сигнала по умолчанию») и соответственно идентификатор таймера.

Текущая реализация ядра заранее выделяет по одной структуре с сигналом реального времени для каждого POSIX-таймера, созданного с помощью вызова `timer_create()`. Это делается для того, чтобы гарантировать, что в момент срабатывания таймера для сохранения сигнала будет доступна как минимум одна такая структура. Это означает, что максимальное количество POSIX-таймеров, которые мы можем создать, не превышает количество сигналов реального времени, которые можно поместить в очередь (см. раздел 22.8).

23.6.2. Запуск и остановка таймера: вызов timer_settime()

Создав таймер, мы можем его запускать и останавливать, используя вызов `timer_settime()`.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_settime(timer_t timerid, int flags, const struct
                  itimerspec *value, struct itimerspec *old_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Аргумент `timerid` принимает идентификатор таймера, полученный ранее из вызова `timer_create()`.

Аргументы `value` и `old_value` аналогичны тем, что используются в вызове `setitimer()`: первый определяет новые параметры таймера, а второй применяется для возвращения предыдущих параметров (см. ниже описание вызова `timer_gettime()`). Если нас не интересуют параметры, установленные ранее, мы можем присвоить аргументу `old_value` значение `NULL`. Аргументы `value` и `old_value` являются указателями на структуры типа `itimerspec`, которые имеют следующий вид:

```
struct itimerspec {
    struct timespec it_interval;    /* Интервал периодического таймера */
    struct timespec it_value;       /* Первое срабатывание */
};
```

Оба поля вышеприведенной структуры, в свою очередь, являются структурами типа `timespec`, которая представляет время в виде секунд и наносекунд:

```
struct timespec {
    time_t tv_sec;                  /* Секунды */
    long tv_nsec;                   /* Наносекунды */
};
```

Поле `it_value` определяет момент первого срабатывания таймера. Если хотя бы одно из вложенных полей `it_interval` не равно `0`, таймер является периодическим и после истечения времени `it_value` будет срабатывать с частотой, заданной с помощью этих полей. Если оба вложенных поля `it_interval` равны `0`, таймер сработает только один раз.

Если аргумент `flags` равен `0`, поле `value.it_value` интерпретируется относительно значения часов на момент вызова `timer_settime()` (по аналогии с `setitimer()`). Если аргумент `flags` равен `TIMER_ABSTIME`, поле `value.it_value` интерпретируется как абсолютное время (то есть отсчитывается с начальной позиции часов). Если согласно часам это время уже прошло, таймер срабатывает немедленно.

Чтобы запустить таймер, нужно сделать вызов `timer_settime()`, передав хотя бы одному вложенному полю `value.it_value` ненулевое значение. Если таймер уже был запущен ранее, вызов `timer_settime()` обновит предыдущие параметры.

Если значение таймера и интервал не являются кратными минимальному отрезку времени в соответствующих часах (который можно получить с помощью `clock_getres()`), эти значения будут округлены в большую сторону до следующего кратного.

При каждом срабатывании таймера процесс уведомляется с помощью метода, который изначально был указан в вызове `timer_create()`. Если внутри `it_interval` содержатся ненулевые значения, они используются для повторной инициализации структуры `it_value`.

Чтобы остановить таймер, нужно опять сделать вызов `timer_settime()`, присвоив `0` обоим полям структуры `value.it_value`.

23.6.3. Получение текущего значения таймера: вызов timer_gettime()

Системный вызов `timer_gettime()` возвращает интервал и время, оставшееся до срабатывания POSIX-таймера с идентификатором `timerid`.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Интервал и время, оставшееся до следующего срабатывания таймера, возвращаются внутри структуры `itimerspec`, на которую указывает аргумент `curr_value`. Поле `curr_value.it_value` содержит оставшееся время, даже если таймер был сделан абсолютным с помощью значения `TIMER_ABSTIME`. Если оба поля возвращенной структуры `curr_value.it_value` равны `0`, таймер не запущен. Если оба поля структуры `curr_value.it_interval` равны `0`, таймер должен сработать только один раз в момент времени `curr_value.it_value`.

23.6.4. Удаление таймера: вызов timer_delete()

Любой POSIX-таймер потребляет какое-то количество ресурсов. То есть, закончив работать с таймером, его нужно удалить и освободить эти ресурсы, используя вызов `timer_delete()`.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_delete(timer_t timerid);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Аргумент `timerid` представляет собой идентификатор, возвращенный предыдущим вызовом `timer_create()`. Если таймер был запущен, перед удалением он автоматически останавливается. Сигнал о срабатывании данного таймера, который уже успел попасть в очередь, сохраняется (этот момент не описан в стандарте SUSv3, поэтому другие системы могут вести себя иначе). Таймеры автоматически удаляются при завершении процесса.

23.6.5. Уведомление с помощью сигнала

Если мы выбрали сигналы в качестве способа доставки уведомлений, для их получения можно применить либо обработчик, либо вызов `sigwaitinfo()` или `sigtimedwait()`. Оба варианта позволяют принимающему процессу получить структуру `siginfo_t` (см. раздел 21.4) с подробностями о сигнале (чтобы воспользоваться этой возможностью в обработчике сигнала, во время его установки нужно указать флаг `SA_SIGINFO`). В структуре `siginfo_t` заполнены следующие поля.

- `si_signo` — содержит сигнал, сгенерированный таймером.
- `si_code` — хранит значение `SI_TIMER`; это означает, что сигнал был сгенерирован при срабатывании POSIX-таймера.
- `si_value` — содержит значение, которое было указано в поле `evp.sigev_value` при создании таймера с помощью вызова `timer_create()`. Использование разных значений в `evp.sigev_value` позволяет различать срабатывания разных таймеров, которые доставляют один и тот же сигнал.

При создании таймера полю `evp.sigev_value.sival_ptr` обычно назначается адрес аргумента `timerid`, который передается в том же вызове `timer_create()` (листинг 23.5). Это позволяет обработчику сигнала (или вызову `sigwaitinfo()`) получить идентификатор таймера, который сгенерировал сигнал (как вариант, полю `evp.sigev_value.sival_ptr` можно было бы назначить адрес структуры, которая содержит тот же аргумент `timerid`).

В Linux структура `siginfo_t` также поддерживает нестандартное поле `si_overrun`. Оно содержит счетчик дополнительных срабатываний данного таймера (описывается в подразделе 23.6.6).

> Linux предоставляет еще одно нестандартное поле: si_timerid. Оно хранит идентификатор, который используется системой для определения таймера (он не совпадает с идентификатором, который возвращает вызов timer_create()). Для прикладных программ он бесполезен.

Использование сигналов в качестве механизма уведомления для POSIX-таймера демонстрируется в листинге 23.5.

Листинг 23.5. Уведомление о срабатывании POSIX-таймера с помощью сигнала
timers/ptmr_sigev_signal.c

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>
#include "curr_time.h"              /* Объявляет currTime() */
```

```
#include "itimerspec_from_str.h"   /* Объявляет itimerspecFromStr() */
#include "tlpi_hdr.h"

#define TIMER_SIG SIGRTMAX      /* Наш сигнал для уведомления о срабатывании таймера */

static void
(1) handler(int sig, siginfo_t *si, void *uc)
{
    timer_t *tidptr;

    tidptr = si->si_value.sival_ptr;

    /* НЕБЕЗОПАСНО: этот обработчик вызывает функцию, не рассчитанную
      на работу с асинхронными сигналами (printf(); см. подраздел 21.1.2) */

    printf("[%s] Got signal %d\n", currTime("%T"), sig);
    printf("    *sival_ptr         = %ld\n", (long) *tidptr);
    printf("    timer_getoverrun() = %d\n", timer_getoverrun(*tidptr));
}

int
main(int argc, char *argv[])
{
    struct itimerspec    ts;
    struct sigaction    sa;
    struct sigevent        sev;
    timer_t *tidlist;
    int j;

    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);
    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

    /* Устанавливаем обработчик для сигнала-уведомления */
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
(2) if (sigaction(TIMER_SIG, &sa, NULL) == -1)
        errExit("sigaction");

    /* Создаем и запускаем по одному таймеру для каждого аргумента командной строки */
    sev.sigev_notify = SIGEV_SIGNAL;  /* Уведомляем с помощью сигнала */
    sev.sigev_signo = TIMER_SIG;      /* Указываем сигнал для уведомления */

    for (j = 0; j < argc - 1; j++) {
(3)     itimerspecFromStr(argv[j + 1], &ts);
        sev.sigev_value.sival_ptr = &tidlist[j];
        /* Позволяем обработчику получить идентификатор этого таймера */

(4)     if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);

(5)     if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }
```

```
❻ for (;;)       /* Ожидаем входящих сигналов, посланных таймером */
        pause();
}
```

timers/ptmr_sigev_signal.c

Каждый из аргументов командной строки в программе из листинга 23.5 представляет собой начальное значение и интервал таймера. Их синтаксис описан в сообщении `usageErr`, которое выводится в сессии командной строки чуть ниже. Программа выполняет такие шаги.

- Устанавливает обработчик для сигнала, который используется таймером для уведомления ❷.
- Создает ❹ и запускает ❺ для каждого аргумента командной строки POSIX-таймер с механизмом уведомления `SIGEV_SIGNAL`. Функция `itimerspecFromStr()`, с помощью которой мы конвертируем ❸ аргументы командной строки в структуры типа `itimerspec`, показана в листинге 23.6.
- При каждом срабатывании таймера процессу доставляется сигнал, указанный в поле `sev.sigev_signo`. Обработчик сигнала выводит содержимое поля `sev.sigev_value.sival_ptr` (то есть идентификатор таймера, `tidlist[j]`) и количество дополнительных срабатываний таймера ❶.
- Создав и запустив таймеры, входит в цикл и ждет, когда они сработают, постоянно делая вызов `pause()` ❻.

В листинге 23.6 показана функция, которая конвертирует каждый аргумент командной строки для нашей программы в соответствующую структуру `itimerspec`. Формат строковых аргументов, интерпретируемых данной функцией, указан в комментарии в верхней части листинга (и продемонстрирован в сессии командной строки чуть ниже).

Листинг 23.6. Преобразование строки с временем и интервалом в значение типа itimerspec

timers/itimerspec_from_str.c

```
#include <string.h>
#include <stdlib.h>
#include "itimerspec_from_str.h"  /* Объявляет определяемую здесь функцию */

/* Конвертирует строку следующего формата в структуру itimerspec:
   "value.sec[/value.nanosec][:interval.sec[/interval.nanosec]]".
   Дополнительные элементы, которые мы опускаем, приводят к обнулению
   соответствующих полей структуры. */

void
itimerspecFromStr(char *str, struct itimerspec *tsp)
{
    char *dupstr ,*cptr, *sptr;

    dupstr = strdup(str);
    cptr = strchr(dupstr, ':');
    if (cptr != NULL)
        *cptr = '\0';
    sptr = strchr(dupstr, '/');
    if (sptr != NULL)
        *sptr = '\0';

    tsp->it_value.tv_sec = atoi(dupstr);
    tsp->it_value.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
    if (cptr == NULL) {
```

```
        tsp->it_interval.tv_sec = 0;
        tsp->it_interval.tv_nsec = 0;
    } else {
        sptr = strchr(cptr + 1, '/');
        if (sptr != NULL)
            *sptr = '\0';
        tsp->it_interval.tv_sec = atoi(cptr + 1);
        tsp->it_interval.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
    }
    free(dupstr);
}
```

timers/itimerspec_from_str.c

В следующей сессии командной строки показан пример использования программы из листинга 23.5. Мы создадим один таймер с начальным временем срабатывания, равным 2 секундам, и интервалом 5 секунд.

```
$ ./ptmr_sigev_signal 2:5
Timer ID: 134524952 (2:5)
[15:54:56]                  В этой системе сигнал SIGRTMAX имеет номер 64
    *sival_ptr         = 134524952 sival_ptr указывает на переменную tid
    timer_getoverrun() = 0
[15:55:01] Got signal 64
    *sival_ptr         = 134524952
    timer_getoverrun() = 0
Нажимаем Ctrl+Z, чтобы приостановить процесс
[1]+ Stopped ./ptmr_sigev_signal 2:5
```

Мы приостанавливаем программу на несколько секунд, позволяя таймеру сработать несколько раз:

```
$ fg
./ptmr_sigev_signal 2:5
[15:55:34] Got signal 64
    *sival_ptr            = 134524952
    timer_getoverrun()    = 5
Нажимаем Ctrl+C, чтобы завершить программу
```

В последней строчке программного вывода видно, что таймер имел пять дополнительных срабатываний, то есть с момента доставки предыдущего сигнала он сработал шесть раз.

23.6.6. Дополнительные срабатывания таймера

Допустим, что мы выбрали доставку сигналов в качестве уведомления о срабатывании таймера (то есть аргумент `sigev_notify` равен `SIGEV_SIGNAL`). Теперь представьте, что прежде, чем этот сигнал будет перехвачен или принят, таймер успеет сработать еще несколько раз. Это может произойти в результате того, что возобновление работы процесса было запланировано с задержкой. Причиной также может быть блокирование сигнала — либо явное, с помощью вызова `sigprocmask()`, либо опосредованное, во время выполнения соответствующего обработчика. Как узнать, были ли у таймера *дополнительные срабатывания*?

Можно предположить, что в решении данной проблемы поможет сигнал реального времени, так как в очередь попало несколько его экземпляров. Однако это предположение неверно, поскольку количество сигналов реального времени, которые могут находиться в очереди, ограничено. Таким образом, комитет, ответственный за стандарт POSIX.1b, решил прибегнуть к другому способу: если выбрать сигнал в качестве средства уведомления о срабатывании таймера, экземпляры этого сигнала никогда не будут складываться в оче-

редь, даже если они работают в режиме реального времени. Вместо этого после получения сигнала (либо через обработчик, либо с помощью вызова `sigwaitinfo()`) мы можем узнать *количество дополнительных срабатываний* таймера, которые произошли между моментом создания сигнала и его доставкой. Например, если с момента получения последнего сигнала таймер сработал три раза, счетчик его дополнительных срабатываний будет равен 2.

Приняв сигнал от таймера, мы можем получить количество дополнительных срабатываний. Для этого есть два способа.

- Сделать вызов `timer_getoverrun()`, который будет описан ниже. Этот вариант определения количества дополнительных срабатываний предусмотрен стандартом SUSv3.
- Воспользоваться полем `si_overrun` из структуры `siginfo_t`, принятой вместе с сигналом. Такой подход является более экономным по сравнению с вызовом `timer_getoverrun()`, но это нестандартное решение, доступное только в Linux.

Счетчик дополнительных срабатываний сбрасывается при каждой доставке сигнала. Если с момента обработки или приема сигнала таймер сработает лишь раз, счетчик будет равен `0` (что говорит об отсутствии дополнительных срабатываний).

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_getoverrun(timer_t timerid);
```

Возвращает количество дополнительных срабатываний при успешном завершении или `-1`, если произошла ошибка

Функция `timer_getoverrun()` возвращает количество дополнительных срабатываний таймера, указанного в аргументе `timerid`. Согласно стандарту SUSv3 она является безопасной для работы с асинхронными сигналами (см. табл. 21.1), поэтому ее можно вызывать внутри обработчика.

23.6.7. Уведомление с помощью потока

Флаг `SIGEV_THREAD` позволяет программе получать уведомления о срабатывании таймера путем вызова функции в отдельном потоке. Для понимания этого флага нужно иметь представление о POSIX-потоках, с которыми вы познакомитесь позже, в главах 29 и 30. Возможно, вам придется прочитать эти главы, прежде чем переходить к демонстрационной программе, представленной в данном разделе.

Применение флага `SIGEV_THREAD` показано в листинге 23.7. Эта программа принимает те же аргументы командной строки, что и пример из листинга 23.5, и выполняет следующие шаги.

- Создает ❻ и запускает ❼ для каждого аргумента командной строки отдельный POSIX-таймер, который использует `SIGEV_THREAD` в качестве механизма уведомления ❸.
- При каждом срабатывании таймера в отдельном потоке вызывается функция, указанная в поле `sev.sigev_notify_function` ❹. Она принимает в качестве аргумента значение поля `sev.sigev_value.sival_ptr`, в котором задан адрес идентификатора таймера (`tidlist[j]`) ❺. Таким образом, функция уведомления может определить таймер, который инициировал ее вызов.
- Создав и запустив все таймеры, программа входит в цикл и ждет, когда они сработают ❽. На каждой итерации цикла делается вызов `pthread_cond_wait()`, который ждет оповещения условной переменно (`cond`) со стороны потока, отвечающего за уведомления таймера.

- При каждом срабатывании таймера вызывается функция `threadFunc()` ❶. После вывода сообщения она инкрементирует значение глобальной переменной `expireCnt`, а также добавляет результат, возвращенный вызовом `timer_getoverrun()`, чтобы учесть возможность дополнительных срабатываний (в подразделе 23.6.6 дополнительные срабатывания рассматривались в контексте механизма уведомлений `SIGEV_SIGNAL`, однако они возможны и при использовании `SIGEV_THREAD`, так как перед вызовом соответствующей функции таймер может успеть сработать несколько раз). Функция, доставляющая уведомления, также оповещает условную переменную `cond`, чтобы главная программа смогла удостовериться в срабатывании таймера ❷.

Программа из листинга 23.7 демонстрируется в следующей сессии командной строки. В этом примере создается два таймера, у которых совпадают начальное время срабатывания и интервал: у одного это 5 секунд, а у второго — 10.

```
$ ./ptmr_sigev_thread 5:5 10:10
Timer ID: 134525024 (5:5)
Timer ID: 134525080 (10:10)
[13:06:22] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 1
[13:06:27] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 2
[13:06:27] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 3
Нажимаем Ctrl+Z, чтобы приостановить процесс
[1]+ Stopped        ./ptmr_sigev_thread 5:5 10:10
$ fg                            Возобновляем выполнение
./ptmr_sigev_thread 5:5 10:10
[13:06:45] Thread notify
    timer ID=134525024
    timer_getoverrun()=2       Обнаружены дополнительные срабатывания
main(): count = 6
[13:06:45] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 7
Нажимаем Ctrl+C, чтобы завершить программу
```

Листинг 23.7. Уведомления POSIX-таймеров на основе функции в отдельном потоке
timers/ptmr_sigev_thread.c

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include "curr_time.h"            /* Объявление currTime() */
#include "tlpi_hdr.h"
#include "itimerspec_from_str.h"  /* Объявляет itimerspecFromStr() */

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int expireCnt = 0;     /* Количество срабатываний всех таймеров */
```

```
static void                   /* Функция уведомления в отдельном потоке */
❶ threadFunc(union sigval sv)
{
    timer_t *tidptr;
    int s;

    tidptr = sv.sival_ptr;
    printf("[%s] Thread notify\n", currTime("%T"));
    printf("    timer ID=%ld\n", (long) *tidptr);
    printf("    timer_getoverrun()=%d\n", timer_getoverrun(*tidptr));

    /* Инкрементируем счетчик, разделяемый с главным потоком, и оповещаем
       условную переменную, чтобы уведомить главный поток об изменении. */
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");
    expireCnt += 1 + timer_getoverrun(*tidptr);

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

❷ s = pthread_cond_signal(&cond);
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    struct itimerspec ts;
    timer_t *tidlist;
    int s, j;
    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);
    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

❸ sev.sigev_notify = SIGEV_THREAD;         /* Уведомляем с помощью потока */
❹ sev.sigev_notify_function = threadFunc;  /* Начальная функция потока */
    sev.sigev_notify_attributes = NULL;
    /* Это мог бы быть указатель на структуру pthread_attr_t */

   /* Создаем и запускаем по одному таймеру для каждого аргумента командной строки */
    for (j = 0; j < argc - 1; j++) {
        itimerspecFromStr(argv[j + 1], &ts);

❺      sev.sigev_value.sival_ptr = &tidlist[j];
        /* Передается в виде аргумента функции threadFunc() */
❻      if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);
❼      if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }
```

```
    /* Главный поток следит за условной переменной, которая оповещается при каждом
       вызове функции уведомления. Мы выводим сообщение, чтобы пользователь знал
       об этом событии. */
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

(8) for (;;) {
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
        printf("main(): expireCnt = %d\n", expireCnt);
    }
}
```

timers/ptmr_sigev_thread.c

23.7. Таймеры, которые уведомляют с помощью файловых дескрипторов: интерфейс timerfd

Начиная с версии 2.6.25, ядро Linux предоставляет еще один программный интерфейс для создания таймеров, `timerfd`. Он позволяет создавать таймеры, уведомления о срабатывании которых можно прочитать из файлового дескриптора. Это может пригодиться, поскольку такой дескриптор можно отслеживать как и любой другой, используя вызовы `select()` и `poll()`, а также интерфейс `epoll` (см. главу 55).

> При обращении к другим программным интерфейсам, описанным в данной главе, одновременное отслеживание одного или нескольких таймеров вместе с набором файловых дескрипторов требует определенных усилий.

Три новых системных вызова, входящих в этот программный интерфейс, по принципу своей работы аналогичны функциям `timer_create()`, `timer_settime()` и `timer_gettime()`, описанным в разделе 23.6.

Для начала рассмотрим системный вызов `timerfd_create()`, который создает новый объект таймера и возвращает ссылающийся на него файловый дескриптор.

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);
```

Возвращает файловый дескриптор при успешном завершении или `-1`, если произошла ошибка

Значение аргумента `clockid` может быть равно либо `CLOCK_REALTIME`, либо `CLOCK_MONOTONIC` (см. табл. 23.1).

В исходной реализации вызова `timerfd_create()` аргумент `flags` был зарезервирован на будущее и должен был быть равен `0`. Но, начиная с Linux 2.6.27, он стал поддерживать два флага.

- `TFD_CLOEXEC` — устанавливает флаг `FD_CLOEXEC` для нового файлового дескриптора. Может быть полезен по тем же причинам, что и флаг `O_CLOEXEC` для вызова `open()`, описанный в подразделе 4.3.1.

- `TFD_NONBLOCK` — устанавливает исходному описанию открытого файла флаг `O_NONBLOCK`, делая будущие операции чтения неблокирующими. Это позволяет избежать дополнительного вызова `fcntl()`, который дает тот же результат.

Закончив использовать таймер, созданный с помощью вызова `timerfd_create()`, мы должны закрыть связанный с ним файловый дескриптор, чтобы ядро могло освободить его ресурсы.

Системный вызов позволяет запускать и останавливать таймер, на который ссылается файловый дескриптор `fd`.

```
#include <sys/timerfd.h>

int timerfd_settime(int fd, int flags, const struct itimerspec *new_value,
                    struct itimerspec *old_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Аргумент `new_value` обозначает новые параметры таймера. Аргумент `old_value` может применяться для возвращения предыдущих параметров (подробности приводятся ниже, при описании вызова `timerfd_gettime()`). Если предыдущие параметры нас не интересуют, аргументу `old_value` можно присвоить `NULL`. Оба этих аргумента представляют собой структуры типа `itimerspec`, которые используются так же, как и в вызове `timer_settime()` (см. подраздел 23.6.2).

Аргумент `flags` аналогичен одноименному аргументу вызова `timer_settime()`. Он может быть равен либо `0` (в этом случае поле `new_value.it_value` интерпретируется относительно времени, полученного с помощью вызова `timerfd_settime()`), либо `TFD_TIMER_ABSTIME` (тогда `new_value.it_value` интерпретируется как абсолютное время, то есть отсчитывается с начальной позиции).

Системный вызов `timerfd_gettime()` возвращает интервал и время, оставшееся до срабатывания часов, на которые ссылается файловый дескриптор `fd`.

```
#include <sys/timerfd.h>

int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Как и в случае с вызовом `timer_gettime()`, интервал и время, оставшееся до срабатывания таймера, возвращаются внутри структуры `itimerspec`, на которую указывает аргумент `curr_value`. Поле `curr_value.it_value` определяет, сколько осталось до следующего срабатывания, даже если таймер был сделан абсолютным с помощью константы `TFD_TIMER_ABSTIME`. Если оба поля итоговой структуры `curr_value.it_value` равны `0`, значит, таймер выключен. Если оба поля итоговой структуры `curr_value.it_interval` равны `0`, это говорит о том, что таймер сработал лишь раз — в момент, указанный в `curr_value.it_value`.

Взаимодействие интерфейса timerfd с вызовами fork() и exec()

Во время вызова `fork()` дочерний процесс наследует копии файловых дескрипторов, созданных с помощью операции `timerfd_create()`. Эти дескрипторы ссылаются на те же объекты таймеров, которые используются родителем, а момент их срабатывания может быть прочитан любым из двух процессов.

Файловые дескрипторы, созданные с помощью операции `timerfd_create()`, сохраняются на протяжении работы вызова `exec()` (разве что они помечены флагом `FD_CLOEXEC`, как описано в разделе 27.4), а запущенные таймеры продолжат срабатывать даже после завершения этого вызова.

Чтение из файлового дескриптора timerfd

Запустив таймер с помощью вызова `timerfd_settime()`, мы можем использовать операцию `read()` для чтения информации о срабатываниях этого таймера из соответствующего файлового дескриптора. При этом буфер, который передается операции `read()`, должен быть достаточно большим, чтобы вместить 8-битное целое число (`uint64_t`).

Если с момента последнего изменения параметров таймера с помощью вызова `timerfd_settime()` или `last read()` он сработал один или несколько раз, операция `read()` немедленно завершается, а в буфер попадает количество произошедших срабатываний. Если срабатываний не было, чтение блокируется, пока таймер не сработает. Мы можем также установить дескриптору неблокирующий флаг `O_NONBLOCK`, воспользовавшись операцией `fcntl() F_SETFL` (см. раздел 5.3), чтобы при отсутствии срабатываний чтение не блокировалось, а сразу же завершалось ошибкой `EAGAIN`.

Как уже говорилось ранее, файловый дескриптор `timerfd` можно отслеживать с помощью вызовов `select()` и `poll()`, а также интерфейса `epoll`. При срабатывании таймера дескриптор становится доступным для чтения.

Пример программы

Пример использования программного интерфейса `timerfd` показан в листинге 23.8. Эта программа принимает два аргумента командной строки. Первый является обязательным; он обозначает начальное время и интервал таймера (он интерпретируется с помощью функции `itimerspecFromStr()`, представленной в листинге 23.6). Второй аргумент определяет максимальное количество срабатываний таймера, которого должна дождаться программа, прежде чем завершиться; если его опустить, будет использоваться значение `1`.

Программа создает и запускает таймер, обращаясь к вызовам `timerfd_create()` и соответственно `timerfd_settime()`. Затем она входит в цикл, считывая из файлового дескриптора уведомления о срабатываниях, пока их количество не достигнет заданного значения. После каждой операции `read()` программа выводит время, прошедшее с момента запуска таймера, количество обнаруженных срабатываний и общее число срабатываний на текущий момент.

В следующей сессии командной строки указывается два аргумента: начальное значение и интервал таймера, равные 1 секунде, а также максимальное количество срабатываний, равное 100.

```
$ ./demo_timerfd 1:1 100
1.000: expirations read: 1; total=1
2.000: expirations read: 1; total=2
3.000: expirations read: 1; total=3
Нажимаем Ctrl+Z, чтобы отправить программу в фоновый режим на несколько секунд
[1]+ Stopped            ./demo_timerfd 1:1 100
$ fg Resume program in foreground    Возвращаем программу в активное состояние
./demo_timerfd 1:1 100
14.205: expirations read: 11; total=14     Срабатывания с момента последнего чтения
15.000: expirations read: 1; total=15
16.000: expirations read: 1; total=16
Нажимаем Ctrl+C, чтобы завершить программу
```

Выше можно видеть, что во время пребывания программы в фоновом режиме произошло несколько срабатываний, каждое из которых было возвращено операцией `read()` после возобновления работы программы.

Листинг 23.8. Использование программного интерфейса timerfd

timers/demo_timerfd.c

```
#include <sys/timerfd.h>
#include <time.h>
#include <stdint.h>                     /* Определение uint64_t */
#include "itimerspec_from_str.h"        /* Объявляет itimerspecFromStr() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct itimerspec ts;
    struct timespec start, now;
    int maxExp, fd, secs, nanosecs;
    uint64_t numExp, totalExp;
    ssize_t s;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]] [max-exp]\n", argv[0]);

    itimerspecFromStr(argv[1], &ts);
    maxExp = (argc > 2) ? getInt(argv[2], GN_GT_0, "max-exp") : 1;
    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        errExit("timerfd_create");

    if (timerfd_settime(fd, 0, &ts, NULL) == -1)
        errExit("timerfd_settime");

    if (clock_gettime(CLOCK_MONOTONIC, &start) == -1)
        errExit("clock_gettime");

    for (totalExp = 0; totalExp < maxExp;) {

        /* Считываем данные о срабатываниях и выводим время, прошедшее с момента запуска
           таймера, и число срабатываний (зафиксированное на текущий момент и общее). */
        s = read(fd, &numExp, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            errExit("read");
        totalExp += numExp;

        if (clock_gettime(CLOCK_MONOTONIC, &now) == -1)
            errExit("clock_gettime");

        secs = now.tv_sec - start.tv_sec;
        nanosecs = now.tv_nsec - start.tv_nsec;
        if (nanosecs < 0) {
            secs--;
            nanosecs += 1000000000;
        }
        printf("%d.%03d: expirations read: %llu; total=%llu\n",
                secs, (nanosecs + 500000) / 1000000,
                (unsigned long long) numExp, (unsigned long long) totalExp);
    }

    exit(EXIT_SUCCESS);
}
```

timers/demo_timerfd.c

23.8. Резюме

Процесс может использовать вызовы `setitimer()` и `alarm()` для установки таймера, чтобы получить сигнал по прошествии реального или процессорного времени. Одно из применений таймеров заключается в ограничении времени блокирования системных вызовов.

Приложения, которым нужно приостанавливать выполнение на определенное количество времени, могут использовать для этого целый ряд функций перехода в режим сна.

Linux 2.6 реализует расширения стандарта POSIX.1b, которые предусматривают программный интерфейс для высокоточных часов и таймеров. Такие таймеры имеют ряд преимуществ перед своими традиционными аналогами (вызов `setitimer()` в UNIX). Они позволяют создавать несколько таймеров, выбирать сигнал, который будет доставляться при срабатывании, извлекать количество дополнительных срабатываний, чтобы определить, срабатывал ли таймер с момента последнего уведомления, а также указывать способ доставки уведомлений — с помощью функции в отдельном потоке или посредством сигнала.

Linux также предоставляет нестандартный программный интерфейс `timerfd`, аналогичный POSIX-интерфейсу. Он тоже содержит несколько вызовов для создания таймеров, но позволяет считывать уведомления о срабатывании с помощью файлового дескриптора. Этот дескриптор можно отслеживать с использованием вызовов `select()` и `poll()`, а также интерфейса `epoll`.

Дополнительная информация

Вместе с обоснованием необходимости отдельных функций в стандарте SUSv3 указаны полезные примечания относительно (стандартных) интерфейсов для таймеров и перехода в режим сна, описанных в этой главе. В книге [Gallmeister, 1995] обсуждаются часы и таймеры стандарта POSIX.1b.

23.9. Упражнения

23.1. Функция `alarm()` реализована в виде системного вызова внутри ядра Linux. Реализуйте ее самостоятельно на основе вызова `setitimer()`.

23.2. Попробуйте запустить программу из листинга 23.3 (`t_nanosleep.c`) в фоне и с 60-секундным интервалом сна; в то же время запустите следующую команду, чтобы послать фоновому процессу как можно больше сигналов `SIGINT`:

```
$ while true; do kill -INT pid; done
```

Вы должны обнаружить, что программа находится в режиме сна дольше, чем ожидалось. Поменяйте вызов `nanosleep()` на функции `clock_gettime()` (используя часы `CLOCK_REALTIME`) и `clock_nanosleep()` с флагом `TIMER_ABSTIME` (для этого упражнения требуется ядро не ниже версии 2.6). Проделайте те же действия с измененной программой и объясните, с чем связаны различия.

23.3. Напишите программу, которая демонстрирует, что передача вызову `timer_create()` аргумента `evp`, равного `NULL`, эквивалентно присвоению этому аргументу указателя на структуру `sigevent`, чьи поля `sigev_notify`, `sigev_signo` и `si_value.sival_int` равны `SIGEV_SIGNAL`, `SIGALRM` и соответственно идентификатору таймера.

23.4. Модифицируйте программу `ptmr_sigev_signal.c` из листинга 23.5, заменив обработчик сигнала вызовом `sigwaitinfo()`.

24 Создание процессов

В этой и следующих четырех главах мы рассмотрим создание и завершение процессов, а также то, как процесс может выполнить новую программу. Эта глава посвящена созданию процессов. Но прежде, чем углубляться в эту тему, начнем с краткого обзора главных системных вызовов, которые нам понадобятся.

24.1. Обзор вызовов fork(), exit(), wait() и execve()

Основными элементами этой и нескольких следующих глав являются системные вызовы `fork()`, `exit()`, `wait()` и `execve()`. Каждый из них имеет свои вариации, которые мы тоже не обойдем стороной. Пока что ознакомимся с кратким описанием этих четырех вызовов и узнаем, как они обычно используются в связке друг с другом.

- Системный вызов `fork()` позволяет одному процессу, родителю, создавать новый, дочерний процесс. Оба этих процесса являются (почти) идентичными: потомок получает копии родительского стека, данных, кучи, копии родительских сегментов стека X(см. раздел 6.3) и текста. Термин *fork* («вилка», «разветвление») стали применять потому, что родительский процесс как бы делится на две копии самого себя.
- Библиотечная функция `exit(status)` завершает процесс, делая все его ресурсы (память, дескрипторы открытых файлов и т. д.) доступными для последующего перераспределения ядром. Аргумент `status` — целое число, которое определяет код завершения процесса. Родительский процесс может извлечь этот код с помощью системного вызова `wait()`.

  Библиотечная функция exit() является оберткой вокруг системного вызова _exit(). В главе 25 вы узнаете разницу между этими двумя интерфейсами. А пока достаточно помнить о том, что обычно с помощью вызова exit() завершают работу только одного родителя или потомка, порожденного вызовом fork(); остальные процессы следует завершать с помощью вызова _exit().

- Системный вызов `wait(&status)` имеет два назначения. Во-первых, если работа потомка текущего процесса еще не была завершена путем вызова `exit()`, функция `wait()` приостанавливает выполнение родителя, пока не будет завершен один из его потомков. Во-вторых, код завершения потомка возвращается через аргумент функции `wait()`.
- Системный вызов `execve(pathname, argv, envp)` загружает в память процесса новую программу (расположенную в `pathname`, с аргументами `argv` и списком переменных среды `envp`). Текст существующей программы сбрасывается, а для новой программы заново создаются сегменты со стеком, данными и кучей. Эту операцию часто называют *выполнением* новой программы. Позже вы познакомитесь с несколькими функциями, которые являются обертками вокруг `execve()`, каждая из которых предоставляет полезную разновидность этого программного интерфейса. В случаях, когда эти разновидности не имеют принципиального значения, мы будем ссылаться на них с помощью обобщенного названия `exec()`, но имейте в виду, что системных вызовов или библиотечных функций с таким именем не существует.

В некоторых других операционных системах возможности функций `fork()` и `exec()` объединены в одну операцию — *порождение (spawn)*; она создает новый процесс, который

выполняет заданную программу. Однако подход, используемый в системах UNIX, обычно является более простым и изящным. Разделение этих двух шагов упрощает программные интерфейсы (системный вызов `fork()` *не* принимает аргументы) и делает программу более гибкой, позволяя ей выполнять определенные действия между этими двумя этапами. Кроме того, вызов `fork()` часто имеет смысл делать без последующего вызова `exec()`.

> Стандарт SUSv3 предусматривает дополнительную функцию posix_spawn(), которая объединяет возможности fork() и exec(). В системе Linux она и еще несколько связанных программных интерфейсов, описанных в стандарте SUSv3, реализована в библиотеке glibc. Функция posix_spawn() позволяет создавать переносимые приложения с поддержкой аппаратных архитектур, которые не предоставляют механизм файла подкачки или блоки управления памятью (что характерно для многих встраиваемых систем). В рамках таких архитектур реализация традиционного вызова fork() является либо затруднительной, либо невозможной в принципе.

На рис. 24.1 показано, как вызовы `fork()`, `exit()`, `wait()` и `execve()` обычно используются вместе (эта диаграмма изображает пошаговые действия командной оболочки, выполняющей команду: создается непрерывный цикл, в котором оболочка считывает команду, обрабатывает ее различными способами, после чего создает дочерний процесс для ее выполнения).

Применение вызова `execve()`, показанное на этой диаграмме, не является обязательным. Иногда имеет смысл позволить потомку продолжить выполнение программы родителя. В любом случае выполнение дочернего процесса в конечном счете завершается вызовом `exit()` (или передачей сигнала) и возвращением кода завершения, доступным родителю через функцию `wait()`. Вызов `wait()` тоже необязателен. Родитель может просто игнорировать своего потомка и продолжать работу. Однако позже мы увидим, что использование функции `wait()` обычно является желательным и выполняется внутри обработчика сигнала `SIGCHLD`, который генерируется ядром для родителя, когда один из его дочерних процессов завершается (по умолчанию сигнал `SIGCHLD` игнорируется, поэтому в диаграмме сказано, что его доставка является опциональной).

24.2. Создание нового процесса: fork()

Во многих приложениях создание нескольких процессов может быть удобным способом разделения задач. Например, сетевой сервер может следить за входящими клиентскими запросами, обрабатывая каждый из них в отдельном дочернем процессе; при этом процесс сервера не прекращает отслеживать дальнейшие подключения со стороны клиента. Разделение задач таким способом упрощает проектирование приложений. Это также улучшает параллелизм (то есть одновременно можно обрабатывать больше задач или запросов).

Системный вызов `fork()` создает новый процесс *(потомок)*, который является почти полной копией вызывающего процесса, *родителя*.

```
#include <unistd.h>

pid_t fork(void);
```

В родителе: возвращает идентификатор потомка при успешном завершении или `-1` при ошибке; в успешно созданном потомке всегда возвращает `0`.

Ключевым моментом в понимании вызова `fork()` является тот факт, что после завершения его работы мы получаем два процесса, каждый из которых продолжает выполнение с момента возврата из этого вызова.

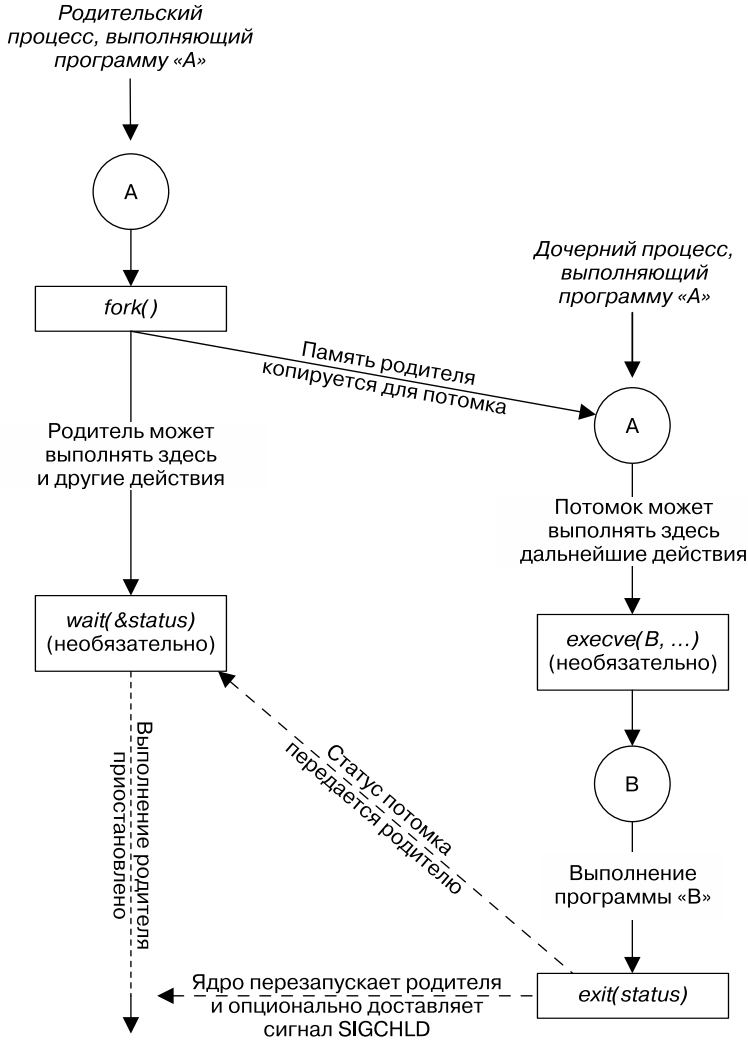


Рис. 24.1. Обор вызовов fork(), exit(), wait() и execve()

Оба процесса выполняют один и тот же программный код, но обладают разными копиями сегментов стека, данных и кучи. Сегменты потомка вначале полностью дублируют соответствующие части памяти своего родителя. Но после завершения вызова `fork()` каждый из процессов может самостоятельно изменять переменные в своих сегментах, не влияя на другой процесс.

Распознавать процессы внутри кода программы можно с помощью значения, возвращенного функцией `fork()`. В случае с родителем это значение равно идентификатору только что созданного потомка. Это полезно, поскольку родитель может создать несколько дочерних процессов, которые ему придется отслеживать (с помощью вызова `wait()` или одной из его разновидностей). В случае с потомком возвращается `0`. При необходимости дочерний процесс может получить свой собственный идентификатор или идентификатор своего родителя, используя функции `getpid()` и соответственно `getppid()`.

Если новый процесс не удается создать, вызов fork() возвращает -1. Одной из причин этого может быть превышение пользователем или всей системой в целом ограничения на определенный вид ресурсов (RLIMIT_NPROC, описанное в разделе 36.3), а именно на количество создаваемых процессов.

Иногда вызов fork() используют следующим образом:

```
pid_t childPid;   /* Используется в родителе после успешного вызова fork()
                     для записи идентификатора потомка */
switch (childPid = fork()) {
case -1:          /* Вызов fork() завершился неудачей */
    /* Обработка ошибки */

case 0:           /* Ветка для потомка после успешного вызова fork() */
    /* Выполнение действий, связанных с потомком */

default:          /* Ветка для родителя, после успешного вызова fork() */
    /* Выполнение действий, связанных с родителем */
}
```

Следует понимать, что после вызова fork() невозможно сказать, какой из двух процессов первым получит от планировщика ресурсы ЦП. В плохо написанных программах такая неопределенность может привести к ошибке, известной под названием *«состояние гонки»* (см. раздел 24.4).

Использование функции fork() продемонстрировано в листинге 24.1. Описанная в нем программа создает дочерний процесс, который изменяет унаследованные им в результате вызова fork() глобальные и автоматические переменные.

С помощью функции sleep() (в нашем коде она вызывается родителем) программа позволяет потомку получить ресурсы процессора раньше родителя; благодаря этому потомок успевает выполнить свою работу и завершиться до того, как выполнение продолжит родительский процесс. Такое использование вызова sleep() не дает никаких гарантий; более надежный подход будет рассмотрен в разделе 24.5.

Запустив программу, описанную в листинге 24.1, вы увидите следующий вывод:

```
$ ./t_fork
PID=28557  (child)  idata=333  istack=666
PID=28556  (parent) idata=111  istack=222
```

Вышеприведенный вывод показывает, что во время вызова fork() дочерний процесс получает собственную копию сегментов стека и данных и может изменять переменные в этих сегментах, не влияя на своего родителя.

Листинг 24.1. Использование функции fork()

procexec/t_fork.c

```
#include "tlpi_hdr.h"

static int idata = 111;    /* Выделена в сегменте данных */

int
main(int argc, char *argv[])
{
    int istack = 222;     /* Выделена в сегменте стека */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");
```

```
    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);    /* Даем потомку шанс выполниться */
        break;
    }

    /* Здесь выполняются и потомок и родитель */

    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
          (childPid == 0) ? "(child) " : "(parent)", idata, istack);

    exit(EXIT_SUCCESS);
}
```

procexec/t_fork.c

24.2.1. Совместный доступ к файлу родителя и потомка

Когда выполняется вызов `fork()`, потомок получает копии всех файловых дескрипторов родителя. Эти копии создаются в той же манере, как работает функция `dup()`; это означает, что соответствующие дескрипторы в родительском и дочернем процессах указывают на один и тот же открытый файл. Как мы уже видели в разделе 5.4, дескриптор открытого файла содержит его текущее смещение (образовавшееся во время редактирования с помощью функций `read()`, `write()` и `lseek()`) и флаги состояния (установленные вызовом `open()` и измененные операцией `fcntl() F_SETFL`). Как следствие, эти атрибуты открытого файла являются общими для родителя и потомка. Например, если дочерний процесс обновляет смещение, это изменение доступно его родителю посредством соответствующего дескриптора.

Тот факт, что после вызова `fork()` эти атрибуты являются общими для родителя и потомка, продемонстрирован в программе из листинга 24.2. Эта программа открывает временный файл, используя функцию `mkstemp()`, и затем вызывает `fork()` для создания дочернего процесса. Потомок изменяет смещение открытого файла и флаги его состояния, после чего завершает работу. Затем родитель извлекает смещение и флаги файла, чтобы удостовериться в том, что он может видеть изменения, внесенные его потомком. Запустив программу, мы получим следующий вывод:

```
$ ./fork_file_sharing
File offset before fork(): 0
O_APPEND flag before fork() is: off
Child has exited
File offset in parent: 1000
O_APPEND flag in parent is: on
```

Объяснение того, зачем мы приводим значение, возвращаемое функцией lseek(), к типу long long, ищите в разделе 5.10.

Листинг 24.2. Совместный доступ к смещению файла и флагам его состояния из родителя и потомка

procexec/fork_file_sharing.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    int fd, flags;
    char template[] = "/tmp/testXXXXXX";

    setbuf(stdout, NULL);    /* Отключаем буферизацию стандартного вывода */

    fd = mkstemp(template);
    if (fd == -1)
        errExit("mkstemp");

    printf("File offset before fork(): %lld\n", (long long) lseek(fd, 0, SEEK_CUR));
    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");
    printf("O_APPEND flag before fork() is: %s\n", (flags & O_APPEND) ? "on" : "off");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:    /* Потомок изменяет сдвиг файла и флаги его состояния */
        if (lseek(fd, 1000, SEEK_SET) == -1)
            errExit("lseek");

        flags = fcntl(fd, F_GETFL);    /* Извлекаем текущие флаги */
        if (flags == -1)
            errExit("fcntl - F_GETFL");
        flags |= O_APPEND;             /* Устанавливаем флаг O_APPEND */
        if (fcntl(fd, F_SETFL, flags) == -1)
            errExit("fcntl - F_SETFL");
        _exit(EXIT_SUCCESS);

    default:    /* Родитель может видеть изменения, внесенные потомком */
        if (wait(NULL) == -1)
            errExit("wait");          /* Ждем завершения работы потомка */
        printf("Child has exited\n");

        printf("File offset in parent: %lld\n", (long long) lseek(fd, 0, SEEK_CUR));

        flags = fcntl(fd, F_GETFL);
        if (flags == -1)
            errExit("fcntl - F_GETFL");
        printf("O_APPEND flag in parent is: %s\n", (flags & O_APPEND) ? "on" : "off");
        exit(EXIT_SUCCESS);
    }
}
```

procexec/fork_file_sharing.c

Совместный доступ к атрибутам файла из родительского и дочернего процессов часто бывает полезен. Например, если родитель и потомок одновременно осуществляют запись в файл, доступ к его смещению позволяет избежать перезаписи процессами вывода друг друга. Однако это не предотвращает случайное смешивание записываемых ими данных. Если такое поведение вас не устраивает, вам придется реализовать некую форму синхронизации между процессами. Например, родитель может использовать системный вызов `wait()`, чтобы дождаться, пока его потомок не завершит работу. Такой подход применяется

в командной оболочке, чтобы приглашение командной строки выводилось только после завершения работы команды в дочернем процессе (за исключением тех случаев, когда пользователь намеренно запускает команду в фоновом режиме, указывая знак амперсанда после ее имени).

Если не требуется такое совместное использование атрибутов файлов, приложение должно быть спроектировано так, чтобы родитель и потомок после вызова `fork()` использовали разные дескрипторы, сразу же закрывая те из них, которые были получены из другого процесса (если один из процессов выполняет вызов `exec()`, вам также может пригодиться флаг `FD_CLOEXEC`, описанный в разделе 27.4). Эти шаги показаны на рис. 24.2.

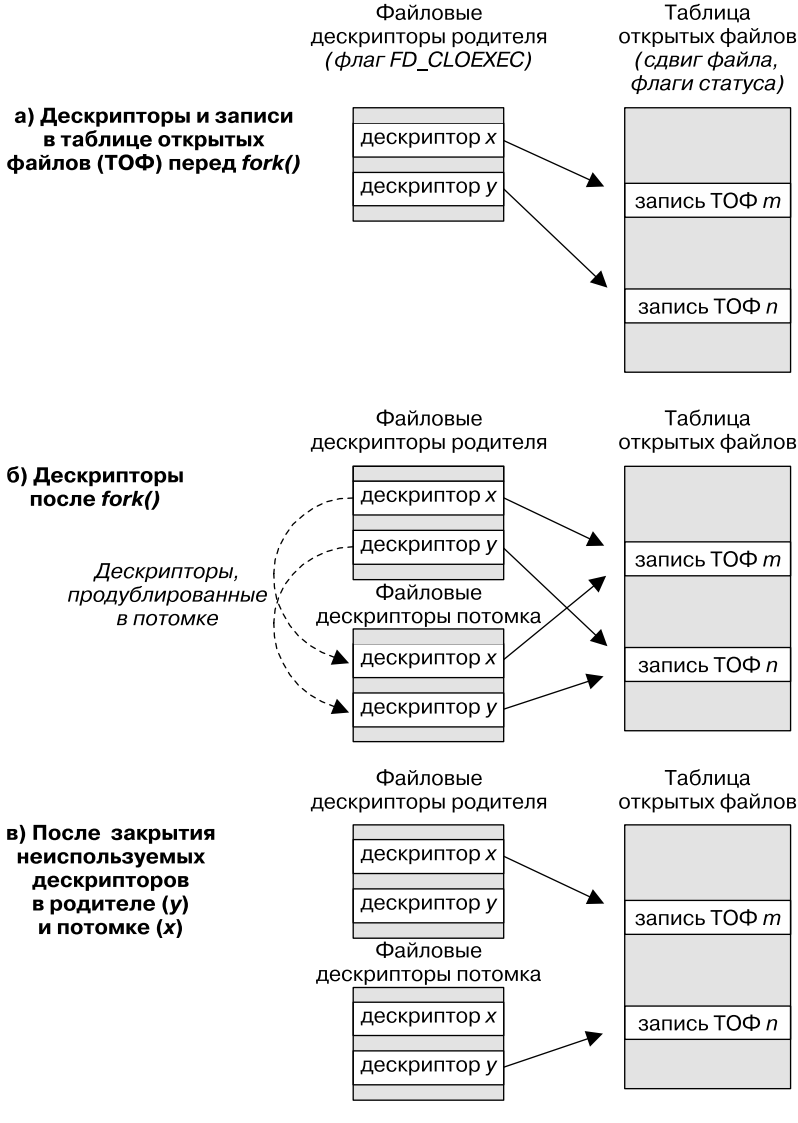


Рис. 24.2. Дублирование файловых дескрипторов во время вызова fork() и закрытие тех из них, которые не используются

24.2.2. Семантика памяти вызова fork()

В принципе, вызов `fork()` можно рассматривать как создание копий родительских сегментов с текстом, данными, кучей и стеком (в некоторых ранних реализациях системы UNIX и в самом деле происходило такое дублирование: образ нового процесса создавался путем копирования памяти родителя в область подкачки, из которого получался дочерний процесс; родитель при этом сохранял свою собственную память). Однако обычное копирование страниц виртуальной памяти родителя в новый дочерний процесс было бы расточительством по целому ряду причин — например, за вызовом `fork()` часто следует функция `exec()`, которая заменяет код процесса новой программой и повторного инициализирует сегменты данных, кучи и стека. Большинство современных реализаций UNIX, в том числе и Linux, пытаются избежать такого избыточного копирования, используя две методики.

- Ядро делает текстовый сегмент каждого процесса доступным только для чтения, чтобы они не могли изменить свой собственный код. Это означает, что родитель и потомок могут иметь общий текстовый сегмент. Системный вызов `fork()` создает текстовый сегмент потомка путем построения записей в таблице страниц памяти для каждого отдельного процесса; каждая запись ссылается на блок страницы физической памяти, который уже используется родителем.
- Для страниц родительского процесса в сегментах с данными, кучей и стеком ядро использует методику, известную как *копирование при записи* (описание ее реализации ищите в книгах [Bach, 1986] и [Bovet & Cesati, 2005]). Изначально ядро подготавливает все для того, чтобы записи таблицы страниц для этих сегментов указывали на те же физические страницы, что и записи родителя, и затем делает сами страницы доступными только для чтения. После вызова `fork()` ядро перехватывает любые попытки родителя или потомка изменить любую из этих страниц, создавая ее копию. Эта новая страница назначается процессу, инициировавшему изменение, а соответствующая запись в таблице другого процесса корректируется должным образом. С этого момента родитель или потомок может изменять свою частную копию страницы, не влияя на страницу другого процесса. Методика копирования при записи проиллюстрирована на рис. 24.3.

Контроль изменения объема занимаемой процессом памяти

Мы можем сочетать использование вызовов `fork()` и `wait()`, чтобы контролировать изменение объема занимаемой процессом памяти. Нас интересует диапазон задействованных процессом виртуальных страниц, который изменяется в результате воздействия таких факторов, как подстройка стека при входе и выходе в функции, вызов `exec()` и, что особенно важно в контексте этого раздела, вызовы `malloc()` и `free()`, приводящие к изменению кучи.

Представьте, что мы поместили некую функцию `func()` между вызовами `fork()` и `wait()`, как это сделано в листинге 24.3. После выполнения этого кода мы знаем, что с момента вызова `func()` память родителя не меняется, поскольку все возможные изменения происходят в дочернем процессе. Это может пригодиться по следующим причинам.

- Если мы знаем, что функция `func()` приводит к утечкам памяти или чрезмерной фрагментации кучи, данный подход устранит проблему (это может быть единственно возможное решение, если у нас нет доступа к исходному коду `func()`).
- Допустим, у нас есть алгоритм, который выделяет память во время анализа дерева (это может быть, к примеру, игровая программа, которая рассчитывает диапазон возможных ходов и реакций на них). Мы можем написать код с применением вызова `free()`, чтобы освободить всю выделенную память. Однако в некоторых случаях проще будет воспользоваться описанной выше методикой; это позволит нам сделать откат к начальной точке, оставляя память вызывающего процесса (родителя) без изменений.

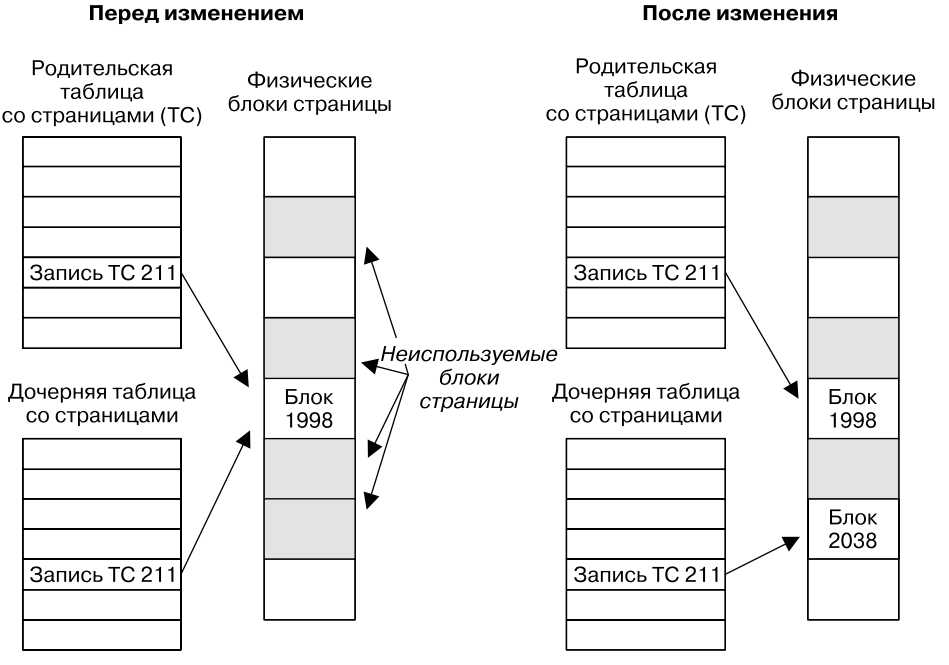


Рис. 24.3. Таблицы со страницами памяти до и после изменения общей страницы, копируемой при записи

В реализации, показанной в листинге 24.3, результат выполнения функции `func()` должен быть выражен в 8 битах, которые посредством вызова `exit()` передаются после завершения дочернего процесса в родительский, вызвавший `wait()`. При необходимости можно передать больший объем данных, наладив межпроцессное взаимодействие на основе файла, конвейера или как-то иначе.

Листинг 24.3. Вызов функции без изменения состояния памяти процесса

Из файла **procexec/footprint.c**

```
pid_t childPid;
int status;

childPid = fork();
if (childPid == -1)
    errExit("fork");

if (childPid == 0)      /* Потомок вызывает func() и использует */
    exit(func(arg));    /* возвращенное значение в качестве */
                        /* кода завершения */

/* Родитель ждет завершения работы потомка.
   Он может узнать результат выполнения
   функции func(), прочитав ее 'status'. */

if (wait(&status) == -1)
    errExit("wait");
```

Из файла **procexec/footprint.c**

24.3. Системный вызов vfork()

Ранние реализации BSD входили в число операционных систем, в которых вызов fork() выполнял полноценное дублирование данных, кучи и стека родителя. Как уже было замечено выше, это довольно расточительно, особенно если за вызовом `fork()` следует функция `exec()`. В связи с этим в более поздних версиях BSD-систем появился системный вызов `vfork()`, который отличается гораздо большей эффективностью, хотя и имеет немного другую (на самом деле довольно странную) семантику. В современных системах UNIX вызов `fork()` выполняет копирование при записи, что значительно более экономно по сравнению со старыми реализациями; благодаря этому необходимость в функции `vfork()` во многом отпала. Тем не менее в случае, если вам нужна самая быстрая операция создания дочерних процессов, операционная система Linux (как и многие другие реализации UNIX) тоже поддерживает вызов `vfork()` с семантикой, принятой в BSD-системах. Но, поскольку эта необычная семантика может привести к неочевидным программным ошибкам, `vfork()` рекомендуется избегать; исключение составляют те редкие случаи, когда он обеспечивает существенный прирост производительности.

Как и `fork()`, вызов `vfork()` применяется вызывающим процессом для создания нового потомка. Однако он специально был спроектирован так, чтобы его можно было использовать в программах, в которых дочерний процесс сразу же делает вызов `exec()`.

```
#include <unistd.h>

pid_t vfork(void);
```

В родителе: возвращает идентификатор потомка при успешном завершении или `-1` при ошибке; в успешно созданном потомке всегда возвращает `0`

Функция `vfork()` имеет два отличия от системного вызова `fork()`, которые делают ее более эффективной.

- Страницы виртуальной памяти или таблицы страниц не дублируются для дочернего процесса. Вместо этого потомок и родитель делят память до тех пор, пока один из них не выполнит успешный вызов `exec()` или `_exit()`, чтобы завершить работу.
- Выполнение родительского процесса приостанавливается до тех пор, пока потомок не вызовет `exec()` или `_exit()`.

Эти два свойства имеют важные последствия. Поскольку потомок использует память родителя, любые изменения, вносимые им в сегменты данных, кучи или стека, будут доступны самому родительскому процессу, как только он возобновит работу. Более того, выполнение потомком функции между вызовами `vfork()` и `exec()` (или `_exit()`) тоже коснется родителя. Это похоже на пример, описанный в разделе 6.8, когда вызов `longjmp()` выполняет переход внутрь функции, которая уже вернула управление (завершилась). Чаще всего это приводит к ошибке сегментации (`SIGSEGV`).

Есть несколько вещей, которые потомок может делать между вызовами `vfork()` и `exec()`, не затрагивая при этом своего родителя. К таковым относится операция открытия дескриптора файла (не путать с потоками stdio (библиотеки)). Поскольку таблица файловых дескрипторов каждого процесса хранится в пространстве ядра (см. раздел 5.4) и дублируется во время вызова `vfork()`, родитель не будет знать об операциях с дескрипторами файлов, выполняемых потомком.

SUSv3 гласит, что поведение программы является неопределенным, если она: а) изменяет любые данные, кроме переменной типа pid_t, применяемой для хранения значения, возвращаемого вызовом vfork(); б) возвращается из функции, в которой произошел вызов vfork(); в) вызывает любую другую функцию до успешного выполнения _exit() или exec().

При рассмотрении системного вызова clone() в разделе 28.2 мы увидим, что потомок, созданный с помощью функций fork() или vfork(), получает собственные копии некоторых других атрибутов процесса.

Семантика функции `vfork()` указывает на то, что после ее вызова потомок гарантированно получит ресурсы центрального процессора раньше родителя. В разделе 24.2 отмечалось, что вызов `fork()` не дает такой гарантии, поэтому после него выполнение может перейти как к родительскому, так и к дочернему процессу.

В листинге 24.4 продемонстрированы обе семантические особенности, которые отличают вызов `vfork()` от `fork()`: потомок получает доступ к памяти родителя, а родитель приостанавливает работу, пока потомок не завершит выполнение или не вызовет функцию `exec()`. Запустив эту программу, мы увидим следующий вывод:

```
$ ./t_vfork
Child executing        Хотя потомок приостановился, родитель не продолжил работу
Parent executing
istack=666
```

Последняя строчка вывода показывает изменение, внесенное потомком в переменную `istack`, которая принадлежит родителю.

Листинг 24.4. Использование вызова vfork()

procexec/t_vfork.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int istack = 222;

    switch (vfork()) {
    case -1:
        errExit("vfork");

    case 0:          /* Первым выполняется потомок (в пространстве памяти родителя) */
        sleep(3);    /* Даже если процесс остановится на некоторое время,
                        родитель все равно не продолжит работу */
        write(STDOUT_FILENO, "Child executing\n", 16);
        istack *= 3;    /* Это изменение будет доступно родителю */
        _exit(EXIT_SUCCESS);

    default:    /* Родитель блокируется, пока существует его потомок */
        write(STDOUT_FILENO, "Parent executing\n", 17);
        printf("istack=%d\n", istack);
        exit(EXIT_SUCCESS);
    }
}
```

procexec/t_vfork.c

Если не брать во внимание случаи, когда скорость является крайне важным фактором, в новых приложениях следует использовать вызов `fork()` вместо `vfork()`. Во-первых,

в большинстве современных UNIX-систем он реализован с поддержкой копирования при записи, что делает его почти таким же быстрым, и во-вторых, он позволяет избежать необычного поведения, свойственного вызову `vfork()` и описанного выше. Сравнение производительности вызовов `fork()` и `vfork()` будет продемонстрировано в разделе 28.3.

В стандарте SUSv3 вызов `vfork()` помечен как устаревший, а в SUSv4 он и вовсе не попал. Его спецификация в SUSv3 оставляет многие подробности неопределенными, позволяя реализовать его в виде вызова `fork()`. В таком случае семантика, принятая в системах BSD, не сохраняется. В некоторых UNIX-системах (в частности, в Linux вплоть до версии ядра 2.0) функция `vfork()` всего лишь вызывает `fork()`.

Везде, где используется `vfork()`, сразу после нее обычно должен следовать вызов `exec()`. Если `exec()` заканчивается неудачно, дочерний процесс должен завершить работу с помощью функции `_exit()` (потомок, созданный функцией `vfork()`, не должен прибегать к вызову `exit()`, потому что это приведет к сбросу и закрытию буферов stdio; мы рассмотрим этот момент более подробно в разделе 25.4).

Применение вызова `vfork()` в других ситуациях (в частности, когда его необычная семантика используется для совместного доступа к памяти и планировании работы процесса) с большой долей вероятности сделает программу непереносимой на другие платформы, особенно если там он реализован просто как вызов `fork()`.

24.4. Состояние гонки после вызова fork()

После вызова `fork()` невозможно точно сказать, какой процесс — родительский или дочерний — первым получит доступ к ЦП (в многопроцессорных системах это может произойти одновременно). Приложения, которые явно или неявно полагаются на определенный порядок выполнения для получения корректных результатов, подвержены ошибкам, связанным с *состоянием гонки* (см. раздел 5.1). Такие ошибки может быть трудно выявить, потому что их возникновение зависит от того, как ядро планирует выполнение процессов под той или иной нагрузкой.

Эту неопределенность можно продемонстрировать на примере программы, представленной в листинге 24.5. Код программы входит в цикл и использует вызов `fork()` для создания множества потомков. После каждого такого вызова родитель и потомок выводят сообщение, содержащее номер итерации цикла и строку, которая указывает, родительский это процесс или дочерний. Например, если попросить программу создать всего лишь одного потомка, можно получить следующий вывод:

```
$ ./fork_whos_on_first 1
0 parent
0 child
```

С помощью этой программы можно создать большое количество потомков и затем проанализировать результат, чтобы увидеть, кто первым выводит свое сообщение на каждом этапе — родитель или потомок. Анализ результатов на системе Linux/x86-32 2.2.19 при 1 миллионе дочерних процессов показал, что родитель выводит свое сообщение первым во всех случаях, кроме 332 (то есть в 99,97 % всех случаев).

> Результаты выполнения программы из листинга 24.5 были проанализированы с помощью скрипта procexec/fork_whos_on_first.count.awk, который можно найти в примерах исходного кода, поставляемых вместе с этой книгой.

Из этих результатов можно было бы предположить, что в системе Linux 2.2.19 после вызова `fork()` выполнение всегда переходит к родительскому процессу. Те 0,03 % случаев,

когда потомок выводит свое сообщение первым, возникают из-за того, что родителю иногда не хватает выделенного ему кванта процессорного времени, чтобы вовремя напечатать сообщение. Иными словами, если бы в этом примере мы полагались на то, что после вызова `fork()` родитель всегда выполняется первым, в целом все бы шло по плану, однако в одном из 3000 случаев мы бы получали неожиданный результат. Естественно, если бы родителю приходилось выполнять больший объем работы, прежде чем передавать выполнение потомку, вероятность неправильного поведения была бы более высокой. Отладка таких ошибок в масштабных программах может оказаться довольно сложной.

Листинг 24.5. Состязание родителя и потомка за возможность вывести сообщение после вызова fork()

procexec/fork_whos_on_first.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numChildren, j;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-children]\n", argv[0]);

    numChildren = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-children") : 1;

    setbuf(stdout, NULL);  /* Делаем стандартный вывод небуферизированным */

    for (j = 0; j < numChildren; j++) {
        switch (childPid = fork()) {
        case -1:
            errExit("fork");

        case 0:
            printf("%d child\n", j);
            _exit(EXIT_SUCCESS);

        default:
            printf("%d parent\n", j);
            wait(NULL);    /* Ждем завершения работы потомка */
            break;
        }
    }

    exit(EXIT_SUCCESS);
}
```

procexec/fork_whos_on_first.c

В результате мы не можем надеяться на определенный порядок выполнения для родителя и потомка после вызова `fork()`. Но если этот порядок нужно обеспечить, нам придется использовать какого-то рода синхронизацию. Несколько таких методик, включая семафоры, блокирование файлов и обмен сообщениями между процессами с помощью конвейеров, будет описано позже в этой главе. В следующем же разделе мы рассмотрим другой подход, который подразумевает применение сигналов.

24.5. Синхронизация с помощью сигналов как способ избежать состояния гонки

Бывает, что после вызова `fork()` одному из процессов нужно подождать, пока не завершится другой. В этом случае после окончания работы активный процесс может послать сигнал, ожидаемый другим процессом.

Этот подход представлен в листинге 24.6. В данном примере подразумевается, что это родитель должен ждать, пока потомок не выполнит какое-то действие. Вызовы, связанные с сигналами, можно поменять местами, если необходимо, чтобы потомок ждал родителя. Можно даже сделать так, чтобы оба процесса многократно обменивались сигналами друг с другом, координируя свои действия, хотя на практике такая координация чаще всего делается с помощью семафоров, блокирования файлов или передачи сообщений.

> В книге [Stevens & Rago, 2005] рекомендуется инкапсулировать блокирование, отправку и перехват сигнала в стандартный набор функций для синхронизации процессов. Преимущество такой инкапсуляции заключается в том, что позже при желании сигналы можно будет заменить другим IPC-механизмом.

Обратите внимание, что в листинге 24.6 сигнал синхронизации (`SIGUSR1`) блокируется до вызова `fork()`. Если бы родитель попытался заблокировать сигнал после этого вызова, он бы оказался подвержен тому самому состоянию гонки, которого мы пытаемся избежать (в данной программе мы исходим из того, что значение маски сигналов в дочернем процессе не играет роли; при необходимости сигнал `SIGUSR1` можно разблокировать в потомке после вызова `fork()`).

Нижепреведенный журнал сессии командной строки показывает результат работы программы из листинга 24.6:

```
$ ./fork_sig_sync
[17:59:02 5173] Child started - doing some work
[17:59:02 5172] Parent about to wait for signal
[17:59:04 5173] Child about to signal parent
[17:59:04 5172] Parent got signal
```

Листинг 24.6. Использование сигналов для синхронизации действий процессов

procexec/fork_sig_sync.c

```
#include <signal.h>
#include "curr_time.h"    /* Объявление функции currTime() */
#include "tlpi_hdr.h"

#define SYNC_SIG SIGUSR1  /* Сигнал синхронизации */

static void   /* Обработчик сигнала, ничего не делает, просто завершается */
handler(int sig)
{
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    sigset_t blockMask, origMask, emptyMask;
    struct sigaction sa;
```

```
    setbuf(stdout, NULL);   /* Отключаем буферизацию стандартного вывода */

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SYNC_SIG); /* Блокируем сигнал */
    if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
        errExit("sigprocmask");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = handler;
    if (sigaction(SYNC_SIG, &sa, NULL) == -1)
        errExit("sigaction");

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0: /* Потомок */

    /* Здесь потомок выполняет некоторые необходимые действия... */

        printf("[%s %ld] Child started - doing some work\n",
               currTime("%T"), (long) getpid());
        sleep(2);    /* Имитируем работу */

        /* И затем сигнализируем родителю о ее завершении */

        printf("[%s %ld] Child about to signal parent\n",
               currTime("%T"), (long) getpid());
        if (kill(getppid(), SYNC_SIG) == -1)
            errExit("kill");

        /* Теперь потомок может заняться другими делами... */

        _exit(EXIT_SUCCESS);

    default: /* Родитель */
        /* Родитель может выполнить здесь какую-то работу, затем он ждет,
           когда потомок выполнит необходимое действие */

        printf("[%s %ld] Parent about to wait for signal\n",
               currTime("%T"), (long) getpid());
        sigemptyset(&emptyMask);
        if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        printf("[%s %ld] Parent got signal\n", currTime("%T"),
               (long) getpid());

        /* При необходимости возвращаем маску сигнала в ее изначальное состояние */

        if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
            errExit("sigprocmask");

      /* Родитель возобновляет работу и может заняться другими делами... */

        exit(EXIT_SUCCESS);
    }
}
```

procexec/fork_sig_sync.c

24.6. Резюме

Системный вызов `fork()` создает новый (дочерний) процесс, делая почти полную копию вызывающего процесса (родителя). Системный вызов `vfork()` является более эффективной версией `fork()`, но из-за необычной семантики его лучше избегать; его особенность заключается в том, что потомок использует память родителя, пока не вызовет `exec()` или не завершит работу, а сам родитель при этом приостанавливает выполнение.

После вызова `fork()` не всегда можно предугадать, кто первый получит ресурсы центрального процессора (или процессоров) — родитель или потомок. Программы, которые в такой ситуации зависят от порядка выполнения, подвержены ошибке, известной под названием *«состояние гонки»*. Поскольку возникновение таких ошибок зависит от внешних факторов (например, от загрузки системы), иногда их бывает сложно искать и отлаживать.

Дополнительная информация

Подробности реализации вызовов `fork()`, `execve()`, `wait()` и `exit()` в системах UNIX можно найти в книгах [Bach, 1986] и [Goodheart & Cox, 1994]. В книгах [Bovet & Cesati, 2005] и [Love, 2010] описаны особенности создания и завершения процессов в системе Linux.

24.7. Упражнения

24.1. Сколько новых процессов появится в результате выполнения программой следующей последовательности вызовов fork() (учитывая, что все они выполнятся успешно)?

```
fork();
fork();
fork();
```

24.2. Напишите программу, которая наглядно показывает, что после вызова `vfork()` дочерний процесс может закрыть файловый дескриптор (например, дескриптор под номером `0`), без влияния на соответствующий дескриптор родителя.

24.3. Каким образом можно получить дамп памяти процесса в заданном месте программы, позволив исходному процессу продолжать выполнение (предполагается, что исходный код программы можно изменять)?

24.4. Поэкспериментируйте с программой из листинга 24.5 (`fork_whos_on_first.c`) на других реализациях UNIX, чтобы определить, как в них планируется выполнение родительского и дочернего процессов после вызова `fork()`.

24.5. Представьте, что дочерний процесс в программе из листинга 24.6 тоже должен ждать, пока родитель не завершит какие-то действия. Какие изменения нужно будет внести в программу, чтобы этого добиться?

25 Завершение работы процесса

В этой главе рассматривается процедура завершения процесса. Мы начнем с описания вызовов `exit()` и `_exit()`, специально для этого предназначенных. Затем обсудим использование обработчиков выхода для автоматического освобождения ресурсов при вызове `exit()`. В заключение будут продемонстрированы некоторые способы взаимодействия между буферами stdio (стандартного ввода/вывода) и вызовами `fork()` и `exit()`.

25.1. Завершение процесса: вызовы _exit() и exit()

Обычно процесс можно завершить двумя способами. Во-первых, это *аварийное* завершение, вызванное передачей сигнала, чьим действием по умолчанию является остановка работы процесса (со сбросом дампа памяти или без); этот вариант описан в разделе 20.1. Возможно также *нормальное* завершение с помощью системного вызова `_exit()`.

```
#include <unistd.h>

void _exit(int status);
```

Аргумент `status`, передаваемый вызову `_exit()`, определяет код завершения процесса, который доступен его родителю посредством вызова `wait()`. Этот аргумент имеет тип `int`, однако родительскому процессу доступны только последние 8 бит. Код `0` принято считать признаком успешного завершения, а любые другие значения сигнализируют о том, что процесс окончил работу с проблемами. Четких правил по интерпретации ненулевых значений не существует; разные приложения следуют своим собственным соглашениям, которые должны быть описаны в их документации. Стандарт SUSv3 определяет две константы, `EXIT_SUCCESS` (`0`) и `EXIT_FAILURE` (`1`), которые используются в большинстве примеров из этой книги.

Процесс всегда завершается успешно, если для этого применяется вызов `_exit()` (то есть `_exit()` никогда не возвращает значения).

> Через аргумент status вызова _exit() родителю можно передать любое значение от 0 до 255, однако номера больше 128 могут вызвать неправильную работу скриптов командной строки. Дело в том, что при завершении программы с помощью сигнала командная оболочка сигнализирует об этом, присваивая переменной $? значение 128 плюс номер самого сигнала; это значение невозможно отличить от аналогичного, переданного в результате вызова _exit().

Программы обычно не вызывают `_exit()` напрямую, используя вместо этого библиотечную функцию `exit()`, которая выполняет различные предварительные действия.

```
#include <stdlib.h>

void exit(int status);
```

Функция `exit()` выполняет следующие приготовления перед вызовом `_exit()`.

- Вызываются обработчики выхода (функции, зарегистрированные с помощью вызовов `atexit()` и `on_exit()`); в порядке, обратном их регистрации (см. раздел 25.3).
- Сбрасываются буферы потоков stdio.
- Выполняется системный вызов `_exit()` со значением, переданным в аргументе `status`.

В отличие от вызова _exit(), характерного для UNIX-систем, функция exit() является частью стандартной библиотеки C; это означает, что она доступна в любой реализации данного языка.

Другим способом завершения работы процесса является возвращение из функции `main()`, явное или неявное (когда достигается конец функции). Явный возврат значения (`return n`) обычно является эквивалентом вызова `exit(n)`, поскольку среда выполнения, которая вызывает функцию `main()`, использует возвращаемое из нее значение для вызова `exit()`.

Существует одно обстоятельство, при котором вызов exit() и возвращение из функции main() не являются эквивалентными. Если во время завершения предпринимаются какие-либо действия с использованием локальных переменных функции main(), то возвращаемое значение этой функции будет неопределенным. Так, например, случается, когда локальная переменная функции main() указывается в вызовах setvbuf() или setbuf() (см. раздел 13.2).

Возвращение из функции без указывания значения или просто завершение главной функции тоже приводит к тому, что программа, вызвавшая функцию `main()`, выполняет вызов `exit()`, однако результат при этом зависит от поддерживаемого стандарта языка C и использованных параметров компиляции:

В стандарте C89 поведение при таких обстоятельствах не определено; программа может завершиться с произвольным статусом. Так, например, происходит на платформе Linux в сочетании с компилятором gcc: код завершения программы берется из какого-то случайного значения в стеке или определенном регистре ЦПУ. Следует избегать завершения программ таким образом.

Стандарт C99 требует, чтобы окончание главной функции было эквивалентно вызову `exit(0)`. Для того чтобы программа в системе Linux вела себя именно так, нужно скомпилировать ее с использованием параметра `gcc -std=c99`.

25.2. Завершение процесса в подробностях

Во время нормального и аварийного завершения процесса выполняются следующие действия.

- Закрываются дескрипторы открытых файлов, потоки каталогов (см. раздел 18.8), дескрипторы каталога сообщений (см. справочные страницы вызовов `catopen(3)` и `catgets(3)`) и дескрипторы преобразования (см. справочную страницу вызова `iconv_open(3)`).
- В результате закрытия дескрипторов снимаются все блокировки файлов, которые удерживал данный процесс (см. главу 51).
- Если это контролирующий процесс в контролирующем терминале, каждому из процессов в активной группе терминала посылается сигнал `SIGHUP`, а сам терминал отключается от сессии. Данный этап будет рассмотрен подробнее в разделе 34.6.
- Закрываются любые именованные семафоры POSIX, открытые в вызывающем процессе, как будто при вызове `sem_close()`.

- Закрываются любые очереди сообщений POSIX, открытые в вызывающем процессе, как будто при вызове `mq_close()`.
- Если в результате завершения процесса его группа становится «осиротевшей», то всем остановленным процессам, которые в ней все еще находятся, по очереди высылаются сигналы `SIGHUP` и `SIGCONT`. Данный этап будет рассмотрен подробнее в подразделе 34.7.4.
- Снимаются любые блокировки памяти, установленные с помощью вызовов `mlock()` или `mlockall()` (см. раздел 46.2).
- Сбрасываются любые отображения в память, созданные с помощью вызова `mmap()`.

25.3. Обработчики выхода

Иногда нужно, чтобы приложение автоматически выполняло некоторые операции перед завершением. Представьте себе программную библиотеку, которой необходимо автоматически освобождать определенные ресурсы перед закрытием приложения. Поскольку библиотека не влияет на то, когда и как процесс завершает свою работу, и не может обязать главную программу вызывать свои функции перед закрытием, она не в состоянии гарантировать освобождение ресурсов. Одним из способов решения этой проблемы является использование *обработчика выхода*.

Обработчик выхода — это функция, предоставляемая программистом, которая регистрируется на каком-то этапе жизненного цикла процесса и затем автоматически вызывается во время его *нормального* завершения посредством функции `exit()`. Обработчики выхода игнорируются, если программа напрямую вызывает `_exit()` или если процесс завершается аварийно с помощью сигнала.

> Факт того, что обработчики выхода не вызываются, если процесс завершается посредством сигнала, в некоторой степени ограничивает их применение. Лучшее, что мы можем сделать, — предусмотреть обработчики для сигналов, которые могут быть посланы процессу, и заставить их устанавливать флаг, который приводит к вызову exit() главной программой. (Поскольку exit() не является одной из безопасной для асинхронных сигналов функций, перечисленных в табл. 21.1, ее, как правило, нельзя вызывать из обработчика сигнала.) Но даже в этом случае обработчик не сможет среагировать на сигнал SIGKILL, для которого нельзя изменить действие по умолчанию. Это одна из причин, по которой мы должны избегать применения этого сигнала для завершения процессов (как отмечено в разделе 20.2); вместо него следует использовать сигнал SIGTERM, который передается по умолчанию командой kill.

Регистрация обработчиков выхода

В GNU библиотеке C предусмотрено два способа регистрации обработчиков выхода. Первый из них описан в стандарте SUSv3 и заключается в использовании функции `atexit()`.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Возвращает `0` при успешном завершении и ненулевое значение при ошибке.

Функция `atexit()` добавляет аргумент `func` в список функций, которые вызываются при завершении процесса. В определении функции `func` не должно быть аргументов и возвращаемого значения, то есть она должна иметь примерно такой вид:

```
void
func(void)
{
    /* Выполняем какие-то действия */
}
```

Стоит отметить, что при возникновении ошибки функция `atexit()` возвращает ненулевое значение (не обязательно `-1`).

Можно зарегистрировать сразу несколько обработчиков выхода (и даже один и тот же обработчик несколько раз). При вызове программой `exit()` эти функции выполняются в *порядке, обратном* их регистрации. Это логично, ведь функции, зарегистрированные раньше, как правило, выполняют более основательное освобождение ресурсов по сравнению с теми, что прошли регистрацию позже.

Внутри обработчика выхода можно выполнить практически любое действие на ваш выбор, включая регистрацию дополнительных обработчиков, которые помещаются в начало списка еще не вызванных обработчиков. Однако если один из обработчиков выхода не сможет вернуть управление (либо из-за вызова `_exit()`, либо в результате завершения процесса с помощью сигнала — например, когда обработчик сделал вызов `raise()`), остальные обработчики не будут вызваны. Кроме того, будут проигнорированы действия, которые обычно выполняются вызовом `exit()` (то есть сброс буферов stdio).

> SUSv3 гласит, что в случае, если обработчик выхода сам вызывает exit(), результат становится неопределенным. В Linux оставшиеся обработчики выхода вызываются как обычно. Однако в некоторых системах это приводит к повторному выполнению всех обработчиков, что может вылиться в бесконечную рекурсию (пока переполнение стека не остановит процесс). В переносимых приложениях следует избегать вызова exit() внутри обработчиков выхода.

SUSv3 требует, чтобы процесс мог зарегистрировать как минимум 32 обработчика выхода. Максимально возможное количество обработчиков, доступных для регистрации в данной реализации, можно узнать с помощью вызова `sysconf(_SC_ATEXIT_MAX)` (однако не существует способа узнать, сколько обработчиков уже было зарегистрировано). Библиотека glibc может почти полностью снять это ограничение, если поместить обработчики выхода в динамически выделяемый связанной список. В системе Linux вызов `sysconf(_SC_ATEXIT_MAX)` возвращает `2147483647` (то есть максимальное целое 32-битное число со знаком). Иными словами, прежде, чем будет достигнуто это ограничение, мы исчерпаем какой-то другой ресурс (например, память).

Дочерний процесс, созданный с помощью вызова `fork()`, наследует копию списка зарегистрированных обработчиков выхода своего родителя. Когда процесс выполняет функцию `exec()`, все обработчики удаляются (это необходимо, потому что функция `exec()` заменяет код обработчиков выхода вместе с остальным программным кодом).

> Если регистрация обработчика выхода была выполнена с помощью вызова atexit() (или on_exit(), описанного ниже), ее уже нельзя отменить. Однако внутри обработчиков перед выполнением каких-либо действий можно проверять, установлен ли глобальный флаг, и отключать их путем сбрасывания этого флага.

У обработчиков выхода, зарегистрированных с помощью вызова `atexit()`, есть два ограничения. Во-первых, вызывающийся обработчик выхода не знает, какой код был передан в функцию `exit()`. Иногда осведомленность об этом коде может оказаться полезной; например, можно выполнять разные действия в зависимости от того, завершается ли процесс удачно или неудачно. Второе ограничение заключается в том, что мы не можем указать аргумент для вызываемого обработчика выхода. Это могло бы пригодиться для

изменения поведения обработчика или для многократной регистрации одной и той же функции, но с разными аргументами.

Для обхода этих ограничений библиотека glibc предоставляет альтернативный (нестандартный) способ регистрации обработчиков выхода: вызов `on_exit()`.

```
#define _BSD_SOURCE /* Или: #define _SVID_SOURCE */
#include <stdlib.h>

int on_exit(void (*func)(int, void *), void *arg);

                                    Возвращает 0 при успешном завершении
                                    или ненулевое значение при ошибке
```

Аргумент `func` вызова `on_exit()` является указателем на функцию следующего вида:

```
void
func(int status, void *arg)
{
    /* Выполняем освобождение ресурсов */
}
```

При вызове функции `func()` передаются два аргумента: `status`, который был передан при вызове `exit()`, и копия аргумента `arg`, который был указан во время регистрации функции. И хотя `arg` объявлен в качестве указателя, он открыт для различных интерпретаций со стороны программиста. Он может указывать на некую структуру данных, но с тем же успехом его можно применять в качестве целого числа или другого скалярного значения, если аккуратно выполнить приведение типов.

Как и `atexit()`, вызов `on_exit()` в случае ошибки возвращает ненулевое значение (не обязательно `-1`).

Точно так же он способен регистрировать множество обработчиков выхода. Функции, зарегистрированные с помощью вызовов `atexit()` и `on_exit()`, попадают в один и тот же список. Если оба этих метода используются в одной и той же программе, зарегистрированные ими обработчики вызываются в порядке, обратном регистрации, независимо от способа регистрации.

Вызов `on_exit()` отличается большей гибкостью по сравнению с `atexit()`, но он не входит ни в один стандарт и лишь всего несколькими альтернативными реализациями системы UNIX, поэтому при написании переносимых приложений его следует избегать.

Пример программы

Применение вызовов `atexit()` и `on_exit()` для регистрации обработчиков выхода демонстрируется в листинге 25.1. При запуске этой программы мы увидим следующий вывод:

```
$ ./exit_handlers
on_exit function called: status=2, arg=20
atexit function 2 called
atexit function 1 called
on_exit function called: status=2, arg=10
```

Листинг 25.1. Использование обработчиков выхода

procexec/exit_handlers.c

```
#define _BSD_SOURCE /* Получаем объявление вызова on_exit() из <stdlib.h> */
#include <stdlib.h>
#include "tlpi_hdr.h"
```

```
static void
atexitFunc1(void)
{
    printf("atexit function 1 called\n");
}

static void
atexitFunc2(void)
{
    printf("atexit function 2 called\n");
}

static void
onexitFunc(int exitStatus, void *arg) {
    printf("on_exit function called: status=%d, arg=%ld\n",
    exitStatus, (long) arg);
}

int
main(int argc, char *argv[])
{
    if (on_exit(onexitFunc, (void *) 10) != 0)
        fatal("on_exit 1");
    if (atexit(atexitFunc1) != 0)
        fatal("atexit 1");
    if (atexit(atexitFunc2) != 0)
        fatal("atexit 2");
    if (on_exit(onexitFunc, (void *) 20) != 0)
        fatal("on_exit 2");

    exit(2);
}
```

procexec/exit_handlers.c

25.4. Взаимодействие между буферами stdio и вызовами fork() и _exit()

Вывод, сгенерированный программой из листинга 25.2, демонстрирует феномен, который на первый взгляд может показаться обескураживающим. Если запустить эту программу со стандартным выводом, направленным в терминал, получится вполне предсказуемый результат:

```
$ ./fork_stdio_buf
Hello world
Ciao
```

Но если перенаправить вывод в файл, мы увидим следующее:

```
$ ./fork_stdio_buf > a
$ cat a
Ciao
Hello world
Hello world
```

В выводе выше можно заметить два странных момента: строка, напечатанная функцией `printf()`, продублирована и ей почему-то предшествует вывод вызова `write()`.

Листинг 25.2. Взаимодействие вызова fork() и буферизации stdio

procexec/fork_stdio_buf.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    /* Здесь продолжают выполнение как потомок, так и родитель */

    exit(EXIT_SUCCESS);
}
```

procexec/fork_stdio_buf.c

Для того чтобы понять, почему сообщение, выводимое вызовом `printf()`, печатается два раза, вспомните, что буферы stdio находятся в пользовательском пространстве памяти процесса (см. раздел 13.2). Следовательно, эти буферы дублируются в дочернем процессе вызовом `fork()`. Когда стандартный вывод направлен в терминал, буферизация в нем по умолчанию происходит построчно, благодаря чему строки, разделенные разрывами, сразу же выводятся вызовом `printf()`. Однако стандартный вывод, направленный в файл, по умолчанию буферизируется по блокам. Таким образом, на момент вызова `fork()` строка, выводимая вызовом `printf()`, все еще находится в родительском буфере stdio и, следовательно, дублируется в потомке. Позже, когда родительский и дочерний процессы вызывают `exit()`, они оба сбрасывают свои копии буферов stdio, что приводит к дублированию результата.

Избежать этого можно одним из двух способов.

- В качестве решения, устраняющего непосредственно проблему буферизации stdio, перед вызовом `fork()` можно использовать функцию `fflush()`, которая сбрасывает соответствующий буфер. Или, как вариант, мы могли бы отключить буферизацию потока stdio с помощью вызовов `setvbuf()` или `setbuf()`.
- Потомок может выполнять вызов `_exit()` вместо `exit()`, чтобы не сбрасывать буферы стандартного ввода/вывода. Этот метод иллюстрирует более общее правило: в приложении, создающем потомка, который не выполняет новой программы, обычно только один из процессов (чаще всего родитель) должен завершаться с помощью функции `exit()`, тогда как для всех остальных следует использовать вызов `_exit()`. Это дает гарантию того, что только один процесс вызывает обработчики выхода и сбрасывает буферы stdio, что обычно и требуется.

  Существуют и другие способы решения этой проблемы, которые позволяют вызывать exit() как родителю, так и потомку; иногда без них нельзя обойтись. Например, можно создать такие обработчики выхода, которые выполняются корректно, даже когда вызываются из нескольких процессов, или регистрировать обработчики вызова лишь после вызова fork(). Более того, иногда сброс буферов stdio после вызова fork() всеми процессами является даже желательным. В этом случае процессы можно завершить с помощью функции exit() или явно вызвать fflush() в каждом из них.

Вывод вызова `write()` в программе из листинга 25.2 не дублируется, поскольку данные этого вызова передаются напрямую в буфер ядра, который не подлежит копированию во время выполнения функции `fork()`.

Теперь вам уже должна быть ясна причина второго странного момента в поведении программы, когда ее вывод направлен в файл. Текст из вызова `write()` появляется раньше, потому что он сразу же доставляется в кэш буфера ядра, тогда как данные в вызове `printf()` передаются только в результате сбрасывания буферов stdio вызовом `exit()` (в целом, как было замечено в разделе 13.7, следует быть осторожными, смешивая функции stdio и системные вызовы при работе с одним и тем же файлом).

25.5. Резюме

Процесс может завершиться нормально или аварийно. Аварийное завершение происходит в результате получения определенных сигналов. Некоторые из них заставляют процесс сгенерировать файл дампа памяти.

Нормальное завершение процесса выполняется с помощью вызова `_exit()` (или его обертки, `exit()`, что более желательно). Вызовы `_exit()` и `exit()` принимают целочисленный аргумент, последние 8 бит которого определяют код завершения процесса. Значение `0` принято считать признаком успешного выполнения, а ненулевой код обычно указывает на какие-то проблемы.

Что бы ни послужило причиной завершения процесса, ядро предпринимает различные шаги по освобождению ресурсов. Кроме того, нормальное завершение с помощью вызова `exit()` приводит к выполнению обработчиков выхода, зарегистрированных с помощью функций `atexit()` и `on_exit()` (в порядке, обратном регистрации), и сбросу буферов stdio.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 24.6.

25.6. Упражнение

25.1. Если дочерний процесс выполняет вызов `exit(-1)`, какой код завершения увидит родитель?

26 Мониторинг дочерних процессов

Часто при проектировании приложений родительский процесс должен знать об изменении состояния своих потомков — то есть когда они завершают работу или останавливаются по сигналу. В этой главе представлено два подхода к мониторингу дочерних процессов: системный вызов `wait()` (и его вариации) и использование сигнала `SIGCHLD`.

26.1. Ожидание дочернего процесса

Во многих приложениях, в которых создаются вложенные процессы, бывает полезно наделить родителя возможностью следить за своими потомками, чтобы знать, когда и как они завершают свою работу. Эта возможность реализована в виде ряда системных вызовов, основным из которых является `wait()`.

26.1.1. Системный вызов wait()

Системный вызов `wait()` ждет, когда один из потомков вызывающего процесса прекратит работу и возвращает код завершения этого дочернего процесса через буфер, на который указывает аргумент `status`.

```
#include <sys/wait.h>

pid_t wait(int *status);
```

Возвращает идентификатор завершенного процесса или `-1` при ошибке

Системный вызов `wait()` делает следующее.

1. Если ни один из потомков вызывающего процесса (который ранее не отслеживался) еще не завершился, вызов блокируется, пока этого не произойдет. Если на момент вызова какой-либо потомок уже прекратил работу, `wait()` сразу же возвращает значение.
2. Если параметр `status` не равен `NULL`, он указывает на целое число, описывающее подробности завершения потомка. Информация, возвращаемая таким образом, будет рассмотрена в разделе 26.1.3.
3. Ядро добавляет процессорное время (см. раздел 10.7) и статистику использования ресурсов (см. раздел 36.1) к общему времени ЦП, затраченному всеми потомками процесса.
4. Вызов `wait()` возвращает идентификатор завершившегося дочернего процесса.

В случае ошибки `wait()` возвращает `-1`. К ошибке может привести, например, отсутствие у вызывающего процесса потомков (которые ранее не отслеживались); в этом случае `errno` присваивается значение `ECHILD`. Это означает, что мы можем ждать завершения всех потомков вызывающего процесса в следующем цикле:

```
while ((childPid = wait(NULL)) != -1)
    continue;
```

```
if (errno != ECHILD)    /* Непредвиденная ошибка... */
    errExit("wait");
```

Использование вызова `wait()` демонстрируется в листинге 26.1. В этой программе создается несколько дочерних процессов — по одному на каждый (целочисленный) аргумент командной строки. Каждый потомок ожидает определенное количество секунд, указанное в соответствующем аргументе, и затем завершается. Тем временем после создания всех потомков родительский процесс начинает их отслеживать, последовательно выполняя вызовы `wait()`. Этот цикл завершается, когда `wait()` возвращает `-1` (это не единственный из возможных вариантов: например, мы могли бы выходить из цикла, когда число завершенных потомков, `numDead`, сравнивается с числом созданных потомков). В журнале сессии, представленном ниже, показано, что происходит, когда мы используем данную программу для создания трех дочерних процессов:

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

Если в какой-то момент у нас имеется несколько завершенных потомков, порядок, в котором они будут обработаны вызовами wait(), является неопределенным (согласно стандарту SUSv3). Это означает, что порядок обработки зависит от реализации. Он может меняться даже в разных версиях ядра Linux.

Листинг 26.1. Создание нескольких потомков и ожидание их завершения

procexec/multi_wait.c

```
#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"       /* Объявление currTIme() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;     /* Количество завершенных потомков на данный момент */
    pid_t childPid;  /* PID завершенного потомка */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);  /* Отключаем буферизацию стандартного вывода */

    for (j = 1; j < argc; j++) {  /* Создаем по одному потомку для каждого аргумента */
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:    /* Потомок ожидает какое-то время, затем завершается */
            printf("[%s] child %d started with PID %ld, sleeping %s "
               "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
```

```
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:    /* Родитель продолжает выполнение цикла */
            break;
        }
    }

    numDead = 0;
    for (;;) {    /* Родитель ждет завершения каждого потомка */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {    /* Какая-то другая (непредвиденная) ошибка */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
               currTime("%T"), (long) childPid, numDead);
    }
}
```

procexec/multi_wait.c

26.1.2. Системный вызов waitpid()

Системный вызов `waitpid()` снимает ряд ограничений, присущих вызову `wait()`.

- Если родительский процесс создал несколько потомков, `wait()` не позволяет ожидать завершения конкретного из них; мы можем отслеживать завершение работы только следующего дочернего процесса.
- Если ни один из потомков еще не был завершен, вызов `wait()` всегда блокируется. Иногда имеет смысл организовать неблокирующее ожидание, чтобы, если все потомки все еще работают, иметь возможность немедленно узнать об этом факте.
- С помощью `wait()` можно отслеживать только потомки, которые завершились. Если потомок был остановлен (по сигналам `SIGSTOP` или `SIGTTIN`) или возобновил свою работу (по сигналу `SIGCONT`), мы об этом не узнаем.

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Возвращает идентификатор потомка: `0` (см. далее) или `-1`, если возникла ошибка

Возвращаемое значение и аргумент `status` у вызова `waitpid()` служат тем же целям, что и `wait()` (описание значений аргумента `status` ищите в разделе 26.1.3). Аргумент `pid` позволяет выбрать потомков, которые будут отслеживаться. Делается это следующим образом.

- Если `pid` больше `0`, ожидаем потомка, чей *идентификатор* равен `pid`.
- Если `pid` равен `0`, ожидаем потомка, входящего *в ту же группу процессов, что и родитель*. Группы процессов описываются в разделе 34.2.

- Если `pid` меньше `-1`, ожидаем любого потомка, идентификатор *группы процессов* которого равен значению `pid` по модулю.
- Если `pid` равен `-1`, ожидаем *любого* потомка. Вызов `wait(&status)` эквивалентен вызову `waitpid(-1, &status, 0)`.

Аргумент `options` представляет собой битовую маску, которая может быть пустой или содержать какое-то количество следующих флагов (каждый из которых входит в стандарт SUSv3), скомбинированных с помощью побитового ИЛИ.

- `WUNTRACED` — позволяет узнать не только о завершенных дочерних процессах, но и о потомках, *остановленных* с помощью сигнала.
- `WCONTINUED` (начиная с Linux 2.6.10) — позволяет узнать о потомках, работа которых была возобновлена после остановки в результате получения сигнала `SIGCONT`.
- `WNOHANG` — если ни один из потомков, указанных в `pid`, все еще не изменил свое состояние, вызов немедленно возвращается, не переходя к блокированию (то есть «опрашивает» потомков). В этом случае возвращаемое значение `waitpid()` равно `0`. Если у родителя не оказывается дочерних процессов, соответствующих значению `pid`, `waitpid()` возвращает ошибку `ECHILD`.
  Использование `waitpid()` проиллюстрировано в листинге 26.3.

В обосновании внесения вызова waitpid() в стандарт SUSv3 отмечается, что название флага WUNTRACED берет свое начало в системе BSD, где выполнение процесса может остановиться по одной из двух причин: из-за отслеживания системным вызовом ptrace() или в результате получения сигнала (то есть без отслеживания). Когда потомок отслеживается вызовом ptrace(), любой полученный им сигнал (кроме SIGKILL) приводит к его остановке и последующей отправке его родителю сигнала SIGCHLD. Так происходит даже в том случае, если потомок игнорирует сигналы. Однако, если сигналы заблокированы, то работа процесса не останавливается (это относится ко всем сигналам, кроме неблокирующегося SIGSTOP).

26.1.3. Статус ожидания

Значение `status`, возвращаемое вызовами `wait()` и `waitpid()`, позволяет различать следующие события, касающиеся потомков.

- Потомок завершил работу с помощью вызова `_exit()` (или `exit()`), вернув целочисленный *код выхода*.
- Потомок был завершен путем доставки необработанного сигнала.
- Потомок был остановлен сигналом, а вызов `waitpid()` был произведен с использованием флага `WUNTRACED`.
- Потомок возобновил работу после сигнала `SIGCONT`, а вызов `waitpid()` был выполнен с использованием флага `WCONTINUED`.

Чтобы охватить все случаи, приведенные выше, мы будем обращаться к термину *«статус ожидания»*. Первые два пункта можно назвать *кодом завершения* (чтобы получить его для последней программы, выполненной в командной строке, можно прочитать содержимое переменной `$?`).

И хотя значение `status` является целым числом, только последних два байта из него действительно используются. Способ их заполнения зависит от того, какое из перечисленных выше событий произошло с потомком (рис. 26.1).

На рис. 26.1 показана схема статуса ожидания в Linux/x86-32. Детали могут отличаться в зависимости от реализации. SUSv3 не предусматривает определенной компоновки для этой информации; он даже не требует, чтобы статус ожидания хранился именно в последних двух байтах, на которые указывает status. Для чтения этих данных в переносимых приложениях всегда нужно использовать макросы из этого раздела, избегая прямого доступа к элементам битовой маски.

Заголовочный файл `<sys/wait.h>` определяет стандартный набор макросов, с помощью которых можно анализировать код завершения. Только один из них вернет `true`, если применить его к значению `status`, возвращаемому вызовами `wait()` и `waitpid()`. Как отмечено в списке, для более глубокого анализа предоставляются дополнительные макросы.

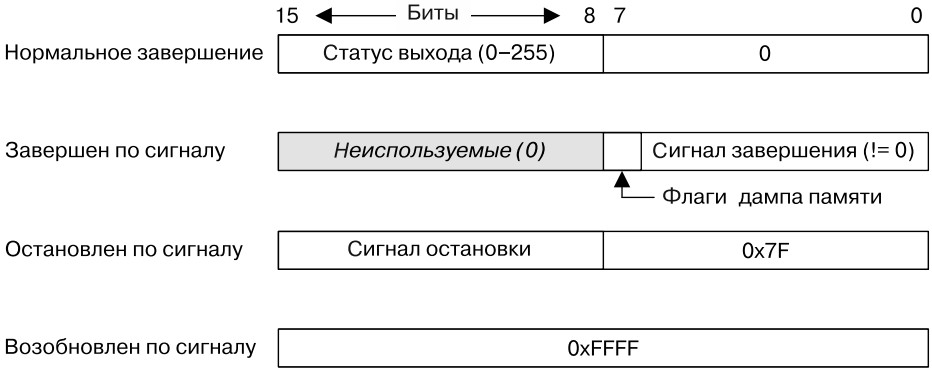


Рис. 26.1. Значение, возвращаемое вызовами wait() и waitpid() в аргументе status

- `WIFEXITED(status)` — возвращает `true`, если дочерний процесс завершился штатно. В этом случае макрос `WEXITSTATUS(status)` возвращает код завершения дочернего процесса (как было отмечено в разделе 25.1, родителю доступен только младший байт код завершения).
- `WIFSIGNALED(status)` — возвращает `true`, если дочерний процесс был завершен с помощью сигнала. В этом случае макрос `WTERMSIG(status)` возвращает номер сигнала, приведшего к завершению процесса, а макрос `WCOREDUMP(status)` возвращает `true`, если потомок сгенерировал файл с дампом памяти. `WCOREDUMP(status)` не входит в стандарт SUSv3, но доступен в большинстве реализаций UNIX.
- `WIFSTOPPED(status)` — возвращает `true`, если дочерний процесс был остановлен по сигналу. В этом случае макрос `WSTOPSIG(status)` возвращает номер сигнала, остановившего процесс.
- `IFCONTINUED(status)` — возвращает `true`, если дочерний процесс возобновил свою работу, получив сигнал `SIGCONT`. Он доступен в системе Linux, начиная с версии 2.6.10.

Обратите внимание, что, хотя вышеприведенные макросы тоже используют для своих аргументов имя `status`, они ожидают получить обычное целое число, а не указатель на него, как в случае с вызовами `wait()` и `waitpi()`.

Пример программы

Функция `printWaitStatus()` из листинга 26.2 задействует все макросы, перечисленные выше. Она считывает и выводит содержимое статуса ожидания.

Листинг 26.2. Вывод значения статуса, возвращенного wait() и другими похожими вызовами
procexec/print_wait_status.c

```
#define _GNU_SOURCE     /* Получаем объявление strsignal() из <string.h> */
#include <string.h>
#include <sys/wait.h>
#include "print_wait_status.h"    /* Объявление printWaitStatus() */
#include "tlpi_hdr.h"
```

```
/* ВАЖНО: в следующей функции используется вызов printf(), который не является
   безопасным с точки зрения асинхронных сигналов (см. подраздел 21.1.2). Это делает
   всю функцию небезопасной для работы с асинхронными сигналами (то есть будьте
   осторожны, когда вызываете ее из обработчика SIGCHLD). */

void    /* Анализируем статус wait() с помощью макросов W* */
printWaitStatus(const char *msg, int status)
{
    if (msg != NULL)
        printf("%s", msg);

    if (WIFEXITED(status)) {
        printf("child exited, status=%d\n", WEXITSTATUS(status));

    } else if (WIFSIGNALED(status)) {
        printf("child killed by signal %d (%s)",
            WTERMSIG(status), strsignal(WTERMSIG(status)));
#ifdef WCOREDUMP    /* Не входит в стандарт SUSv3, может отсутствовать
                       в некоторых системах */
    if (WCOREDUMP(status))
        printf(" (core dumped)");
#endif
        printf("\n");
    } else if (WIFSTOPPED(status)) {
        printf("child stopped by signal %d (%s)\n",
            WSTOPSIG(status), strsignal(WSTOPSIG(status)));

#ifdef WIFCONTINUED  /* Входит в стандарт SUSv3, но может отсутствовать
                        в ранних версиях Linux и некоторых реализациях UNIX */
    } else if (WIFCONTINUED(status)) {
        printf("child continued\n");
#endif

    } else {    /* Этого никогда не должно случиться */
        printf("what happened to this child? (status=%x)\n",
               (unsigned int) status);
    }
}
```

procexec/print_wait_status.c

В листинге 26.3 применяется функция `printWaitStatus()`. Она создает дочерний процесс, который либо входит в цикл, постоянно вызывая `pause()` (в этот момент потомку можно слать сигналы), либо, если был предоставлен целочисленный аргумент командной строки, сразу же завершается, используя этот аргумент в качестве кода завершения. Тем временем родитель следит за потомком с помощью вызова `waitpid()`, выводит возвращенный код и передает его в функцию `printWaitStatus()`. Родитель завершает работу, обнаружив, что потомок уже остановлен — либо штатно, либо с помощью сигнала.

Следующая сессия командной оболочки демонстрирует несколько запусков программы из листинга 26.3. Для начала создадим дочерний процесс, который сразу же завершается с кодом 23:

```
$ ./child_status 23
Child started with PID = 15807
waitpid() returned: PID=15807; status=0x1700 (23,0)
child exited, status=23
```

Теперь запустим программу в фоновом режиме и отправим дочернему процессу сигналы `SIGSTOP` и `SIGCONT`:

```
$ ./child_status &
[1] 15870
$ Child started with PID = 15871
kill -STOP 15871
$ waitpid() returned: PID=15871; status=0x137f (19,127)
child stopped by signal 19 (Stopped (signal))
kill -CONT 15871
$ waitpid() returned: PID=15871; status=0xffff (255,255)
child continued
```

Последние две строчки вывода появляются только в Linux версии 2.6.10 и выше, поскольку более старые ядра не поддерживают параметр `WCONTINUED` для `waitpid()` (сессия командной оболочки получается немного хаотичной, поскольку программа выполняется в фоновом режиме и ее вывод может смешиваться с приглашением командной строки).

Продолжаем сессию командной оболочки. Отправим сигнал `SIGABRT`, чтобы завершить работу потомка:

```
kill -ABRT 15871
$ waitpid() returned: PID=15871; status=0x0006 (0,6)
child killed by signal 6 (Aborted)
Нажмите Enter, чтобы увидеть уведомление командной оболочки о завершении фоновой задачи
[1]+ Done          ./child_status
$ ls -l core
ls: core: No such file or directory
$ ulimit -c           Выводим ограничение RLIMIT_CORE
0
```

И хотя по умолчанию сигнал `SIGABRT` генерирует дамп памяти перед завершением процесса, в данном случае этого не произошло. Дело в том, что дамп памяти был отключен; согласно команде `ulimit`, приведенной выше, ограничение на программные ресурсы `RLIMIT_CORE` (см. раздел 36.3), которое задает максимальный размер файла с дампом памяти, было установлено в `0`.

Повторим тот же эксперимент, но на этот раз перед отправкой потомку сигнала `SIGABRT` включим дамп памяти:

```
$ ulimit -c unlimited          Включаем дампы памяти
$ ./child_status &
[1] 15902
$ Child started with PID = 15903
kill -ABRT 15903               Передаем потомку сигнал SIGABRT
$ waitpid() returned: PID=15903; status=0x0086 (0,134)
child killed by signal 6 (Aborted) (core dumped)
Нажмите Enter, чтобы увидеть уведомление командной оболочки о завершении фоновой задачи
[1]+ Done ./child_status
$ ls -l core                   На этот раз мы получаем дамп памяти
-rw-------  1 mtk   users   65536 May   6 21:01  core
```

Листинг 26.3. Использование вызова waitpid() для получения статуса дочернего процесса
procexec/child_status.c

```
#include <sys/wait.h>
#include "print_wait_status.h"    /* Объявление printWaitStatus() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
```

```
    int status;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);

    switch (fork()) {
    case -1: errExit("fork");

    case 0:   /* Потомок либо немедленно завершается с заданным кодом,
                 либо входит в цикл ожидания сигналов */
        printf("Child started with PID = %ld\n", (long) getpid());
        if (argc > 1)  /* Был ли предоставлен статус в командной строке? */
            exit(getInt(argv[1], 0, "exit-status"));
        else           /* Если нет, ждем сигналов */
            for (;;)
                pause();
        exit(EXIT_FAILURE);    /* В нашем случае никогда не выполнится,
                                  но считается хорошим тоном */

    default:        /* Родитель ждет в цикле, пока потомок не будет
                       завершен — либо штатно, либо по сигналу */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED
#ifdef WCONTINUED    /* Отсутствует в старых версиях Linux */
                       | WCONTINUED
#endif
                );
            if (childPid == -1)
                errExit("waitpid");
            /* Выводит статус как восьмеричное число
               и десятичные значения отдельных байтов */

            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                (long) childPid,
                (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);

            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

procexec/child_status.c

26.1.4. Завершение процесса из обработчика сигнала

Как показано в табл. 20.1, некоторые сигналы по умолчанию завершают процесс. В определенных обстоятельствах перед завершением работы возникает необходимость в освобождении ресурсов. Для этого можно предусмотреть обработчик, который будет перехватывать такие сигналы, освобождать ресурсы и только потом завершать процесс. Делая это, мы должны иметь в виду, что код завершения процесса доступен его родителю посредством вызовов `wait()` и `waitpid()`. Например, если сделать из обработчика сигнала вызов `_exit(EXIT_SUCCESS)`, родитель будет знать, что потомок завершился успешно.

Если потомку нужно сообщить о том, что он завершается из-за сигнала, его обработчик должен сначала себя отключить, и затем еще раз послать тот же сигнал, которому

уже ничто не помешает завершить работу процесса. В этом случае обработчик сигнала будет содержать примерно такой код:

```
void
handler(int sig)
{
    /* Освобождаем ресурсы */

    signal(sig, SIG_DFL);  /* Отключаем обработчик */
    raise(sig);            /* Генерируем сигнал повторно */
}
```

26.1.5. Системный вызов waitid()

Как и `waitpid()`, вызов `waitid()` возвращает статус дочернего процесса. Однако, кроме этого, он предоставляет дополнительные возможности, недоступные в `waitpid()`. Этот системный вызов происходит из System V, хотя и не является частью стандарта SUSv3. Он поддерживается в ядре Linux с версии 2.6.9.

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Возвращает `0` при успешном завершении или если был указан флаг `WNOHANG` и больше не осталось потомков для ожидания; возвращает `-1` при ошибке

Аргументы `idtype` и `id` определяют, какого из потомков нужно ждать:

- если `idtype` равен `P_ALL`, ждем любого потомка; `id` игнорируется;
- если `idtype` равен `P_PID`, ждем потомка, чей идентификатор равен `id`;
- если `idtype` равен `P_PGID`, ждем любого потомка, идентификатор группы процессов которого равен `id`.

Стоит отметить, что waitid(), в отличие от waitpid(), не позволяет охватить всех потомков в группе вызывающего процесса, указав для id значение 0. Вместо этого следует явно задать идентификатор группы вызывающего процесса, используя значение, возвращенное вызовом getpgrp().

Главным отличием вызова `waitid()` от `waitpid()` является то, что он предоставляет более тонкий контроль за событиями ожидаемого потомка. Это обеспечивается за счет задания внутри параметра `options` одного или нескольких флагов, скомбинированных с помощью побитового ИЛИ.

- `WEXITED` — позволяет ждать потомка, который завершился штатным образом или по сигналу.
- `WSTOPPED` — позволяет ждать потомка, который был остановлен по сигналу.
- `WCONTINUED` — позволяет ждать потомка, который возобновил работу, получив сигнал `SIGCONT`.

Параметр `options` также может содержать следующие флаги, разделенные побитовым ИЛИ.

- `WNOHANG` — означает то же самое, что и в случае с вызовом `waitpid()`.Если ни один из потомков, указанных в `id`, все еще не изменил свое состояние, вызов немедленно возвращается, не переходя к блокированию (то есть «опрашивает» потомков). В этом

случае возвращаемое значение `waitid()` равно `0`. Если у родителя не оказывается дочерних процессов, соответствующих значению `id`, `waitid()` возвращает ошибку `ECHILD`.

- `WNOWAIT` — в обычной ситуации `waitid()` возвращает статус потомка лишь однократно. Но если указать флаг `WNOWAIT`, возвращается статус потомка, а сам потомок остается доступным для ожидания, и позже мы можем опять воспользоваться вызовом `waitid()`, чтобы получить ту же самую информацию.

В случае успеха `waitid()` возвращает `0`, а структуре `siginfo_t` (см. раздел 21.4), на которую ссылается параметр `infop`, присваивается информация о потомке. В структуре `siginfo_t` заполняются следующие поля.

- `si_code` — содержит одно или несколько значений такого вида: `CLD_EXITED` (говорит о том, что потомок был завершен путем вызова `_exit()`), `CLD_KILLED` (сигнализирует об остановке потомка с помощью сигнала), `CLD_STOPPED` (указывает на остановку потомка посредством сигнала) или `CLD_CONTINUED` (говорит о том, что ранее остановленный потомок продолжил выполнение в результате получения сигнала `SIGCONT`).
- `si_pid` — содержит идентификатор дочернего процесса, у которого изменилось состояние.
- `si_signo` — всегда равно `SIGCHLD`.
- `si_status` — содержит либо код завершения потомка, переданный в `_exit()`, либо сигнал, который заставил потомка остановиться, продолжить работу или завершиться. Чтобы определить, с каким именно видом информации мы имеем дело, можно проанализировать значение поля `si_code`.
- `si_uid` — содержит реальный пользовательский идентификатор потомка. В большинстве реализаций UNIX оно остается пустым.

В операционной системе Solaris заполняются два дополнительных поля: si_stime и si_utime. Они содержат, соответственно, системное и пользовательское время ЦП, затраченное потомком. Стандарт SUSv3 не требует от вызова waitid() заполнения данных полей.

На одном из аспектов работы вызова `waitid()` следует остановиться отдельно. Если в аргументе `options` указан флаг `WNOHANG`, тогда возвращаемое значение `0` может означать одно из двух: либо потомок уже успел поменять состояние на момент вызова (а информация о нем передается через аргумент `infop`, указывающий на структуру `siginfo_t`), либо потомка, чье состояние изменилось, просто не было. Во втором варианте некоторые реализации Unix (включая Linux) обнуляют структуру `siginfo_t`. Так, сравнивая `si_pid` с нулем, мы можем различить эти два случая. К сожалению, такой подход не является обязательным с точки зрения стандарта SUSv3 и в некоторых системах UNIX структура `siginfo_t` остается неизменной (в 2013 году в стандарт SUSv4 было внесено требование, согласно которому в этом случае `si_pid` и `si_signo` должны обнуляться). Единственным переносимым способом отличать эти два случая является обнуление структуры `siginfo_t` перед вызовом `waitid()`, как показано в следующем коде:

```
siginfo_t info;
...
memset(&info, 0, sizeof(siginfo_t));
if (waitid(idtype, id, &info, options | WNOHANG) == -1)
    errExit("waitid");
if (info.si_pid == 0) {     /* Ни один из потомков не изменил состояние */
} else {
    /* Потомок изменил состояние: подробности доступны в 'info' */
}
```

26.1.6. Системные вызовы wait3() и wait4()

Системные вызовы `wait3()` и `wait4()` своим назначением похожи на `waitpid()`. Однако принципиальная их семантическая особенность заключается в том, что в структуре, на которую указывает аргумент `rusage`, они возвращают сведения об использовании ресурсов завершенным потомком. Среди прочего это касается процессорного времени и статистики о работе с памятью. Подробное описание структуры `rusage` отложим до раздела 36.1, где рассматривается системный вызов `getrusage()`.

```
#define _BSD_SOURCE /* Или #define _XOPEN_SOURCE 500 для wait3() */
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Оба вызова возвращают идентификатор потомка или –1 при ошибке

Если не считать наличия аргумента `rusage`, вызов `wait3()` эквивалентен следующему вызову `waitpid()`:

```
waitpid(-1, &status, options);
```

Аналогично ниже представлен эквивалент вызова `wait4()`:

```
waitpid(pid, &status, options);
```

Иными словами, `wait3()` ожидает любого потомка, а с помощью `wait4()` можно ждать определенных дочерних процессов.

В некоторых реализациях UNIX вызовы `wait3()` и `wait4()` возвращают сведения о потреблении ресурсов только для завершенных потомков. В Linux также может быть доступна информация об остановленных дочерних процессах, если в аргументе `options` указан флаг `WUNTRACED`.

Имена этих двух системных вызовов связаны с количеством принимаемых ими аргументов. Оба они берут свое начало в системе BSD, но в наши дни доступны в большинстве реализаций UNIX. Ни один из них не входит в стандарт SUSv3 (`wait3()` был описан в стандарте SUSv2, с пометкой LEGACY).

В этой книге мы, как правило, избегаем использования данных вызовов. В большинстве случаев нам не нужна дополнительная информация, которую они возвращают. К тому же нехватка стандартизации ограничивает их переносимость.

26.2. Процессы-«сироты» и процессы-«зомби»

Жизненные циклы родительского и дочернего процессов обычно не совпадают — один из них живет дольше, чем другой. Из этого вытекает два вопроса.

- Кто становится родителем *«осиротевшего»* потомка? Ответ: `init` — предок всех процессов, имеющий идентификатор `1`. Иными словами, после того, как родитель потомка завершит работу, вызов `getppid()` начнет возвращать `1`. С помощью этого вызова можно узнать, жив ли еще настоящий родитель потомка (подразумевается, что потомок был создан любым другим процессом, кроме `init`).

Применение операции PR_SET_PDEATHSIG системного вызова prctl() (доступного только в Linux) позволяет сделать так, чтобы в случае потери родителя процесс получал выбранный сигнал.

- Что происходит с потомком, который завершается до того, как его родитель имел возможность выполнить `wait()`? Дело в том, что, хоть потомок и закончил свою работу, родителю все равно должно быть позволено сделать вызов `wait()`, чтобы определить причину завершения. Ядро решает эту проблему путем превращения потомка в *«зомби»*. Это означает, что большинство ресурсов, занимаемых потомком, возвращаются системе и могут быть задействованы другими процессами. Единственная часть дочернего процесса, которая остается нетронутой, — это запись в таблице процессов ядра, хранящая (помимо прочего) идентификатор потомка, код завершения и статистику использования ресурсов (см. раздел 36.1).

Что касается «зомби», системы Unix следуют канонам, принятым в художественных фильмах, — процесс-«зомби» нельзя убить сигналом, даже если это `SIGKILL` (своеобразная серебряная пуля). Благодаря этому родитель всегда может выполнить вызов `wait()`.

Когда родитель наконец делает вызов `wait()`, ядро удаляет процесс-«зомби», поскольку информация о нем больше не требуется. С другой стороны, если родитель завершает работу, так и не выполнив вызов `wait()`, процесс `init` удочеряет потомка и автоматически делает этот вызов самостоятельно, удаляя таким образом «зомби» из системы.

Если родитель создает потомка, но не успевает вызвать `wait()`, запись о дочернем процессе-«зомби» все равно продолжит храниться в таблице процессов ядра. Если будет создано слишком большое количество «зомби», они в какой-то момент переполнят эту таблицу, мешая созданию новых процессов. Поскольку «зомби» нельзя убить по сигналу, единственный способ удалить их из системы заключается в принудительном (или штатном) завершении работы их родителя; в этом случае они удочеряются процессом `init`, который начинает их ожидать, что приводит к их удалению из системы.

Последствия такой семантики имеют большое значение при проектировании долгоживущих родительских процессов, таких как сетевые серверы и командные оболочки, создающих большое количество потомков. Иначе говоря, родительский процесс в таких приложениях должен выполнять вызовы `wait()`, чтобы гарантировать удаление отработанных потомков из системы и не дать им превратиться в неубиваемых «зомби». Такие вызовы можно делать синхронно или асинхронно в ответ на сигнал `SIGCHLD`, как описывается в разделе 26.3.1.

В листинге 26.4 показан процесс создания «зомби» и тот факт, что их нельзя принудительно завершить с помощью сигнала `SIGKILL`. При запуске этой программы мы увидим следующий вывод:

```
$ ./make_zombie
Parent PID=1013
Child (PID=1014) exiting
  1013 pts/4    00:00:00 make_zombie      Вывод команды ps(1)
  1014 pts/4    00:00:00 make_zombie <defunct>
After sending SIGKILL to make_zombie (PID=1014):
  1013 pts/4    00:00:00 make_zombie      Вывод команды ps(1)
  1014 pts/4    00:00:00 make_zombie <defunct>
```

В выводе, представленном выше, видно, что команда `ps(1)` выводит строку `<defunct>`, обозначающую процесс-«зомби».

Программа из листинга 26.4 использует функцию system() для выполнения консольной команды, передаваемой ей в виде символьно-строкового аргумента. Подробно эта функция описывается в разделе 27.6.

Листинг 26.4. Создание дочерних процессов-«зомби»

procexec/make_zombie.c

```
#include <signal.h>
#include <libgen.h>        /* Для объявления basename() */
#include "tlpi_hdr.h"

#define CMD_SIZE 200

int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    pid_t childPid;

    setbuf(stdout, NULL);   /* Отключаем буферизацию стандартного вывода */

    printf("Parent PID=%ld\n", (long) getpid());

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:         /* Потомок немедленно завершается, чтобы стать «зомби» */
        printf("Child (PID=%ld) exiting\n", (long) getpid());
        _exit(EXIT_SUCCESS);

    default:        /* Родитель */
        sleep(3);   /* Даем потомку шанс начать выполнение и завершиться */
        snprintf(cmd, CMD_SIZE, "ps | grep %s", basename(argv[0]));
        system(cmd);    /* Видим потомка-«зомби» */

        /* Теперь отправляем «зомби» сигнал о безусловном завершении */

        if (kill(childPid, SIGKILL) == -1)
            errMsg("kill");

        sleep(3);    /* Даем потомку шанс отреагировать на сигнал */
        printf("After sending SIGKILL to zombie (PID=%ld):\n",
               (long) childPid);
        system(cmd);    /* Опять видим потомка-«зомби» */

        exit(EXIT_SUCCESS);
    }
}
```

procexec/make_zombie.c

26.3. Сигнал SIGCHLD

Принудительное завершение дочернего процесса — это событие, которое происходит асинхронно. Родитель не может предвидеть момент его возникновения (даже если он сам шлет потомку сигнал `SIGKILL`, точное время завершения зависит от того, когда потомок получит доступ к ЦП). Мы уже знаем, что для предотвращения накопления потомков-«зомби» родитель должен использовать вызов `wait()` (или аналогичный ему), и сделать это можно двумя способами.

- Родитель может вызвать `wait()` или `waitpid()`, не указывая флаг `WNOHANG`. В этом случае вызов блокируется, если потомок все еще не завершил работу.
- Родитель может периодически выполнять неблокирующую проверку (опрос) завершившихся потомков с помощью вызова `waitpid()`, указывая флаг `WNOHANG`.

Оба этих метода могут оказаться неудобными. С одной стороны, мы не хотим блокировать родителя, пока тот ждет завершения работы своего потомка. С другой — постоянное выполнение неблокирующих вызовов `waitpid()` расходует процессорное время и усложняет структуру приложения. Чтобы обойти эту проблему, мы можем воспользоваться обработчиком сигнала `SIGCHLD`.

26.3.1. Установка обработчика сигнала SIGCHLD

Сигнал `SIGCHLD` передается родительскому процессу всякий раз, когда один из его потомков завершает работу. По умолчанию он игнорируется, но мы можем его перехватить, установив соответствующий обработчик. Внутри этого обработчика можно сделать вызов `wait()` (или аналогичный ему), чтобы убрать дочерний процесс-«зомби». Однако у этого подхода есть небольшая тонкость.

В разделах 20.10 и 20.12 мы видели, что при вызове обработчика сигнал, который этот вызов спровоцировал, временно блокируется (если при вызове `sigaction()` не был указан флаг `SA_NODEFER`); кроме того, стандартные сигналы, одним из которых и является `SIGCHLD`, не ставятся в очередь. Следовательно, если во время того, как обработчик `SIGCHLD` выполняется для уже завершенного потомка, еще несколько потомков успевают завершиться, сигнал `SIGCHLD` хоть и будет сгенерирован два раза, но дойдет до родителя в единственном экземпляре. В результате, если родительский обработчик `SIGCHLD` при каждом выполнении делает только один вызов `wait()`, он может пропустить некоторые дочерние процессы-«зомби».

Для решения этой проблемы внутри обработчика `SIGCHLD` можно поместить цикл и вызывать в нем `waitpid()` с флагом `WNOHANG`, пока больше не останется завершенных потомков. Тело обработчика `SIGCHLD` часто состоит из следующего кода, который просто утилизирует завершенных потомков без проверки их статуса:

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    continue;
```

Цикл, представленный выше, продолжается до тех пор, пока `waitpid()` не вернет либо `0`, что говорит о том, что потомков-«зомби» больше не осталось, либо `-1`, что свидетельствует об ошибке (вероятно, со статусом `ECHILD`, который указывает на отсутствие дочерних процессов).

Проблемы проектирования обработчиков SIGCHLD

Представьте, что на момент установки обработчика `SIGCHLD` у процесса уже есть завершенный потомок. Должно ли ядро немедленно сгенерировать сигнал `SIGCHLD` для родителя? Стандарт SUSv3 оставляет этот вопрос без ответа. Одни системы, в основном на базе System V, генерируют этот сигнал, а другие, включая Linux, — нет. Переносимое приложение может нивелировать эти отличия, установив обработчик `SIGCHLD` до создания каких-либо потомков (обычно это наиболее очевидное решение).

Еще одним моментом, на который стоит обратить внимание, является реентерабельность. В подразделе 21.1.2 отмечалось, что использование системного вызова (например, `waitpid()`) внутри обработчика сигнала может изменить значение глобальной переменной `errno`. Такое изменение может помешать главной программе задать это значение (в качестве примера см. обсуждение вызова `getpriority()` в разделе 35.1) или проверить

его после неудачного системного вызова. По этой причине иногда требуется, чтобы обработчик `SIGCHLD` в самом начале сохранял `errno` в локальной переменной, и затем восстанавливал ее значение перед самым возвращением. Пример этого подхода показан в листинге 26.5.

Пример программы

В листинге 26.5 приводится пример более сложного обработчика `SIGCHLD`. Этот обработчик выводит идентификатор процесса и статус ожидания каждого освобожденного потомка ❶. Чтобы продемонстрировать, что во время выполнения обработчика сигналы `SIGCHLD` не ставятся в очередь, мы искусственно продлеваем время работы обработчика, делая вызов `sleep()` ❷. Главная программа создает по одному дочернему процессу на каждый (целочисленный) аргумент командной строки ❹. Все эти потомки приостанавливаются на время, заданное в соответствующем аргументе (в секундах), после чего завершаются ❺. В листинге сессии данной программы, приведенном ниже, видно, что сигнал `SIGCHLD` поступает родителю всего два раза, хотя завершено было три потомка:

```
$ ./multi_SIGCHLD 1 2 4
16:45:18 Child 1 (PID=17767) exiting
16:45:18 handler: Caught SIGCHLD        Первый вызов обработчика
16:45:18 handler: Reaped child 17767 - child exited, status=0
16:45:19 Child 2 (PID=17768) exiting    Эти потомки завершаются во время…
16:45:21 Child 3 (PID=17769) exiting    первого вызова обработчика
16:45:23 handler: returning             Конец первого вызова обработчика
16:45:23 handler: Caught SIGCHLD        Второй вызов обработчика
16:45:23 handler: Reaped child 17768 - child exited, status=0
16:45:23 handler: Reaped child 17769 - child exited, status=0
16:45:28 handler: returning
16:45:28 All 3 children have terminated; SIGCHLD was caught 2 times
```

Обратите внимание на вызов `sigprocmask()` в листинге 26.5 ❸, который блокирует сигнал `SIGCHLD` до создания потомков. Это делается для того, чтобы обеспечить корректную работу цикла `sigsuspend()` внутри родителя. Если бы мы этого не сделали, а потомок завершился бы между проверкой значения `numLiveChildren` и выполнением вызова `sigsuspend()` (или как вариант вызова `pause()`), тогда вызов `sigsuspend()` заблокировался бы навсегда в ожидании сигнала, который уже был перехвачен ❻. Подробности о том, как решается проблема с этим видом состояния гонки, приводятся в разделе 22.9.

Листинг 26.5. Снятие завершенных дочерних процессов с помощью обработчика SIGCHLD
procexec/multi_SIGCHLD.c

```
#include <signal.h>
#include <sys/wait.h>
#include "print_wait_status.h"
#include "curr_time.h"
#include "tlpi_hdr.h"

static volatile int numLiveChildren = 0;
    /* Количество запущенных потомков, которых еще не дождались */

static void
sigchldHandler(int sig)
{
```

```
    int status, savedErrno;
    pid_t childPid;

    /* НЕБЕЗОПАСНО: этот обработчик использует функции printf(), printWaitStatus(),
       currTime(), небезопасные в контексте асинхронных сигналов; (подраздел 21.1.2) */

    savedErrno = errno;  /* На случай, если мы изменим значение 'errno' */

    printf("%s handler: Caught SIGCHLD\n", currTime("%T"));

    while ((childPid = waitpid(-1, &status, WNOHANG)) > 0) {
1       printf("%s handler: Reaped child %ld - ",
            currTime("%T"), (long) childPid);
        printWaitStatus(NULL, status);
        numLiveChildren--;
    }

    if (childPid == -1 && errno != ECHILD)
        errMsg("waitpid");
2   sleep(5);   /* Искусственно увеличиваем время выполнения обработчика */
    printf("%s handler: returning\n", currTime("%T"));

    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    int j, sigCnt;
    sigset_t blockMask, emptyMask;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s child-sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);   /* Отключаем буферизацию стандартного вывода */

    sigCnt = 0;
    numLiveChildren = argc - 1;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigchldHandler;
    if (sigaction(SIGCHLD, &sa, NULL) == -1)
        errExit("sigaction");

    /* Блокируем сигнал SIGCHLD, чтобы предотвратить его доставку в случае, если
       потомок завершится до начала цикла sigsuspend() в родителе (см. ниже) */

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SIGCHLD);
3   if (sigprocmask(SIG_SETMASK, &blockMask, NULL) == -1)
        errExit("sigprocmask");

4   for (j = 1; j < argc; j++) {
        switch (fork()) {
        case -1:
            errExit("fork");
```

```
        case 0:    /* Потомок приостанавливается, и затем завершается */
❺           sleep(getInt(argv[j], GN_NONNEG, "child-sleep-time"));
            printf("%s Child %d (PID=%ld) exiting\n",
                   currTime("%T"), j, (long) getpid());
            _exit(EXIT_SUCCESS);

        default:    /* Родитель входит в цикл для создания следующего потомка */
            break;
        }
    }
    /* Сюда доходит родитель: ждем сигнала SIGCHLD, пока не завершатся все потомки */

    sigemptyset(&emptyMask);
    while (numLiveChildren > 0) {
❻       if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        sigCnt++;
    }

    printf("%s All %d children have terminated; SIGCHLD was caught "
            "%d times\n", currTime("%T"), argc - 1, sigCnt);

    exit(EXIT_SUCCESS);
}
```

procexec/multi_SIGCHLD.c

26.3.2. Доставка сигнала SIGCHLD для остановленных потомков

По аналогии с тем, как вызов `waitpid()` может быть использован для отслеживания остановленных потомков, родитель может получать сигнал `SIGCHLD` в случае остановки одного из его дочерних процессов (тоже по сигналу). Этого можно добиться с помощью флага `SA_NOCLDSTOP`, когда вызов `sigaction()` устанавливает обработчик `SIGCHLD`. Если этот флаг опущен, сигнал `SIGCHLD` доставляется родителю при остановке одного из потомков; если флаг присутствует, этого не происходит (реализация вызова `signal()`, показанная в разделе 22.7, не предусматривает флага `SA_NOCLDSTOP`).

> Поскольку сигнал SIGCHLD по умолчанию игнорируется, флаг SA_NOCLDSTOP имеет значение только в том случае, если установить обработчик SIGCHLD. Более того, SIGCHLD — это единственный сигнал, на который влияет флаг SA_NOCLDSTOP.

Стандарт SUSv3 также позволяет отправлять родителю сигнал `SIGCHLD`, если один из его остановленных потомков возобновил работу (получив сигнал `SIGCONT`) — по аналогии с тем, как действует флаг `WCONTINUED` в вызове `waitpid()`. Эта возможность доступна в ядре Linux, начиная с версии 2.6.9.

26.3.3. Игнорирование завершенных дочерних процессов

Существует еще один способ работы с завершенными процессами. Явное изменение диспозиции сигнала `SIGCHLD` на `SIG_IGN` приводит к тому, что любой потомок, который впоследствии завершает работу, немедленно удаляется из системы, вместо того чтобы превратиться в «зомби». В этом случае последующий вызов `wait()` (или аналогичный)

не может вернуть никаких сведений о завершенном дочернем процессе, поскольку его статус просто сбрасывается.

> Стоит отметить, что, хотя диспозицией сигнала SIGCHLD по умолчанию является игнорирование, явная установка диспозиции SIG_IGN приводит к описанному выше поведению, отличному от поведения по умолчанию. SIGCHLD — единственный сигнал, который ведет себя таким образом.

В Linux, как и во многих реализациях UNIX, изменение диспозиции сигнала `SIGCHLD` на `SIG_IGN` не влияет на статус уже имеющихся потомков-«зомби», которых по-прежнему нужно ждать. Хотя в некоторых версиях UNIX (таких как Solaris 8) это приводит к удалению существующих дочерних процессов-«зомби».

Семантика `SIG_IGN` для сигнала `SIGCHLD` имеет длинную историю, берущую свое начало в System V. Поведение, описанное выше, предусмотрено стандартом SUSv3, однако данная семантика не является частью оригинального стандарта POSIX.1. Таким образом, игнорирование сигнала `SIGCHLD` в некоторых реализациях UNIX никак не влияет на создание процессов-«зомби». Единственный полностью переносимый способ предотвратить появление зомби заключается в использовании вызовов `wait()` или `waitpid()`, возможно даже внутри обработчика, установленного для `SIGCHLD`.

Флаг SA_NOCLDWAIT при вызове sigaction()

В стандарте SUSv3 описан флаг `SA_NOCLDWAIT` вызова `sigaction()`, с помощью которого можно изменять действие сигнала `SIGCHLD`. Этот флаг обеспечивает поведение, похожее на то, к которому приводит изменение действия `SIGCHLD` на `SIG_IGN`. Он был реализован в Linux 2.6.

Использование флага `SA_NOCLDWAIT` принципиально отличается от изменения действия `SIGCHLD` на `SIG_IGN` тем, что стандарт SUSv3 умалчивает, должен ли сигнал `SIGCHLD` передаваться родителю при завершении потомка. Иными словами, в отдельной реализации, если указан флаг `SA_NOCLDWAIT`, сигнал `SIGCHLD` может доставляться, и приложение может его перехватить (хотя обработчик не сможет получить статус потомка с помощью вызова `wait()`, поскольку ядро успевает уничтожить процесс-«зомби»). В некоторых реализациях UNIX, в том числе и в Linux, ядро действительно генерирует сигнал `SIGCHLD` для родителя. Но есть системы, в которых этого не происходит.

26.4. Резюме

Использование вызовов `wait()` и `waitpid()` (и других связанных с ними функций) позволяет родительскому процессу получать статус его завершенных и остановленных потомков. Этот статус сигнализирует, завершился ли дочерний процесс в штатном режиме (успешно или нет), завершился ли аварийно, был ли он остановлен или возобновлен по сигналу (во втором случае это сигнал `SIGCONT`).

Если родитель завершает свою работу, его потомки становятся «сиротами» и «удочеряются» процессом `init`, чей идентификатор равен `1`.

Когда завершается дочерний процесс, он становится «зомби» и удаляется из системы только после того, как его родитель получит его статус с помощью вызова `wait()` (или аналогичного). Долгоживущие программы, такие как командные оболочки и демоны, нужно проектировать так, чтобы они всегда могли получить статус созданного ими потомка, поскольку процессы в состоянии «зомби» не могут быть завершены, и в какой-то момент они переполнят таблицу процессов ядра.

Стандартный способ снятия завершенных дочерних процессов заключается в установлении обработчика для сигнала SIGCHLD. Этот сигнал доставляется родителю всякий раз, когда один из его потомков завершает работу, и, опционально, когда потомок останавливается по сигналу. Есть и другой, менее переносимый вариант: процесс может изменить диспозицию сигнала SIGCHLD на SIG_IGN, в результате чего статус завершенных потомков сразу же сбрасывается (и больше не может быть получен родителем) и они не становятся «зомби».

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 24.6.

26.5. Упражнения

26.1. Напишите программу, в которой проверяется утверждение, что при завершении родительского процесса вызов getppid() в его потомках возвращал 1 (идентификатор процесса init).

26.2. Представьте, что у вас имеется три процесса, которые соотносятся между собой как прародитель, родитель и потомок; при этом прародитель не вызывает wait() сразу же после завершения родителя, в результате чего последний становится «зомби». В какой момент потомок будет «удочерен» процессом init (на что будет указывать вызов getppid(), возвращающий 1): после завершения родителя или после того, как прародитель выполнит вызов wait()? Напишите программу, чтобы проверить свой ответ.

26.3. Замените в листинге 26.3 вызов waitpid() на waitid() (child_status.c). Вызов функции printWaitStatus() должен быть заменен кодом, который выводит подходящие поля из структуры siginfo_t, возвращенной из waitid().

26.4. В листинге 26.4 (make_zombie.c) используется вызов sleep(), благодаря которому дочерний процесс получает шанс выполниться и завершить работу до того, как родитель вызовет system(). Такой подход потенциально может привести к состоянию гонки. Измените программу, чтобы исключить эту возможность; используйте сигналы для синхронизации родителя и потомка.

27 Выполнение программы

Эта глава является логическим продолжением предыдущих глав, посвященных созданию и завершению процессов. Здесь мы рассмотрим системный вызов `execve()`, с помощью которого процесс может заменить текущую программу какой-то другой. Затем мы покажем, как реализовать функцию `system()`, которая позволяет вызывающему ее процессу выполнить произвольную консольную команду.

27.1. Выполнение новой программы: execve()

Системный вызов `execve()` загружает в память процесса новую программу. Во время этой операции старая программа удаляется вместе со стеком, данными и кучей и заменяется аналогичными частями новой программы. Выполнив инициализационный и загрузочный код из библиотеки C (например, статические конструкторы в C++ или функции языка C, объявленные с помощью атрибута `constructor`, доступного в `gcc` и описанного в разделе 42.4), новая программа начинает работу со своей функции `main()`.

Чаще всего вызов `execve()` применяется в потомке, созданном с помощью функции `fork()`, хотя его можно использовать и без предварительного запуска этой функции.

Системный вызов `execve()` задействуется в качестве основы для различных библиотечных функций, имена которых начинаются с `exec`. Каждая из них предоставляет свой особый интерфейс к одним и тем же возможностям. Загрузка новой программы посредством любой из этих функций обычно называется операцией `exec` или просто обозначается как `exec()`. Мы начнем с описания вызова `execve()`, после чего перейдем к библиотечным функциям.

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Ничего не возвращает при успешном завершении;
возвращает `-1` при возникновении ошибки

Аргумент `pathname` содержит путь к новой программе, которая должна быть загружена в память процесса. Этот путь может быть абсолютным (на что указывает знак `/` в начале) или относительным (то есть зависеть от текущего каталога вызывающей программы).

Аргумент `argv` содержит параметры командной строки, которые будут переданы новой программе. Этот аргумент соответствует второму аргументу (`argv`) функции `main()` в языке C и имеет ту же форму; это список указателей, ссылающихся на символьные строки, который завершается значением `NULL`. Значение `argv[0]` соответствует имени команды и обычно совпадает с названием исполняемого файла (то есть последней части `pathname`).

Последний аргумент, `envp`, предоставляет новой программе список переменных среды. Он соответствует массиву `environ` новой программы; это список указателей, ссылающихся на символьные строки вида *`ИМЯ=значение`*, который завершается значением `NULL` (см. раздел 6.7).

Linux предоставляет файл /proc/PID/exe, который является символьной ссылкой и содержит абсолютный путь к исполняемому файлу, запущенному соответствующим процессом.

После вызова `execve()` идентификатор процесса остается неизменным, поскольку сам процесс продолжает существовать. Не меняются и некоторые другие атрибуты, как описывается в разделе 28.4.

Если для файла программы, указанной в аргументе `pathname`, выставлен бит установки ID пользователя (или группы), значение соответствующего действующего ID становится равным значению ID пользователя (группы) владельца файла. Это механизм временного предоставления привилегий пользователям на период выполнения программы (см. раздел 9.3).

Вне зависимости от того, были ли изменены действующие идентификаторы (имена) пользователя и группы, вызов `execve()` копирует их в свои сохраненные биты.

Поскольку в случае успеха вызов `execve()` заменяет вызвавшую его программу, он никогда не возвращает значения. Нам не нужно проверять результат выполнения `execve()`, поскольку он всегда будет равен `-1`. Иначе говоря, сам факт того, что вызов что-то вернул, уже свидетельствует об ошибке, причину которой обычно можно узнать с помощью аргумента `errno`. Среди ошибок, возвращаемых таким образом, можно выделить следующие.

- `EACCES` — аргумент `pathname` не указывает на обычный файл, файл не является исполняемым или запрещено чтение содержимого одного из каталогов, являющегося частью `pathname` (то есть для каталога не выставлен признак «на исполнение»). Как вариант, файл может храниться в файловой системе, подключенной с использованием флага `MS_NOEXEC` (см. подраздел 14.8.1).
- `ENOENT` — файл, на который ссылается `pathname`, не существует.
- `ENOEXEC` — файл, на который ссылается `pathname`, помечен как исполняемый, но система не распознает его формат. Возможно, это скрипт, в первой строке которого не был указан интерпретатор (такая строка должна начинаться с символов `#!`).
- `ETXTBSY` — файл, на который ссылается `pathname`, открыт для записи другим процессом (см. подраздел 4.3.2).
- `E2BIG` — общий объем памяти, требуемый для хранения списка аргументов и переменных среды, превышает допустимое ограничение.

Ошибки, перечисленные выше, могут также возникнуть в случае, если любое из этих условий будет исполнено для интерпретатора, который должен выполнить скрипт (см. раздел 27.3), или для ELF-интерпретатора, с помощью которого выполняется программа.

Формат ELF (Executable and Linking Format) является широко распространенной спецификацией, описывающей структуру исполняемый файлов. Обычно во время выполнения образ процесса строится на основе сегментов исполняемого файла (см. раздел 6.3). Однако спецификация ELF также позволяет указать интерпретатор (заголовочный элемент PT_INTERP) для выполнения программы. Если интерпретатор определен, ядро строит образ процесса из сегментов указанного исполняемого файла интерпретатора. Затем уже сам интерпретатор должен загрузить и выполнить программу. Больше о роли интерпретаторов в формате ELF можно узнать в главе 41; там же будут даны отсылки к более углубленной информации.

Пример программы

Пример использования вызова `execve()` показан в листинге 27.1. Данный код создает списки аргументов и переменных среды для новой программы и затем вызывает `execve()`, задействуя в качестве пути к исполняемому файлу аргумент командной строки (`argv[1]`).

В листинге 27.2 приводится программа, которая должна быть запущена кодом из предыдущего листинга. Все, что она делает, — это выводит аргументы командной строки

и переменные среды (доступ к последним осуществляется посредством глобальной переменной `environ`, как описывается в разделе 6.7).

Работа программ из листингов 27.1 и 27.2 показана в следующей сессии командной оболочки (в этом примере для задания исполняемого файла используется относительный путь):

```
$ ./t_execve ./envargs
argv[0] = envargs       Весь этот текст выводится командой envargs
argv[1] = hello world
argv[2] = goodbye
environ: GREET=salut
environ: BYE=adieu
```

Листинг 27.1. Использование вызова execve() для запуска новой программы

procexec/t_execve.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *argVec[10];    /* С запасом */
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);
    argVec[0] = strrchr(argv[1], '/');
                    /* Получаем последнюю часть имени файла из argv[1] */
    if (argVec[0] != NULL)
        argVec[0]++;
    else
        argVec[0] = argv[1];
    argVec[1] = "hello world";
    argVec[2] = "goodbye";
    argVec[3] = NULL;    /* В конце списка должно быть значение NULL */
    execve(argv[1], argVec, envVec);
    errExit("execve");   /* Если мы сюда добрались, что-то пошло не так */
}
```

procexec/t_execve.c

Листинг 27.2. Вывод списка аргументов и переменных среды

procexec/envargs.c

```
#include "tlpi_hdr.h"

extern char **environ;

int
main(int argc, char *argv[])
{
    int j;
    char **ep;

    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);

    for (ep = environ; *ep != NULL; ep++)
        printf("environ: %s\n", *ep);
```

```
    exit(EXIT_SUCCESS);
}
```

procexec/envargs.c

27.2. Библиотечные функции семейства exec()

Библиотечные функции, описанные в этом разделе, предоставляют альтернативный интерфейс к операции `exec()`. Все они работают поверх вызова `execve()`, отличаются от него и различаются между собой только тем, каким образом указываются имя программы, список аргументов и переменные среды.

```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
           /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
           /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
           /* , (char *) NULL */);
```

Ни одна из этих функций не возвращает результат при успешном выполнении; при ошибке все они возвращают `-1`

На разницу между этими функциями указывают последние буквы их названий. Различия между ними собраны в табл. 27.1 и подробно описаны в следующем списке.

- Большинство функций семейства `exec()` ожидают получить путь к программе, которую нужно загрузить. Но `execlp()` и `execvp()` позволяют указать лишь имя файла. В этом случае файл ищется в каталогах из списка, заданного в переменной среды `PATH` (подробнее о ней читайте ниже). Такой же поиск выполняется командной оболочкой, когда в нее вводят название команды. Чтобы обозначить эту особенность, имена этих функций содержат букву `p` (от `PATH`). Переменная `PATH` игнорируется, если в имени файла присутствует слеш (`/`): в этом случае имя воспринимается как относительный или абсолютный путь.
- Вместо того чтобы передавать новой программе список аргументов `argv` в виде массива, функции `execle()`, `execlp()` и `execl()` используют для этого список. Первый из этих аргументов соответствует элементу `argv[0]` в функции `main()` новой программы и, следовательно, совпадает с аргументом `filename` или последней частью аргумента `pathname`. В конце списка аргументов должен находиться нулевой указатель, чтобы данные вызовы могли определить конец (в прототипах, приведенных выше, это требование обозначено закомментированным `(char *) NULL`). Чтобы отличить данные функции от тех, что передают список аргументов в виде массива с `NULL` в конце, их имена содержат букву `l` (от list — «список»). Функции, которые применяют для этой цели массив (`execve()`, `execvp()` и `execv()`), обозначены буквой `v` (от vector — «вектор»).
- Функции `execve()` и `execle()` позволяют программисту явно задать переменные среды для новой программы, используя аргумент `envp` — массив указателей на символьные строки, который заканчивается значением `NULL`. Чтобы обозначить этот факт, имена данных функций заканчиваются буквой `e` (от environment — «среда»). Остальные функции семейства `exec()` применяют для новой программы среду вызывающего процесса (то есть содержимое глобальной переменной `environ`).

В библиотеке glibc версии 2.11 появилась нестандартная функция execvpe(file, argv, envp). Она похожа на execvp(), но вместо того, чтобы наследовать переменные среды для новой программы из environ, она принимает их в виде аргумента envp (по аналогии с execve() и execle()).

На следующих нескольких страницах будут демонстрироваться примеры использования этих разновидностей вызова `exec()`.

Таблица 27.1. Краткое перечисление различий между разными функциями вида exec()

Функция	Задание программного файла (-, p)	Задание аргументов (v, l)	Источник переменных среды (e, -)
execve()	pathname	Массив	Аргумент envp
execle()	pathname	Список	Аргумент envp
execlp()	filename + PATH	Список	environ вызывающего процесса
execvp()	filename + PATH	Массив	environ вызывающего процесса
execv()	pathname	Массив	environ вызывающего процесса
execl()	pathname	Список	environ вызывающего процесса

27.2.1. Переменная среды PATH

Функции `execvp()` и `execlp()` позволяют нам указать только имя файла, который нужно выполнить. Для его поиска они используют переменную среды `PATH`. Значение `PATH` представляет собой строку с именами каталогов *(префиксы путей)*, которые разделяются двоеточиями. Например, в следующем значении `PATH` содержится пять каталогов:

```
$ echo $PATH
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin:.
```

При входе в систему значение `PATH` настраивается общесистемными и пользовательскими скриптами. Поскольку потомок наследует копию переменных среды своего родителя, каждый процесс, создаваемый командной оболочкой для выполнения команды, получает дубликат переменной `PATH` из оболочки.

Пути, заданные в `PATH`, могут быть либо абсолютными (если начинаются со знака `/`), либо относительными. Относительный путь интерпретируется с учетом текущего каталога вызывающего процесса. Текущий каталог можно задать с помощью символа `.` (точка), как в вышеприведенном примере.

Текущий каталог в переменной PATH можно также задать в виде префикса нулевой длины — двух двоеточий подряд или начального/завершающего двоеточия (например, /usr/bin:/bin:). В стандарте SUSv3 такой подход считается устаревшим; текущий каталог следует указывать явно с помощью символа . (точка).

Если переменная `PATH` не определена, функции `execvp()` и `execlp()` по умолчанию используют список путей вида `.:/usr/bin:/bin`.

В качестве меры предосторожности учетная запись администратора (root) обычно настраивается таким образом, чтобы текущий каталог не попал в переменную `PATH`. Благодаря этому администратор не может случайно выполнить файлы из каталога, в котором находится (и содержимое которого мог изменить злоумышленник), если имена этих файлов совпадают со стандартными командами или рассчитаны на опечатку в написании распространенных команд (например, `sl` вместо `ls`). В некоторых дистрибутивах Linux

текущий каталог исключается из значения PATH и для обычных пользователей. Именно этот вариант подразумевается при выводе всех сессий командной оболочки в этой книге; по этой причине мы всегда добавляем префикс ./ к именам программ, которые запускаются из текущего каталога (полезным побочным эффектом этого является возможность визуально отличать наши программы от стандартных команд в сессиях командной оболочки, которые здесь приводятся).

Функции execvp() и execlp() ищут файл в каждом каталоге, указанном в переменной PATH, начиная с первого элемента списка и дальше, пока не найдут. Такое использование переменной PATH является полезным, если мы не знаем местоположение выполняемого файла или не хотим создавать в своем коде жесткую зависимость от этого местоположения.

Применения функций execvp() и execlp() в программах с установленным идентификатором пользователя или группы следует избегать или по крайней мере относиться к ним с большой осторожностью. В частности, нужно тщательно следить за использованием переменной среды PATH, чтобы случайно не запустить вредоносную программу. На практике это означает, что приложение должно перезаписывать ранее определенное значение PATH, оставляя в нем только хорошо известные и безопасные каталоги.

В листинге 27.3 приводится пример использования вызова execlp(). Сессия командной оболочки, показанная ниже, демонстрирует выполнение программы для запуска команды echo (/bin/echo):

```
$ which echo
/bin/echo
$ ls -l /bin/echo
-rwxr-xr-x 1 root 15428 Mar 19 21:28 /bin/echo
$ echo $PATH                  Выводим содержимое переменной среды PATH
/home/mtk/bin:/usr/local/bin:/usr/bin:/bin   /bin находится в PATH
$ ./t_execlp echo execlp()    execlp() использует PATH для успешного
                              нахождения команды echo
hello world
```

Строка hello world, которая выводится выше, была передана вызову execlp() в качестве третьего аргумента (см. листинг 27.3).

Теперь переопределим значение PATH, исключив из него каталог /bin, в которой находится программа echo:

```
$ PATH=/home/mtk/bin:/usr/local/bin:/usr/bin
$ ./t_execlp echo
ERROR [ENOENT No such file or directory] execlp
$ ./t_execlp /bin/echo
hello world
```

Как можно заметить, при передаче вызову execlp() имени файла (то есть строки без слешей), он выдает ошибку, так как файл с именем echo не был найден ни в одном каталоге, перечисленном в переменной PATH. С другой стороны, если передать вызову execlp() путь, содержащий один или несколько слешей, содержимое PATH будет проигнорировано.

Листинг 27.3. Использование вызова execlp() для поиска файла в PATH

procexec/t_execlp.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);
```

```
    execlp(argv[1], argv[1], "hello world", (char *) NULL);
    errExit("execlp");  /* Если мы добрались сюда, то что-то пошло не так */
}
```
procexec/t_execlp.c

27.2.2. Задание аргументов программы в виде списка

Если на момент написания программы нам известно количество аргументов для вызова внешней команды, мы можем указывать их внутри вызовов `execle()`, `execlp()` или `execl()`. Это может оказаться удобным, поскольку требует меньше кода по сравнению с формированием списка аргументов `argv`. Программа, представленная ниже, делает то же, что и листинг 27.1, но вместо вызова `execve()` использует `execle()`.

Листинг 27.4. Использование вызова execle() для задания аргументов программы в виде списка

procexec/t_execle.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };
    char *filename;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    filename = strrchr(argv[1], '/');    /* Получаем имя файла из argv[1] */
    if (filename != NULL)
        filename++;
    else
        filename = argv[1];

    execle(argv[1], filename, "hello world", "goodbye", (char *) NULL, envVec);
    errExit("execle");  /* Если мы добрались сюда, то что-то пошло не так */
}
```
procexec/t_execle.c

27.2.3. Передача переменных среды вызывающего процесса новой программе

Функции `execlp()`, `execvp()`, `execl()` и `execv()` не позволяют программисту явно указывать список переменных среды; вместо этого новая программа наследует их от вызывающего процесса (см. раздел 6.7). В одних случаях это нам подходит, а в других — нет. Из соображений безопасности иногда лучше снабжать программу заранее известным списком переменных среды. Более детально об этом рассказывается в разделе 38.8.

Процедура наследования новой программой среды вызывающего процесса во время вызова `execl()` продемонстрирована в листинге 27.5. Сначала эта программа использует функцию `putenv()`, чтобы изменить переменные среды, которые она наследует от командной оболочки в результате вызова `fork()`. Затем запущенная программа `printenv` выводит значения переменных среды `USER` и `SHELL`. Запустив этот код, мы увидим следующее:

```
$ echo $USER $SHELL           Выводим некоторые из переменных среды командной оболочки
blv /bin/bash
$ ./t_execl
Initial value of USER: blv    Копия среды была унаследована от командной оболочки
Britta                        Эти две строки выводятся запущенной программой printenv
/bin/bash
```

Листинг 27.5. Передача среды вызывающего процесса новой программе с помощью execl()

procexec/t_execl.c

```
#include <stdlib.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Initial value of USER: %s\n", getenv("USER"));
    if (putenv("USER=britta") != 0)
        errExit("putenv");

    execl("/usr/bin/printenv", "printenv", "USER", "SHELL", (char *) NULL);
    errExit("execl");  /* Если мы добрались сюда, то что-то пошло не так */
}
```

procexec/t_execl.c

27.2.4. Выполнение файла через ссылку на его дескриптор: fexecve()

С версии 2.3.2 библиотека `glibc` предоставляет функцию `fexecve()`, которая похожа на `execve()`, но позволяет указать исполняемый файл в виде дескриптора `fd`, а не с помощью пути к файлу. Использование `fexecve()` подходит для приложений, которые перед выполнением файла хотят его открыть и проверить его контрольную сумму.

```
#define _GNU_SOURCE
#include <unistd.h>

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Не возвращает значение при успешном завершении; при ошибке возвращает `-1`

Мы можем открыть и прочитать/проверить содержимое файла без функции `fexecve()` (используя вызов `open()`). Однако это сделало бы возможным подмену файла между его открытием и выполнением, потому что удержание дескриптора открытого файла не мешает создать новый файл с тем же именем. Таким образом, проверенное содержимое могло бы отличаться от того, что непосредственно запускается.

27.3. Интерпретируемые скрипты

Интерпретатор — это программа, которая считывает и выполняет текстовые команды (в отличие от *компилятора*, транслирующего исходный код в машинный, который затем может быть выполнен процессором или виртуальной машиной). В качестве примеров

интерпретатора можно привести различные командные оболочки UNIX, а также такие программы, как awk, sed, perl, python и ruby. Обычно интерпретаторы позволяют считывать и выполнять команды не только в интерактивном режиме, но и путем загрузки их из текстового файла, который называют *скриптом* (или *сценарием*).

Ядра в системах UNIX позволяют запускать интерпретируемые скрипты тем же способом, что и скомпилированные программы. Для этого должно выполняться два условия: во-первых, файл скрипта должен быть исполняемым и, во-вторых, в начале файла должна находиться строка, в которой указан путь к подходящему интерпретатору. Эта строка имеет следующий вид.

```
#! путь-к-интерпретатору [ опциональные-аргументы ]
```

Символы `#!` («шебанг») должны находиться в начале строки; при желании между ними и путем к интерпретатору можно указать пробел. Переменная окружения `PATH` *не* используется при анализе этого пути, поэтому обычно путь должен быть абсолютным. Относительные пути тоже допускаются, хотя их принято избегать; они вычисляются относительно текущего каталога процесса, который запускает интерпретатор. Путь к интерпретатору отделяется от опциональных аргументов (назначение которых мы объясним чуть ниже) с помощью пробельного символа. Сами аргументы не должны содержать пробельных символов.

Скрипты командной строки в UNIX (или просто shell-скрипты) обычно начинаются со следующей строчки, в которой указана командная оболочка для их выполнения:

```
#!/bin/sh
```

> Опциональный аргумент в первой строчке интерпретируемого скрипта не должен содержать пробельных символов, потому что в противном случае поведение будет сильно зависеть от конкретной реализации. В Linux пробельный символ в опциональном аргументе не означает ничего особенного — весь текст от начала аргумента и до конца строки воспринимается как единое слово (которое передается интерпретатору в качестве параметра, как будет описано ниже). Стоит отметить, что такое обращение с пробельными символами контрастирует с командной оболочкой, в которой их используют для разделения строк на отдельные слова.
>
> Одни реализации UNIX воспринимают пробельные символы в опциональном аргументе так же, как Linux, другие — иначе. Раньше в системе FreeBSD после пути к интерпретатору можно было указывать несколько аргументов, разделенных пробелами (после чего те передавались интерпретатору в виде отдельных слов); но, начиная с версии 6.0, FreeBSD ведет себя так же, как Linux. В Solaris 8 пробельные символы завершают опциальный аргумент; любой текст, следующий после них в начальной строке, игнорируется.

В ядре Linux существует ограничение на длину начальной строки скрипта — 127 символов (не считая символа перехода на новую строку в конце). Все, что выходит за его пределы, игнорируется.

Способ определения интепретатора скрипта с помощью `#!` не описан в стандарте SUSv3, но доступен в большинстве реализаций UNIX.

> Ограничение, накладываемое на длину начальной строки, варьируется в зависимости от конкретной системы. Например, OpenBSD 3.1 имеет ограничение 64 символа, а Tru64 5.1 — 1024. В некоторых старых реализациях (таких как SunOS 4) ограничение составляло 32 символа.

Выполнение интерпретируемых скриптов

Поскольку скрипт не содержит двоичного машинного кода, очевидно, что при его запуске с помощью вызова `execve()` должно происходить что-то необычное. Если `execve()`

обнаруживает в начале скриптового файла двухбайтную последовательность `#!`, он извлекает оставшуюся часть строки (путь и аргументы) и выполняет файл интерпретатора со следующим списком аргументов:

```
путь-к-интерпретатору [ опциональные-аргументы ] путь-к-скрипту аргументы…
```

Путь к интерпретатору и опциональные аргументы берутся из начальной строки файла, путь к скрипту передается вызову `execve()`, а все дальнейшие аргументы указываются через параметр `argv` в том же вызове (при этом элемент `argv[0]` игнорируется). Происхождение каждого аргумента в скрипте показано на рис. 27.1.

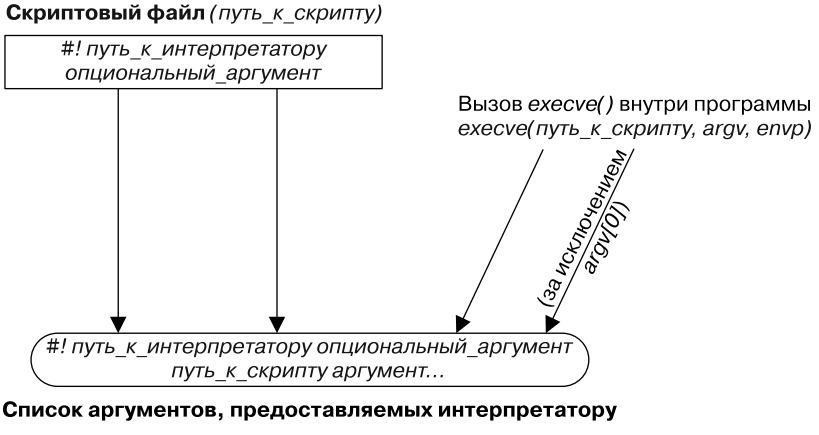


Рис. 27.1. Список аргументов, который передается выполняемому скрипту

Происхождение аргументов интерпретатора можно продемонстрировать на примере скрипта, который использует программу из листинга 6.2 (`necho.c`). Эта программа просто выводит все аргументы командной строки, которые ей переданы. Воспользовавшись программой из листинга 27.1, чтобы запустить наш скрипт, мы увидим следующее:

```
$ cat > necho.script              Создаем скрипт
#!/home/mtk/bin/necho some argument
Some junk
Нажмите Ctrl-D
$ chmod +x necho.script           Делаем скрипт исполняемым
$ ./t_execve necho.script         И выполняем скрипт
argv[0] = /home/mtk/bin/necho     Первых три аргумента сгенерированы ядром
argv[1] = some argument           Аргумент скрипта воспринимается как единое слово
argv[2] = necho.script            Это путь к скрипту
argv[3] = hello world             Это был элемент argVec[1], переданный в execve()
argv[4] = goodbye                 А это был элемент argVec[2]
```

В этом примере наш «интерпретатор» (`necho`) игнорирует содержимое скрипта (`necho.script`), поэтому вторая его строка (`Some junk`) никак не влияет на его выполнение.

Большинство командных оболочек и интерпретаторов в UNIX воспринимают символ `#` как начало комментария, поэтому при интерпретации скриптов они игнорируют начальную строку `#!`.

Использование необязательных аргументов скрипта

С помощью необязательных аргументов в начальной строке скрипта можно указать параметры командной строки для интерпретатора. Это бывает полезно при работе с такими интерпретаторами, как awk.

> Интерпретатор awk стал частью системы UNIX в конце 1970-х годов. Язык awk описан во множестве книг, одна из которых написана его непосредственными создателями [Aho et al., 1988]. Из их инициалов, к слову, состоит название языка. Его сильной стороной является быстрое создание прототипов приложений, предназначенных для обработки текста. По своей архитектуре (слабая типизация, богатый набор инструкций для работы с текстом, синтаксис, основанный на C) awk является предком таких широко используемых на сегодня языков, как JavaScript и PHP.

Передать скрипт интерпретатору awk можно двумя способами. По умолчанию скрипт просто указывается в качестве первого аргумента командной строки:

```
$ awk 'script' input-file...
```

Но скрипт также может находиться в файле. Например, следующий awk-скрипт выводит длину самой длинной входящей строки:

```
$ cat longest_line.awk
#!/usr/bin/awk
length > max    { max = length; }
END             { print max; }
```

Попытаемся запустить этот скрипт с помощью следующего кода на языке C:

```
execl("longest_line.awk", "longest_line.awk", "input.txt", (char *) NULL);
```

Этот вызов execl(), в свою очередь, запускает функцию execve(), которая выполняет команду awk со следующими аргументами:

```
/usr/bin/awk longest_line.awk input.txt
```

Данный вызов execve() завершается неудачей, поскольку awk интерпретирует строку longest_line.awk как скрипт, содержащий некорректную команду. Нам нужно как-то сообщить awk о том, что этот аргумент является именем файла со скриптом. Для этого можно воспользоваться параметром -f, указав его в качестве опционального аргумента в начальной строке скрипта. Так awk сможет понять, что в следующем аргументе находится скриптовый файл:

```
#!/usr/bin/awk -f
length > max    { max = length; }
END             { print max; }
```

Теперь наш вызов execl() генерирует следующий список аргументов:

```
/usr/bin/awk -f longest_line.awk input.txt
```

Это приводит к успешному запуску интерпретатора awk, который обрабатывает файл input.txt.

Выполнение скриптов с помощью вызовов execlp() и execvp()

Обычно отсутствие в скрипте начальной строки (начинающейся с #!) приводит к неудачному завершению функций exec(). Однако вызовы execlp() и execvp() ведут себя немного иначе. Как вы помните, с помощью переменной среды PATH они получают список каталогов,

в которых ищется исполняемый файл. Если найденный файл действительно является исполняемым, но состоит не из бинарного кода и первая строка не начинается с символов `#!`, эти вызовы обращаются к командной оболочке, чтобы та интерпретировала файл. В контексте Linux это означает, что такие файлы обрабатываются так, как будто начинаются с `#!/bin/sh`.

27.4. Дескрипторы файлов и вызовы exec()

Все файловые дескрипторы, открываемые программой, которая вызывает `exec()`, остаются открытыми на протяжении выполнения этого вызова и доступны для использования в новой программе. Часто это может быть полезным, потому что файлы, открытые вызывающей программой в определенных дескрипторах, автоматически становятся доступными для новой программы (которая при этом не должна знать их имена или открывать их заново).

Командная оболочка пользуется этой возможностью для перенаправления ввода/вывода программ, которые в ней выполняются. Представьте, к примеру, что мы ввели следующую команду:

```
$ ls /tmp > dir.txt
```

Для ее выполнения командная оболочка проделывает следующие операции.

1. Выполняется вызов `fork()`, который создает дочерний процесс, представляющий собой копию командной оболочки (то есть в том числе и копию команды).
2. Дочерняя командная оболочка открывает для вывода файл `dir.txt`, используя файловый дескриптор 1 (стандартный вывод). Это можно сделать одним из двух способов.
   - Дочерняя командная оболочка закрывает дескриптор 1 (`STDOUT_FILENO`), после чего открывает файл `dir.txt`. Поскольку вызов `open()` всегда использует наименьший доступный номер дескриптора, а стандартный ввод (дескриптор 0) остается открытым, файл будет открыт в дескрипторе 1.

     Код будет выглядеть примерно так:

```
fd = open("dir.txt", O_WRONLY | O_CREAT,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
          /* rw-rw-rw- */
if (fd != STDOUT_FILENO) {
    dup2(fd, STDOUT_FILENO);
    close(fd);
}
```

   - Командная оболочка открывает файл `dir.txt`, получая новый файловый дескриптор. Затем, если этот дескриптор не соответствует стандартному выводу, она использует вызов `dup2()`, чтобы принудить стандартный вывод стать дубликатом нового дескриптора. После этого новый дескриптор закрывается за ненадобностью (этот подход является более безопасным, чем предыдущий, поскольку он не полагается на открытие дескриптора с наименьшим номером).
3. Дочерняя командная оболочка выполняет команду `ls`, которая записывает результат своей работы в стандартный вывод, то есть в файл `dir.txt`.

Приводимое здесь объяснение того, как командная оболочка выполняет перенаправление ввода/вывода, упрощает некоторые моменты. В частности, некоторые так называемые встроенные команды выполняются оболочкой напрямую, без вызовов fork() или exec(). В контексте перенаправления ввода/вывода с ними нужно работать особым образом.

Консольную команду делают встроенной по одной из двух причин: повышенная эффективность или доступ к ресурсам оболочки. Некоторые часто используемые команды, такие как pwd, echo и test, достаточно простые для того, чтобы их имело смысл реализовывать как встроенные. Другие команды встраиваются в оболочку для того, чтобы иметь возможность изменять ее внутреннюю информацию, устанавливать атрибуты ее процесса или как-то влиять на ее работу. Например, команда cd должна изменять текущий каталог самой командной оболочки, поэтому ее нельзя выполнять в рамках отдельного процесса. Среди прочих команд, встроенных для того, чтобы пользоваться внутренними ресурсами оболочки, можно выделить exec, exit, read, set, source, ulimit, umask, wait и команды управления заданиями оболочки — jobs, fg и bg. Полный перечень встроенных команд, поддерживаемых вашей командной оболочкой, ищите на соответствующей справочной странице.

Флаг закрытия при выполнении (close-on-exit) FD_CLOEXEC

Иногда имеет смысл сделать так, чтобы файловые дескрипторы закрывались до вызова `exec()`. В частности, если мы запускаем неизвестную программу (то есть написанную не нами) из привилегированного процесса или программу, которой не нужны дескрипторы для открытых нами файлов, правила безопасного программирования требуют от нас убедиться в том, чтобы все ненужные файловые дескрипторы были закрыты до загрузки новой программы. Этого можно было бы достичь, вызывая `close()` для всех таких дескрипторов, но такой подход имеет следующие ограничения.

- Дескриптор файла может быть открыт библиотечной функцией. Эта функция не может принудить главную программу к закрытию дескриптора до выполнения вызова `exec()` (следует взять за правило, что библиотечные функции всегда должны устанавливать флаг `FD_CLOEXEC` для любых открываемых ими файлов, используя описанный ниже метод).
- Если вызов `exec()` по какой-то причине завершается неудачно, может иметь смысл оставить файловые дескрипторы открытыми. Если они уже закрылись, их восстановление со ссылками на те же файлы может оказаться затруднительным или вообще невозможным.

В связи с этим ядро предоставляет для каждого файлового дескриптора флаг `FD_CLOEXEC`. Если он установлен, дескриптор автоматически закрывается после успешного выполнения `exec()`, но остается открытым, если вызов завершается неудачей. Доступ к данному флагу можно получить с помощью системного вызова `fcntl()` (см. раздел 5.2). `fcntl()` поддерживает операцию `F_GETFD`, которая извлекает копию флагов файлового дескриптора:

```
int flags;

flags = fcntl(fd, F_GETFD);
if (flags == -1)
    errExit("fcntl");
```

После извлечения этих флагов мы можем их обновить с помощью второго вызова `fcntl()` с аргументом `F_SETFD`, предварительно изменив бит `FD_CLOEXEC`:

```
flags |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, flags) == -1)
   errExit("fcntl");
```

FD_CLOEXEC на самом деле является единственным битом, который используется в флагах файлового дескриптора. Он соответствует значению 1. В старых программах иногда можно встретить установку флага FD_CLOEXEC с помощью кода fcntl(fd, F_SETFD, 1), который полагается на тот факт, что при этом не будут затронуты никакие другие биты. Теоретически

это может быть не так (в будущем какая-нибудь UNIX-система может обзавестись дополнительными битами в списке флагов), поэтому вам следует использовать методику, приведенную в основном тексте.

Многие реализации UNIX, включая Linux, позволяют изменять флаг FD_CLOEXEC с помощью нестандартных вызовов ioctl(). Установить флаг FD_CLOEXEC для дескриптора fd можно посредством кода ioctl(fd, FIOCLEX); для его сброса достаточно вызвать ioctl(fd, FIONCLEX).

Когда для создания дубликата файлового дескриптора используются вызовы `dup()`, `dup2()` или `fcntl()`, в полученной копии флаг `FD_CLOEXEC` всегда сбрасывается (это поведение сложилось исторически и закреплено в SUSv3).

Изменение флага `FD_CLOEXEC` продемонстрировано в листинге 27.6. В зависимости от наличия аргумента командной строки (любого набора символов) данная программа сначала устанавливает флаг `FD_CLOEXEC` для стандартного вывода, а затем выполняет команду `ls`. Вот что мы увидим после ее запуска:

```
$ ./closeonexec          Выполняем ls, не закрывая стандартный вывод
-rwxr-xr-x 1 mtk users 28098 Jun 15 13:59 closeonexec
$ ./closeonexec n        Устанавливаем для стандартного вывода флаг FD_CLOEXEC
ls: write error: Bad file descriptor
```

Мы можем видеть, что при втором запуске команда `ls` обнаруживает факт закрытия стандартного вывода и возвращает сообщение в стандартный вывод ошибок.

Листинг 27.6. Установка флага FD_CLOEXEC для файлового дескриптора

procexec/closeonexec.c

```
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int flags;

    if (argc > 1) {
        flags = fcntl(STDOUT_FILENO, F_GETFD);       /* Извлекаем флаги */
    if (flags == -1)
        errExit("fcntl - F_GETFD");

    flags |= FD_CLOEXEC;                             /* Устанавливаем FD_CLOEXEC */

    if (fcntl(STDOUT_FILENO, F_SETFD, flags) == -1)  /* Обновляем флаги */
        errExit("fcntl - F_SETFD");
    }

    execlp("ls", "ls", "-l", argv[0], (char *) NULL);
    errExit("execlp");
}
```

procexec/closeonexec.c

27.5. Сигналы и вызов exec()

Во время выполнения вызова `exec()` текст существующего процесса удаляется. Этот текст мог содержать обработчики сигналов, установленные вызывающей программой. Поскольку обработчики исчезли, ядро изменяет диспозиции для всех перехватываемых

сигналов на SIG_DFL. Диспозиции всех остальных сигналов (то есть SIG_IGN или SIG_DFL) остаются без изменений. Это требование стандарта SUSv3.

В стандарте SUSv3 игнорирование сигнала SIGCHLD оговаривается отдельно (в подразделе 26.3.3 уже отмечалось, что это позволяет избежать возникновения процессов-«зомби»). Однако в нем не уточняется, игнорируется ли данный сигнал на протяжении всего вызова exec(), или же его действие сбрасывается к SIG_DFL. В Linux применяется первый вариант, но в некоторых реализациях UNIX (таких как Solaris) происходит сбрасывание. Из этого следует, что для достижения максимальной переносимости программ, которые игнорируют сигнал SIGCHLD, нам следует выполнять signal(SIGCHLD, SIG_DFL) перед вызовом exec() и исходить из того, что исходной диспозицией этого сигнала является SIG_DFL.

Удаление данных, кучи и стека старой программы также означает, что любой альтернативный стек сигналов, полученный в результате вызова sigaltstack() (см. раздел 21.3), теряется. Поскольку этот стек не сохраняется во время выполнения exec(), бит SA_ONSTACK тоже сбрасывается для всех сигналов.

Во время выполнения exec() у процесса сохраняются сигнальная маска и сигналы, находящиеся в состоянии ожидания. Это позволяет блокировать и складывать в очередь сигналы для новой программы. Однако в стандарте SUSv3 отмечается, что многие существующие приложения ошибочно предполагают, что определенные сигналы при их запуске изначально имели диспозицию SIG_DFL или были разблокированными (в частности, стандарты языка C содержат куда менее конкретную спецификацию сигналов, в которой не упоминается их блокирование; следовательно, программы на этом языке, написанные на несовместимых с UNIX системах, не знают о том, что сигналы нужно разблокировать). По этой причине в стандарте SUSv3 рекомендуется не блокировать и не игнорировать сигналы во время выполнения сторонних программ с помощью exec(). Под сторонними подразумеваются программы, написанные не нами. Если приложение разработали мы сами или если мы знаем, как оно ведет себя с сигналами, от этого правила можно отойти.

27.6. Выполнение консольных команд: system()

Функция system() позволяет вызывающей программе выполнять произвольные консольные команды. В этом разделе мы опишем принцип ее работы, а в следующем — покажем, как ее можно реализовать с помощью вызовов fork(), exec(), wait() и exit().

> В разделе 44.5 рассматриваются функции popen() и pclose(), которые тоже могут быть использованы для вызова консольных команд, но при этом позволяют вызывающей программе либо считывать вывод команды, либо отправлять ей ввод.

```
#include <stdlib.h>

int system(const char *command);
```

Описание возвращаемого значения ищите в основном тексте

Функция system() создает дочерний процесс для выполнения команды *command*. Ниже показан пример ее вызова:

```
system("ls | wc");
```

Принципиальными преимуществами функции system() являются простота и удобство.

- Нам не нужно иметь дело с деталями вызовов `fork()`, `exec()`, `wait()` и `exit()`.
- Обработка ошибок и сигналов выполняется за нас самой функцией `system()`.
- Поскольку `system()` использует для выполнения *команды* командную оболочку, перед ее запуском выполняются все стандартные процедуры обработки, подстановки и перенаправления. Это упрощает добавление в приложение возможности вида «выполнить консольную команду» (подобная функция доступна во многих интерактивных программах по команде !).

Главным недостатком функции `system()` является ее низкая производительность. Запуск команды с ее помощью требует создания как минимум двух процессов — одного для командной оболочки, и еще одного — для выполняемой команды; каждый такой процесс требует вызова `exec()`. Если эффективность или скорость являются важным требованием, для запуска нужной программы лучше воспользоваться непосредственно вызовами `fork()` и `exec()`.

Значение, возвращаемое функцией `system()`, зависит от следующих факторов.

- Если в аргументе `command` передали нулевой указатель, `system()` возвращает ненулевое значение (если доступна командная оболочка) или `0` (если доступа к оболочке нет). Этот случай является следствием того факта, что стандарты языка C не привязаны ни к какой конкретной операционной системе, поэтому `system()` может запускаться в среде, несовместимой с UNIX, в которой отсутствует командная строка. Более того, командная оболочка может оказаться недоступной даже в UNIX-системе, если перед выполнением `system()` программа вызвала `chroot()`. Если в аргументе `command` не нулевой указатель, вывод функции `system()` обусловливается оставшимися пунктами данного списка.
- Если дочерний процесс не может быть создан или его код завершения нельзя получить, `system()` возвращает `-1`.
- Если командная оболочка не может быть запущена в дочернем процессе, `system()` возвращает значение, которое мы бы получили при завершении дочерней командной оболочки, вызывав `_exit(127)`.
- Если все системные вызовы выполнились успешно, `system()` возвращает код завершения дочерней командной оболочки, с помощью которого запускалась внешняя команда. (Код завершения командной оболочки совпадает с кодом заверешния последней выполненной команды. Если эта команда была завершена с помощью сигнала, то большинство оболочек завершаются с кодом, равным `128+`*n*, где *n* — номер сигнала. Код завершения потомка, завершенного с помощью сигнала, рассматривается в разделе 26.1.3).

  Имея лишь значение, возвращенное system(), невозможно точно сказать, когда ей не удается запустить командную оболочку, а когда командная оболочка просто завершается со статусом 127 (последний вариант возможен, если оболочка не может найти и запустить программу с заданным именем).

В последних двух случаях значение, возвращаемое функцией `system()`, является статусом ожидания того же формата, который возвращается вызовом `waitpid()`. Это означает, что для анализа данного значения следует использовать функции, описанные в подразделе 26.1.3; кроме того, мы можем вывести его с помощью нашей функции `printWaitStatus()` (см. листинг 26.2).

Пример программы

Использование функции `system()` демонстрируется в листинге 27.7. Эта программа запускает цикл, который считывает строку с командой, выполняет ее с помощью `system()`, и затем анализирует и выводит полученный результат. Пример ее выполнения показан ниже:

```
$ ./t_system
Command: whoami
mtk
system() returned: status=0x0000 (0,0)
child exited, status=0
Command: ls | grep XYZ                  Код завершения оболочки совпадает с кодом…
system() returned: status=0x0100 (1,0)  …выхода ее последней команды (grep), которая…
child exited, status=1                  …не нашла совпадений, что привело к вызову exit(1)
Command: exit 127
system() returned: status=0x7f00 (127,0)
(Probably) could not invoke shell       Но в данном случае это не так
Command: sleep 100
Нажимаем Ctrl-Z, чтобы приостановить активную группу процессов
[1]+ Stopped ./t_system
$ ps | grep sleep                       Находим PID команды sleep
29361 pts/6 00:00:00 sleep
$ kill 29361                            И отправляем сигнал, чтобы ее завершить
$ fg                                    Опять делаем программу t_system активной
./t_system
system() returned: status=0x000f (0,15)
child killed by signal 15 (Terminated)
Command: ^D$                            Нажимаем Ctrl+D для завершения программы
```

Листинг 27.7. Выполнение консольной команды с помощью функции system()

procexec/t_system.c

```
#include <sys/wait.h>
#include "print_wait_status.h"
#include "tlpi_hdr.h"
#define MAX_CMD_LEN 200

int
main(int argc, char *argv[])
{
    char str[MAX_CMD_LEN];  /* Команда, которую нужно выполнить посредством system() */
    int status;       /* Статус, возвращаемый из system() */

    for (;;) {        /* Считываем и выполняем консольную команду */
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            break;    /* Конец файла */
        status = system(str);
        printf("system() returned: status=0x%04x (%d,%d)\n",
            (unsigned int) status, status >> 8, status & 0xff);

        if (status == -1) {
            errExit("system");
        } else {
            if (WIFEXITED(status) && WEXITSTATUS(status) == 127)
                printf("(Probably) could not invoke shell\n");
            else    /* Оболочка успешно выполнила команду */
                printWaitStatus(NULL, status);
        }
    }

    exit(EXIT_SUCCESS);
}
```

procexec/t_system.c

Избегайте функции system() в программах с установленным ID пользователя или группы

Программы с установленным ID пользователя или группы никогда не должны применять функцию `system()` во время работы в режиме повышенных привилегий. Даже если пользователь не может вводить текст выполняемой команды, командная оболочка все равно полагается на различные переменные среды, что в сочетании с вызовом `system()` открывает потенциальную брешь в безопасности системы.

Например, в старых версиях оболочки Bourne shell переменная среды `IFS`, которая устанавливает внутренний разделитель между отдельными словами командной строки, была источником целого ряда успешных взломов. Если присвоить `IFS` значение `a`, строка `shar` будет восприниматься командной оболочкой как слово `sh`, за которым следует аргумент `r`; Это, в свою очередь, приведет к вызову еще одной оболочки, которая вместо ожидаемого действия (запуска команды под названием `shar`) выполнит скрипт с именем `r` в текущем каталоге. Для исправления этой конкретной уязвимости переменную `IFS` стали применять только к словам, получаемым в результате расширения командной строки. Кроме того, современные командные оболочки сбрасывают переменную `IFS` при каждом запуске (присваивая ей строку с тремя символами: пробелом, табуляцией и переходом на новую строку), гарантируя тем самым предсказуемое поведение скриптов, которые наследуют необычное значение `IFS`. В качестве дополнительной меры безопасности оболочка bash при вызове из программы с установленным ID пользователя или группы сбрасывает соответствующие идентификаторы к их реальным значениям.

Безопасные приложения, которым необходимо запускать другие программы, должны использовать для этого непосредственно вызов `fork()` и одну из функций семейства `exec()`: `execlp()` или `execvp()`.

27.7. Реализация функции system()

В этом разделе рассказывается, как реализовать функцию `system()`. Для начала рассмотрим упрощенный вариант, разберем, каких элементов ему не хватает, и затем перейдем к полноценной реализации.

Упрощенная реализация функции system()

Ключ `-c` команды `sh` позволяет выполнить произвольные консольные команды, записанные в строку:

```
$ sh -c "ls | wc"
     38        38         444
```

Таким образом, для реализации функции `system()` нам потребуется вызов `fork()`. Он будет создавать потомка, который вызывает `execl()` с описанными выше аргументами команды `sh`:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

Для получения статуса потомка, созданного с помощью `system()`, воспользуемся вызовом `waitpid()` с заданным идентификатором дочернего процесса (вызова `wait()` было бы недостаточно, поскольку он ожидает завершения любого потомка и может случайно получить статус не того дочернего процесса). Простая и неполная реализация функции `system()` показана в листинге 27.8.

Листинг 27.8. Реализация функции system(), не предусматривающая обработки сигналов

procexec/simple_system.c

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int
system(char *command)
{
    int status;
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:    /* Ошибка */
        return -1;

    case 0:    /* Потомок */
        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
        _exit(127);        /* Неудачный запуск */

    default:    /* Родитель */
        if (waitpid(childPid, &status, 0) == -1)
            return -1;
        else
            return status;
    }
}
```

procexec/simple_system.c

Корректная обработка сигналов внутри system()

Корректная работа с сигналами усложняет реализацию функции `system()`.

Прежде всего следует учитывать сигнал `SIGCHLD`. Допустим, программа, вызывающая `system()` и, помимо прочего, создающая напрямую дочерние процессы, устанавливает обработчик для `SIGCHLD`, который сам делает вызов `wait()`. В этой ситуации, если `SIGCHLD` сгенерирован в результате завершения потомка, созданного с помощью `system()`, может быть вызван обработчик из главной программы, который получит статус потомка до того, как `system()` сможет вызвать `waitpid()` (это пример состояния гонки). Из этого следует два нежелательных обстоятельства.

- Вызывающая программа будет ошибочно полагать, что один из ее потомков был завершен.
- Функция `system()` не сможет получить код завершения созданного ею потомка.

Таким образом, функция `system()` во время своей работы должна блокировать доставку сигнала `SIGCHLD`.

Также следует обратить внимание на сигналы, генерируемые терминалом в результате набора специальных симовольных последовательностей прерывания (обычно Ctrl+C) и выхода (обычно Ctrl+\), — соответственно `SIGINT` и `SIGQUIT`. Посмотрим, что произойдет, если выполнить следующий вызов:

```
system("sleep 20");
```

На данном этапе выполняется три процесса: процесс главной программы, командная оболочка и команда `sleep` (рис. 27.2).

Если вместе с параметром -c передается простая команда (без конвейера или последовательности), некоторые командные оболочки (включая bash) для повышения эффективности вызывают команду exec напрямую, не создавая дочернего процесса. Рисунок 27.2 не совсем точно описывает оболочки, выполняющие такую оптимизацию, поскольку в их случае процессов будет два — вызывающий и команда sleep. Однако это не отменяет замечаний о том, как функция system() должна заниматься обработкой сигналов.

Все процессы, показанные на рис. 27.2, являются частью активной группы процессов в терминале (группы процессов будут подробно рассмотрены в разделе 34.2). Следовательно, если набрать в терминале последовательсть *прерывания* или *выхода*, то все процессы получат соответствующий сигнал. Сигналы `SIGINT` и `SIGQUIT` игнорируются командной оболочкой, которая ждет своих потомков, но по умолчанию приводят к завершению вызывающей программы и процесса `sleep`.

Каким образом вызывающий процесс и выполняемая команда должны реагировать на эти сигналы? Стандарт SUSv3 гласит следующее.

- Пока выполняется команда, сигналы `SIGINT` и `SIGQUIT` должны игнорироваться вызывающим процессом.
- Потомок должен обрабатывать сигналы `SIGINT` и `SIGQUIT` так, как это делает вызывающий процесс при использовании вызовов `fork()` и `exec()`; то есть диспозиция обрабатываемых сигналов сбрасывается к значениям по умолчанию, а диспозиции других сигналов остаются без изменений.

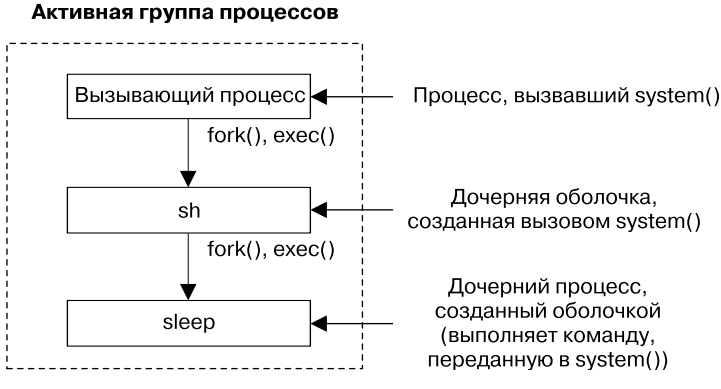


Рис. 27.2. Организация процессов во время выполнения system("sleep 20")

Работа с сигналами в соответствии со стандартом SUSv3 является наиболее разумным подходом, и вот почему.

- Не следует отвечать на эти сигналы сразу в двух процессах, поскольку, с точки зрения пользователя, это может привести к неожиданному поведению приложения.
- Аналогично эти сигналы нельзя игнорировать в процессе, выполняющем команду, и при этом оставлять для них диспозицию по умолчанию в вызывающем процессе. Это позволило бы пользователю, например, завершить вызывающий процесс, но оставить работать запущенную им команду. Это также идет вразрез с тем фактом, что во время выполнения команды, переданной в функцию `system()`, вызывающий процесс на самом деле теряет управление (то есть блокируется в вызове `waitpid()`).
- Команда, выполняемая функцией `system()`, может оказаться интерактивным приложением, и в этом случае имеет смысл позволить ей реагировать на сигналы, генерируемые терминалом.

Стандарт SUSv3 требует, чтобы сигналы SIGINT и SIGQUIT обрабатывались вышеописанным образом, но отмечает, что это может привести к нежелательным последствиям в программах, которые применяют функцию system() в фоновом режиме для выполнения некоторых задач. Нажатие Ctrl+C или Ctrl+\ во время выполнения такой команды приведет к завершению только потомка system(), тогда как само приложение продолжит работу (неожиданно для пользователя). Программа, которая использует функцию system() таким образом, должна проверять код завершения, которая та возвращает, и предпринимать соответствующие меры в случае завершения команды по сигналу.

Улучшенная реализация функции system()

В листинге 27.9 приводится реализация функции system(), отвечающая вышеописанным правилам. Обратите внимание на следующие замечания относительно этой реализации.

- Как было отмечено ранее, если в аргументе command передан нулевой указатель, функция system() должна вернуть ненулевое значение или 0 в зависимости от того, доступна командная оболочка или нет. Единственный надежный способ проверить эту информацию — это попытаться запустить командную оболочку. Для этого следует сделать рекурсивный вызов system() для выполнения встроенной команды :, которая ничего не делает, но всегда возвращает код успешного завершения; после этого нужно проверить, равняется ли полученный результат 0 (1). Того же результата можно было бы достичь путем запуска консольной команды exit 0. Заметьте, что для этого было бы недостаточно воспользоваться вызовом access(), чтобы проверить, существует ли файл /bin/sh и является ли он исполняемым. В среде после вызова chroot(), даже если исполняемый файл оболочки окажется на месте и он динамически скомпонован, его не получится выполнить, пока не удастся найти необходимые совместно используемые библиотеки.
- В родительском процессе (вызывающем system()) нужно блокировать только сигнал SIGCHLD (2), а сигналы SIGINT и SIGQUIT следует игнорировать (3). Однако эти действия необходимо выполнить до вызова fork(), в противном случае возникнет состояние гонки (представьте, к примеру, что потомок завершился до того, как родитель смог заблокировать SIGCHLD). Следовательно, потомок должен отменить изменения атрибутов сигнала, о чем мы поговорим чуть ниже.
- Родитель игнорирует ошибки выполнения вызовов sigaction() и sigprocmask(), с помощью которых изменяются диспозиция и маска сигналов процесса (2), (3), (9). Это делается по двум причинам. Во-первых, эти вызовы крайне редко завершаются неудачей. Но единственная проблема, которая действительно может с ними возникнуть, — некорректное задание аргументов, от которого нужно избавляться еще на стадии начальной отладки. Во-вторых, мы предполагаем, что вызывающий процесс больше заинтересован в получении сведений об ошибках, связанных с вызовами fork() или waitpid(). По схожим причинам вызовы для работы с сигналами в конце system() помещаются между строчками кода, сохраняющими (8) и восстанавливающими значение errno (10); если вызовы fork() или waitpid() завершатся неудачей, вызывающий процесс сможет определить источник ошибки. Если в результате неудачных вызовов для работы с сигналами вернуть -1, вызывающий процесс может ошибочно предположить, что функции system() не удалось выполнить команду.

  В стандарте SUSv3 просто говорится, что функция system() должна возвращать –1, если дочерний процесс не удалось создать, иначе его статус не может быть получен. Нет никаких упоминаний о том, что –1 может возвращаться из system() из-за ошибок при операциях с сигналами.

- Для системных вызовов, связанных с сигналами и выполняемых внутри потомка ❹, ❺, проверка на ошибки не проводится. С другой стороны, о таких ошибках просто невозможно сообщить (вызов `_exit(127)` зарезервирован для ошибок, происходящих во время работы командной оболочки); однако стоит отметить, что такие ошибки не затрагивают вызывающий `system()` процесс, так как он является отдельным.
- При возврате из `fork()` в потомке диспозицией сигналов `SIGINT` и `SIGQUIT` является `SIG_IGN` (наследуется от родителя). Но, как было отмечено ранее, потомок должен обращаться с этими сигналами так, как будто функция `system()` сделала вызовы `fork()` и `exec()`. Первый не влияет на работу с сигналами в потомке, а второй сбрасывает диспозиции необрабатываемых сигналов до значений по умолчанию, а все остальные сигналы оставляет без изменений (см. раздел 27.5). Следовательно, если диспозиции сигналов `SIGINT` и `SIGQUIT` в вызывающем процессе отличаются от `SIG_IGN`, потомок сбрасывает их к `SIG_DFL` ❹.

  Некоторые реализации функции system() вместо сбрасывания диспозиций сигналов SIGINT и SIGQUIT к тем, что имели место в вызывающем процессе, полагаются на тот факт, что последующий вызов execl() автоматически сбросит диспозиции обрабатываемых сигналов к значениям по умолчанию. Однако это может привести к нежелательным последствиям, если любой из этих сигналов обрабатывается в вызывающем процессе. В таком случае, если сигнал был доставлен потомку перед вызовом execl(), обработчик будет вызван внутри потомка, но после того, как вызов sigprocmask() разблокирует сигнал.

- Если вызов `execl()` в потомке заканчивается неудачей, мы завершаем процесс с помощью `_exit()` ❻ вместо `exit()`, чтобы предотвратить сброс на диск любых незаписанных данных из дочерней копии буферов stdio.
- Родитель должен использовать вызов `waitpid()`, чтобы отслеживать именно того потомка, которого мы создали ❼. Если бы мы прибегли к вызову `wait()`, родитель мог бы случайно извлечь статус какого-нибудь другого созданного им дочернего процесса.
- Реализация `system()` не требует использования обработчиков сигналов, однако такие обработчики могут быть установлены вызывающей программой, из-за чего может прерваться работа заблокированного вызова `waitpid()`. В стандарте SUSv3 явно обозначено, что операция ожидания в этом случае должна быть запущена заново. Поэтому `waitpid()` вызывается в цикле, который заканчивает свою работу при любой ошибке, кроме `EINTR` ❼, и завершается при любой другой.

Листинг 27.9. Реализация функции system()

procexec/system.c

```
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <errno.h>

int
system(const char *command)
{
    sigset_t blockMask, origMask;
    struct sigaction saIgnore, saOrigQuit, saOrigInt, saDefault;
    pid_t childPid;
    int status, savedErrno;
❶   if (command == NULL)        /* Доступна ли командная оболочка? */
        return system(":") == 0;
```

```
    sigemptyset(&blockMask);      /* Блокируем SIGCHLD */
    sigaddset(&blockMask, SIGCHLD);
❷   sigprocmask(SIG_BLOCK, &blockMask, &origMask);

    saIgnore.sa_handler = SIG_IGN;    /* Игнорируем SIGINT и SIGQUIT */
    saIgnore.sa_flags = 0;
    sigemptyset(&saIgnore.sa_mask);
❸   sigaction(SIGINT, &saIgnore, &saOrigInt);
    sigaction(SIGQUIT, &saIgnore, &saOrigQuit);

    switch (childPid = fork()) {
    case -1:      /* Вызов fork() завершился неудачей */
        status = -1;
        break;    /* Идем дальше, чтобы сбросить атрибуты сигнала */

    case 0:       /* Потомок запускает команду */
        saDefault.sa_handler = SIG_DFL;
        saDefault.sa_flags = 0;
        sigemptyset(&saDefault.sa_mask);

❹       if (saOrigInt.sa_handler != SIG_IGN)
            sigaction(SIGINT, &saDefault, NULL);
        if (saOrigQuit.sa_handler != SIG_IGN)
            sigaction(SIGQUIT, &saDefault, NULL);

❺       sigprocmask(SIG_SETMASK, &origMask, NULL);
        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
❻       _exit(127);    /* Нам не удалось запустить командную оболочку */

    default:          /* Родитель ждет завершения нашего потомка */
❼       while (waitpid(childPid, &status, 0) == -1) {
            if (errno != EINTR) {    /* Ошибка, отличная от EINTR */
                status = -1;
                break;    /* Выходим из цикла */
            }
        }
        break;
    }

/* Разблокируем SIGCHLD, восстанавливаем диспозиции для SIGINT и SIGQUIT */

❽   savedErrno = errno;  /* Следующий код может изменить значение 'errno' */

❾   sigprocmask(SIG_SETMASK, &origMask, NULL);
    sigaction(SIGINT, &saOrigInt, NULL);
    sigaction(SIGQUIT, &saOrigQuit, NULL);

❿   errno = savedErrno;

    return status;
}
```

procexec/system.c

Дополнительные подробности о функции system()

Переносимые приложения должны следить за тем, чтобы функция `system()` не вызывалась, когда диспозицией сигнала `SIGCHLD` является `SIG_IGN`, поскольку вызов `waitpid()` в этом случае не может получить статус потомка (если игнорировать сигнал `SIGCHLD`, статус потомка сразу же сбрасывается, как было описано в подразделе 26.3.3).

В некоторых реализациях UNIX `system()`, обнаружив установленную диспозицию `SIG_IGN` сигнала `SIGCHLD`, временно меняет ее на `SIG_DFL`. Такой подход работает в системах, которые (в отличие от Linux) снимают дочерние процессы-«зомби», если диспозицией `SIGCHLD` является `SIG_IGN` (в остальных системах такая реализация `system()` имела бы отрицательные последствия: любой другой потомок вызывающего процесса, завершающийся во время ее работы, превращался бы в «зомби», которого нельзя уничтожить).

В некоторых реализациях UNIX (в частности, в Solaris) `/bin/sh` не является стандартной командной оболочкой POSIX. Если мы хотим вызывать именно стандартную оболочку, нам следует использовать библиотечную функцию `confstr()`, которая позволяет получить значение конфигурационной переменной `_CS_PATH`. Это значение представляет собой список каталогов (того же формата, что и в `PATH`) со стандартными системными утилитами. Мы можем присвоить этот список переменной `PATH` и затем вызвать стандартную команду оболочку с помощью `execlp()`:

```
char path[PATH_MAX];

if (confstr(_CS_PATH, path, PATH_MAX) == 0)
    _exit(127);
if (setenv("PATH", path, 1) == -1)
    _exit(127);
execlp("sh", "sh", "-c", command, (char *) NULL);
_exit(127);
```

27.8. Резюме

С помощью вызова `execve()` процесс может заменить программу, которая в нем выполняется, на новую. Этот вызов принимает списки аргументов (`argv`) и переменных среды для новой программы. Вокруг него существуют различные библиотечные функции-обертки, которые предоставляют разные интерфейсы к одним и тем же возможностям.

Все функции семейства `exec()` позволяют загружать бинарные исполняемые файлы или выполнять скрипты интерпретатора. В последнем случае его интерпретатор заменяет собой текущую программу процесса. Интерпретатор, а точнее, путь к нему обычно определяется первой строчкой скрипта (начинающейся с символов `#!`). Если такой строчки нет, скрипт может быть выполнен только с помощью функций `execlp()` или `execvp()`, которые используют в качестве интерпретатора командную оболочку.

Мы продемонстрировали, как с помощью сочетания вызовов `fork()`, `exec()`, `exit()` и `wait()` можно реализовать функцию `system()`, которая способна запускать произвольные консольные команды.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 24.6

27.9. Упражнения

27.1. Последняя команда в следующей сессии командной строки использует программу из листинга 27.3 для запуска `xyz`. К чему это приведет?

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:./dir1:./dir2
$ ls -l dir1
total 8
```

```
-rw-r--r-- 1 mtk    users    7860 Jun 13 11:55 xyz
$ ls -l dir2
total 28
-rwxr-xr-x 1 mtk    users    27452 Jun 13 11:55 xyz
$ ./t_execlp xyz
```

27.2. Используйте вызов `execve()`, чтобы реализовать функцию `execlp()`. Вам нужно применить программный интерфейс `stdarg(3)` для обработки списка аргументов переменной длины, передаваемого в `execlp()`. Вам также понадобятся функции из пакета `malloc`, чтобы выделить память для списка аргументов и списка с переменными среды. Также стоит отметить, что для проверки существования файла в определенном каталоге и определения того, является ли он исполняемым, вы можете просто попытаться его запустить.

27.3. Какой бы вывод мы получили, если бы следующий скрипт был исполняемым и запущенным с помощью вызова `exec()`?

```
#!/bin/cat -n
Hello world
```

27.4. Какой результат будет при выполнении следующего кода? В каких обстоятельствах он может пригодиться?

```
childPid = fork();
if (childPid == -1)
    errExit("fork1");
if (childPid == 0) { /* Потомок */
    switch (fork()) {
    case -1: errExit("fork2");

    case 0: /* Потомок потомка */
        /* ----- Здесь выполняется реальная работа ----- */
        exit(EXIT_SUCCESS);  /* После выполнения реальной работы */

    default:
        exit(EXIT_SUCCESS);  /* Делаем дочерний процесс потомка «сиротой» */
    }
}
/* Родитель перескакивает сюда */
if (waitpid(childPid, &status, 0) == -1)
    errExit("waitpid");
/* Родитель приступает к выполнению других задач */
```

27.5. Запустив эту программу, мы обнаружим, что она не генерирует никакого вывода. Почему?

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world");
    execlp("sleep", "sleep", "0", (char *) NULL);
}
```

27.6. Представьте, что родительский процесс установил обработчик для `SIGCHLD` и заблокировал этот сигнал. После того как один из его потомков завершается, родитель выполняет вызов `wait()`, чтобы получить статус потомка. Что произойдет, если родитель разблокирует `SIGCHLD`? Напишите программу, чтобы проверить свой ответ. Имеет ли это какое-либо отношение к программе, вызывающей функцию `system()`?

28 Подробнее о создании процесса и выполнении программы

Эта глава дополняет материал, изложенный в главах 24–27, охватывая различные темы, касающиеся создания процесса и выполнения программы. Будет описан учет используемых ресурсов — функция, ядра, которая в момент завершения любого процесса в системе записывает его учетную информацию. Мы рассмотрим системный вызов `clone()`, который представляет собой низкоуровневый программный интерфейс для создания потоков выполнения в Linux. За этим последует сравнение производительности `fork()`, `vfork()` и `clone()`. В конце будет рассмотрено влияние вызовов `fork()` и `exec()` на атрибуты процесса.

28.1. Учет ресурсов, используемых процессом

Когда система учета ресурсов включена, ядро записывает в общесистемный файл данные о каждом процессе, который оно завершает. Каждая такая запись содержит различные сведения, собираемые ядром, такие как код завершения процесса и сколько тот затратил процессорного времени. Учетный файл можно анализировать с помощью стандартных инструментов (вызов `sa(8)` предоставляет краткую сводку, а `lastcomm(1)` показывает информацию о ранее выполненных командах) или специализированных приложений.

В силу исторических причин учет ресурсов изначально применяли для взимания платы с пользователей за работу в многопользовательских UNIX-системах. Однако с его помощью также можно получать сведения о процессах, которые не собираются и не могут предоставляться их родителями.

Система учета ресурсов не описывается в стандарте SUSv3, хотя и доступна в большинстве разновидностей UNIX. Формат записей и местоположение файла, в котором они хранятся, могут варьироваться в зависимости от реализации. В этом разделе мы сосредоточимся на деталях, характерных для Linux, не забывая при этом отмечать некоторые различия, встречающиеся в других системах UNIX.

> В Linux система учета ресурсов является опциональным компонентом ядра, который настраивается посредством параметра CONFIG_BSD_PROCESS_ACCT.

Включение и отключение учета ресурсов

Для включения и отключения учета ресурсов привилегированный процесс (`CAP_SYS_PACCT`) должен воспользоваться системным вызовом `acct()`. Этот вызов редко применяется в прикладных программах. Обычно учет ресурсов включается при каждом запуске системы путем размещения подходящих команд в загрузочных скриптах.

```
#define _BSD_SOURCE
#include <unistd.h>

int acct(const char *acctfile);
```

Возвращает 0 при успешном завершении и –1 при ошибке

Чтобы включить учет ресурсов, аргументу `acctfile` нужно передать путь к обычному *существующему* файлу. Типичный путь к учетному файлу имеет вид `/var/log/pacct` или `/usr/account/pacct`. Для отключения учета ресурсов аргументу `acctfile` достаточно передать `NULL`.

Программа из листинга 28.1 использует вызов `acct()` для включения и отключения учета ресурсов. По своим возможностям она похожа на консольную команду `accton(8)`.

Листинг 28.1. Включение и выключение учета ресурсов

procexec/acct_on.c

```
#define _BSD_SOURCE
#include <unistd.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc > 2 || (argc > 1 && strcmp(argv[1], "--help") == 0))
        usageErr("%s [file]\n", argv[0]);
    if (acct(argv[1]) == -1)
        errExit("acct");

    printf("Process accounting %s\n",
       (argv[1] == NULL) ? "disabled" : "enabled");
    exit(EXIT_SUCCESS);
}
```

procexec/acct_on.c

Записи в системе учета ресурсов

После включения учета ресурсов данные, записанные в структуру `acct`, сбрасываются в файл при завершении каждого процесса. Структура `acct` объявлена в заголовочном файле `<sys/acct.h>` и имеет следующий вид:

```
typedef u_int16_t comp_t;  /* См. текст */
struct acct {
    char ac_flag;          /* Флаги учета (см. текст) */
    u_int16_t ac_uid;      /* Пользовательский идентификатор процесса */
    u_int16_t ac_gid;      /* Групповой идентификатор процесса */
    u_int16_t ac_tty;      /* Управляющий терминал процесса (может
                              равняться 0, если процесс является демоном) */
    u_int32_t ac_btime;    /* Начальное время (time_t; секунды с начала эры UNIX) */
    comp_t ac_utime;       /* Пользовательское процессорное время
                              (такты системного времени) */
    comp_t ac_stime;       /* Системное процессорное время
                              (такты системного времени) */
    comp_t ac_etime;       /* Прошедшее (реальное) время (такты системного времени) */
    comp_t ac_mem;         /* Среднее потребление памяти (килобайты) */
    comp_t ac_io;          /* Байты, переданные вызовами read(2) и write(2)
                              (не используется) */
```

```
    comp_t ac_rw;          /* Прочитанные/записанные блоки (не используется) */
    comp_t ac_minflt;      /* Незначительные отказы страницы (есть лишь в Linux) */
    comp_t ac_majflt;      /* Значительные отказы страницы
                              (существует только в Linux) */
    comp_t ac_swaps;       /* Количество сбросов в файл подкачки
                              (не используется; существует лишь в Linux) */
    u_int32_t ac_exitcode; /* Код завершения процесса */
#define ACCT_COMM 16
    char ac_comm[ACCT_COMM+1];
    /* Имя команды с NULL в конце (имя последнего запущенного файла) */
    char ac_pad[10];      /* Смещение (зарезервировано для будущего использования) */
};
```

Обратите внимание на следующие особенности данной структуры.

- Типы данных `u_int16_t` и `u_int32_t` являются 16- и 32-битными беззнаковыми целыми числами.
- Поле `ac_flag` — это битовая маска, которая хранит различные события процесса. Биты, которые могут попасть в это поле, приведены в табл. 28.1. Как видите, некоторые из них представлены не во всех реализациях UNIX. Кроме того, некоторые системы предоставляют дополнительные биты для этого поля.
- Поле `ac_comm` хранит имя последней команды (программного файла), выполненной данным процессом. Ядро записывает это значение при каждом вызове `execve()`. В некоторых других реализациях UNIX данное поле ограничено восьмью символами.
- Тип `comp_t` представляет собой число с плавающей запятой. Его значения иногда называют *сжатыми тактами системного времени*. Это вещественное значение состоит из 3-битной экспоненты с основанием 8- и 13-битной мантиссы; экспонента может представлять фактор в диапазоне от $8^0 = 1$ до 8^7 (2 097 152). Например, мантисса 125 и экспонента от 1 представляет значение 1000. В листинге 28.2 показана функция `comptoLL()`, предназначенная для преобразования этого типа в `long long`. Мы должны использовать тип `long long`, поскольку 32 бит, выделяемых для `unsigned long` на платформе x86-32, недостаточно для хранения наибольшего значения, которое может оказаться в поле `comp_t` ($(2^{13} - 1) \times 8^7$).
- Три временных поля типа `comp_t` представляют время в тактах системных часов. Чтобы конвертировать эти значения в секунды, их следует разделить на число, возвращаемое вызовом `sysconf(_SC_CLK_TCK)`.
- Поле `ac_exitcode` хранит код завершения процесса (описанный в подразделе 26.1.3). В большинстве других реализаций вместо него предоставляется однобайтное поле с именем `ac_stat`, которое вмещает в себя только сигнал, который завершил процесс (если это был сигнал), и бит, указывающий на то, привел ли этот сигнал к сбрасыванию на диск дампа памяти. Системы, основанные на BSD, не предоставляют ни одно из этих полей.

Таблица 28.1. Битовые значения для поля ac_flag в записях системы учета ресурсов

Бит	Описание
AFORK	Процесс был создан с помощью fork(), но перед завершением не выполнил exec()
ASU	Процесс воспользовался привилегиями администратора
AXSIG	Процесс был завершен по сигналу (отсутствует в некоторых реализациях)
ACORE	Процесс сгенерировал дамп памяти (отсутствует в некоторых реализациях)

Поскольку записи учета ресурсов размещаются в порядке завершения процессов (значение, которое отсутствует в записи), а не их запуска (`ac_btime`), поскольку именно в этот момент они создаются.

Если в системе произошел сбой, данные учета ресурсов для еще выполняющихся процессов не записываются.

Хранение записей в учетном файле может привести к быстрому исчерпыванию дискового пространства, поэтому для управления операциями учета ресурсов в Linux предоставляется виртуальный файл `/proc/sys/kernel/acct`. Он содержит три числа, описывающих *верхнее ограничение*, *нижнее ограничение* и *частоту*. Обычно значениями по умолчанию являются `4`, `2` и `30`. Если учет ресурсов включен и процент свободного дискового пространства опускается за пределы *нижнего ограничения*, учет приостанавливается. Если позже процент свободного пространства начинает превышать *верхнее ограничение*, учет возобновляется. *Частота* указывает на длину интервала (в секундах) между проверками процента свободного пространства на диске.

Пример программы

Программа, представленная в листинге 28.2, выводит отдельные поля записей учетного файла. Ниже приводится сессия командной строки, которая демонстрирует использование этой программы. Начнем с создания нового пустого учетного файла и включения учета ресурсов:

```
$ su                Для включения учета ресурсов нужны повышенные привилегии
Password:
# touch pacct
# ./acct_on pacct   Запись об этом процессе первой попадет в учетный файл
Process accounting enabled
# exit              Отказываемся от администраторских привилегий
```

С момента включения системы учета ресурсов было завешено три процесса. С их помощью были выполнены программы `acct_on`, `su` и `bash`. Процесс `bash` был запущен из `su` для создания привилегированной сессии командной строки.

Добавим новые записи в учетный файл, запустив еще несколько команд:

```
$ sleep 15 &
[1] 18063
$ ulimit -c unlimited    Разрешаем создание дампов памяти (команда встроена в оболочку)
$ cat                    Создаем процесс
Нажимаем Ctrl+\ (генерирует SIGQUIT, сигнал 3), чтобы завершить процесс cat
Quit (core dumped)
$
Нажимаем Enter, чтобы увидеть уведомление командной оболочки о завершении программы
sleep перед появлением приглашения командной строки
[1]+ Done sleep 15
$ grep xxx badfile       grep завершается ошибкой со статусом 2
grep: badfile: No such file or directory
$ echo $?                Командная оболочка получила статус grep
                         (команда встроена в оболочку)
2
```

Следующие две команды запускают программы, представленные в предыдущих главах (листинги 27.1 и 24.1). Первая из них выполняет файл `/bin/echo`, в результате чего в учетный файл попадает запись о процессе с именем `echo`. Вторая создает дочерний процесс, который не выполняет вызов `exec()`.

```
$ ./t_execve /bin/echo
hello world goodbye
$ ./t_fork
PID=18350 (child) idata=333 istack=666
PID=18349 (parent) idata=111 istack=222
```

Наконец, выведем содержимое учетного файла, воспользовавшись программой из листинга 28.2:

```
$ ./acct_view pacct
Command    flags term.   user  start       time      CPU   elapsed
                status                                time    time
acct_on    -S--       0  root  2010-07-23  17:19:05  0.00    0.00
bash       ----       0  root  2010-07-23  17:18:55  0.02   21.10
su         -S--       0  root  2010-07-23  17:18:51  0.01   24.94
cat        --XC    0x83  mtk   2010-07-23  17:19:55  0.00    1.72
sleep      ----       0  mtk   2010-07-23  17:19:42  0.00   15.01
grep       ----  0x200   mtk   2010-07-23  17:20:12  0.00    0.00
echo       ----       0  mtk   2010-07-23  17:21:15  0.01    0.01
t_fork     F---       0  mtk   2010-07-23  17:21:36  0.00    0.00
t_fork     ----       0  mtk   2010-07-23  17:21:36  0.00    3.01
```

В вышеприведенном выводе для каждого процесса, созданного в сессии командной оболочки, выделяется отдельная строка. Команды `ulimit` и `echo` встроены в оболочку, поэтому они не приводят к созданию новых процессов. Обратите внимание на то, что команда `sleep` идет после записи с командой `cat`, потому что именно в таком порядке они были завершены.

Большая часть этого вывода не нуждается в дополнительных объяснениях. Столбец `flags` хранит одиночные буквы, указывая на то, какие биты `ac_flag` установлены для каждой записи (см. табл. 28.1). О том, как интерпретировать значения кода завершения, хранящиеся в столбце `term. status`, шла речь в подразделе 26.1.3.

Листинг 28.2. Вывод данных из учетного файла

procexec/acct_view.c

```
#include <fcntl.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/acct.h>
#include <limits.h>
#include "ugid_functions.h"    /* Объявление функции userNameFromId() */
#include "tlpi_hdr.h"

#define TIME_BUF_SIZE 100

static long long     /* Приводим значение comp_t к типу long long */
comptToLL(comp_t ct)
{
    const int EXP_SIZE = 3;       /* Трехбитная экспонента с основанием 8 */
    const int MANTISSA_SIZE = 13; /* За ней следует 13-битная мантисса */
    const int MANTISSA_MASK = (1 << MANTISSA_SIZE) - 1;
    long long mantissa, exp;

    mantissa = ct & MANTISSA_MASK;
    exp = (ct >> MANTISSA_SIZE) & ((1 << EXP_SIZE) - 1);
    return mantissa << (exp * 3);   /* Степень 8 означает сдвиг влево на три бита */
}

int
main(int argc, char *argv[])
{
    int acctFile;
    struct acct ac;
    ssize_t numRead;
```

```
    char *s;
    char timeBuf[TIME_BUF_SIZE];
    struct tm *loc;
    time_t t;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    acctFile = open(argv[1], O_RDONLY);
    if (acctFile == -1)
        errExit("open");

    printf("command    flags    term.    user     "
        "start time            CPU    elapsed\n");
    printf("             status             "
        "                time    time\n");

    while ((numRead = read(acctFile, &ac, sizeof(struct acct))) > 0) {
        if (numRead != sizeof(struct acct))
            fatal("partial read");

        printf("%-8.8s    ", ac.ac_comm);

        printf("%c", (ac.ac_flag & AFORK) ? 'F' : '-') ;
        printf("%c", (ac.ac_flag & ASU)   ? 'S' : '-') ;
        printf("%c", (ac.ac_flag & AXSIG) ? 'X' : '-') ;
        printf("%c", (ac.ac_flag & ACORE) ? 'C' : '-') ;

#ifdef __linux__
        printf(" %#6lx   ", (unsigned long) ac.ac_exitcode);
#else
        /* Во многих других реализациях вместо этого предоставляется поле ac_stat */
        printf(" %#6lx   ", (unsigned long) ac.ac_stat);
#endif

        s = userNameFromId(ac.ac_uid);
        printf("%-8.8s ", (s == NULL) ? "???" : s);

        t = ac.ac_btime;
        loc = localtime(&t);
        if (loc == NULL) {
            printf("???Unknown time???  ");
        } else {
            strftime(timeBuf, TIME_BUF_SIZE, "%Y-%m-%d %T ", loc);
            printf("%s ", timeBuf);
        }

        printf("%5.2f %7.2f ", (double) (comptToLL(ac.ac_utime) +
           comptToLL(ac.ac_stime)) / sysconf(_SC_CLK_TCK), (double)
           comptToLL(ac.ac_etime) / sysconf(_SC_CLK_TCK));
        printf("\n");
    }

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);
}
```

procexec/acct_view.c

Формат учетного файла версии 3

Начиная с версии 2.6.8, ядро Linux поддерживает альтернативную версию учетного файла (известную как *версия 3*), лишенную ограничений традиционной версии. Чтобы воспользоваться ею, перед сборкой ядра следует включить параметр `CONFIG_BSD_PROCESS_ACCT_V3`.

Единственным отличием версии 3 является формат записей, хранящихся в учетном файле. Его можно описать следующим образом:

```
struct acct_v3 {
    char      ac_flag;      /* Флаги учета */
    char      ac_version;   /* Версия учетного файла (3) */
    u_int16_t ac_tty;       /* Управляющий терминал процесса */
    u_int32_t ac_exitcode;  /* Код завершения процесса */
    u_int32_t ac_uid;       /* 32-битный пользовательский идентификатор процесса */
    u_int32_t ac_gid;       /* 32-битный групповой идентификатор процесса */
    u_int32_t ac_pid;       /* Идентификатор процесса */
    u_int32_t ac_ppid;      /* Идентификатор родительского процесса */
    u_int32_t ac_btime;     /* Начальное время (time_t) */
    float     ac_etime;     /* Прошедшее (реальное) время (такты системного времени) */
    comp_t    ac_utime;     /* Пользовательское процессорное время
                               (такты системного времени) */
    comp_t    ac_stime;     /* Системное процессорное время
                               (такты системного времени) */
    comp_t    ac_mem;       /* Среднее потребление памяти (килобайты) */
    comp_t    ac_io;        /* Количество прочитанных/записанных байтов
                               (не используется) */
    comp_t    ac_rw;        /* Количество прочитанных/записанных блоков
                               (не используется) */
    comp_t    ac_minflt;    /* Незначительные отказы страницы */
    comp_t    ac_majflt;    /* Значительные отказы страницы */
    comp_t    ac_swaps;     /* Количество сбросов в файл подкачки
                               (не используется; существует лишь в Linux) */
    #define ACCT_COMM 16
    char      ac_comm[ACCT_COMM];    /* Имя команды */
};
```

Ниже перечислены главные отличия новой структуры `acct_v3` от традиционной `acct`.

- Добавлено поле `ac_version`, содержащее номер версии этого формата записей. Для `acct_v3` оно всегда равно `3`.
- Добавлены поля `ac_pid` и `ac_ppid`, содержащие идентификаторы процесса и его родителя.
- Поля `ac_uid` и `ac_gid` расширены с 16 до 32 бит, чтобы вместить 32-битные пользовательские и групповые идентификаторы, появившиеся в Linux 2.4 (удлиненные идентификаторы не могут быть корректно представлены в традиционном файле формата `acct`).
- Тип поля `ac_etime` поменялся с `comp_t` на `float`, чтобы можно было записывать более длинные периоды прошедшего времени.

Аналог программы из листинга 28.2 с поддержкой версии 3 доступен в файле procexec/acct_v3_view.c внутри архива с исходными кодами, прилагающемся к этой книге.

28.2. Системный вызов clone()

В Linux доступен уникальный для этой системы вызов `clone()`, предназначенный для создания новых процессов. От `fork()` и `vfork()` он отличается тем, что позволяет более точно контролировать этапы создания процесса. Его в основном применяют в библиотеках, отвечающих за работу с потоками выполнения. Поскольку вызов `clone()` не является

переносимым, вам следует избегать его непосредственного использования в прикладных программах. Мы упоминаем его здесь, потому что он послужит хорошим фоном для описания потоков выполнения POSIX в главах 29–33, а также потому, что он помогает осветить дополнительные аспекты вызовов `fork()` и `vfork()`.

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags,
          void *func_arg, ...
         /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Возвращает идентификатор потомка при успешном завершении или `-1` в результате ошибки

Как и в случае с `fork()`, новый процесс, создаваемый вызовом `clone()`, является почти полной копией своего родителя.

Отличие от `fork()` состоит в том, что клонированный потомок не продолжает работу с момента вызова, а сначала вызывает функцию, указанную в аргументе `func`; мы будем называть ее *дочерней функцией*. Итак, при вызове дочерней функции ей передается значение, указанное в `func_arg`. Этот аргумент можно свободно интерпретировать как целое число или, например, указатель на структуру, используя подходящее приведение типов (интерпретация аргумента в качестве указателя становится возможным благодаря тому, что клонированный потомок либо получает копию памяти вызывающего процесса, либо разделяет ее с ним).

Клонированный дочерний процесс завершается либо при возврате функции `func` (в этом случае возвращаемое значение становится кодом завершения процесса), либо при вызове `exit()` (или `_exit()`). Родительский процесс может ожидать завершения потомка как обычно — с помощью вызова `wait()` или аналогичного ему.

Поскольку клонированный потомок может разделять память с родителем (как в случае с `vfork()`), он не может использовать родительский стек. Вместо этого он должен выделить для своего стека блок памяти подходящего размера и передать указатель на него в аргумент `child_stack`. В большинстве аппаратных архитектур стек растет сверху вниз, поэтому аргумент `child_stack` должен указывать на верхний конец выделенного блока.

Тот факт, что направление роста стека зависит от архитектуры, является концептуальным недостатком вызова clone(). Архитектура Intel IA-64 предоставляет улучшенный программный интерфейс клонирования в виде вызова clone2(). Он описывает диапазон дочернего стека таким образом, что направление его роста не имеет значения, поскольку передается сразу два адреса — начальный и конечный. Подробности ищите на справочной странице.

Аргумент `flags` вызова `clone()` служит сразу двум целям. Во-первых, его нижний байт содержит *сигнал завершения* потомка, который передается родителю в момент завершения дочернего процесса (если клонированный потомок останавливается по сигналу, родитель все равно получает `SIGCHLD`). Байт может быть равен `0`; в этом случае сигнал не генерируется (сигнал завершения любого процесса можно определить с помощью файла вида `/proc/PID/stat`, доступного в Linux; дальнейшие подробности ищите на справочной странице `proc(5)`).

Вызовы fork() и vfork() не позволяют выбрать сигнал завершения; это всегда будет SIGCHLD.

Оставшиеся байты аргумента `flags` отводятся под битовую маску, которая определяет поведение вызова `clone()`. Ее значения собраны в табл. 28.2 и подробно описаны в подразделе 28.2.1.

Таблица 28.2. Значения битовой маски flags вызова clone()

Флаг	**Действие**
CLONE_CHILD_CLEARTID	Очищает ctid, когда потомок вызывает exec() или _exit() (начиная с 2.6) (здесь и далее в скобках версия Linux)
CLONE_CHILD_SETTID	Записывает идентификатор дочернего потока в ctid (начиная с 2.6)
CLONE_FILES	Родитель и потомок разделяют таблицу дескрипторов открытых файлов
CLONE_FS	Родитель и потомок разделяют атрибуты, относящиеся к файловой системе
CLONE_IO	Потомок получает доступ к контексту ввода/вывода родителя (начиная с 2.6.25)
CLONE_NEWIPC	Потомок получает новое пространство имен в System V IPC (начиная с 2.6.19)
CLONE_NEWNET	Потомок получает сетевое пространство имен (начиная с 2.4.24)
CLONE_NEWNS	Потомок получает копию родительского пространства имен файловой системы (2.4.19)
CLONE_NEWPID	Потомок получает новое пространство имен с идентификаторами процессов (начиная с 2.6.19)
CLONE_NEWUSER	Потомок получает новое пространство имен с идентификаторами пользователей (начиная с 2.6.23)
CLONE_NEWUTS	Потомок получает новое пространство имен UTS (utsname()) (начиная с 2.6.19)
CLONE_PARENT	Делает родителя потомка тождественным вызывающему процессу родителя (начиная с 2.4)
CLONE_PARENT_SETTID	Записывает идентификатор дочернего потока в ptid (начиная с 2.6)
CLONE_PID	Устаревший флаг, который используется только процессом загрузки системы (до 2.4)
CLONE_PTRACE	Трассирует потомка, если трассируется его родитель
CLONE_SETTLS	tls описывает для потомка хранилище на уровне потока выполнения (начиная с 2.6)
CLONE_SIGHAND	Родитель и потомок разделяют действия сигналов
CLONE_SYSVSEM	Родитель и потомок разделяют значения отмены семафоров (начиная с 2.6)
CLONE_THREAD	Помещает потомка в одну группу потоков выполнения с его родителем (начиная с 2.4)
CLONE_UNTRACED	Не позволяет применить к потомку значение CLONE_PTRACE (начиная с 2.6)
CLONE_VFORK	Родитель приостанавливает работу, пока потомок не сделает вызов exec() или _exit()
CLONE_VM	Родитель и потомок разделяют виртуальную память

Оставшиеся аргументы вызова `clone()` — `ptid`, `tls` и `ctid`. Они относятся к реализации потока выполнения — в частности, к использованию его идентификатора и локального хранилища. Мы рассмотрим применение этих аргументов, когда будем описывать значения битовой маски `flags` в подразделе 28.2.1 (в версиях Linux до 2.4 включительно эти три аргумента отсутствуют в вызове `clone()`; их специально добавили в Linux 2.6 для поддержки библиотеки потоков POSIX NPTL).

Пример программы

В листинге 28.3 показан простой пример использования вызова `clone()` для создания дочернего процесса. Главная программа делает следующее.

- Открывает файловый дескриптор (для `/dev/null`), который будет закрыт потомком (2).
- Присваивает аргументу `flags` вызова `clone()` значение `CLONE_FILES` (3) (3). Если был предоставлен аргумент командной строки, родитель и потомок будут использовать единую таблицу файловых дескрипторов. В противном случае `flags` получает значение `0`.
- Выделяет стек для потомка (4).
- Если значение `CHILD_SIG` не равно `0` или `SIGCHLD`, оно будет игнорироваться в случае завершения процесса по сигналу. Мы не игнорируем `SIGCHLD`, поскольку это помешало бы нам ждать потомка для получения его статуса.
- Вызывает `clone()` для создания потомка (6). Третий аргумент (битовая маска) включает в себя сигнал завершения. Четвертый аргумент (`func_arg`) указывает на дескриптор ранее открытого файла (2).
- Ждет завершения работы потомка (7).
- Проверяет, открыт ли все еще файловый дескриптор (созданный на шаге (2)), пытаясь записать в него что-нибудь с помощью вызова `write()` (8), и сообщает результат этой попытки.

Выполнение клонированного потомка начинается с функции `childFunc()`, которая получает файловый дескриптор (посредством аргумента `arg`), открытый главной программой (на шаге (2)). Потомок закрывает этот дескриптор и завершается, выполняя операцию `return` (1).

Листинг 28.3. Использование clone() для создания дочерних процессов

procexec/t_clone.c

```
#define _GNU_SOURCE
#include <signal.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sched.h>
#include "tlpi_hdr.h"

#ifndef CHILD_SIG
#define CHILD_SIG SIGUSR1    /* При завершении клонированного потомка
                                будет сгенерирован сигнал */
#endif

static int    /* Начальная функция для клонированного потомка */
childFunc(void *arg)
{
(1) if (close(*((int *) arg)) == -1)
        errExit("close");
```

```
    return 0;    /* Здесь потомок завершается */
}

int
main(int argc, char *argv[])
{
    const int STACK_SIZE = 65536;   /* Размер стека для клонированного потомка */
    char *stack;           /* Начало буфера стека */
    char *stackTop;        /* Конец буфера стека */
    int s, fd, flags;

❷   fd = open("/dev/null", O_RDWR);   /* Потомок закроет дескриптор fd */
    if (fd == -1)
        errExit("open");

  /* Если argc > 1, потомок будет разделять таблицу файловых дескрипторов с родителем */

❸   flags = (argc > 1) ? CLONE_FILES : 0;

    /* Выделяем стек для потомка */

❹   stack = malloc(STACK_SIZE);
    if (stack == NULL)
        errExit("malloc");
    stackTop = stack + STACK_SIZE;  /* Предполагается, что стек растет сверху вниз */

    /* Игнорируем CHILD_SIG, если это сигнал, который по умолчанию завершает процесс;
       при этом не игнорируем SIGCHLD (который игнорируется по умолчанию), иначе это
       могло бы помешать созданию процесса-"зомби". */

❺   if (CHILD_SIG != 0 && CHILD_SIG != SIGCHLD)
        if (signal(CHILD_SIG, SIG_IGN) == SIG_ERR)
        errExit("signal");

    /* Создаем потомка; потомок начинает выполнение с функции childFunc() */
❻   if (clone(childFunc, stackTop, flags | CHILD_SIG, (void *) &fd) == -1)
        errExit("clone");
  /* Родитель переходит сюда и ждет потомка; потомку, который применяет
     для уведомления любой сигнал, кроме SIGCHLD, нужно значение _WCLONE. */

❼   if (waitpid(-1, NULL, (CHILD_SIG != SIGCHLD) ? __WCLONE : 0) == -1)
        errExit("waitpid");
    printf("child has terminated\n");

    /* Повлияло ли на родителя закрытие дескриптора в потомке? */

❽   s = write(fd, "x", 1);
    if (s == -1 && errno == EBADF)
        printf("file descriptor %d has been closed\n", fd);
    else if (s == -1)
        printf("write() on file descriptor %d failed "
            "unexpectedly (%s)\n", fd, strerror(errno));
    else
        printf("write() on file descriptor %d succeeded\n", fd);

    exit(EXIT_SUCCESS);
}
```

procexec/t_clone.c

Запустив программу из листинга 28.3 без аргумента командной строки, мы получим следующий результат:

```
$ ./t_clone                                  Не использует CLONE_FILES
child has terminated
write() on file descriptor 3 succeeded       Вызов close() в потомке не влияет на родителя
```

Если запустить программу с аргументом командной строки, можно увидеть, что оба процесса используют одну и ту же таблицу файловых дескрипторов:

```
$ ./t_clone x                              Использует CLONE_FILES
child has terminated
file descriptor 3 has been closed          Вызов close() в потомке затрагивает родителя
```

Более сложный пример использования вызова clone() представлен в файле procexec/demo_clone.c, входящем в архив с исходным кодом для этой книги.

28.2.1. Аргумент flags вызова clone()

Аргумент `flags` вызова `clone()` представляет собой сочетание (побитовое ИЛИ) значений битовой маски, описанных на следующих страницах. Мы рассмотрим их в порядке, который облегчает объяснение; начнем с тех флагов, что применяются в реализации потоков выполнения POSIX. С этой точки зрения термин *«процесс»*, многократно упоминающийся ниже, часто можно заменить словом *«поток»*.

На данном этапе стоит отметить, что наши попытки провести грань между терминами *«процесс»* и *«поток»* в некоторой степени являются жонглированием словами. Это немного помогает при знакомстве с понятием *единицы планирования ядра* (Kernel Scheduling Entity, KSE); с его помощью в технической литературе иногда описывают объекты, с которыми работает планировщик ядра. На самом деле потоки и процессы являются всего лишь экземплярами KSE, поддерживающими разную степень совместного использования атрибутов (виртуальной памяти, дескрипторов открытых файлов, действий сигналов, идентификатора процесса и т. д.). Спецификация POSIX-потоков описывает лишь один из множества возможных способов разделения атрибутов между потоками.

По мере описания далее мы иногда будем упоминать две главные реализации POSIX-потоков, доступные в Linux: LinuxThreads (старую) и NTPL (более современную). Детальную информацию о них можно найти в разделе 33.5.

Начиная с версии 2.6.16 ядро Linux предоставляет новый системный вызов unshare(), который позволяет потомку, созданному с помощью clone() (или fork(), или vfork()), отменить общее использование некоторых атрибутов (то есть аннулировать эффект от действия флагов falgs вызова clone()), установленное во время создания потомка. Подробности ищите на справочной странице unshare(2).

Разделение таблиц файловых дескрипторов: CLONE_FILES

Если указать флаг `CLONE_FILES`, родитель и потомок будут применять общую таблицу дескрипторов. Это означает, что результат выделения или утилизации файлового дескриптора (вызовы `open()`, `close()`, `dup()`, `pipe()`, `socket()` и т. д.) в одном из процессов будет доступен и в другом. Если флаг `CLONE_FILES` не указан, таблица файловых дескрипторов не разделяется, а потомок в момент вызова `clone()` получает копию родительской табли-

цы. Оригинальные и скопированные дескрипторы ссылаются на одни и те же открытые файлы (как в случае с `fork()` и `vfork()`).

Спецификация POSIX-потоков требует, чтобы все потоки в процессе пользовались общими дескрипторами открытых файлов.

Разделение информации, связанной с файловой системой: CLONE_FS

Если указать флаг `CLONE_FS`, родитель и потомок будут использовать общую информацию, связанную с файловой системой: umask, корневой и текущий каталог. Это означает, что вызовы `umask()`, `chdir()` или `chroot()`, сделанные в одном из процессов, затронут другой. Если флаг `CLONE_FS` не указан, родитель и потомок получают отдельные копии этой информации (как в случае с `fork()` и `vfork()`).

Разделение атрибутов, предоставляемое флагом `CLONE_FS`, является обязательным условием для POSIX-потоков.

Разделение действий сигналов: CLONE_SIGHAND

Если указать флаг `CLONE_SIGHAND`, родитель и потомок будут использовать общую таблицу действий сигналов. Применение вызовов `sigaction()` или `signal()` для изменения действия сигнала в одном из процессов отразится на другом. Если флаг `CLONE_SIGHAND` не указан, действия сигналов не разделяются; вместо этого потомок получает копию родительской таблицы (как в случае с `fork()` и `vfork()`). Флаг `CLONE_SIGHAND` не затрагивает маску сигнала и набор сигналов, находящихся в состоянии ожидания — эти данные всегда уникальны для каждого процесса. Начиная с Linux 2.6, при использовании `CLONE_SIGHAND` нужно также указывать флаг `CLONE_VM`.

Разделение действий сигналов является обязательным условием для POSIX-потоков.

Разделение виртуальной памяти родителя: CLONE_VM

Если указать флаг `CLONE_VM`, родитель и потомок будут использовать общие страницы виртуальной памяти (как в случае с `vfork()`). Изменения, вносимые в память, или вызовы `mmap()` и `munmap()` будут касаться сразу обоих процессов. Если флаг `CLONE_VM` не указан, потомок получает копию виртуальной памяти родителя (как в случае с `fork()`).

Разделение виртуальной памяти является одной из основополагающих характеристик потоков в целом и обязательным требованием POSIX-потоков в частности.

Группы потоков выполнения: CLONE_THREAD

Если указать флаг `CLONE_THREAD`, потомок будет помещен в одну группу потоков с родителем. В противном случае потомок помещается в новую группу, выделенную специально для него.

Группы потоков выполнения были добавлены в Linux 2.4, чтобы программные библиотеки могли выполнять требование стандарта POSIX, согласно которому все потоки в процессе должны иметь единый идентификатор PID (то есть вызов `getpid()` в каждом из потоков должен возвращать одно и то же значение). Как показано на рис. 28.1, группа потоков представляет собой группу экземпляров KSE, которые разделяют идентификатор группы потоков (Thread Group Identifier, или TGID). Далее в этом подразделе мы будем называть эти экземпляры просто *потоками*.

Начиная с Linux 2.4, вызов `getpid()` возвращает идентификатор TGID вызывающего потока. Иными словами, TGID ничем не отличается от идентификатора процесса.

Каждый поток имеет *уникальный идентификатор* (TID) в рамках своей группы. В Linux 2.4 был представлен новый системный вызов `gettid()`, который позволяет потоку получить свой собственный TID (это то же значение, что возвращается в поток, который вызывает `clone()`). Идентификатор потока представлен с помощью `pid_t` — того же типа данных, что используется для идентификатора процесса. TID является уникальным в рамках всей системы; ядро гарантирует, что любой такой идентификатор никогда не совпадет с идентификатором любого процесса в системе, за исключением тех случаев, когда поток в группе является лидирующим для процесса.

Рис. 28.1. Группа, состоящая из четырех потоков

Первый поток в новой группе имеет тот же идентификатор, что и сама группа. Такой поток называют *лидирующим*.

> Идентификаторы потоков, которые здесь обсуждаются, не являются тождественными тем идентификаторам, которые используются в POSIX-потоках (и имеют тип pthread_t). Последние генерируются и обслуживаются внутренним POSIX-механизмом, реализованным в пользовательском пространстве.

Все потоки в группе имеют один и тот же идентификатор родительского процесса — тот, что принадлежит лидеру группы. Родительский процесс получает сигнал `SIGCHLD` (или сигнал завершения) то после того, как завершатся все потоки в группе. Такое поведение соответствует требованиям, предъявляемым к POSIX-потокам.

Когда поток с флагом `CLONE_THREAD` завершает работу, поток, который создал его с помощью вызова `clone()`, не получает никакого сигнала. Соответственно, вызов `wait()` (и аналогичные ему) не подходит для ожидания потоков, созданных таким образом. Это отвечает требованиям стандарта POSIX. POSIX-поток не тождественен процессу; его нельзя отследить с помощью `wait()`. Вместо этого его следует присоединить, применив вызов `pthread_join()`. Для определения того, что поток, созданный с использованием флага `CLONE_THREAD`, завершил работу, применяется специальное средство синхронизации под названием *«фьютекс»* (см. описание флага `CLONE_PARENT_SETTID`, приведенное ниже).

Если какой-либо поток выполнит `exec()`, все потоки в его группе, кроме лидера, завершатся (это поведение соответствует требованием стандарта POSIX), а новая программа будет выполнена в потоке-лидере. Иными словами, вызов `gettid()` в новой программе вернет идентификатор потока, являющегося лидером группы. Во время выполнения вы-

зова exec() сигнал о завершении работы, который этот процесс должен послать своему родителю, сбрасывается к SIGCHLD.

Потомок, созданный в потоке с помощью вызовов fork() или vfork(), может быть отслежен любым участником группы этого потока с использованием вызова wait() или аналогичного.

Начиная с Linux 2.6, при использовании CLONE_THREAD в аргументе flags необходимо также указывать флаг CLONE_SIGHAND. Это отвечает требованиям стандарта POSIX; больше подробностей можно найти в описании взаимодействия POSIX-потоков и сигналов в разделе 33.2 (способ обработки ядром сигналов для групп потоков, созданных с помощью флага CLONE_THREAD, отражает требования стандарта POSIX, касающиеся того, как потоки внутри процесса должны реагировать на сигналы).

Поддержка библиотек для работы с потоками: CLONE_PARENT_SETTID, CLONE_CHILD_SETTID и CLONE_CHILD_CLEARTID

Флаги CLONE_PARENT_SETTID, CLONE_CHILD_SETTID и CLONE_CHILD_CLEARTID были добавлены в Linux 2.6 для поддержки реализации POSIX-потоков. Они влияют на то, как вызов clone() обращается со своими аргументами ptid и ctid. Флаги CLONE_PARENT_SETTID и CLONE_CHILD_CLEARTID также используются в реализации NPTL-потоков.

Если указать флаг CLONE_PARENT_SETTID, то ядро запишет идентификатор дочернего потока в место, на которое указывает ptid. Этот идентификатор копируется в ptid перед дублированием памяти родителя. Это означает, что даже при отсутствии флага CLONE_VM значение с этим местоположением будет доступно как родителю, так и потомку (как уже отмечалось выше, флаг CLONE_VM указывается при создании POSIX-потоков).

Флаг CLONE_PARENT_SETTID нужен для того, чтобы поточная библиотека имела надежный способ получения идентификатора нового потока. Стоит отметить, что использование для этого значения, возвращаемого вызовом clone() (как показано ниже) не является достаточным:

```
tid = clone(...);
```

Дело в том, что данный код может привести к различным видам состояния гонки, поскольку присваивание выполняется только после возвращения вызова clone(). Представим, к примеру, что новый поток завершается и что его обработчик сигнала завершения вызывается до того, как переменной tid присвоится соответствующее значение. В этом случае обработчик не может получить доступ к tid эффективным способом (в рамках поточной библиотеки tid может быть всего лишь элементом глобальной структуры, с помощью которой отслеживается статус всех потоков). Часто программы, вызывающие clone() напрямую, способны обойти это состояние гонки. Однако поточная библиотека не может контролировать действия вызывающей ее программы. Этой проблемы можно избежать, если до возврата вызова clone() гарантировать, что идентификатор нового потока сохранен по адресу, на который указывает ptid; и именно этого позволяет добиться флаг CLONE_PARENT_SETTID.

Если указать флаг CLONE_CHILD_SETTID, вызов clone() запишет идентификатор дочернего потока в место, на которое указывает ctid. Это происходит только в памяти потомка, но родитель тоже может быть затронут, если дополнительно указать флаг CLONE_VM. И хотя CLONE_CHILD_SETTID не предусмотрен в библиотеке NPTL, с его помощью можно повысить гибкость альтернативных реализаций.

Если указать флаг CLONE_CHILD_CLEARTID, во время завершения потомка вызов clone() обнулит участок памяти, на который указывает ctid.

Аргумент ctid представляет собой механизм, с помощью которого библиотека NPTL получает уведомление о завершении потока. Это уведомление необходимо для функции pthread_join(), которая позволяет одному POSIX-потоку ждать завершения другого.

При создании потока с помощью функции pthread_join() библиотека NPTL делает вызов clone(), в котором аргументы ptid и ctid указывают на один и тот же участок памяти (именно по этой причине CLONE_CHILD_SETTID не входит в состав NPTL). Благодаря флагу CLONE_PARENT_SETTID этот участок инициализируется идентификатором нового потока. Когда потомок завершает работу, а ctid очищается, данное изменение доступно всем потокам в процессе (поскольку был дополнительно указан флаг CLONE_VM).

Ядро обращается с участком памяти, на который указывает ctid, как с *фьютексом* — высокопроизводительным механизмом синхронизации (больше подробностей о фьютексах можно узнать на справочной странице futex(2)). Уведомление о завершении потока можно получить с помощью системного вызова futex(), который блокирует ожидание изменения в содержимом участка памяти, связанного с ctid (внутри происходит то же самое, что и в функции pthread_join()). В то же время ядро очищает ctid и возобновляет работу любых единиц планирования ядра (то есть потоков), заблокированных из-за ожидания фьютексом этого адреса (на уровне POSIX-потоков это приводит к разблокированию с помощью вызова pthread_join()).

Локальное хранилище на уровне потока: CLONE_SETTLS

Если указать флаг CLONE_SETTLS, аргумент tls будет указывать на структуру user_desc, описывающую буфер локального хранилища, которое будет использоваться в этом потоке. Этот флаг был добавлен в Linux 2.6 для поддержки локальных хранилищ на уровне потока в библиотеке NPTL (см. раздел 31.4). Подробные сведения о структуре user_desc можно почерпнуть в исходных кодах ядра версии 2.6 и на справочной странице set_thread_area(2).

Отдельные пространства имен файловой системы для каждого процесса: CLONE_NEWNS

Начиная с версии 2.4.19, ядро Linux поддерживает отдельные *пространства имен файловой системы* для каждого процесса, которые представляют собой набор точек подключения, обслуживаемых вызовами mount() и umount(). Пространство имен файловой системы отвечает за то, каким образом из путей получаются соответствующие файлы, а также за работу таких системных вызовов, как chdir() и chroot().

Родитель и потомок по умолчанию используют общее пространство имен файловой системы. Это означает, что изменения, внесенные с помощью mount() и umount() в одном процессе будут видны другому (как в случае с fork() и vfork()). Привилегированный процесс (CAP_SYS_ADMIN) может указать флаг CLONE_NEWNS, чтобы потомок получил копию пространства имен ФС своего родителя. В результате этого изменения будут доступны только тому процессу, который их вносит (в ядрах версий 2.4.x и ниже можно считать, что все процессы в системе работают с единым системным пространством имен ФС).

С помощью разделения пространств имен ФС можно создавать изолированные среды, похожие на «тюрьмы» chroot() *(chroot jails)*, но более безопасные и гибкие; например, изолированному процессу можно предоставить точку подключения, недоступную для любого другого процесса в системе. Пространства имен ФС также могут пригодиться при создании среды для виртуального сервера.

Одновременное использование флагов `CLONE_NEWNS` и `CLONE_FS` в одном и том же вызове `clone()` не имеет смысла и является недопустимым.

Назначение родителя вызывающего процесса родителем потомка: CLONE_PARENT

Родителем процесса, который создается с помощью `clone()` (и который соответствует значению, возвращаемому вызовом `getppid()`), по умолчанию является процесс, вызвавший `clone()` (как в случае с `fork()` и `vfork()`). Но если указать флаг `CLONE_PARENT`, родителем потомка будет родитель вызывающего процесса. Иными словами, флаг `CLONE_PARENT` тождественен присваиванию `child.PPID = caller.PPID` (в стандартной ситуации, без флага `CLONE_PARENT`, это выглядело бы как `child.PPID = caller.PID`). Родительским (`child.PPID`) является процесс, которому приходит сигнал при завершении потомка.

Флаг `CLONE_PARENT` доступен в Linux 2.4 и выше. Изначально он задумывался для удобства реализации POSIX-потоков, однако в ядре версии 2.6 был сделан акцент на вспомогательных потоках (с применением флага `CLONE_THREAD`, описанного выше), благодаря чему необходимость в данном флаге отпала.

Назначение потомку идентификатора его родителя: CLONE_PID (устаревший)

Если указать флаг CLONE_PID, потомок будет иметь такой же идентификатор, как его родитель. В противном случае родитель и потомок получают разные идентификаторы PID (как в случае с `fork()` и `vfork()`). Этот флаг может устанавливать только процесс загрузки системы (тот, чей PID равен `0`); он используется во время инициализации мультипроцессорных систем.

Флаг `CLONE_PID` не предназначен для применения в пользовательских приложениях. Он больше не доступен в Linux 2.6, а ему на замену пришел флаг `CLONE_IDLETASK`, который приводит к тому, что идентификатор нового процесса равняется 0. Флаг `CLONE_IDLETASK` доступен только для внутреннего применения в ядре и игнорируется вызовом `clone()`. Он используется для создания невидимых процессов, находящихся в режиме ожидания; каждый такой процесс отводится для отдельного ЦПУ (в многопроцессорных системах их может быть несколько).

Трассировка процессов: CLONE_PTRACE и CLONE_UNTRACED

Если указать флаг `CLONE_PTRACE`, потомок трассируемого процесса тоже будет трассироваться. Подробности о трассировке процессов (которая используется при отладке и в команде `strace`) ищите на справочной странице `ptrace(2)`.

В версиях Linux 2.6 и выше можно указывать флаг `CLONE_UNTRACED`, который не позволяет трассируемому процессу применить флаг `CLONE_PTRACE` к своему потомку. Он используется внутри системы для создания потоков ядра.

Приостановка родителя до тех пор, пока потомок не завершится или не выполнит exec: CLONE_VFORK

Если указать флаг CLONE_VFORK, выполнение родителя приостанавливается до тех пор, пока потомок не освободит ресурсы своей виртуальной памяти с помощью вызовов `exec()` или `_exit()` (как в случае с `vfork()`).

Новые флаги вызова clone() для поддержки контейнеров

В Linux 2.6.19 и выше появилась поддержка целого ряда новых флагов для вызова `clone()`: `CLONE_IO`, `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWUSER` и `CLONE_NEWUTS` (см. справочную страницу `clone(2)`).

Большинство из этих флагов обеспечивают поддержку *контейнеров* ([Bhattiprolu et al., 2008]). Контейнер — это разновидность легковесной виртуализации, при которой группы процессов, выполняющихся на одном ядре, могут быть изолированы друг от друга в средах, напоминающих отдельные компьютеры. Контейнеры также могут быть вложенными, один внутри другого. Данный подход отличается от полноценной виртуализации, при которой каждая виртуальная среда работает на отдельном ядре.

Для реализации контейнеров разработчикам Linux пришлось создать в ядре виртуальный слой вокруг каждого глобального ресурса системы — это, например, касается идентификаторов процессов, сетевого стека, идентификаторов, возвращаемых вызовом `uname()`, IPC-объектов из System V, а также пространств имен для идентификаторов пользователей и групп. Благодаря этому каждый контейнер может получить собственный экземпляр этих ресурсов.

Контейнеры имеют множество различных применений, включая следующие:

- контроль над выделением ресурсов системы, таких как пропускная способность сети или процессорное время (например, одному контейнеру можно выдать 75 % процессорного времени, а другому — 25 %);
- предоставление нескольких легковесных виртуальных серверов на одном и том же компьютере;
- замораживание контейнера, чтобы приостановить выполнение всех процессов внутри него и позже, возможно после миграции на другой компьютер, запустить их заново;
- возможность сбросить на диск состояние приложения (создав тем самым контрольную точку) и затем восстановить его (возможно, после аварийного завершения программы или плановой/неплановой перезагрузки системы), чтобы продолжить выполнение с того самого места.

Использование флагов вызова clone()

Вызов `fork()`, грубо говоря, соответствует вызову `clone()`, аргумент `flags` которого равен `SIGCHLD`, тогда как `vfork()` соответствует `clone()` со следующими флагами:

```
CLONE_VM | CLONE_VFORK | SIGCHLD
```

> Обертка fork() библиотеки glibc версии 2.3.3 и выше, которая входит в состав реализации NPTL-потоков, обходит одноименный системный вызов и обращается вместо него к вызову clone(). Эта функция запускает любые обработчики, установленные вызывающим процессом с помощью pthread_atfork() (см. раздел 33.3).

В библиотеке LinuxThreads для создания потоков вызов `clone()` (но только с первыми четырьмя его аргументами) применяется в сочетании со следующими флагами:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

В библиотеке NPTL вызов `clone()` (вместе со всеми семью его аргументами) создает потоки с помощью таких флагов:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS |
CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

28.2.2. Расширения к вызову waitpid() для клонированных потомков

Для ожидания потомков, созданных с помощью вызова `clone()`, в аргумент `options` вызовов `waitpid()`, `wait3()` или `wait4()` (который является битовой маской) можно передавать следующие значения (актуальные только для Linux).

- `__WCLONE` — в зависимости от того, установлен этот флаг или нет, родитель ожидает только *клонированных* или соответственно только *неклонированных* потомков. В данном контексте клонированным является потомок, который при завершении работы отправляет родителю сигнал, отличающийся от `SIGCHLD`. Этот бит игнорируется, если использовать его в сочетании с `__WALL`.
- `__WALL` (начиная с Linux 2.4) — приводит к ожиданию любых потомков, независимо от того, *клонированные* они или *нет*.
- `__WNOTHREAD` (начиная с Linux 2.4) — по умолчанию вызовы `wait()` позволяют ждать потомков не только вызывающего, но и любого другого процесса в одной с ним группе. Флаг `__WNOTHREAD` ограничивает процедуру ожидания потомками вызывающего процесса. Эти флаги нельзя использовать в вызове `waitid()`.

28.3. Скорость создания процессов

В табл. 28.3 приводится сравнение производительности разных методов создания процессов. Результаты были собраны с помощью тестовой программы, которая создает в цикле дочерние процессы и ждет, когда те завершатся. В сравнении используются три разных объема памяти, выделяемой для процесса (столбец «Общая виртуальная память»). Это было достигнуто путем выделения дополнительного места в куче с помощью вызова `malloc()` до начала замеров.

Значения размеров процессов («Общая виртуальная память») в табл. 28.3 взяты из столбца VSZ при выводе команды ps -o "pid vsz cmd".

Таблица 28.3. Время, необходимое для создания 100 000 процессов с помощью вызовов fork(), vfork() и clone()

Метод создания процесса	Общая виртуальная память					
	1,70 Мбайт		2,70 Мбайт		11,70 Мбайт	
	Время (с)	Количество	Время (с)	Количество	Время (с)	Количество
fork()	22,27 (7,99)	4544	26,38 (8,98)	4135	126,93 (52,55)	1276
vfork()	3,52 (2,49)	28 955	3,55 (2,50)	28621	3,53 (2,51)	28 810
clone()	2,97 (2,14)	34 333	2,98 (2,13)	34217	2,93 (2,10)	34 688
fork() + + exec()	135,72 (12,39)	764	146,15 (16,69)	719	260,34 (61,86)	435
vfork() + + exec()	107,36 (6,27)	969	107,81 (6,35)	964	107,97 (6,38)	960

Для каждого размера в табл. 28.3 представлено два вида статистики.

- Первый вид состоит из двух временных измерений. Первое (то, что больше) представляет общее (реальное) время, затраченное на создание 100 000 процессов. Второе, заключенное в скобки, является процессорным временем, потребленным родительским процессом. Поскольку эти тесты выполнялись на свободном от других задач компьютере, разница между этими двумя значениями представляет собой общее время существования дочерних процессов.
- Второй вид статистики в каждом из тестов показывает количество создаваемых процессов за одну (реальную) секунду. Для каждого случая выполнялось 20 тестовых прогонов; статистика была получена в системе с архитектурой x86-32 и ядром 2.6.27.

Первые три строки с данными содержат время, уходящее на простое создание процесса (без выполнения новой программы внутри потомка). В каждом из случаев дочерний процесс завершает свою работу сразу же после создания, а родитель ожидает этого момента, чтобы создать следующий процесс.

В первом столбце находятся значения для системного вызова `fork()`. По этим данным видно, что чем больше процесс, тем дольше он создается. Разница в значениях отражает дополнительное время, необходимое для копирования все более объемных страниц памяти родителя и перевод всех страничных записей (данных, кучи и стека) в режим только для чтения (сами *страницы* не копируются, так как потомок не изменяет свои сегменты с данными и стеком).

Второй столбец предоставляет такую же статистику для `vfork()`. Мы видим, что, несмотря на увеличение размера процесса, время остается неизменным. Дело в том, что вызов `vfork()` не копирует таблицы со страницами памяти или сами страницы, поэтому размер виртуальной памяти вызывающего процесса не играет никакой роли. Разница между статистикой для вызовов `fork()` и `vfork()` представляет общее время, необходимое в каждом из случаев для копирования таблицы со страницами.

> Небольшая разница между значениями для vfork() и clone(), показанными в табл. 28.3, связана с выборкой ошибок и отклонениями в планировщике. Даже если увеличить размер создаваемых процессов до 300 Мбайт, временные показатели этих двух вызовов не изменятся.

Третий столбец показывает статистику создания процессов с помощью вызова `clone()` и следующего сочетания флагов:

```
CLONE_VM | CLONE_VFORK | CLONE_FS | CLONE_SIGHAND | CLONE_FILES
```

Первые два из них эмулируют поведение вызова `vfork()`. Оставшиеся флаги указывают на то, что родитель и потомок должны иметь общие атрибуты файловой системы (`umask`, корневой и текущий каталог), таблицу действий сигналов и таблицу дескрипторов открытых файлов. Разница между результатами для вызовов `clone()` и `vfork()` представляет небольшой объем дополнительной работы, выполняемой последней для копирования этих данных в дочерний процесс. Затраты на копирование атрибутов файловой системы и таблицы действий сигналов остаются постоянными. Однако время копирования таблицы дескрипторов открытых файлов зависит от количества этих дескрипторов. Например, открытие в родительском процессе 100 файлов увеличит реальное время выполнения вызова `vfork()` (первый столбец в таблице) с 3,52 до 5,04 секунды, но никак не повлияет на вызов `clone()`.

> Здесь приводится время выполнения функции-обертки clone() из библиотеки glibc, а не самого системного вызова sys_clone(). Другие тесты (которые здесь не приводятся) показали пренебрежительно малую разницу между sys_clone() и clone() в случае, когда дочерняя функция немедленно завершает работу.

Различия между вызовами `fork()` и `vfork()` довольно значительные. Однако необходимо учитывать следующие обстоятельства.

- Последний столбец таблицы, в котором `vfork()` показывает более чем 30-кратный прирост производительности по сравнению с `fork()`, относится к большому процессу. Обычные процессы имели бы показатели, близкие к первым двум столбцам.
- Поскольку время, необходимое для создания процесса, обычно меньше того, что требуется для выполнения `exec()`, разница между вызовами `fork()` и `vfork()` становится гораздо менее заметной, если после них запустить новую программу. Этот факт проиллюстрирован в последних двух столбцах табл. 28.3, в которых каждый потомок вместо немедленного завершения выполняет `exec()`. В качестве запускаемой программы использовалась команда `true` (`/bin/true`, выбранная потому, что она не генерирует вывода). В этом случае видно, что различия между `fork()` и `vfork()` получились намного меньшими.

  На самом деле данные, показанные в табл. 28.3, не отражают всех затрат, связанных с вызовом exec(), поскольку потомок запускает одну и ту же программу в каждой итерации цикла. В результате ресурсы, затрачиваемые на ввод/вывод для считывания программы с диска в память, практически нивелируются, поскольку программа будет скопирована в кэш буфера ядра при первом выполнении exec() и останется там. Если бы на каждой итерации цикла вызывалась новая программа (например, копия одной и той же команды, но с другим именем), мы бы могли увидеть, что на вызов exec() тратится больше ресурсов.

28.4. Влияние вызовов exec() и fork() на атрибуты процесса

Процесс обладает множеством атрибутов. Часть из них уже была описана ранее, а остальные мы рассмотрим в последующих главах. В этом контексте встает два вопроса.

- Что происходит с этими атрибутами, когда процесс выполняет `exec()`?
- Какие атрибуты наследуются потомком при вызове `fork()`?

Ответы собраны в табл. 28.4. Столбец `exec()` показывает, какие атрибуты сохраняются во время вызова `exec()`. В столбце `fork()` перечисляются атрибуты, наследуемые (или в некоторых случаях разделяемые) потомком после вызова `fork()`. Все перечисленные ниже атрибуты (кроме тех, что обозначены как уникальные для Linux) входят в стандартную реализацию UNIX, а их обработка во время выполнения `exec()` и `fork()` соответствует стандарту SUSv3.

Таблица 28.4. Влияние вызовов exec() и fork() на атрибуты процесса

Атрибут процесса	**exec()**	**fork()**	**Интерфейсы, затрагивающие атрибуты; дополнительные заметки**
Адресное пространство процесса			
Текстовый сегмент	Нет	Общий	Дочерний процесс разделяет текстовый сегмент с родителем
Сегмент стека	Нет	Да	Входные/выходные точки функций; alloca(), longjmp(), siglongjmp()

Продолжение ⇙

Таблица 28.4 *(продолжение)*

Атрибут процесса	exec()	fork()	Интерфейсы, затрагивающие атрибуты; дополнительные заметки
Сегменты данных и кучи	Нет	Да	brk(), sbrk()
Переменные среды	См. заметки	Да	putenv(), setenv(); непосредственное изменение environ. Перезаписываются вызовами execle() и execve(), остаются неизменными после вызовов exec()
Отображение в память	Нет	См. заметки	mmap() и munmap(). Флаг MAP_NORESERVE отображения наследуется во время выполнения fork(). Отображения, помеченные madvise(MADV_DONTFORK), не наследуются во время выполнения fork()
Блокировки памяти	Нет	Нет	mlock(), munlock()
Идентификаторы и привилегии процесса			
Идентификатор процесса	Да	Нет	
Идентификатор родительского процесса	Да	Нет	
Идентификатор группы процесса	Да	Да	setpgid()
Идентификатор сессии	Да	Да	setsid()
Реальные идентификаторы	Да	Да	setuid(), setgid() и связанные с ними вызовы
Действующий и сохраненный идентификаторы	См. заметки	Да	setuid(), setgid() и связанные с ними вызовы. То, как exec() влияет на эти идентификаторы, объясняется в главе 9
Дополнительные групповые идентификаторы	Да	Да	setgroups(), initgroups()
Файлы, файловый ввод/вывод и каталоги			
Дескрипторы открытых файлов	См. заметки	Да	open(), close(), dup(), pipe(), socket() и т. д. Файловые дескрипторы сохраняются во время exec(), если не установлен флаг FD_CLOEXEC. Дескрипторы в родителе и потомке ссылаются на одни и те же открытые файлы; см. раздел 5.4
Флаг FD_CLOEXEC	Да (если сброшен)	Да	fcntl(F_SETFD)
Файловые сдвиги	Да	Разделяемые	lseek(), read(), write(), readv(), writev(). Потомок разделяет файловые сдвиги с родителем
Статусные флаги открытого файла	Да	Разделяемые	open(), fcntl(F_SETFL). Потомок разделяет статусные флаги открытого файла с родителем
Асинхронные операции ввода/вывода	См. заметки	Нет	aio_read(), aio_write() и связанные с ними вызовы. Незавершенные задачи отменяются во время exec()

Атрибут процесса	exec()	fork()	Интерфейсы, затрагивающие атрибуты; дополнительные заметки
Потоки каталогов	Нет	Да; см. заметки	opendir(), readdir()
Файловая система			
Текущий каталог	Да	Да	chdir()
Корневой каталог	Да	Да	chroot()
Маска режима создания файла	Да	Да	umask()
Сигналы			
Действия сигналов	См. заметки	Да	signal(), sigaction(). Во время выполнения exec(), действия сигналов, которые игнорируются или где используются значения по умолчанию, остается без изменений; перехваченные сигналы изменяют действия на значения по умолчанию. См. раздел 27.5
Маски сигналов	Да	Да	Доставка сигналов, igprocmask(), sigaction()
Набор ожидающих сигналов	Да	Нет	Доставка сигналов; raise(), kill(), sigqueue()
Альтернативный стек сигналов	Нет	Да	sigaltstack()
Таймеры			
Интервальные таймеры	Да	Нет	setitimer()
Таймеры, установленные вызовом alarm()	Да	Нет	alarm()
POSIX-таймеры	Нет	Нет	timer_create() и связанные с ним вызовы
POSIX-потоки			
Потоки	Нет	См. заметки	Во время выполнения fork() в потомке копируется только вызывающий поток
Тип и состояние потока с точки зрения его «отменяемости»	Нет	Да	После вызова exec() тип и состояние «отменяемости» сбрасываются к значениям соответственно PTHREAD_CANCEL_ENABLE и PTHREAD_CANCEL_DEFERRED
Мьютексы и условные переменные	Нет	Да	Подробности об обращении с мьютексами и другими ресурсами потоков во время выполнения fork() можно найти в разделе 33.3
Приоритет и планирование			
Значение nice	Да	Да	nice(), setpriority()
Политика планирования и приоритет	Да	Да	sched_setscheduler(), sched_setparam()

Продолжение ⇙

Таблица 28.4 *(продолжение)*

Атрибут процесса	exec()	fork()	Интерфейсы, затрагивающие атрибуты; дополнительные заметки
Ресурсы и процессорное время			
Ограничения на ресурсы	Да	Да	setrlimit()
Процессорное время процесса и потомка	Да	Нет	Значения, возвращаемые вызовом times()
Потребление ресурсов	Да	Нет	Значения, возвращаемые вызовом getrusage()
Межпроцессное взаимодействие			
Общие сегменты памяти в System V	Нет	Да	shmat(), shmdt()
Разделяемая память в POSIX	Нет	Да	shm_open() и связанные с ним вызовы
Очереди сообщений в POSIX	Нет	Да	mq_open() и связанные с ним вызовы. Дескрипторы потомка и родителя ссылаются на одну и ту же очередь открытых сообщений. Потомок не наследует от родителя его регистрации уведомлений о сообщениях
Именованные семафоры в POSIX	Нет	разделяемые	sem_open() и связанные с ним вызовы. Потомок разделяет ссылки на те же семафоры, что и родитель
Неименованные семафоры в POSIX	Нет	См. заметки	sem_init() и связанные с ним вызовы. Если семафоры находятся на участке разделяемой памяти, потомок и родитель используют общие семафоры; в противном случае потомок получает собственную копию семафоров
Корректировки семафоров в System V	Да	Нет	semop()
Блокировки файлов	Да	См. заметки	flock(). Потомок наследует от родителя ссылку на ту же блокировку
Блокировки записей	См. заметки	Нет	fcntl(F_SETLK). Блокировки сохраняются во время выполнения exec(), если только файловый дескриптор, ссылающийся на файл, не помечен флагом FD_CLOEXEC; см. подраздел 51.3.5
Разное			
Настройки локализации	Нет	Да	setlocale(). После запуска новой программы выполняется эквивалент setlocale(LC_ALL, “C”); это часть процедуры инициализации среды выполнения C
Настройки плавающей запятой	Нет	Да	Когда выполняется новая программа, состояние настроек плавающей точки сбрасывается к значению по умолчанию; см. fenv(3)
Управляющий терминал процесса	Да	Да	

Атрибут процесса	exec()	fork()	Интерфейсы, затрагивающие атрибуты; дополнительные заметки
Обработчики выхода	Нет	Да	atexit(), on_exit()
Относящееся только к Linux			
Идентификаторы ФС	См. заметки	Да	setfsuid(), setfsgid(). Эти идентификаторы меняются при каждом изменении действующих идентификаторов
Таймеры timerfd	Да	См. заметки	timerfd_create(); потомок наследует файловые дескрипторы, ссылающиеся на те же таймеры, что и родитель
Возможности (Capabilities)	См. заметки	Да	capset(). Работа с возможностями во время выполнения exec() описана в разделе 39.5
Набор привязок для возможностей	Да	Да	
Флаги securebits для возможностей	См. заметки	Да	Во время выполнения exec() в securebits сохраняются все флаги, кроме SECBIT_KEEP_CAPS, который всегда удаляется
Привязка к ЦПУ	Да	Да	sched_setaffinity()
SCHED_RESET_ON_FORK	Да	Нет	См. подраздел 35.3.2
Разрешенные ЦПУ	Да	Да	См. cpuset(7)
Разрешенные узлы памяти	Да	Да	См. cpuset(7)
Политика памяти	Да	Да	См. set_mempolicy(2)
Аренда файлов	Да	См. заметки	fcntl(F_SETLEASE). Потомок наследует ссылку на ту же аренду, что и родитель
Уведомления об изменении каталога	Да	Нет	Программный интерфейс dnotify, доступный через fcntl(F_NOTIFY)
prctl(PR_SET_DUMPABLE)	См. заметки	Да	Флаг PR_SET_DUMPABLE устанавливается во время выполнения exec(), если запускаемая программа не устанавливает пользовательские или групповые идентификаторы. В ином случае он удаляется
prctl(PR_SET_PDEATHSIG)	Да	Нет	
prctl(PR_SET_NAME)	Нет	Да	
oom_adj	Да	Да	
coredump_filter	Да	Да	См. раздел 22.1

28.5. Резюме

Когда включен учет ресурсов, ядро записывает в файл учетные сведения о каждом завершающемся процессе в системе.

По аналогии с `fork()`, системный вызов `clone()` (доступный только в Linux) создает новый процесс, но при этом позволяет более тонко управлять разделением атрибутов

между родителем и потомком. Он в основном используется для реализации поточных библиотек.

Мы сравнили скорость создания процессов с помощью `fork()`, `vfork()` и `clone()`. Вызов `vfork()` опережает `fork()` по скорости, однако разница между этими системными вызовами небольшая, если сравнивать с временем, которое уходит у потомка на выполнение последующего вызова `exec()`.

Дочерний процесс, созданный с помощью вызова `fork()`, наследует копии (или в некоторых случаях разделяет) определенные атрибуты своего родителя. Например, потомок наследует от родителя копии таблицы файловых дескрипторов и действия сигналов, но не интервалы таймеров, блокировки записей или набор ожидающих сигналов. Соответственно, когда процесс выполняет `exec()`, одни его атрибуты остаются без изменений, а другие сбрасываются к значениям по умолчанию. Например, идентификатор процесса сохраняется прежним, файловые дескрипторы остаются открытыми (если не указан флаг `FD_CLOEXEC`), интервалы таймеров не меняются, а сигналы остаются в режиме ожидания; однако обрабатываемые сигналы сбрасывают свои действия к значениям по умолчанию, а сегменты общей памяти разъединяются.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 24.6. В книге [Frisch, 2002] в главе 17 описывается управление системой учета ресурсов, а также рассматриваются некоторые отличия между разными реализациями UNIX. В книге [Bovet & Cesati, 2005] описывается реализация системного вызова `clone()`.

28.6. Упражнение

28.1. Напишите программу, чтобы увидеть, насколько быстро работают системные вызовы `fork()` и `vfork()` на вашем компьютере. Каждый дочерний процесс должен немедленно завершаться, а родитель перед созданием следующего потомка должен ждать этого момента с помощью вызова `wait()`. Сравните различия между этими двумя вызовами с данными из табл. 28.3. Для измерения времени выполнения программы можно воспользоваться встроенной в оболочку командой `time`.

29 Потоки выполнения: введение

Эта и несколько следующих глав посвящены POSIX-потокам, известным также как *Pthreads*. Однако мы не ставим перед собой цели рассмотреть весь программный интерфейс Pthreads, поскольку он довольно объемный. В конце этой главы будут представлены различные источники дополнительной информации о потоках.

В данных главах в основном описывается стандартное поведение потоков, предусмотренное программным интерфейсом Pthreads. В разделе 33.5 мы обсудим различия между двумя главными реализациями потоков в Linux — LinuxThreads и NPTL (Native POSIX Threads Library).

В этой главе мы сначала ознакомимся с кратким обзором работы потоков, а затем сосредоточимся на том, как они создаются и завершаются. В конце будут рассмотрены некоторые факторы, которые следует учитывать при выборе между двумя разными подходами к проектирования приложений — многопоточным и многопроцессным.

29.1. Краткий обзор

По аналогии с процессами потоки выполнения представляют собой механизм для одновременного выполнения нескольких параллельных задач в рамках одного приложения. Как показано на рис. 29.1, один процесс может содержать несколько потоков. Все они выполняются внутри одной программы независимо друг от друга, разделяя общую глобальную память — в том числе инициализированные/неинициализированные данные и сегменты кучи (традиционный для UNIX процесс является всего лишь частным случаем многопоточного процесса; он состоит из одного потока).

> На рис. 29.1 допущены некоторые упрощения. В частности, местоположение стеков у каждого из потоков может пересекаться с разделяемыми библиотеками и общими участками памяти; это зависит от порядка, в котором создавались потоки, загружались библиотеки и присоединялись общие участки памяти. Кроме того, местоположение стеков у потоков может меняться в зависимости от дистрибутива Linux.

Потоки в процессе могут выполняться одновременно. В многопроцессорных системах возможно параллельное выполнение потоков. Если один поток заблокирован из-за ввода/вывода, другой может продолжать работу (иногда имеет смысл создать отдельный поток, который занимается исключительно вводом/выводом, хотя, часто более подходящими оказываются альтернативные модели ввода/вывода; подробней об этом — в главе 59).

В некоторых ситуациях потоки имеют преимущество перед процессами. Рассмотрим традиционный для UNIX подход к обеспечению конкурентного выполнения за счет создания нескольких процессов. Возьмем для примера модель сетевого сервера, в которой родительский процесс принимает входящие подключения и создает с помощью вызова `fork()` отдельные дочерние процессы для общения с каждым клиентом (см. раздел 56.3). Это позволяет одновременно обслуживать сразу несколько подключений. Такой подход обычно хорошо себя проявляет, но в некоторых ситуациях он приводит к следующим ограничениям.

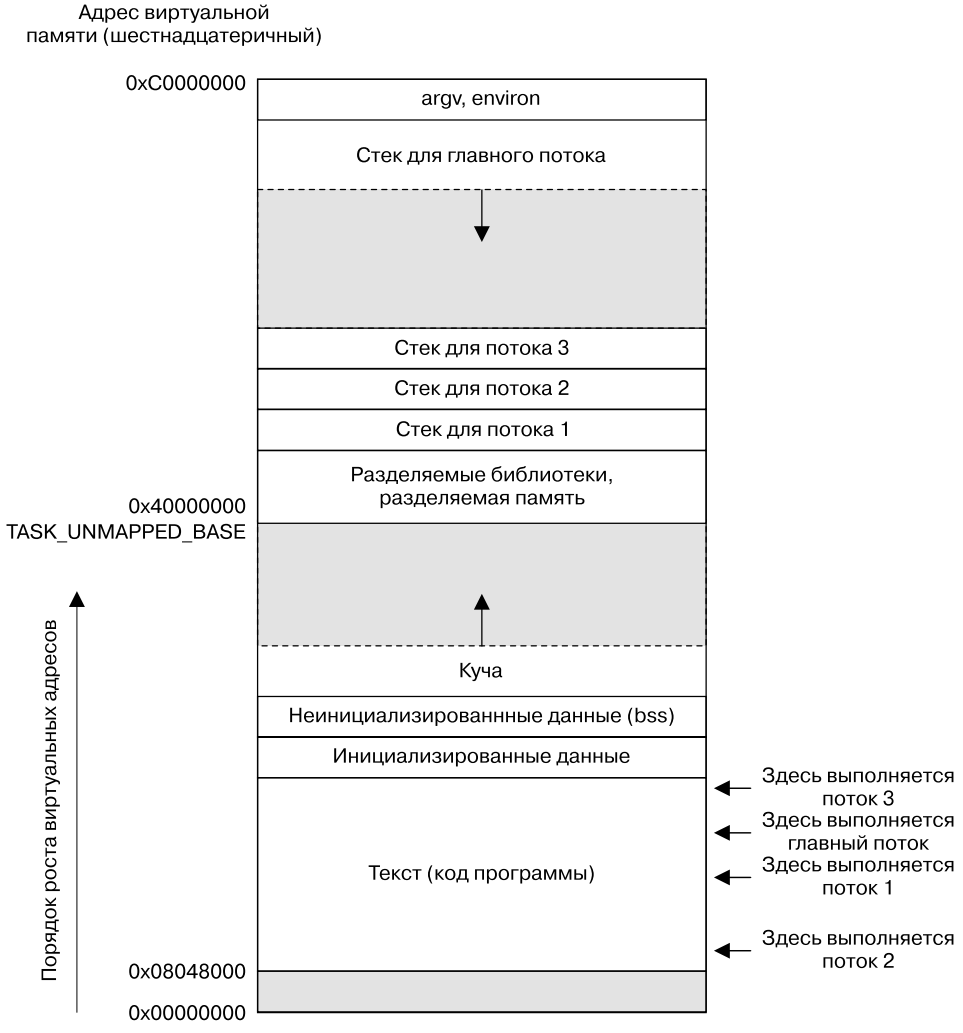


Рис. 29.1. Четыре потока, выполняющиеся внутри процесса (Linux/x86-32)

- Обмен информацией между процессами имеет свои сложности. Поскольку родитель и потомок не разделяют память (помимо текстового сегмента, предназначенного сугубо для чтения), мы вынуждены использовать некую форму межпроцессного взаимодействия для обмена данными.
- Создание процесса с помощью `fork()` потребляет относительно много ресурсов. Даже если использовать метод копирования при записи, описанный в подразделе 24.2.2, нам все равно приходится дублировать различные атрибуты процесса, такие как таблицы со страницами памяти и файловыми дескрипторами; это означает, что вызов `fork()` по-прежнему занимает существенное время.

  Потоки помогают избавиться от обеих этих проблем.
- Обмен информации между потоками является простым и быстрым. Для этого всего лишь нужно скопировать данные в общие переменные (глобальные или в куче).

Но чтобы избежать проблем, которые могут возникнуть в ситуации, когда сразу несколько потоков пытаются обновить одну и ту же информацию, приходится применять методы синхронизации, описанные в главе 30.

- Создание потока занимает меньше времени, чем создание процесса — обычно имеем как минимум десятикратный выигрыш в производительности (в Linux потоки реализованы с помощью системного вызова `clone()`; отличия в скорости между ним и вызовом `fork()` показаны в табл. 28.3). Процедура создания потока является более быстрой, поскольку многие атрибуты вместо непосредственного копирования, как в случае с `fork()`, просто разделяются. В частности, отпадает потребность в дублировании страниц памяти (с помощью копирования при записи) и таблиц со страницами.

Помимо глобальной памяти, потоки также разделяют целый ряд других атрибутов (это когда атрибуты являются глобальными для всего процесса, а не для отдельных потоков). Среди них можно выделить атрибуты, перечисленные ниже.

- Идентификаторы процесса и его родителя.
- Идентификаторы группы процессов и сессии.
- Управляющий терминал.
- Учетные данные процесса (идентификаторы пользователя и группы).
- Дескрипторы открытых файлов.
- Блокировки записей, созданные с помощью вызова `fcntl()`.
- Действия сигналов.
- Информация, относящаяся к файловой системе: umask, текущий и корневой каталог.
- Интервальные таймеры (`setitimer()`) и POSIX-таймеры (`timer_create()`).
- Значения отмены семафоров (`semadj`) в System V.
- Ограничения на ресурсы.
- Потребленное процессорное время (полученное из `times()`).
- Потребленные ресурсы (полученные из `getrusage()`).
- Значение `nice` (установленное с помощью `setpriority()` и `nice()`).

Ниже перечислены атрибуты, которые являются уникальными для каждого отдельного потока:

- Идентификатор потока (см. раздел 29.5).
- Маска сигнала.
- Данные, относящиеся к определенному потоку (см. раздел 31.3).
- Альтернативный стек сигналов (`sigaltstack()`).
- Переменная `errno`.
- Настройки плавающей запятой (см. `env(3)`).
- Политика и приоритет планирования в режиме реального времени (см. разделы 35.2 и 35.3).
- Привязка к ЦПУ (относится только к Linux, описывается в разделе 35.4).
- Возможности (относится только к Linux, описывается в главе 39).
- Стек (локальные переменные и сведения о компоновке вызовов функций).

Как можно видеть на рис. 29.1, все стеки, относящиеся к отдельным потокам, находятся внутри одного и того же виртуального адресного пространства. Это означает, что потоки, имея подходящие указатели, могут обмениваться данными через стеки друг друга. Это бывает удобно, но требует осторожности при написании кода, чтобы уладить зависимость, вытекающую из того факта, что локальная переменная остается действительной только на время существования стека, в котором она находится (если функция возвращает значение, участок памяти, использовавшийся ее стеком, может быть повторно задействован во время последующего вызова функции; если поток завершается, участок

памяти, в котором находился его стек, формально становится доступным другому потоку). Неправильная работа с этой зависимостью может привести к ошибкам, которые будет сложно отследить.

29.2. Общие сведения о программном интерфейсе Pthreads

В конце 1980-х и начале 1990-х годов существовало несколько разных программных интерфейсов для работы с потоками. В 1995 году в стандарте POSIX.1 был описан API-интерфейс POSIX-потоков, который позже вошел в состав SUSv3.

Программный интерфейс Pthreads основывается на нескольких концепциях. Мы познакомимся с ними, подробно рассматривая его реализацию.

Типы данных в Pthreads

Программный интерфейс Pthreads определяет целый ряд типов данных, часть из которых перечислена в табл. 29.1. Большинство из них будет описано на следующих страницах.

Таблица 29.1. Типы данных в Pthreads

Тип данных	Описание
pthread_t	Идентификатор потока
pthread_mutex_t	Мьютекс
pthread_mutexattr_t	Объект с атрибутами мьютекса
pthread_cond_t	Условная переменная
pthread_condattr_t	Объект с атрибутами условной переменной
pthread_key_t	Ключ для данных, относящихся к определенному потоку
pthread_once_t	Одноразовый контекст управления инициализацией
pthread_attr_t	Объект с атрибутами потока

Стандарт SUSv3 не содержит подробностей о том, как именно должны быть представлены эти типы данных, поэтому переносимые приложения должны считать их непрозрачными. Это означает, что программа не должна зависеть от структуры или содержимого переменных любого из этих типов. В частности, мы не можем сравнивать такие переменные с помощью оператора `==`.

Потоки и переменная errno

В традиционном программном интерфейсе UNIX переменная `errno` является глобальной и целочисленной. Однако этого недостаточно для многопоточных программ. Если поток вызвал функцию, записавшую ошибку в глобальную переменную `errno`, это может ввести в заблуждение другие потоки, которые тоже вызывают функции и проверяют значение `errno`. Иными словами, результатом будет состояние гонки. Таким образом, в многопоточных программах каждый поток имеет свой отдельный экземпляр `errno`. В Linux (и в большинстве реализаций UNIX) для этого используется примерно один подход: `errno` объявляется в виде макроса, который разворачивается в вызов функции, возвращающий изменяемое значение вида `lvalue`, уникальное для

каждого отдельного потока (поскольку значение `lvalue` доступно для изменения, мы по-прежнему можем записывать в многопоточных программах операции присваивания вида `errno = value`).

Резюмируя вышесказанное: механизм `errno` был интегрирован в потоки так, что процедура оповещения об ошибках полностью соответствует традиционному подходу, применяющемуся в программных интерфейсах UNIX.

> Оригинальный стандарт POSIX.1 следовал рекомендациям Кернигана и Ритчи, позволяя программе объявлять переменную errno как extern int. В стандарте SUSv3 это запрещено (на самом деле данное изменение было внесено в 1995 году в стандарт POSIX.1c). Сейчас для объявления errno программа должна подключить заголовочный файл <errno.h>, который поддерживает многопоточную реализацию.

Значение, возвращаемое функциями в Pthreads

Традиционно системные вызовы и некоторые функции возвращают `0` в случае успеха и `-1`, если произошла ошибка. Для обозначения самой ошибки применяется переменная `errno`. Функции в программном интерфейсе Pthreads ведут себя иначе. В случае успеха так же возвращается `0`, но при ошибке используется положительное значение. Это одно из тех значений, которые можно присвоить переменной `errno` в традиционных системных вызовах UNIX.

Поскольку каждая ссылка на `errno` в многопоточной программе несет в себе накладные расходы, связанные с вызовом функции, наша программа не присваивает значение, возвращенное функцией из состава Pthreads, непосредственно этой переменной. Вместо этого мы используем промежуточную переменную и нашу диагностическую функцию `errExitEN()` (см. подраздел 3.5.2), как показано ниже:

```
pthread_t *thread;
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
errExitEN(s, "pthread_create");
```

Компиляция программ на основе Pthreads

В Linux программы, которые используют программный интерфейс Pthreads, должны компилироваться с параметром `cc -pthread`. Среди действий этого параметра можно выделить следующие.

- Объявляется макрос препроцессора `_REENTRANT`. Это открывает доступ к объявлениям нескольких реентерабельных (повторно входимых) функций.
- Программа компонуется с библиотекой libpthread (эквивалент параметра `-lpthread`).

> Конкретные параметры компиляции многопоточных программ варьируются в зависимости от реализации (и компилятора). В некоторых системах (таких как Tru64) тоже используется cc -pthread, хотя в Solaris и HP-UX применяется параметр cc -mt.

29.3. Создание потоков

Сразу после запуска программы итоговый процесс состоит из одного потока, который называют *исходным* или *главным*. В этом разделе вы узнаете, как создаются дополнительные потоки.

Для создания нового потока используется функция `pthread_create()`.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start)(void *), void *arg);
```

Возвращает `0` при успешном завершении
и положительное число, если произошла ошибка

Новый поток начинает выполнение с вызова функции, указанной в виде значения `start` и принимающей аргумент `arg` (то есть `start(arg)`). Поток, который вызвал `pthread_create()`, продолжает работу, выполняя инструкцию, следующую за данным вызовом (это соответствует поведению функции-обертки вокруг системного вызова `clone()` из состава библиотеки glibc, описанной в разделе 28.2).

Аргумент `arg` объявлен как `void *`. Это означает, что мы можем передать функции `start` указатель на объект любого типа. Обычно он указывает на переменную, находящуюся в глобальном пространстве или в куче, но мы также можем использовать значение `NULL`. Если нам нужно передать функции `start` несколько аргументов, мы можем предоставить в качестве `arg` указатель на структуру, содержащую эти аргументы в виде отдельных полей. Мы даже можем указать `arg` как целое число (`int`), воспользовавшись подходящим приведением типов.

Строго говоря, стандарты языка C не описывают результатов приведения int к void * и наоборот. Однако большинство компиляторов допускают эту операцию и генерируют предсказуемый результат — то есть int j == (int) ((void *) j).

Значение, возвращаемое функцией `start`, тоже имеет тип `void *` и может быть интерпретировано, как и аргумент `arg`. Ниже, при рассмотрении функции `pthread_join()`, вы узнаете, как используется это значение.

Во время приведения значения, возвращенного начальной функцией потока, к целому числу следует соблюдать осторожность. Дело в том, что значение PTHREAD_CANCELED, возвращаемое при отмене потока (см. главу 32), обычно реализуется в виде целого числа, приведенного к типу void *. Если начальная функция вернет это значение, другой поток, выполняющий pthread_join(), ошибочно воспримет это как уведомление об отмене потока. В приложениях, которые допускают отмену потоков и используют целые числа в качестве значений, возвращаемых из начальных функций, необходимо следить за тем, чтобы в потоках, завершающихся в штатном режиме, эти значения не совпадали с константой PTHREAD_CANCELED (чему бы она ни равнялась в текущей реализации Pthreads). Переносимые приложения должны делать то же самое, но с учетом всех реализаций, с которыми они могут работать.

Аргумент `thread` указывает на буфер типа `pthread_t`, в который перед возвращением функции `pthread_create()` записывает уникальный идентификатор созданного потока. С помощью этого идентификатора можно будет ссылаться на данный поток в дальнейших вызовах Pthreads.

В стандарте SUSv3 отдельно отмечается, что буфер, на который указывает thread, не нужно инициализировать до начала выполнения нового потока. То есть новый поток может начать работу до того, как вернется функция pthread_create(). Если новому потоку нужно получить свой собственный идентификатор, он должен использовать для этого функцию pthread_self() (описанную в разделе 29.5).

Аргумент attr является указателем на объект pthread_attr_t, содержащий различные атрибуты нового потока (к которым мы еще вернемся в разделе 29.8). Если присвоить attr значение NULL, поток будет создан с атрибутами по умолчанию, — именно это мы и будем делать в большинстве примеров в этой книге.

Программа не знает, какому потоку планировщик выделит процессорное время после вызова pthread_create() (в многопроцессорных системах оба потока могут выполняться одновременно на разных ЦПУ). Программы, которые явно полагаются на определенный порядок планирования, подвержены тем же видам состояния гонки, что были описаны в разделе 24.4. Если нам необходимо гарантировать тот или иной порядок выполнения, мы должны использовать один из методов синхронизации, рассмотренных в главе 30.

29.4. Завершение потоков

Выполнение потока прекращается по одной из следующих причин.

- Начальная функция выполняет инструкцию return, указывая возвращаемое значение для потока.
- Поток вызывает функцию pthread_exit() (описанную ниже).
- Поток отменяется с помощью функции pthread_cancel() (описанной в разделе 32.1).
- Любой из потоков вызывает exit() или главный поток выполняет инструкцию return (внутри функции main()), что приводит к немедленному завершению всех потоков в процессе.

Функция pthread_exit() завершает вызывающий поток и указывает возвращаемое значение, которое может быть получено из другого потока с помощью функции pthread_join().

```
include <pthread.h>

void pthread_exit(void *retval);
```

Вызов pthread_exit() эквивалентен выполнению инструкции return внутри начальной функции потока, за исключением того, что pthread_exit() можно вызвать из любого кода, запущенного начальной функцией.

Аргумент retval хранит значение, возвращаемое потоком. Значение, на которое указывает retval, не должно находиться в стеке самого потока, поскольку по окончании вызова pthread_exit() его содержимое становится неопределенным (этот участок виртуальной памяти процесса может быть сразу же выделен под стек для нового потока). То же самое касается и значения, передаваемого с помощью инструкции return в начальной функции.

Если главный поток вызовет pthread_exit() вместо exit() или инструкции return, остальные потоки продолжат выполнение.

29.5. Идентификаторы потоков

Каждый поток внутри процесса имеет свой уникальный идентификатор. Он возвращается в вызывающий поток функцией pthread_create(). Кроме того, с помощью функции pthread_self() поток может получить собственный идентификатор.

```
include <pthread.h>

pthread_t pthread_self(void);
```

Возвращает идентификатор вызывающего потока

Идентификаторы потоков внутри приложения можно использовать следующим образом.

- Задействуются различными функциями из состава Pthreads для определения того, в каком потоке они выполняются. В качестве примера можно привести функции `pthread_join()`, `pthread_detach()`, `pthread_cancel()` и `pthread_kill()`; все они описаны в этой и последующих главах.
- В некоторых приложениях может иметь смысл маркировать динамические структуры данных идентификатором определенного потока. Так мы можем определить создателя и «владельца» структуры; это также позволяет определить поток, который должен выполнить какие-то последующие действия со структурой данных.

Функция `pthread_equal()` позволяет проверить на тождественность два идентификатора потоков.

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Возвращает ненулевое значение, если *t1* и *t2* равны, в противном случае возвращает `0`

Например, чтобы проверить, совпадает ли идентификатор вызывающего потока со значением, сохраненным в переменной `tid`, можно написать следующий код:

```
if (pthread_equal(tid, pthread_self())
    printf("tid matches self\n");
```

Необходимость в функции `pthread_equal()` возникает из-за того, что тип данных `pthread_t` должен восприниматься как непрозрачный. В Linux он имеет тип `unsigned long`, но в других системах он может быть указателем или структурой.

> В библиотеке NPTL pthread_t в самом деле является указателем, который приводится к типу unsigned long.

Стандарт SUSv3 не требует, чтобы тип `pthread_t` был скалярным. Это может быть структура. Таким образом, код для вывода идентификатора потока, представленный выше, не является переносимым (хотя он работает во многих системах, включая Linux, и может пригодиться во время отладки):

```
pthread_t thr;

printf("Thread ID = %ld\n", (long) thr);   /* Неправильно! */
```

В Linux идентификаторы потоков являются уникальными для всех процессов. Однако в других системах это может быть не так. В стандарте SUSv3 отдельно отмечается, что переносимые приложения не могут полагаться на эти идентификаторы для определения потоков в других процессах. Там же указывается, что поточные библиотеки могут повторно исполь-

зовать эти идентификаторы после присоединения завершенного потока с помощью функции `pthread_join()` или после завершения отсоединенного потока (функция `pthread_join()` будет рассмотрена в следующем разделе, а отсоединенные потоки — в разделе 29.7).

> Идентификаторы POSIX- и обычных потоков, возвращаемые системным вызовом gettid() (доступным только в Linux), — это не одно и то же. Идентификатор POSIX-потока присваивается и обслуживается реализацией поточной библиотеки. Идентификатор обычного потока возвращается вызовом gettid() и представляет собой число (похожее на идентификатор процесса), которое назначается ядром. И хотя в реализации библиотеки NPTL используются уникальные идентификаторы потоков, выданные ядром, приложениям зачастую не нужно о них знать (к тому же работа с ними лишает приложения возможности переносимости между разными системами).

29.6. Присоединение к завершенному потоку

Функция `pthread_join()` ждет завершения потока, обозначенного аргументом `thread` (если поток уже завершился, она сразу же возвращается). Эта операция называется *присоединением*.

```
include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Возвращает `0` при успешном завершении или положительное число, если случилась ошибка

Если аргумент `retval` является ненулевым указателем, функция получает копию возвращаемого значения завершенного потока, то есть значение, указанное при выполнении потоком инструкции `return` или вызове `pthread_exit()`.

Вызов функции `pthread_join()` для идентификатора уже присоединенного потока может привести к непредсказуемым последствиям; например, мы можем присоединиться к потоку, который был создан позже и повторно использует тот же идентификатор.

Если поток не отсоединен (см. раздел 29.7), мы должны присоединиться к нему с помощью функции `pthread_join()`. Если нам не удастся это сделать, завершенный поток превратится в аналог процесса-«зомби» (см. раздел 26.2). Помимо пустой траты ресурсов, это может привести к тому, что мы больше не сможем создавать новые потоки (в случае если накопится достаточное количество потоков-«зомби»).

Процедура, которую выполняет функция `pthread_join()` для потоков, похожа на действие вызова `waitpid()` в контексте процессов. Но между ними есть и заметные различия.

- Потоки не имеют иерархии. Любой поток в процессе может воспользоваться функцией `pthread_join()` для присоединения к другому потоку в том же процессе. Представьте, к примеру, что поток А создал поток Б, который позже создал поток В; в этом случае поток А может присоединиться к потоку В и наоборот. Это отличается от иерархических отношений между процессами. Если родительский процесс создал потомка с помощью `fork()`, он и только он может ожидать этого потомка, используя вызов `wait()`. Между потоком, который вызывает функцию `pthread_create()`, и потоком, который при этом создается, нет таких отношений.
- Невозможно присоединиться к «любому потоку» (в случае с процессами мы можем это сделать с помощью вызова `waitpid(-1, &status, options)`); также не

существует неблокирующей операции присоединения (аналога вызова `waitpid()` с флагом `WNOHANG`). Похожих результатов можно добиться с помощью условных переменных; пример этого будет показан в подразделе 30.2.4.

Ограничение, связанное с тем, что функция pthread_join() может присоединять потоки только при наличии конкретного идентификатора, было создано умышленно. Идея заключается в том, что программа должна присоединяться только к тем потокам, о которых она «знает». Проблема операции «присоединения к произвольному потоку» произрастает из того факта, что потоки не имеют иерархии, поэтому таким образом мы действительно могли бы присоединиться к любому потоку, в том числе и к приватному, созданному библиотечной функцией (применение условных переменных, описанное в подразделе 30.2.4, позволяет присоединяться только к известным потокам). В результате библиотека не смогла бы больше присоединиться к этому потоку, чтобы получить его статус, а попытки присоединения к уже присоединенному потоку привели бы к ошибкам. Иными словами, операция «присоединения к произвольному потоку» несовместима с модульной архитектурой приложения.

Пример программы

Программа, показанная в листинге 29.1, создает новый поток и присоединяется к нему.

Листинг 29.1. Простая программа, использующая библиотеку Pthreads

threads/simple_thread.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}

int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

threads/simple_thread.c

Запустив эту программу, мы увидим следующее:

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

Порядок вывода первых двух строчек зависит от того, как планировщик распорядится двумя потоками.

29.7. Отсоединение потока

По умолчанию потоки являются *присоединяемыми*; это означает, что после завершения их статус можно получить из другого потока с помощью функции `pthread_join()`. Иногда статус, возвращаемый потоком, не имеет значения; нам просто нужно, чтобы система автоматически освободила ресурсы и удалила поток, когда тот завершится. В этом случае мы можем пометить поток как *отсоединенный*, воспользовавшись функцией `pthread_detach()` и указав идентификатор потока в аргументе `thread`.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Возвращает `0` при успешном завершении или положительное число, если возникла ошибка

В качестве примера использования функции `pthread_detach()` можно привести следующий вызов, в котором поток отсоединяет сам себя:

```
pthread_detach(pthread_self());
```

Если поток уже был отсоединен, мы больше не можем получить его возвращаемый статус с помощью функции `pthread_join()`. Мы также не можем снова сделать его присоединяемым.

Отсоединенный поток не становится устойчивым к вызову `exit()`, сделанному в другом потоке, или к инструкции `return`, выполненной в главной программе. В любой из этих ситуаций все потоки внутри процесса немедленно завершаются, вне зависимости от того, присоединяемые они или нет. Иными словами, функция `pthread_detach()` просто отвечает за поведение потока после его завершения, но не за то, в каких обстоятельствах тот завершается.

29.8. Атрибуты потоков

Ранее уже упоминалось, что аргумент `attr` функции `pthread_create()`, имеющий тип `pthread_attr_t`, может быть использован для задания атрибутов, которые применяются при создании нового потока. Мы не будем углубляться в рассмотрение этих атрибутов (подробности о них ищите в ссылках, перечисленных в конце главы) или изучать прототипы различных функций из состава Pthreads, которые позволяют работать с объектом `pthread_attr_t`. Мы просто отметим, что данные атрибуты содержат такие сведения, как местоположение и размер стека потока, политику его планирования и приоритет (это похоже на политики планирования в режиме реального времени и приоритеты процессов,

описанные в разделах 35.2 и 35.3), а также информацию о том, является ли поток присоединяемым или отсоединенным.

Пример использования этих атрибутов приводится в листинге 29.2, где создается поток, являющийся отсоединенным на момент своего появления (а не в результате последующего вызова `pthread_detach()`). В самом начале этот код инициализирует структуру с атрибутами с помощью значений по умолчанию, потом устанавливает атрибуты, необходимые для создания отсоединенного потока, и затем создает с помощью этой структуры новый поток. По окончании процедуры создания объект с атрибутами удаляется за ненадобностью.

Листинг 29.2. Создание потока с «отсоединяющим» атрибутом

Из файла **threads/detached_attrib.c**

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *x)
{
    return x;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    pthread_attr_t attr;
    int s;

    s = pthread_attr_init(&attr);    /* Присваиваем значения по умолчанию */
     if (s != 0)
        errExitEN(s, "pthread_attr_init");

    s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (s != 0)
        errExitEN(s, "pthread_attr_setdetachstate");

    s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_attr_destroy(&attr);    /* Больше не нужен */
    if (s != 0)
        errExitEN(s, "pthread_attr_destroy");

    s = pthread_join(thr, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join failed as expected");
    exit(EXIT_SUCCESS);
}
```

Из файла **threads/detached_attrib.c**

29.9. Сравнение потоков и процессов

В этом разделе мы кратко рассмотрим несколько факторов, которые следует учитывать при выборе между потоками и процессами в качестве основы для своего приложения. Начнем с обсуждения преимуществ многопоточного подхода.

- Обмениваться данными между потоками просто. Такой же обмен данными, но между процессами требует больше затрат (например, создания общего сегмента памяти или использование конвейера).
- Создание потока занимает меньше времени, чем создание процесса; потоки также могут выигрывать в плане скорости переключения контекста.

  Однако потоки имеют определенные недостатки по сравнению с процессами.
- Создавая программу на основе потоков, мы должны следить за тем, чтобы используемые функции были потокобезопасными или вызывались совместимым с потоками способом (понятие потоковой безопасности будет описано в разделе 31.1). Многопроцессные приложения могут об этом не заботиться.
- Ошибка в одном потоке (например, изменение памяти через неправильный указатель) может повредить остальные потоки в процессе, поскольку все они используют общее адресное пространство и некоторые другие атрибуты. Для сравнения, процессы являются более изолированными друг от друга.
- Каждый поток соперничает за возможность использования конечного адресного пространства своего процесса. В частности, стек и локальное хранилище каждого потока потребляет часть виртуальной памяти процесса, что делает ее недоступной для других потоков. И хотя адресное пространство имеет довольно большой объем (например, 3 Гбайт для архитектуры x86-32), этот фактор может стать существенным ограничением, сдерживающим процесс от создания большого количества потоков или потоков, требующих много памяти. Отдельные процессы, с другой стороны, могут задействовать весь диапазон свободной виртуальной памяти, ограниченный лишь размерами RAM и пространства подкачки.

Ниже приведены дополнительные моменты, которые могут повлиять на выбор между потоками и процессами.

- Работа с сигналами в многопоточном приложении требует тщательного проектирования (как правило, рекомендуется избегать использования сигналов в многопоточных программах). Больше о потоках и сигналах будет рассказано в разделе 33.2.
- В многопоточном приложении все потоки должны выполнять одну и ту же программу (возможно, задействуя разные ее функции). В многопроцессных приложениях разные процессы способны выполнять разные программы.
- Помимо самих данных, потоки разделяют и другую информацию (например, файловые дескрипторы, действия сигналов, текущий каталог, идентификаторы (имена) пользователя и группы). Это может быть как преимуществом, так и недостатком, в зависимости от приложения.

29.10. Резюме

В многопоточных процессах в одной и той же программе одновременно выполняются разные потоки. Все они имеют общие глобальные переменные и кучу, но у каждого из них есть свой отдельный стек для локальных переменных. Разные потоки одного и того же процесса также разделяют целый ряд атрибутов, включая идентификатор процесса, дескриптор открытых файлов, действия сигналов, текущего каталога и ограничения на ресурсы.

Ключевой особенностью потоков является более простой обмен информацией по сравнению с процессами; по этой причине некоторые программные архитектуры лучше ложатся на многопоточный подход, нежели на многопроцессный. Кроме того, в некоторых ситуациях потоки могут демонстрировать лучшую производительность (например,

поток создается быстрее, чем процесс), однако этот фактор обычно является вторичным при выборе между потоками и процессами.

Потоки создаются с помощью функции `pthread_create()`. Любой поток может завершиться независимо от других, используя функцию `pthread_exit()` (если вызвать `exit()` в любом из потоков, все они будут немедленно завершены). Если поток не был помечен как отсоединенный (например, с помощью вызова `pthread_detach()`), он должен быть присоединен другим потоком посредством функции `pthread_join()`, которая возвращает код завершения присоединенного потока.

Дополнительная информация

[Butenhof, 1996] содержит понятное и в то же время глубокое описание библиотеки Pthreads. Эта библиотека также хорошо освещена в [Robbins & Robbins, 2003]. [Tanenbaum, 2007] предоставляет теоретическое введение в понятие потоков, рассматривая такие темы, как мьютексы, критические регионы, условные переменные, а также обнаружение и обход взаимного блокирования. [Vahalia, 1996] содержит общие сведения о реализации потоков.

29.11. Упражнения

29.1. К каким возможным последствиям может привести выполнение потоком следующего кода?

```
pthread_join(pthread_self(), NULL);
```

Напишите программу, чтобы увидеть, что же в действительности происходит в системе Linux. Представьте, что у нас есть переменная `tid`, содержащая идентификатор потока; каким образом поток может избежать выполнения вызова `pthread_join(tid, NULL)`, являющегося эквивалентом вышеприведенной операции?

29.2. Что не так со следующей программой, если не брать во внимание отсутствие проверок на ошибки и объявлений различных переменных и структур?

```
static void *
threadFunc(void *arg)
{
    struct someStruct *pbuf = (struct someStruct *) arg;

    /* Выполняем какие-то действия со структурой, на которую указывает 'pbuf' */
}

int
main(int argc, char *argv[])
{
    struct someStruct buf;

    pthread_create(&thr, NULL, threadFunc, (void *) &buf);
    pthread_exit(NULL);
}
```

30 Потоки выполнения: синхронизация

В этой главе мы опишем два инструмента, с помощью которых потоки могут синхронизировать свои действия: мьютексы и условные переменные. Мьютексы позволяют потокам синхронизировать использование общих ресурсов, чтобы, к примеру, один поток не пытался получить доступ к разделяемой переменной в момент, когда другой поток изменяет ее значение. Условные переменные дополняют это решение, позволяя потокам оповещать друг друга о том, что разделяемая переменная (или другой общий ресурс) изменила свое состояние.

30.1. Защита доступа к разделяемым переменным: мьютексы

Одно из принципиальных преимуществ потоков заключается в том, что они могут делиться информацией посредством глобальных переменных. Но у этого простого механизма есть и обратная сторона: мы должны следить за тем, чтобы одну и ту же переменную одновременно не пытались изменить сразу несколько потоков или чтобы один поток не пытался прочитать ее содержимое, пока другой поток его обновляет. Термин *«критический участок»* относится к участку кода, который работает с общими ресурсами и чье выполнение должно быть атомарным. То есть выполнение не должно быть прервано другим потоком, который в этот самый момент получает доступ к тому же ресурсу.

В листинге 30.1 показаны проблемы, которые могут возникнуть при неатомарном доступе к разделяемым ресурсам. Эта программа создает два потока, каждый из которых выполняет одну и ту же функцию. Функция содержит цикл, который постепенно инкрементирует глобальную переменную `glob`. Для этого ее значение копируется в локальную переменную `loc`, там же инкрементируется и копируется обратно в `glob` (поскольку переменная `loc` автоматически попадает в локальный стек потока, каждый поток имеет ее уникальную копию). Количество итераций цикла определяется аргументом командной строки, который передается программе, или значением по умолчанию, если такого аргумента не обнаружено.

Листинг 30.1. Некорректная инкрементация глобальной переменной из двух потоков

threads/thread_incr.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;

static void *            /* 'arg' раз инкрементируем 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;
```

```
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

threads/thread_incr.c

Если запустить программу из листинга 30.1, указав, что каждый поток должен инкрементировать переменную 1000 раз, все должно выглядеть нормально:

```
$ ./thread_incr 1000
glob = 2000
```

Что же произошло? Первый поток, скорее всего, успел завершить всю свою работу до того, как стартовал второй поток. Если существенно увеличить объем работы, мы увидим совсем другой результат:

```
$ ./thread_incr 10000000
glob = 16517656
```

В конце этой последовательности переменная `glob` должна быть равна 20 миллионам. Проблема здесь кроется в способе выполнения потоков (см. также рис. 30.1).

1. Поток 1 копирует текущее значение `glob` в свою локальную переменную `loc`. Будем считать, что это значение равно 2000.
2. Временной отрезок, отведенный планировщиком для потока 1, исчерпывается, после чего начинает выполнение поток 2.
3. Поток 2 выполняет множество итераций, в которых он копирует текущее значение `glob` в свою локальную переменную `loc`, инкрементирует эту переменную и передает

результат обратно в `glob`. В первой из этих итераций значение, полученное из `glob`, будет равно 2000. Представим, что на момент исчерпания времени, отведенного потоку 2, значение `glob` увеличилось до 3000.

4. Поток 1 получает еще один временной отрезок и продолжает выполнение с того места, на котором он был прерван. Имея в переменной `loc` ранее скопированное значение `glob` (шаг 1), равное 2000, он теперь инкрементирует `loc` и передает результат (2001) обратно в `glob`. На этом этапе результат инкрементации значения потоком 2 теряется.

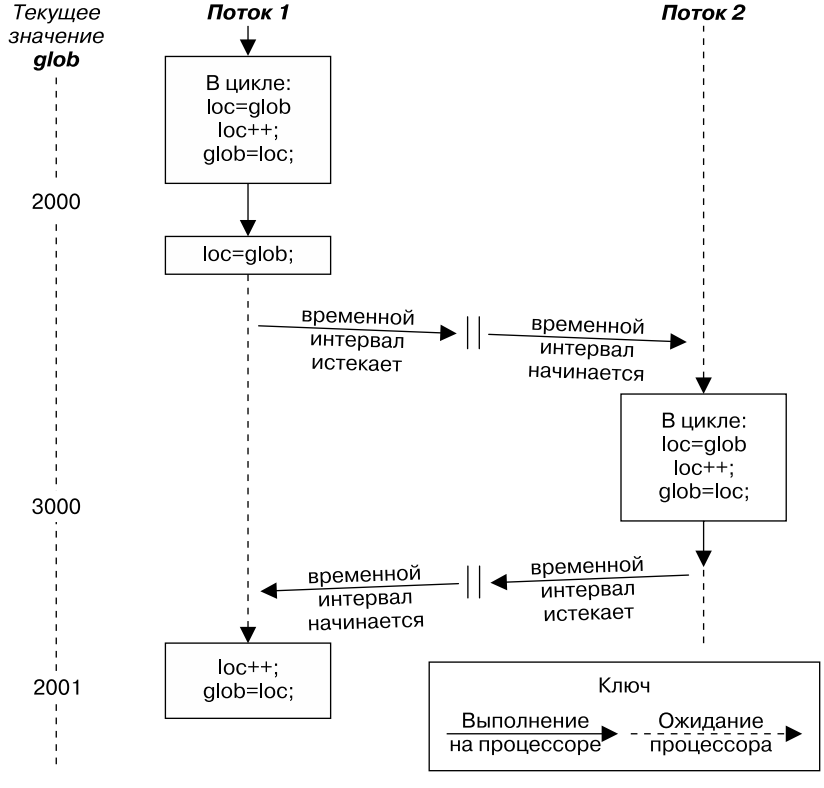


Рис. 30.1. Два потока, инкрементирующих глобальную переменную без синхронизации

Если несколько раз запустить программу из листинга 30.1 с одним и тем же аргументом командной строки, можно увидеть, что выводимое значение `glob` существенно варьируется:

```
$ ./thread_incr 10000000
glob = 10880429
$ ./thread_incr 10000000
glob = 13493953
```

Такое непредсказуемое поведение является результатом причудливой работы планировщика ядра. В сложных программах подобные ошибки возникают редко и их бывает сложно воспроизвести — а значит, сложно найти.

На первый взгляд может показаться, что проблема легко решается путем замены трех инструкций функции `threadFunc()` внутри цикла `for` одним-единственным выражением:

```
glob++;         /* или ++glob; */
```

Однако во многих программных архитектурах (таких как RISC) компилятору все равно бы пришлось преобразовать это выражение в команды машинного кода, которые являются эквивалентом трех инструкций из цикла в функции `threadFunc()`. Иными словами, несмотря на внешнюю простоту, даже простая операция инкремента может оказаться неатомарной и демонстрировать поведение, описанное выше.

Чтобы избежать проблем, которые возникают при попытке обновления разделяемой переменной из разных потоков, следует использовать мьютекс (от *mutual exclusion* — «взаимное исключение»); это позволит гарантировать, что только один поток имеет доступ к переменной в определенный промежуток времени. В целом мьютексы можно использовать для обеспечения атомарного доступа к любым разделяемым ресурсам, но чаще всего они применяются для защиты общих переменных.

Мьютекс имеет два состояния: *закрытое* (блокированное) и *открытое* (разблокированное). В любой момент времени максимум один поток может удерживать мьютекс закрытым. Попытки закрыть уже закрытый мьютекс либо отклоняются, либо приводят к ошибке — в зависимости от того, как выполнялось закрытие. Когда поток закрывает мьютекс, он становится его владельцем. Только владелец мьютекса может его открыть. Это улучшает структуру кода, в котором используются мьютексы, а также позволяет проделывать с ними некоторые оптимизации. Благодаря концепции владения вместо *«закрыть»* и *«открыть»* иногда применяются термины *«приобрести»* и *«освободить»*.

В целом для каждого разделяемого ресурса (который может состоять из нескольких связанных между собой переменных) устанавливается отдельный мьютекс. Для доступа к такому ресурсу каждый поток использует такую последовательность действий:

- закрыть мьютекс для разделяемого ресурса;
- получить доступ к разделяемому ресурсу;
- открыть мьютекс.

Если сразу несколько потоков попытаются выполнить этот блок кода *(критический участок)*, только один из них сможет получить этот мьютекс (другие останутся заблокированными); это означает, что в заданный момент времени только один поток сможет войти в этот блок, как проиллюстрировано на рис. 30.2.

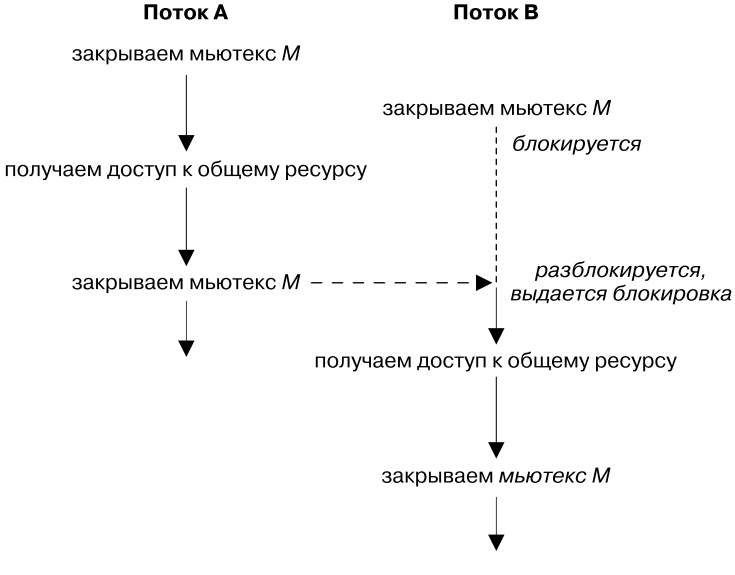


Рис. 30.2. Использование мьютекса для защиты критического участка

Наконец, стоит отметить, что закрытие мьютекса является скорее рекомендацией, а не требованием. Иными словами, поток может проигнорировать использование мьютекса и просто обратиться к соответствующим разделяемым переменным. Но, чтобы задействовать эти переменные безопасным образом, все потоки должны совместно пользоваться мьютексом, соблюдая правила закрытия, которые тот навязывает.

30.1.1. Статически выделяемые мьютексы

Мьютекс должен быть выделен в виде статической переменной или создан динамически во время выполнения программы (например, в блоке памяти выделенном с помощью вызова `malloc()`). Динамическое создание мьютексов является несколько более сложным, поэтому мы отложим его обсуждение до подраздела 30.1.5.

Мьютекс представляет собой переменную типа `pthread_mutex_t`. Прежде чем вы сможете его использовать, он должен быть инициализирован. В случае со статически выделенным мьютексом это можно сделать путем присваивания ему значения `PTHREAD_MUTEX_INITIALIZER`, как показано ниже:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

Согласно стандарту SUSv3 применение операций, которые будут описаны в оставшейся части этого раздела, к копии мьютекса приводит к неопределенным результатам. Эти операции всегда должны выполняться с оригиналом мьютекса, инициализированным статически, с помощью значения PTHREAD_MUTEX_INITIALIZER, или динамически, путем вызова pthread_mutex_init() (описанного в разделе 30.1.5).

30.1.2. Закрытие и открытие мьютекса

После инициализации мьютекс находится в открытом состоянии. Для его закрытия и открытия используются функции `pthread_mutex_lock()` и `pthread_mutex_unlock()`.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Обе возвращают `0` при успешном завершении
или положительное число, если случилась ошибка

Чтобы закрыть *мьютекс*, его нужно передать в вызов `pthread_mutex_lock()`. Если он в данный момент открыт, вызов сразу же возвращается. Если же мьютекс закрыт другим потоком, `pthread_mutex_lock()` блокируется, пока тот не откроется; в момент открытия данная функция снова блокирует мьютекс и возвращается.

Если вызывающий поток сам закрыл мьютекс, переданный в функцию `pthread_mutex_lock()`, тогда, если это стандартный тип мьютекса, тогда случится одно из двух, в зависимости от реализации: либо поток войдет в состояние взаимного блокирования, пытаясь закрыть мьютекс, которым он уже владеет, либо вызов завершится неудачей и вернет ошибку `EDEADLK`. В Linux по умолчанию происходит взаимное блокирование (другие возможные варианты будут описаны после рассмотрения разных типов мьютексов в подразделе 30.1.7).

Функция `pthread_mutex_unlock()` открывает мьютекс, закрытый ранее вызывающим потоком. Если мьютекс уже открыт или был закрыт другим потоком, данная функция возвращает ошибку.

Если мьютекс, открытый `pthread_mutex_unlock()`, ожидают сразу несколько потоков, получить его может любой из них.

Пример программы

Ниже представлена измененная версия листинга 30.1, которая использует мьютекс для защиты доступа к глобальной переменной `glob`. Запустив эту программу с тем же аргументом командной строки, что и раньше, мы увидим, что инкрементация значения `glob` всегда происходит предсказуемо:

```
$ ./thread_incr_mutex 10000000
glob = 20000000
```

Листинг 30.2. Использование мьютекса для защиты доступа к глобальной переменной
threads/thread_incr_mutex.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *        /* 'arg' раз инкрементируем 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
```

```
    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

threads/thread_incr_mutex.c

Функции pthread_mutex_trylock() и pthread_mutex_timedlock()

Программный интерфейс Pthreads предоставляет две разновидности функции `pthread_mutex_lock()`: `pthread_mutex_trylock()` и `pthread_mutex_timedlock()` (их прототипы ищите на соответствующих справочных страницах).

Функция `pthread_mutex_trylock()` делает то же самое, что и `pthread_mutex_lock()`, но если мьютекс закрыт, возвращает ошибку `EBUSY`.

Особенностью функции `pthread_mutex_timedlock()` является то, что она позволяет указать дополнительный аргумент `abstime`, который ограничивает время, отводящееся на получение мьютекса. Если это время истекло, а вызывающий поток все еще не стал владельцем мьютекса, функция `pthread_mutex_timedlock()` возвращает ошибку `ETIMEDOUT`.

Функции `pthread_mutex_trylock()` и `pthread_mutex_timedlock()` являются гораздо менее популярными, чем `pthread_mutex_lock()`. В большинстве хорошо спроектированных приложений поток должен удерживать мьютекс на протяжении короткого отрезка времени, чтобы не мешать параллельному выполнению других потоков. Благодаря этому другие потоки вскоре смогут сами открыть этот мьютекс. Поток, который периодически вызывает функцию `pthread_mutex_trylock()`, чтобы проверить, можно ли закрыть мьютекс, рискует стать наименее «удачливым» с точки зрения частоты получения доступа к ресурсу по сравнению с потоками, которые находятся в очереди и закрывают мьютекс с помощью функции `pthread_mutex_lock()`.

30.1.3. Производительность мьютексов

Какова цена использования мьютексов? Мы рассмотрели две разные версии одной программы, которая инкрементирует разделяемую переменную: одну с мьютексами (см. листинг 30.2), а другую — без (см. листинг 30.1). Запустив их на компьютере с архитектурой x86-32 под управлением Linux 2.6.31 (с NPTL), мы обнаружим, что для выполнения 10 миллионов итераций в каждом из потоков версии без мьютексов (которая выдает неправильный результат) понадобилось 0,35 секунды, а версии с мьютексами — 3,1 секунды.

На первый взгляд это может показаться расточительством. Но давайте рассмотрим главный цикл, выполняемый версией без мьютексов (см. листинг 30.1). В этой версии функция `threadFunc()` выполняет цикл `for`, который инкрементирует контролирующую его переменную, сравнивает ее значение с другой переменной, производит два присваивания и еще одну инкрементацию, после чего возвращается в начало цикла. Версия с мьютексами (см. листинг 30.2) выполняет те же шаги, но, кроме этого, закрывает и открывает мьютекс на каждой итерации. Иными словами, затраты на закрытие и открытие мьютекса примерно в десять раз меньше, чем затраты на операции, перечисленные выше в контексте первой программы. Это требует относительно небольших ресурсов. Более того, в типичной ситуации поток тратит намного больше времени на выполнение другой

работы и производит меньше закрытий и открытий, поэтому в большинстве приложений влияние мьютексов на производительность не является значительным.

Для сравнения, запуск некоторых простых тестовых программ в той же системе с 20 миллионами итераций, в которых происходит блокирование и разблокирование файлового участка с помощью вызова `fcntl()` (см. раздел 51.3), занимает 44 секунды, в то время как для такого же количества итераций с инкрементацией и декрементацией семафора из System V требуется 28 секунд. Проблема с файловыми блокировками и семафорами состоит в том, что для операций блокирования и разблокирования всегда требуется системный вызов, который неизменно влечет к небольшим, но заметным затратам (см. раздел 3.1). Мьютексы, с другой стороны, реализованы с помощью атомарных операций на машинном языке (выполняемых на участках памяти, видимых для всех потоков) и требуют выполнения системного вызова только в случае, если за установление блокировки конкурируют сразу несколько потоков.

В Linux мьютексы реализованы с применением фьютексов (акроним от fast user space mutexes — «быстрые мьютексы пользовательского пространства»), а конфликты при блокировании разрешаются с помощью системного вызова futex(). В этой книге мьютексы не описываются (они не предназначены для непосредственного применения в пространстве пользователя), но подробности о них можно найти в книге [Drepper, 2004 (a)], где также идет речь о реализации мьютексов на основе фьютексов. [Franke et al., 2002] — это (уже устаревший) документ, написанный разработчиками фьютексов, в котором содержится описание ранней реализации этой технологии и ее преимуществ в плане производительности.

30.1.4. Взаимное блокирование мьютексов

Иногда потоку нужно получить доступ сразу к двум или более разделяемым ресурсам, каждый из которых управляется отдельным мьютексом. Если один и тот же набор мьютексов закрывается несколькими потоками, может произойти взаимное блокирование. Пример такой ситуации показан на рис. 30.3, где каждый поток успешно закрывает по одному мьютексу, после чего пытает закрыть мьютекс, которым уже владеет другой поток. В итоге оба потока навсегда остаются заблокированными.

Поток А	**Поток Б**
1. pthread_mutex_lock(mutex1);	*1. pthread_mutex_lock(mutex2);*
2. pthread_mutex_lock(mutex2);	*2. pthread_mutex_lock(mutex1);*
блокируется	блокируется

Рис. 30.3. Взаимное блокирование в ситуации, когда два потока закрывают два мьютекса

Самый простой способ избежать подобного взаимного блокирования заключается в определении иерархии мьютексов. Если потоки могут закрыть один и тот же набор мьютексов, они всегда должны делать это в одном и том же порядке. Например, в сценарии, описанном на рис. 30.3, взаимного блокирования можно было бы избежать, если бы оба потока всегда закрывали сначала первый мьютекс, а потом второй. Иногда иерархия мьютексов является логически очевидной. Но если это не так, вы можете установить произвольный порядок, которому должны следовать все потоки.

Существует и другая, менее популярная стратегия, которую вкратце можно описать как «попробуй, а затем отойди». Согласно ей поток закрывает первый мьютекс с помощью функции `pthread_mutex_lock()`, после чего использует функцию `pthread_mutex_trylock()` для закрытия оставшихся мьютексов. Если хотя бы один вызов `pthread_mutex_trylock()` завершается ошибкой (`EBUSY`), поток освобождает все мьютексы и затем пытается повто-

рить те же действия — возможно, после некоторой задержки. Данный подход является менее эффективным по сравнению с иерархией, поскольку для него может потребоваться несколько итераций. С другой стороны, он может оказаться более гибким ввиду отсутствия жесткой иерархии мьютексов. Пример этой стратегии показан в [Butenhof, 1996].

30.1.5. Динамическая инициализация мьютексов

Статическое значение `PTHREAD_MUTEX_INITIALIZER` можно использовать только для инициализации мьютексов, статически выделенных атрибутами по умолчанию. Во всех других случаях следует применять динамическую инициализацию с помощью функции `pthread_mutex_init()`.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

Возвращает `0` при успешном выполнении или положительное число, если случилась ошибка

Аргумент `mutex` обозначает мьютекс, который должен быть инициализирован. Аргумент `attr` — это указатель на объект `pthread_mutexattr_t`, предварительно инициализированный подходящими аргументами (мы вернемся к атрибутам мьютекса в следующем разделе). Если указать аргументу `attr` значение `NULL`, мьютексу будут назначены различные атрибуты по умолчанию.

В стандарте SUSv3 сказано, что инициализация уже инициализированного мьютекса приводит к непредсказуемым последствиям; этого не следует делать.

Ниже представлены некоторые из случаев, когда вместо статического инициализатора нужно использовать функцию `pthread_mutex_init()`.

- Мьютекс был динамически выделен в куче. Представьте, к примеру, что мы создали динамически выделенный связный список структур и каждая из них содержит поле `pthread_mutex_t` с мьютексом, который защищает доступ к этой структуре.
- Мьютекс является автоматической переменной, выделенной в стеке.
- Мы хотим инициализировать статически выделенный мьютекс нестандартными атрибутами.

Когда мьютекс, выделенный автоматически или динамически, больше не нужен, его следует уничтожить с помощью функции `pthread_mutex_destroy()` (для мьютексов, инициализированных статически с использованием значения `PTHREAD_MUTEX_INITIALIZER`, ее можно не вызывать).

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Безопасное уничтожение мьютекса возможно, только когда он находится в открытом состоянии и нет ни одного потока, который бы впоследствии планировал его закрыть.

Если мьютекс находится на участке динамически выделяемой памяти, его нужно уничтожить до того, как он освободит этот участок. Автоматически выделенный мьютекс нужно уничтожать до возврата функции, в которой он выполняется.

Мьютекс, уничтоженный с помощью функции `pthread_mutex_destroy()`, в дальнейшем можно будет повторно инициализировать, используя функцию `pthread_mutex_init()`.

30.1.6. Атрибуты мьютексов

Как уже отмечалось ранее, аргумент `attr` функции `pthread_mutex_init()` можно применять для задания объекта `pthread_mutexattr_t`, который определяет атрибуты мьютекса. Различные функции из состава библиотеки Pthreads способны извлекать и записывать атрибуты, используя этот объект. Мы не станем вдаваться в подробности относительно всех атрибутов мьютекса или показывать прототипы функций, которые позволяют их инициализировать внутри объекта `pthread_mutexattr_t`. Но на одном атрибуте, описывающем тип мьютекса, мы все же остановимся.

30.1.7. Типы мьютексов

На предыдущих страницах приводится ряд утверждений о поведении мьютексов.

- Поток не может закрывать один и тот же мьютекс дважды.
- Поток не может открыть мьютекс, которым он не владеет (то есть мьютекс, который он не закрывал).
- Поток не может открыть мьютекс, который не является закрытым.

Что именно происходит в каждом из этих случаев, зависит от *типа* мьютекса. В стандарте SUSv3 определены следующие типы.

- `PTHREAD_MUTEX_NORMAL` — этот тип мьютексов не поддерживает обнаружение взаимного блокирования. Если поток попытается закрыть мьютекс, который он уже закрыл, его работа будет заблокирована. Открытие мьютекса, который не является закрытым или закрыт другим потоком, приводит к неопределенным результатам (в Linux обе эти операции с данным типом мьютекса заканчиваются успешно).
- `PTHREAD_MUTEX_ERRORCHECK` — проверка ошибок выполняется для всех операций. Все три сценария, перечисленные выше, приводят к возврату ошибок соответствующими функциями из состава Pthreads. Этот тип мьютексов обычно отличается пониженной производительностью, но может пригодиться во время отладки, позволяя найти те участки приложения, которые нарушают правила использования мьютексов.
- `PTHREAD_MUTEX_RECURSIVE` — рекурсивный мьютекс реализует концепцию счетчика закрытий. Когда поток впервые получает мьютекс, счетчику устанавливается значение 1. При каждой следующей операции закрытия тем же потоком счетчик инкрементируется, а при открытии — декрементируется. Мьютекс освобождается (то есть становится доступным для других потоков), только когда счетчик закрытий достигает значения 0. Открытие мьютекса, который не является закрытым или закрыт другим потоком, заканчивается неудачей.

В Linux реализация потоков поддерживает нестандартные статические инициализаторы для каждого из типов мьютексов, приведенных выше (например, `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`), поэтому для инициализации статически выделенных мьютексов не требуется вызывать функцию `pthread_mutex_init()`. Однако в переносимых приложениях следует воздержаться от использования этих инициализаторов.

Помимо типов, описанных выше, стандарт SUSv3 определяет дополнительный тип, `PTHREAD_MUTEX_DEFAULT`, который устанавливается по умолчанию в случае использования

инициализатора PTHREAD_MUTEX_INITIALIZER или если мы указали NULL для аргумента attr в функции pthread_mutex_init(). Поведение этих типов мьютексов намеренно оставлено неопределенным во всех трех сценариях, описанных в начале данного раздела; это обеспечивает максимальную гибкость при реализации эффективных мьютексов. В Linux поведение мьютексов PTHREAD_MUTEX_DEFAULT и PTHREAD_MUTEX_NORMAL является идентичным.

В листинге 30.3 показано, как задать тип мьютекса (в этом примере мы используем мьютекс *с проверкой ошибок*).

Листинг 30.3. Задание типа мьютекса

```
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
int s, type;

s = pthread_mutexattr_init(&mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_init");

s = pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_ERRORCHECK);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_settype");

s = pthread_mutex_init(mtx, &mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutex_init");

s = pthread_mutexattr_destroy(&mtxAttr); /* Больше не требуется */
if (s != 0)
    errExitEN(s, "pthread_mutexattr_destroy");
```

30.2. Оповещение об изменении состояния: условные переменные

Мьютекс предотвращает одновременный доступ к переменной из разных потоков. Условные переменные позволяют потокам информировать друг друга об изменениях в состоянии разделяемых переменных (или других общих ресурсов) и ждать (блокируясь) получения таких уведомлений.

Ниже показан простой пример без задействования условных переменных, который призван продемонстрировать их пользу. Представьте, что у вас есть набор потоков, которые генерируют некоторые «итоговые элементы», потребляемые главным потоком. Количество полученных элементов, ожидающих потребления, будет представлено с помощью переменной avail, защищенной мьютексом:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
```

Участки кода, представленные в данном разделе, можно найти в файле threads/prod_no_condvar.c, который входит в состав архива с исходным кодом, прилагающимся к этой книге.

Потоки, генерирующие элементы, будут состоять примерно из такого кода:

```
/* Код для генерирования элементов опущен */
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");
```

```
avail++; /* Уведомляем потребителя о готовности еще одного элемента */
s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");
```

А в главном потоке (потребителе) мы воспользуемся следующим кодом:

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail > 0) { /* Потребляем все доступные элементы */
        /* Делаем что-нибудь со сгенерированным элементом */
        avail--;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}
```

Код, показанный выше, работает, но при этом тратит впустую процессорное время, поскольку главный поток постоянно проверяет состояние переменной `avail`, используя цикл. Эту проблему можно устранить с помощью *условной переменной*. Это позволит потоку приостанавливать свою работу (ждать), пока другой поток не оповестит его (просигнализирует) о том, что ему нужно что-то сделать (то есть что возникло некое «условие», на которое он должен отреагировать).

Условные переменные всегда используются в сочетании с мьютексами. Мьютекс обеспечивает взаимоисключающий доступ к разделяемому ресурсу, тогда как условная переменная сигнализирует об изменении его состояния (это *сигнализирование* не имеет ничего общего с сигналами, описанными в главах 20–22; речь здесь скорее идет об *оповещении*).

30.2.1. Статически выделяемые условные переменные

По аналогии с мьютексами, уловные переменные могут выделяться статически и динамически. Здесь будут рассмотрены статически выделяемые переменные, а динамические мы рассмотрим в подразделе 30.2.5.

Условная переменная имеет тип `pthread_cond_t`. Как и в случае с мьютексом, перед использованием ее следует инициализировать. В случае со статически выделяемыми условными переменными это делается путем присваивания им значения `PTHREAD_COND_INITIALIZER`, как показано в следующем примере:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

> Согласно стандарту SUSv3 применение операций, которые будут описаны в оставшейся части этого раздела, к копии условной переменной приводит к неопределенным результатам. Эти операции всегда должны выполняться с оригиналом условной переменной, инициализированным статически, с помощью значения PTHREAD_COND_INITIALIZER, или динамически, путем вызова pthread_cond_init() (описанного в подразделе 30.2.5).

30.2.2. Оповещение и ожидание условных переменных

Основными операциями с условными переменными являются *оповещение* и *ожидание*. Первая дает понять одному или нескольким ожидающим потокам, что состояние разде-

ляемого ресурса изменилось. Операция ожидания позволяет блокировать поток до тех пор, пока не будет получено оповещение.

Оповещения передаются с помощью аргумента `cond` в функциях `pthread_cond_signal()` и `pthread_cond_broadcast()`. Функция `pthread_cond_wait()` блокирует поток, пока условная переменная `cond` не просигнализирует об изменении.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Все возвращают `0` при успешном завершении или положительное число, если произошла ошибка

Разница между функциями `pthread_cond_signal()` и `pthread_cond_broadcast()` заключается в том, что происходит, когда в вызове `pthread_cond_wait()` заблокировано несколько потоков. `pthread_cond_signal()` просто гарантирует, что по меньшей мере один поток возобновит свою работу; `pthread_cond_broadcast()` разблокирует все потоки сразу.

Использование `pthread_cond_broadcast()` всегда приводит к корректному результату (поскольку все потоки должны уметь справляться с лишними и ложными возобновлениями работы). Функция `pthread_cond_signal()` может оказаться более эффективной, но ее следует применять только в том случае, если на изменение состояния разделяемого ресурса должен реагировать только один поток, неважно, какой именно. Этот сценарий обычно относится к случаям, когда все ожидающие потоки спроектированы для выполнения одной и той же задачи. Исходя из этого, функция `pthread_cond_signal()` может демонстрировать более высокую производительность по сравнению с `pthread_cond_broadcast()`, поскольку позволяет избежать следующих ситуаций.

1. Все ожидающие потоки возобновили работу.
2. Один из потоков первым получает процессорное время. Он проверяет состояние разделяемой переменной (под защитой соответствующего мьютекса) и видит, что ему необходимо выполнить некие действия. Поток выполняет все, что от него требуется, изменяет состояние разделяемой переменной, чтобы уведомить о завершении работы, и открывает соответствующий мьютекс.
3. Оставшиеся потоки по очереди закрывают мьютекс и проверяют состояние разделяемой переменной. Однако из-за изменений, внесенных первым потоком, им больше нечего делать, поэтому они открывают мьютекс и возвращаются к состоянию ожидания (то есть снова вызывают `pthread_cond_wait()`).

В отличие от `pthread_cond_signal()`, функция `pthread_cond_broadcast()` рассчитана в том числе и на потоки, которые выполняют разные задачи (в этом случае они, скорее всего, имеют разные предикаты, связанные с условной переменной).

Условная переменная не содержит никакой информации о состоянии. Это всего лишь механизм обмена данными о состоянии приложения. Если в момент создания уведомления условную переменную не ожидает ни один поток, это уведомление теряется. Если позже какой-то поток начнет ждать условную переменную, он будет разблокирован только после создания нового уведомления.

Функция `pthread_cond_timedwait()` делает то же самое, что и `pthread_cond_wait()`, однако ее аргумент `abstime` позволяет ограничить время ожидания потоком уведомления от условной переменной.

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Возвращает 0 при успешном завершении или положительное число, если произошла ошибка

Аргумент `timespec` представляет собой структуру `timespec` (см. подраздел 23.4.2), которая обозначает абсолютное время, выраженное в секундах и наносекундах, прошедших с момента начала эры UNIX (см. раздел 10.1). Если временной интервал, указанный в `timespec`, истекает до отправки уведомления условной переменной, функция `pthread_cond_timedwait()` возвращает ошибку `ETIMEDOUT`.

Применение условной переменной в примере с производителем-потребителем

Откорректируем наш предыдущий пример с учетом использования условной переменной. Ее объявление, равно как и объявление глобальной переменной и связанного с ней мьютекса, выглядит следующим образом:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;


static int avail = 0;
```

Участки кода, представленные в данном разделе, можно найти в файле `threads/prod_condvar.c`, который входит в состав архива с исходным кодом, прилагающимся к этой книге.

Код потоков, генерирующих элементы, остается почти без изменений, если не считать появления вызова `pthread_cond_signal()`:

```
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

avail++; /* Уведомляем потребителя о готовности еще одного элемента */
s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

s = pthread_cond_signal(&cond); /* Возобновляем работу заблокированного потребителя */
if (s != 0)
    errExitEN(s, "pthread_cond_signal");
```

Прежде чем приступать к рассмотрению кода потребителя, следует подробней остановиться на функции `pthread_cond_wait()`. Ранее уже отмечалось, что у условной переменной всегда есть связанный с ней мьютекс. Оба эти объекта передаются качестве аргументов в функцию `pthread_cond_wait()`, которая выполняет следующие действия:

- открывает мьютекс, указанный в аргументе `mutex`;
- блокирует вызывающий поток до тех пор, пока другой поток не уведомит условную переменную `cond`;
- заново закрывает `mutex`.

Для выполнения этих шагов была создана функция `pthread_cond_wait()`, потому что обычно для доступа к разделяемой переменной приходится писать следующий код:

```
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

while (/* Проверяем, не находится ли разделяемая переменная в нужном нам состоянии */)
    pthread_cond_wait(&cond, &mtx);

    /* Теперь разделяемая переменная в нужном состоянии; выполняем какую-то работу */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");
```

Причину размещения вызова pthread_cond_wait() внутри цикла while, а не в инструкции if вы узнаете в следующем разделе.

В приведенном выше коде обе попытки доступа к разделяемой переменной должны быть защищены мьютексом (почему — было рассказано выше). Иными словами, мьютекс и условная переменная имеют естественную связь.

1. Поток закрывает мьютекс в ходе подготовки к проверке состояния разделяемой переменной.
2. Состояние разделяемой переменной проверено.
3. Если разделяемая переменная находится не в том состоянии, которое нам нужно, поток должен открыть мьютекс (чтобы другие потоки могли получить доступ к разделяемой переменной) до начала ожидания условной переменной.
4. Когда поток снова возобновляет работу в результате получения уведомления от условной переменной, мьютекс опять должен быть закрыт, потому что следующим шагом поток, как правило, пытается получить доступ к разделяемой переменной.

Функция `pthread_cond_wait()` автоматически выполняет операции открытия и закрытия мьютекса, необходимые в последних двух пунктах. В третьем пункте открытие мьютекса и блокирование посредством условной переменной происходит автоматически. Иными словами, пока поток, вызывающий `pthread_cond_wait()`, не будет заблокирован через условную переменную, ни один другой поток не сможет получить мьютекс и послать уведомление.

Между условной переменной и мьютексом существует естественная связь. Из этого наблюдения можно сделать такой вывод: все потоки, которые одновременно ожидают определенную условную переменную, должны указывать в своих вызовах pthread_cond_wait() (или pthread_cond_timedwait()) один и тот же мьютекс. В сущности, вызов pthread_cond_wait() на период своей работы динамически привязывает условную переменную к уникальному мьютексу. В стандарте SUSv3 отмечается, что результат использования более чем одного мьютекса в вызовах pthread_cond_wait(), конкурирующих за одну и ту же условную переменную, является неопределенным.

Собрав воедино вышеперечисленные детали, мы теперь можем отредактировать главный поток (потребитель) так, чтобы он использовал функцию `pthread_cond_wait()`:

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail == 0) { /* Ждем элементов, которые можно потребить */
        s = pthread_cond_wait(&cond, &mtx);
```

```
    if (s != 0)
        errExitEN(s, "pthread_cond_wait");
    }

    while (avail > 0) { /* Потребляем все доступные элементы */
        /* Делаем что-нибудь со сгенерированным элементом */
        avail--;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

    /* Возможно, выполняем другую работу, не требующую закрытия мьютекса */
}
```

Завершим этот подраздел последним наблюдением касательно использования функции `pthread_cond_wait()` (и `pthread_cond_timedwait()`). В коде потока, генерирующего элементы, делается вызов `pthread_mutex_unlock()` и затем `pthread_cond_signal()`; мы сначала открываем мьютекс, связанный с разделяемой переменной, после чего оповещаем соответствующую условную переменную. Эти два шага можно было бы поменять местами; стандарт SUSv3 позволяет выполнять их в произвольном порядке.

> В книге [Butenhof, 1996] отмечается, что в некоторых реализациях открытие мьютекса и последующее уведомление условной переменной может обеспечивать лучшую производительность, чем если бы эти действия осуществлялись в обратном порядке. Если мьютекс открывается только после уведомления условной переменной, поток, выполняющий pthread_cond_wait(), может возобновить работу в момент, когда мьютекс все еще закрыт, и, обнаружив это, сразу же вернуться к состоянию ожидания. Это приводит к двум лишним переключениям контекста. Некоторые реализации устраняют эту проблему, используя методику называемую трансформацией ожидания, которая позволяет переместить поток с оповещением из очереди ожидания условной переменной в очередь ожидания мьютекса, не переключая при этом контекст (если мьютекс закрыт).

30.2.3. Проверка предиката условной переменной

У каждой условной переменной есть свой предикат на основе одной или нескольких разделяемых переменных. Например, в коде из предыдущего раздела предикатом, связанным с аргументом `cond`, было выражение `(avail == 0)`. Этот участок кода демонстрирует общий принцип проектирования: вызов `pthread_cond_wait()` должен выполняться циклом `while`, а не инструкцией `if`. Дело в том, что при возврате из функции `pthread_cond_wait()` состояние предиката может быть произвольным; следовательно, мы должны сразу же его перепроверить и продолжить ожидание, если он не находится в нужном нам состоянии.

Предположения о состоянии предиката при возвращении из функции `pthread_cond_wait()` являются несостоятельными по следующим причинам.

- *Другие потоки могут возобновить работу раньше*. Вероятно, сразу несколько потоков ждали получения мьютекса, связанного с условной переменной. Даже если поток, отправивший мьютексу уведомление, установит предикат в нужное состояние, это не исключает того, что другой поток может первым закрыть мьютекс и изменить состояние связанных с ним разделяемых ресурсов и, как следствие, состояние самого предиката.
- *Написание кода со «свободными» предикатами может быть проще*. Иногда приложения легче проектировать на основе условных переменных, которые сигнализируют о *возможности*, а не об *определенности*. Иными словами, передача уведомления

условной переменной означала бы *возможное*, а не гарантированное наличие работы для потока. С помощью такого подхода мы можем оповещать условную переменную, исходя из нашей оценки состояния предиката; поток, получивший уведомление, может узнать, есть ли что-то, что можно сделать путем перепроверки предиката.

- *Могут возникнуть ложные срабатывания.* В некоторых реализациях поток, ожидающий условную переменную, может возобновить работу, даже если никакой другой поток не передал этой переменной уведомление. Такие ложные срабатывания являются (редким) последствием применения методик, необходимых для эффективной реализации многопоточности на некоторых многопроцессорных системах, и специально допускаются стандартом SUSv3.

30.2.4. Пример программы: подсоединение любого завершенного потока

Ранее отмечалось, что функция `pthread_join()` может быть задействована только для присоединения конкретных потоков. Она не позволяет присоединить *любой* завершающийся поток. Теперь будет показано, как обойти это ограничение с помощью условной переменной.

Программа из листинга 30.4 создает по одному потоку для каждого аргумента командной строки. Каждый поток ждет определенное количество секунд, указанное с помощью соответствующего аргумента, и затем завершает работу. Задавая временные интервалы, мы можем имитировать поток, который работает на протяжении какого-то времени.

Программа содержит набор глобальных переменных, в которые записывается информация обо всех создаваемых потоках. Каждому из них соответствует элемент в глобальном массиве `thread`, содержащем идентификатор потока (поле `tid`) и его текущее состояние (поле `state`). Поле `state` может принимать одно из следующих значений: `TS_ALIVE` (означает, что поток выполняется), `TS_TERMINATED` (означает, что поток завершил работу, но еще не был присоединен) и `TS_JOINED` (означает, что поток завершился и был присоединен).

Во время завершения каждый поток присваивает полю `state` в соответствующем элементе массива `thread` значение `TS_TERMINATED`, инкрементирует глобальный счетчик завершенных, но еще не присоединенных потоков (`numUnjoined`) и уведомляет условную переменную `threadDied`.

Главный поток содержит цикл, который постоянно ждет изменения условной переменной `threadDied`. Каждый раз, когда эта переменная получает уведомление, главный поток сканирует массив `thread` в поисках элементов с полем `state`, равным `TS_TERMINATED`. Для каждого потока с этим состоянием (который определяется с помощью поля `tid`) вызывается функция `pthread_join()`, после чего полю `state` присваивается значение `TS_JOINED`. Главный цикл завершается, когда все созданные потоки завершили работу — то есть когда глобальная переменная `numLive` равна 0.

Сессия выполнения программы из листинга 30.4 показана ниже:

```
$ ./thread_multijoin 1 1 2 3 3    Создаем пять потоков
Thread 0 terminating
Thread 1 terminating
Reaped thread 0 (numLive=4)
Reaped thread 1 (numLive=3)
Thread 2 terminating
Reaped thread 2 (numLive=2)
Thread 3 terminating
Thread 4 terminating
Reaped thread 3 (numLive=1)
Reaped thread 4 (numLive=0)
```

Наконец, стоит отметить, что, хоть в данном примере все потоки создаются присоединяемыми и при завершении работы сразу же утилизируются функцией `pthread_join()`, нам не обязательно использовать этот подход для отслеживания завершающихся потоков. Мы могли бы сделать потоки отсоединенными, избавиться от функции `pthread_join()` и просто записывать сведения о завершении потоков в массив `thread` (и соответствующие глобальные переменные).

Листинг 30.4. Главный поток, способный присоединить любой другой завершающийся поток
threads/thread_multijoin.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_cond_t threadDied = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t threadMutex = PTHREAD_MUTEX_INITIALIZER;
               /* Защищает все глобальные переменные, указанные ниже */

static int totThreads = 0;   /* Общее количество созданных потоков */
static int numLive = 0;      /* Общее количество созданных выполняемых
                                или уже завершенных потоков, которые еще
                                не были присоединены */
static int numUnjoined = 0;  /* Количество завершенных потоков, которые
                                еще не были присоединены */
enum tstate {                /* Состояния потоков */
    TS_ALIVE,                /* Поток выполняется */
    TS_TERMINATED,           /* Поток завершен, но еще не присоединен */
    TS_JOINED                /* Поток завершен и присоединен */
};

static struct {              /* Данные о каждом потоке */
    pthread_t tid;           /* Идентификатор этого потока */
    enum tstate state;       /* Состояние потока (константы TS_*, указанные выше */
    int sleepTime;           /* Количество секунд, оставшихся до завершения */
} *thread;

static void *                /* Начальная функция потока */
threadFunc(void *arg)
{
    int idx = (int) arg;
    int s;

    sleep(thread[idx].sleepTime);    /* Имитируем некую работу */
    printf("Thread %d terminating\n", idx);

    s = pthread_mutex_lock(&threadMutex);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    numUnjoined++;
    thread[idx].state = TS_TERMINATED;

    s = pthread_mutex_unlock(&threadMutex);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
    s = pthread_cond_signal(&threadDied);
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");
```

```
    return NULL;
}

int
main(int argc, char *argv[])
{
    int s, idx;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-secs...\n", argv[0]);

    thread = calloc(argc - 1, sizeof(*thread));
    if (thread == NULL)
        errExit("calloc");

    /* Создаем все потоки */

    for (idx = 0; idx < argc - 1; idx++) {
        thread[idx].sleepTime = getInt(argv[idx + 1], GN_NONNEG, NULL);
        thread[idx].state = TS_ALIVE;
        s = pthread_create(&thread[idx].tid, NULL,
            threadFunc, (void *) idx);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    totThreads = argc - 1;
    numLive = totThreads;

    /* Присоединяем завершенные потоки */

    while (numLive > 0) {
        s = pthread_mutex_lock(&threadMutex);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        while (numUnjoined == 0) {
            s = pthread_cond_wait(&threadDied, &threadMutex);
            if (s != 0)
                errExitEN(s, "pthread_cond_wait");
        }

        for (idx = 0; idx < totThreads; idx++) {
            if (thread[idx].state == TS_TERMINATED) {
                s = pthread_join(thread[idx].tid, NULL);
                if (s != 0)
                    errExitEN(s, "pthread_join");

                thread[idx].state = TS_JOINED;
                numLive--;
                numUnjoined--;

                printf("Reaped thread %d (numLive=%d)\n", idx, numLive);
            }
        }
        s = pthread_mutex_unlock(&threadMutex);
        if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
    }
```

```
    exit(EXIT_SUCCESS);
}
```

threads/thread_multijoin.c

30.2.5. Динамически выделяемые условные переменные

Для динамической инициализации условных переменных используется функция `pthread_cond_init()`. Обстоятельства ее применения аналогичны тем, при которых мы динамически инициализируем мьютексы с помощью функции `pthread_mutex_init()` (см. подраздел 30.1.5). Иными словами, `pthread_cond_init()` применяется для инициализации автоматически и динамически выделенных условных переменных, а также переменных, которые выделены статически, но требуют нестандартных атрибутов.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Аргумент `cond` определяет условную переменную, которую нужно инициализировать. Как и в случае с мьютексами, мы можем указать предварительно инициализированный аргумент `attr`, чтобы назначить условной переменной нужные атрибуты. Для инициализации атрибутов в объекте `pthread_condattr_t`, на который указывает `attr`, можно использовать разные функции из состава библиотеки Pthreads. Если аргумент `attr` равен `NULL`, условной переменной присваиваются атрибуты по умолчанию.

Стандарт SUSv3 гласит, что повторная инициализация условной переменной приводит к непредсказуемому проведению; этого не следует делать.

Когда автоматически или динамически выделенная условная переменная больше не нужна, ее необходимо удалить с помощью функции `pthread_cond_destroy()`. Однако если она была статически инициализирована с использованием значения `PTHREAD_COND_INITIALIZER`, этого можно не делать.

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Уничтожение условной переменной является безопасным только в том случае, если ее не ожидает ни один из потоков. Если условная переменная находится на участке динамически выделяемой памяти, ее следует уничтожить до освобождения этого участка. Автоматически выделенную условную переменную нужно уничтожать до возврата функции, в которой она выполняется.

Условную переменную, уничтоженную с помощью функции `pthread_cond_destroy()`, в дальнейшем можно будет повторно инициализировать, используя функцию `pthread_cond_init()`.

30.3. Резюме

Улучшенный обмен данных, который обеспечивают потоки, имеет свою цену. Для координированного доступа к общим ресурсам многопоточные приложения должны использовать инструменты синхронизации, такие как мьютексы и условные переменные. Мьютекс обеспечивает эксклюзивный доступ к разделяемому ресурсу. Условная переменная позволяет одному или нескольким потокам ожидать уведомление о том, что какой-то другой поток изменил состояние разделяемой переменной.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 29.10.

30.4. Упражнения

30.1. Измените программу из листинга 30.1 (`thread_incr.c`) так, чтобы каждая итерация цикла в начальной функции потока выводила текущее значение переменной `glob` и какой-либо идентификатор, уникальный для текущего потока. Уникальный идентификатор можно указать в виде аргумента функции `pthread_create()`, с помощью которой создавался поток. В данном примере это потребует изменения аргумента начальной функции — это должен быть указатель на структуру, содержащую уникальный идентификатор и граничное значение цикла. Запустите программу, перенаправив ее вывод в файл; изучите полученный результат, чтобы понять, что происходит с переменной `glob`, когда планировщик ядра распределяет выполнение между двумя потоками.

30.2. Реализуйте набор потокобезопасных функций, которые позволяют искать внутри несбалансированного двоичного дерева и обновлять его содержимое. Итоговая библиотека должна включать в себя функции следующего вида (назначение понятно из их названий):

```
initialize(tree);
add(tree, char *key, void *value);
delete(tree, char *key)
Boolean lookup(char *key, void **value)
```

В приведенных выше прототипах `tree` является структурой, которая указывает на корень дерева (вам нужно будет определить ее должным образом). Каждый элемент дерева содержит пару «ключ-значение». Кроме того, структура каждого элемента должна предусматривать мьютекс, который защищает этот элемент, позволяя использовать его только одному потоку в заданный момент времени. Реализация функций `initialize()`, `add()` и `lookup()` не должна вызвать затруднений. Операция `delete()` потребует немного больше усилий.

Избавившись от необходимости обслуживать сбалансированное дерево, мы существенно упрощаем требования к реализации блокирования, но рискуем ухудшить производительность дерева, если входящие данные соответствуют определенному шаблону. Обслуживание сбалансированного дерева подразумевает перемещение узлов между поддеревьями во время операций add() и delete(), что влечет за собой использование куда более сложных стратегий блокирования.

31 Потоки выполнения: потоковая безопасность и локальное хранилище

В этой главе обсуждение программного интерфейса POSIX-потоков будет дополнено описанием потокобезопасных функций и единовременной инициализации. Мы также рассмотрим, как с помощью локального хранилища потока сделать существующие функции потокобезопасными, не изменяя их интерфейсов.

31.1. Потоковая безопасность (и новый взгляд на реентерабельность)

Функция считается *потокобезопасной*, если она может быть безопасно вызвана сразу из двух потоков. К этому можно подойти и с обратной стороны: если функция не является потокобезопасной, мы не можем вызывать ее из одного потока, пока она выполняется в другом. Следующую функцию (похожую на код из листинга 30.1), к примеру, нельзя назвать потокобезопасной:

```
static int glob = 0;

static void
incr(int loops)
{
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
}
```

Если несколько потоков одновременно вызовут данную функцию, мы не сможем предсказать итоговое значение переменной `glob`. Здесь мы видим типичный признак того, что функция не является потокобезопасной: она использует глобальные или статические переменные, которые разделяются всеми потоками.

Сделать функцию потокобезопасной можно различными способами. Например, к ней (или ко всем функциям в библиотеке, если в них применяются одни и те же глобальные переменные) можно привязать мьютекс, который будет закрываться при ее вызове и открываться при завершении. Преимущество данного подхода заключается в его простоте. С другой стороны, это означает, что в любой момент времени функцию может выполнять только один поток — в этом случае говорят, что доступ к функции *сериализованный*. Если на выполнение этой функции уходит значительное количество времени, сериализация приводит к невозможности одновременной работы, поскольку потоки программы больше не могут выполняться параллельно.

Более сложным решением является связывание мьютекса с разделяемой переменной. После этого нужно определить, какие участки функции являются критическими (в ко-

торых происходит доступ к разделяемой переменной), и затем закрывать/открывать мьютекс только при их выполнении. Это позволяет нескольким потокам одновременно выполнять одну и ту же функцию, действуя параллельно, за исключением тех моментов, когда разные потоки пытаются выполнить критический участок.

Функции, небезопасные для параллельного выполнения

Для облегчения разработки многопоточных приложений все функции стандарта SUSv3 (кроме тех, что перечислены в табл. 31.1) должны быть реализованы потокобезопасными (многие из этих функций не рассматриваются в данной книге).

Помимо исключений, представленных в табл. 31.1, стандарт SUSv3 имеет следующие уточнения.

- Функции `ctermid()` и `tmpnam()` могут не быть потокобезопасными, если им передается значение `NULL`.
- Функции `wcrtomb()` и `wcsrtombs()` могут не быть потокобезопасными, если их последний аргумент (`ps`) равен `NULL`.

Стандарт SUSv4 вносит такие изменения в табл. 31.1.

- Удалены функции `ecvt()`, `fcvt()`, `gcvt()`, `gethostbyname()` и `gethostbyaddr()`, поскольку они не попали в эту версию стандарта.
- Добавлены функции `strsignal()` и `system()`. Последняя не является реентерабельной, поскольку изменения действий сигналов, которые она должна выполнять, влияют на весь процесс в целом.

Вышеупомянутые стандарты не запрещают делать функции из табл. 31.1 потокобезопасными. Но даже если некоторые из них являются таковыми, переносимые приложения не могут полагаться на то, что они будут потокобезопасными во всех системах.

Таблица 31.1. Функции, которые могут не быть потокобезопасными согласно стандарту SUSv3

asctime()	fcvt()	getpwnam()	nl_langinfo()
basename()	ftw()	getpwuid()	ptsname()
catgets()	gcvt()	getservbyname()	putc_unlocked()
crypt()	getc_unlocked()	getservbyport()	putchar_unlocked()
ctime()	getchar_unlocked()	getservent()	putenv()
dbm_clearerr()	getdate()	getutxent()	pututxline()
dbm_close()	getenv()	getutxid()	rand()
dbm_delete()	getgrent()	getutxline()	readdir()
dbm_error()	getgrgid()	gmtime()	setenv()
dbm_fetch()	getgrnam()	hcreate()	setgrent()
dbm_firstkey()	gethostbyaddr()	hdestroy()	setkey()
dbm_nextkey()	gethostbyname()	hsearch()	setpwent()
dbm_open()	gethostent()	inet_ntoa()	setutxent()
dbm_store()	getlogin()	l64a()	strerror()
dirname()	getnetbyaddr()	lgamma()	strtok()
dlerror()	getnetbyname()	lgammaf()	ttyname()

Продолжение ⇨

Таблица 31.1 *(продолжение)*

drand48()	getnetent()	lgammal()	unsetenv()
ecvt()	getopt()	localeconv()	wcstombs()
encrypt()	getprotobyname()	localtime()	wctomb()
endgrent()	getprotobynumber()	lrand48()	
endpwent()	getprotoent()	mrand48()	
endutxent()	getpwent()	nftw()	

Реентерабельные и нереентерабельные функции

Использование критических участков для реализации потоковой безопасности является значительным прорывом по сравнению с назначением каждой функции отдельного мьютекса. Однако этот подход все равно не очень эффективен, поскольку на закрытие и открытие мьютексов тоже тратятся ресурсы. *Реентерабельные* (или повторно входимые) функции позволяют достичь потоковой безопасности без применения мьютексов. Это делается за счет отказа от глобальных и статических переменных. Любая информация, которую следует вернуть в вызывающий поток или сохранить между вызовами функции, хранится в буфере, выделенном вызывающим потоком (впервые с реентерабельностью мы столкнулись в подразделе 21.1.2, когда рассматривали работу с глобальными переменными внутри обработчиков сигналов). Однако не все функции можно сделать реентерабельными. Обычно это объясняется следующими причинами.

- Некоторые функции ввиду своего назначения должны получать доступ к глобальным структурам данных. В качестве характерного примера можно привести функции из библиотеки `malloc`, которые хранят в куче связный список со свободными блоками. Эти функции реализованы потокобезопасными с помощью мьютексов.
- Некоторые функции (созданные до изобретения потоков) имеют интерфейс, который делает их нереентерабельными по определению; причиной тому может быть возвращение указателей на хранилище, статически выделенное самой функцией, или использование статического хранилища для хранения информации между последовательными вызовами одних и тех же (или родственных) функций. К этой категории можно причислить большинство функций, описанных в табл. 31.1. Например, функция `asctime()` (см. подраздел 10.2.3) возвращает указатель на статически выделенный буфер, содержащий строку с датой и временем.

Для некоторых функций, имеющих нереентерабельные интерфейсы, в стандарте SUSv3 предусмотрены реентерабельные аналоги, имена которых заканчиваются суффиксом `_r`. Такие функции требуют от вызывающего потока выделить буфер и передать им его адрес; он будет использован для возвращения результата. Это позволяет вызывающему потоку применить для итогового буфера функции локальную переменную (находящуюся в стеке). Для этих целей в стандарт SUSv3 входят следующие функции: `asctime_r()`, `ctime_r()`, `getgrgid_r()`, `getgrnam_r()`, `getlogin_r()`, `getpwnam_r()`, `getpwuid_r()`, `gmtime_r()`, `localtime_r()`, `rand_r()`, `readdir_r()`, `strerror_r()`, `strtok_r()` и `ttyname_r()`.

> Некоторые реализации также предоставляют дополнительные реентерабельные аналоги для других традиционных функций. Например, библиотека glibc содержит функции crypt_r(), gethostbyname_r(), getservbyname_r(), getutent_r(), getutid_r(), getutline_r() и ptsname_r(). Однако переносимые приложения не могут полагаться на их наличие в других реализациях. В некоторых случаях эти реентерабельные аналоги не входят в стандарт SUSv3, поскольку

существуют альтернативы, которые имеют преимущества по сравнению с традиционными функциями и тоже являются реентерабельными. Например, у функций gethostbyname() и getservbyname() есть современная реентерабельная альтернатива, getaddrinfo().

31.2. Единовременная инициализация

Иногда многопоточному приложению нужно сделать так, чтобы определенные процедуры инициализации выполнялись только один раз, независимо от количества создаваемых потоков. Например, вам может потребоваться инициализировать мьютекс специальными атрибутами, используя функцию `pthread_mutex_init()`, и это должно быть единовременное действие. Этого обычно легко достичь, если потоки создаются из главной программы — любые потоки, зависящие от инициализации, должны быть созданы после ее проведения. Однако это невозможно сделать в библиотечной функции, потому что перед ее первым запуском вызывающая программа может уже успеть создать потоки. Следовательно, библиотечной функции нужен механизм выполнения инициализации при первом вызове из любого потока.

Единовременная инициализация реализована в виде функции `pthread_once()`.

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control, void (*init)(void));
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

С помощью аргумента `once_control` функция `pthread_once()` гарантирует, что `init` вызывается только один раз, независимо от того, что несколько разных потоков могут сделать вызов `pthread_once()`.

Функция `init` вызывается без каких-либо аргументов и имеет следующий вид:

```
void
init(void)
{
    /* Тело функции */
}
```

Аргумент `once_control` является указателем на переменную, которая должна быть статически инициализирована с помощью значения `PTHREAD_ONCE_INIT`:

```
pthread_once_t once_var = PTHREAD_ONCE_INIT;
```

Первый вызов `pthread_once()`, определяющий указатель на конкретную структуру `pthread_once_t`, изменяет значение переменной, на которую указывает `once_control`, поэтому последующие вызовы `pthread_once()` не запускают `init`.

Функцию `pthread_once()` часто применяют в сочетании с данными, относящимися к определенному потоку (о чем мы поговорим ниже).

Главная причина существования функции pthread_once() заключается в том, что ранние версии библиотеки Pthreads не позволяли инициализировать мьютекс статически. Вместо этого использовался вызов pthread_mutex_init() (см. [Butenhof, 1996]). Учитывая последующее добавление поддержки статически выделенных мьютексов, для единовременной инициализации теперь достаточно взять статическую булеву переменную. Тем не менее функция pthread_once() — для удобства.

31.3. Данные уровня потока

Самый эффективный способ обеспечения потоковой безопасности функции — это сделать ее реентерабельной. Так должны быть реализованы все новые библиотечные функции. Но если существующая функция нереентерабельна (возможно, она была создана до того, как потоки получили широкое распространение), данный подход обычно требует изменения ее интерфейса, что влечет за собой изменение всех программ, которые ее используют.

Задействование данных, относящихся к отдельному потоку, позволяет сделать функцию потокобезопасной, не изменяя при этом ее интерфейс. Функции, использующие такие данные, могут работать чуть медленней реентерабельных, но позволяют оставлять ранее написанный код без изменений.

Этот способ позволяет функции иметь отдельную копию переменной для каждого потока, который ее вызывает (рис. 31.1). Данные уровня потока являются постоянными; они продолжают существовать между вызовами. Благодаря этому функция в случае необходимости может передавать каждому вызывающему потоку отдельный итоговый буфер.

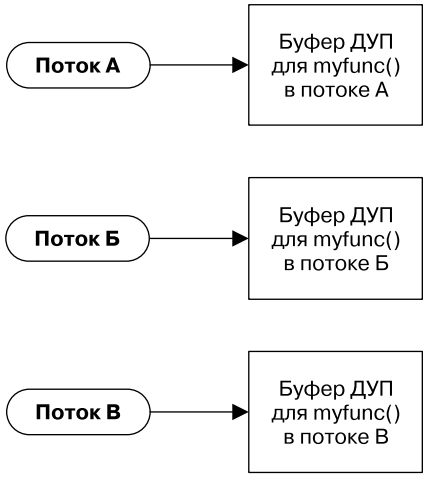


Рис. 31.1. Данные уровня потока (ДУП) предоставляют функции хранилище, относящееся к отдельному потоку

31.3.1. Данные уровня потока с точки зрения библиотечной функции

Чтобы понять, как использовать программный интерфейс для работы с данными уровня потока, необходимо взглянуть на вещи с точки зрения библиотечной функции, которая эти данные потребляет.

- Функция должна выделить отдельный блок хранилища для каждого вызывающего ее потока. Этот блок должен выделяться только один раз — в момент вызова функции из потока.
- При каждом последующем вызове из того же потока функция должна иметь возможность получить адрес блока хранилища, который был выделен во время первого вызова из этого потока. Мы не можем хранить указатель на блок в автоматической переменной, потому что она исчезнет сразу после возвращения функции; мы также не

можем присвоить этот указатель статической переменной, поскольку она имеет только один экземпляр на весь процесс. Программный интерфейс Pthreads предоставляет функции для решения этой задачи.

- Разным (то есть независимым) функциям могут понадобиться данные уровня потока. Каждой функции может потребоваться механизм определения данных уровня потока (ключ), чтобы отличить их от данных, используемых другими функциями.
- Функция не контролирует завершения потока, потому что в этот момент поток, скорее всего, выполняет код за ее пределами. Тем не менее должен существовать некий механизм (деструктор), который бы позволил автоматически освобождать блок хранилища, выделенный для потока, когда тот завершается. В противном случае может возникнуть утечка памяти, поскольку потоки постоянно создаются, вызывают функцию и завершаются.

31.3.2. Обзор программного интерфейса для работы с данными уровня потока

Для использования данных уровня потока функции обычно выполняют следующие шаги.

1. Функция создает *ключ*, с помощью которого можно отличить элементы данных уровня потока, задействуемые разными функциями. Ключ создается посредством вызова `pthread_key_create()`. Эту процедуру необходимо выполнить всего один раз, при первом вызове функции. Для этого применяется вызов `pthread_once()`. Создание ключа не приводит к выделению блоков в сегменте данных уровня потока.
2. Вызов `pthread_key_create()` имеет дополнительное назначение: он позволяет вызывающему потоку указать адрес функции-деструктора, предварительно объявленной программистом, которая используется для освобождения каждого блока хранилища, выделенного с помощью текущего ключа (см. следующий шаг). Когда поток, владеющий блоком хранилища, завершается, программный интерфейс Pthreads автоматически вызывает этот деструктор, передавая ему указатель на блок данных этого потока.
3. Функция выделяет блок данных для каждого потока, в котором она вызывается. Это делается с помощью вызова `malloc()` (или аналогичного ему). Такое выделение производится только один раз в каждом потоке при первом вызове функции.
4. Для сохранения указателя на хранилище, выделенное в предыдущем шаге, используются две функции из состава Pthreads: `pthread_setspecific()` и `pthread_getspecific()`. Первая выполняет запрос к реализации Pthreads, который можно сформулировать так: «сохрани этот указатель и запиши, что он связан с определенным ключом (назначенным этой функции) и определенным потоком (вызывающим)». Функция `pthread_getspecific()` решает обратную задачу, возвращая указатель на заданный ключ, связанный с вызывающим потоком. Если с заданными ключом и потоком не был связан ни один указатель, функция `pthread_getspecific()` возвращает `NULL`. Это позволяет нам определить, что в текущем потоке наша функция вызывается впервые и что нам нужно выделить для этого потока блок хранилища.

31.3.3. Подробности о программном интерфейсе для работы с данными уровня потока

Здесь мы подробно обсудим каждую функцию, упомянутую в предыдущем разделе, а также разберем принцип работы данных уровня потока на примере того, как они обычно реализованы. В следующем разделе будет продемонстрировано использование данных

уровня потока для написания потокобезопасной версии функции `strerror()`, входящей в стандартную библиотеку C.

Вызов функции `pthread_key_create()` приводит к созданию нового ключа, связанного с данными уровня потока; этот ключ возвращается в вызывающий поток внутри буфера, на который указывает аргумент `key`.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Поскольку возвращенный ключ используется во всех потоках процесса, аргумент `key` должен указывать на глобальную переменную.

Аргумент `destructor` указывает на предварительно объявленную функцию следующего вида:

```
void
dest(void *value)
{
    /* Освобождаем хранилище, на которое указывает 'value' */
}
```

При завершении потока, значение `key` которого не равно `NULL`, программный интерфейс Pthreads автоматически вызывает функцию-деструктор, передавая ей это значение в качестве аргумента. `key` обычно является указателем на блок данных уровня потока для этого ключа. Если деструктор не требуется, аргументу `destructor` можно присвоить `NULL`.

> Если поток содержит несколько локальных блоков данных, деструкторы для каждого из них вызываются в произвольном порядке. Функции-деструкторы должны уметь действовать независимо друг от друга.

Изучение реализации данных уровня потока помогает понять, как они используются на практике. Типичная реализация (например, библиотека NPTL) подразумевает наличие таких массивов, как:

- единый глобальный (то есть доступный на уровне всего процесса) массив с информацией о ключах для данных уровня потока;
- набор отдельных массивов в рамках отдельных потоков, каждый из которых содержит указатели на все блоки данных, выделенные для определенного потока (другими словами, в этом массиве находятся указатели, создаваемые в результате вызовов `pthread_setspecific()`).

В этой реализации значение `pthread_key_t`, возвращаемое функцией `pthread_key_create()`, представляет собой обычный индекс в глобальном массиве; этот массив имеет название `pthread_keys`, а его структура показана на рис. 31.2. Каждый элемент массива является структурой с двумя полями. Первое поле говорит, используется ли данный элемент (то есть был ли он выделен предыдущим вызовом `pthread_key_create()`). Второе поле применяется для хранения указателя на функцию-деструктор для блоков данных уровня потока, связанных с текущим ключом (другими словами, это копия аргумента `destructor` из функции `pthread_key_create()`).

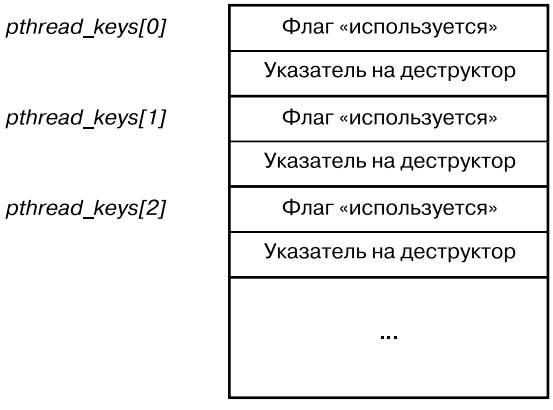


Рис. 31.2. Реализация ключей для данных уровня потока

Функция `pthread_setspecific()` делает запрос к программному интерфейсу Pthreads, чтобы тот сохранил копию `value` в структуре данных, которая связывает ее с вызывающим потоком и ключом `key`, возвращенным предыдущим вызовом `pthread_key_create()`. Функция `pthread_getspecific()` выполняет обратную операцию, возвращая значение, которое ранее было связано с заданным ключом этого потока.

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
```

Возвращает `0` при успешном завершении или положительное число, если случилась ошибка

```
void *pthread_getspecific(pthread_key_t key);
```

Возвращает указатель или `NULL`, если с ключом не связаны данные уровня потока

Аргумент `value`, передающийся функции `pthread_setspecific()`, обычно является указателем на блок памяти, ранее выделенный вызывающим потоком. При завершении потока этот указатель будет передан в функцию-деструктор для текущего ключа.

> Аргумент value не обязательно должен быть указателем на блок памяти. Он может представлять собой некое скалярное значение, которое можно привести к типу void *. В этом случае в ранее сделанном вызове функции pthread_key_create() аргумент destructor должен равняться NULL.

На рис. 31.3 показана типичная реализация структуры данных, которая используется для хранения `value`. На этой диаграмме подразумевается, что элемент `pthread_keys[1]` был выделен функции с именем `myfunc()`. Программный интерфейс Pthreads предоставляет массив указателей на блоки данных каждого отдельного потока. Элементы этих массивов напрямую соответствуют элементам глобального массива `pthread_keys`, показанного на рис. 31.2. Функция `pthread_setspecific()` привязывает каждый элемент массива к соответствующему ключу вызывающего потока.

При создании потока указатели на все его данные инициализируются значением `NULL`. Это означает, что, когда поток впервые вызывает нашу функцию, он должен сначала

сделать вызов `pthread_getspecific()`, чтобы проверить, имеет ли он значение, связанное с ключом. Если такого значения нет, функция выделяет блок памяти и сохраняет указатель на него с помощью вызова `pthread_setspecific()`. В следующем разделе эта процедура будет продемонстрирована на примере реализации потокобезопасной разновидности функции `strerror()`.

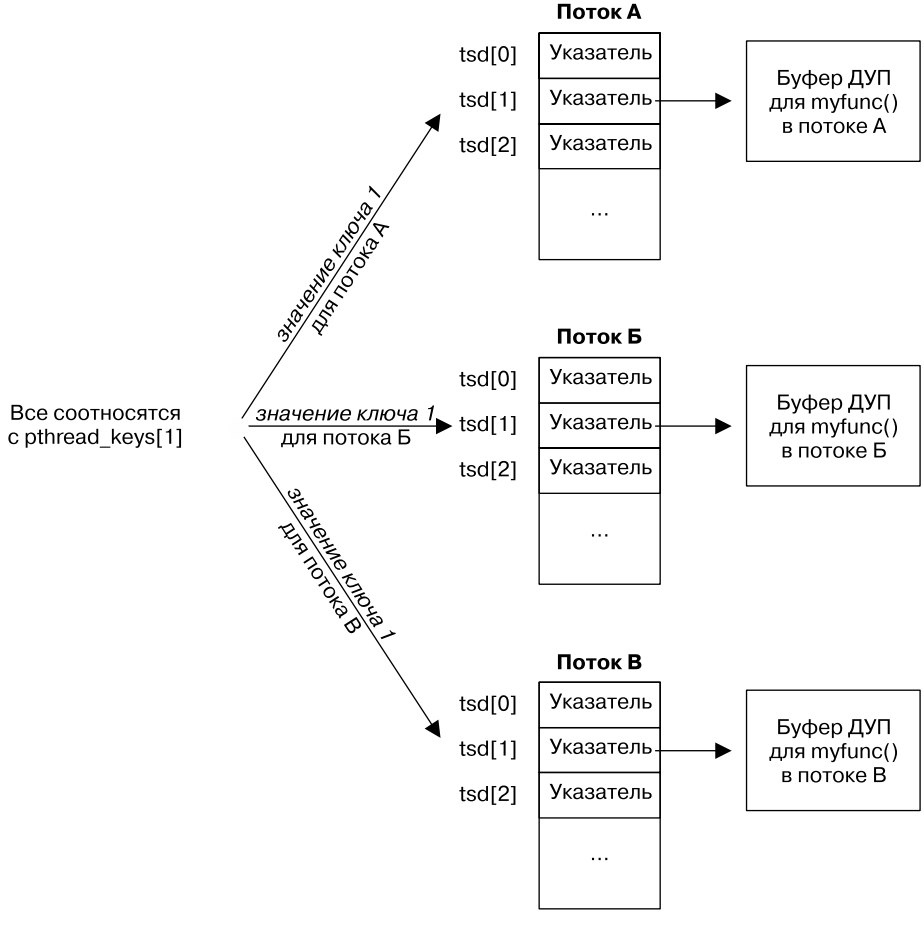


Рис. 31.3. Структура данных, которая используется для реализации указателей на данные уровня потока (ДОП)

31.3.4. Использование программного интерфейса для работы с данными уровня потока

При первом знакомстве со стандартной функцией `strerror()` (см. раздел 3.4) мы отмечали, что она может вернуть указатель на статически выделенную строку. Это означает, что данная функция может не быть потокобезопасной. На нескольких следующих страницах мы рассмотрим стандартную реализацию `strerror()` и затем увидим, как сделать ее потокобезопасной с помощью данных уровня потока.

Во многих реализациях UNIX, включая Linux, функция strerror(), входящая в состав стандартной библиотеки C, является потокобезопасной. Она выбрана в качестве демонстрации, потому что стандарт SUSv3 не требует применять к ней правила потоковой безопасности; к тому же ее реализация послужит простым примером использования данных уровня потока.

В листинге 31.1 показана простая реализация `strerror()`, которая не является потокобезопасной. Она задействует две глобальные переменные, объявленные библиотекой `glibc`: одна, `_sys_errlist`, представляет собой массив указателей на строки, соответствующие номерам ошибок в `errno` (хотя элемент `_sys_errlist[EINVAL]`, к примеру, указывает на строку `Invalid operation`), а другая, `_sys_nerr`, указывает, сколько элементов находится в `_sys_errlist`.

Листинг 31.1. Реализация strerror() без учета потоковой безопасности

threads/strerror.c

```
#define _GNU_SOURCE          /* Получаем объявления '_sys_nerr'
                                и '_sys_errlist' из файла <stdio.h> */
#include <stdio.h>
#include <string.h>          /* Получаем объявление strerror() */

#define MAX_ERROR_LEN 256    /* Максимальная длина строки,
                                возвращаемой функцией strerror() */

static char buf[MAX_ERROR_LEN];  /* Статически выделенный итоговый буфер */

char *
strerror(int err)
{
    if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
    } else {
        strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';  /* Завершаем строку символом '\0' */
    }

    return buf;
}
```

threads/strerror.c

На примере данной программы можно продемонстрировать последствия того, что реализация `strerror()` из листинга 31.1 пренебрегает потоковой безопасностью. Эта программы вызывает `strerror()` из двух разных потоков, но выводит возвращаемое значение только после того, как оба потока завершили выполнение этой функции. И хотя каждый поток передает функции `strerror()` разные значения (`EINVAL` и `EPERM`), полученный результат будет следующим (при условии, если мы скомпилируем и скомпонуем программу с версией `strerror()` из листинга 31.1):

```
$ ./strerror_test
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x804a7c0) = Operation not permitted
Main thread: str (0x804a7c0) = Operation not permitted
```

Оба потока вывели значение `errno`, равное `EPERM`, потому что вызов `strerror()`, сделанный вторым потоком (в функции `threadFunc`), перезаписал буфер, который был записан посредством вызова `strerror()` внутри главного потока. Изучив вывод, можно увидеть, что локальная переменная `str` в обоих потоках указывает на один и тот же адрес.

Листинг 31.2. Вызов strerror() из двух разных потоков

threads/strerror_test.c

```
#include <stdio.h>
#include <string.h>        /* Получаем объявление функции strerror() */
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *str;

    printf("Other thread about to call strerror()\n");
    str = strerror(EPERM);
    printf("Other thread: str (%p) = %s\n", str, str);

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t;
    int s;
    char *str;

    str = strerror(EINVAL);
    printf("Main thread has called strerror()\n");

    s = pthread_create(&t, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Main thread:  str (%p) = %s\n", str, str);

    exit(EXIT_SUCCESS);
}
```

threads/strerror_test.c

В листинге 31.3 показана улучшенная реализация `strerror()`, которая обеспечивает потоковую безопасность с помощью данных уровня потока.

Первым делом наша новая версия `strerror()` вызывает `pthread_once()` ❹, чтобы первый вызов этой функции (сделанный из любого потока) выполнил вызов `createKey()` ❷. Функция `createKey()` вызывает `pthread_key_create()`, чтобы выделить ключ для данных уровня потока, который будет храниться в глобальной переменной `strerrorKey` ❸. Вызов `pthread_key_create()` также записывает адрес деструктора ❶, с помощью которого мы освободим буфер уровня потока, относящийся к текущему ключу.

Затем функция `strerror()` вызывает `pthread_getspecific()` ❺, чтобы получить адрес уникального буфера текущего потока, соотносящегося с ключом `strerrorKey`. Если `pthread_getspecific()` возвращает `NULL`, значит, данный поток вызывает `strerror()` впервые, поэтому функция выделяет новый буфер с помощью `malloc()` ❻ и сохраняет его адрес с помощью вызова `pthread_setspecific()` ❼. Если `pthread_getspecific()`

возвращает ненулевое значение, значит, указатель ссылается на существующий буфер, который был выделен в результате предыдущего вызова `strerror()`.

Оставшийся код этой реализации `strerror()` похож на ранее показанную версию, с той лишь разницей, что `buf` теперь является адресом буфера с данными уровня потока, а не статической переменной.

Листинг 31.3. Потокобезопасная реализация strerror() с использованием данных уровня потока

threads/strerror_tsd.c

```
#define _GNU_SOURCE        /* Получаем объявления '_sys_nerr'
                              и '_sys_errlist' из файла <stdio.h> */
#include <stdio.h>
#include <string.h>        /* Получаем объявление strerror() */
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t strerrorKey;

#define MAX_ERROR_LEN 256    /* Максимальная длина строки буфера уровня
                                потока, возвращаемого функцией strerror() */

static void       /* Освобождаем буфер с данными уровня потока */
(1) destructor(void *buf)
{
    free(buf);
}

static void    /* Функция для единовременного создания ключа */
(2) createKey(void)
{
    int s;

    /* На уровне потока выделяем уникальный ключ для буфера
       и сохраняем адрес деструктора этого буфера */

(3) s = pthread_key_create(&strerrorKey, destructor);
    if (s != 0)
        errExitEN(s, "pthread_key_create");
}

char *
strerror(int err)
{
    int s;
    char *buf;

    /* Выделяем ключ для данных уровня потока при первом вызове */

(4) s = pthread_once(&once, createKey);
    if (s != 0)
        errExitEN(s, "pthread_once");

(5) buf = pthread_getspecific(strerrorKey);
    if (buf == NULL) {    /* Если это первый вызов из данного потока,
                             выделяем буфер и сохраняем его адрес */
```

```
❻       buf = malloc(MAX_ERROR_LEN);
        if (buf == NULL)
            errExit("malloc");

❼       s = pthread_setspecific(strerrorKey, buf);
        if (s != 0)
            errExitEN(s, "pthread_setspecific");
    }

    if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
    } else {
        strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';  /* Завершаем строку символом '\0' */
    }

    return buf;
}
```

threads/strerror_tsd.c

Если скомпилировать и скомпоновать нашу тестовую программу (см. листинг 31.2) с новой версией `strerror()` (листинг 31.3), чтобы получить исполняемый файл `strerror_test_tsd`, при его запуске можно будет увидеть следующий результат:

```
$ ./strerror_test_tsd
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x804b158) = Operation not permitted
Main thread: str (0x804b008) = Invalid argument
```

По этому выводу видно, что новая версия `strerror()` является потокобезопасной. Также можно заметить, что в двух потоках используются разные адреса, на которые указывают локальные переменные `str`.

31.3.5. Ограничения реализации данных уровня потока

При описании типичной реализации данных уровня потока подразумевалось, что количество ключей для них, возможно, придется ограничить. Стандарт SUSv3 требует, чтобы программа поддерживала как минимум 128 ключей (константа `_POSIX_THREAD_KEYS_MAX`). Чтобы понять, сколько именно ключей поддерживается в текущей реализации, мы можем либо свериться с объявлением `PTHREAD_KEYS_MAX` (в заголовочном файле `<limits.h>`), либо сделать вызов `sysconf(_SC_THREAD_KEYS_MAX)`. В Linux количество ключей ограничено числом 1024.

Но даже 128 ключей должно быть более чем достаточно для большинства приложений. Дело в том, что любая библиотечная функция должна использовать небольшую часть от этого количества, обычно ограничиваясь всего одним ключом. Если функции понадобилось несколько значений уровня потока, их, как правило, можно легко поместить внутрь структуры и связать с ней только один ключ.

31.4. Локальное хранилище потока

По аналогии с данными уровня потока локальное хранилище потока предоставляет средство постоянного хранения информации. Это нестандартная концепция, но она реализована в похожем (или одном и том же) виде в других системах UNIX (таких как Solaris и FreeBSD).

Главное преимущество локального хранилища заключается в том, что его значительно проще использовать по сравнению с данными уровня потока. Чтобы сделать глобальную или статическую переменную локальной для каждого потока, нужно просто указать при ее объявлении спецификатор `__thread`:

```
static __thread buf[MAX_ERROR_LEN];
```

Каждый поток имеет свою собственную копию переменных, объявленных с этим спецификатором. Переменные в локальном хранилище существуют до тех пор, пока не завершится поток, который ими владеет — в этот момент хранилище автоматически утилизируется.

Стоит отметить следующие моменты касательно объявления и использования локальных переменных потока.

- Ключевое слово `__thread` должно быть указано сразу после ключевых слов `static` или `extern`, если какое-либо из них используется при объявлении.
- Объявление локальной переменной потока может содержать инициализатор — по аналогии с тем, как это делается в обычных глобальных или статических переменных.
- Оператор адреса в языке C (`&`) может использоваться для получения указателя на локальную переменную потока.

Локальное хранилище потока требует поддержки со стороны ядра (предоставляется, начиная с Linux 2.6), реализации Pthreads (предоставляется библиотекой NPTL) и компилятора языка C (на платформе x86-32 обеспечивается компилятором gcc версии 3.3 и выше).

Потокобезопасная реализация `strerror()` на основе локального хранилища потока показана в листинге 31.4. Если скомпилировать и скомпоновать нашу тестовую программу (см. листинг 31.2) с этой версией `strerror()`, чтобы получить исполняемый файл `strerror_test_tls`, при его запуске можно будет увидеть следующий результат:

```
$ ./strerror_test_tls
Main thread has called strerror()
Other thread about to call strerror()
Other thread: str (0x40376ab0) = Operation not permitted
Main thread: str (0x40175080) = Invalid argument
```

Листинг 31.4. Потокобезопасная реализация strerror() на основе локального хранилища потока

threads/strerror_tls.c

```
#define _GNU_SOURCE     /* Получаем объявления '_sys_nerr'
                           и '_sys_errlist' из файла <stdio.h> */
#include <stdio.h>
#include <string.h>     /* Получаем объявление strerror() */
#include <pthread.h>

#define MAX_ERROR_LEN 256  /* Максимальная длина строки буфера уровня
                              потока, возвращаемого функцией strerror() */

static __thread char buf[MAX_ERROR_LEN];
/* Итоговый локальный буфер потока */

char *
strerror(int err)
{
    if (err < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", err);
    } else {
        strncpy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
```

```
        buf[MAX_ERROR_LEN - 1] = '\0';  /* Завершаем строку символом '\0' */
    }

    return buf;
}
```

threads/strerror_tls.c

31.5. Резюме

Функция считается потокобезопасной, если ее можно безопасно вызывать одновременно из нескольких потоков. Чаще всего потоковая безопасность нарушается в результате задействования глобальных или статических переменных. Один из способов «обезопасить» функцию — защитить все ее вызовы с помощью мьютекса. Недостатком этого подхода является ухудшение параллелизма, потому что только один поток может выполнять функцию в любой заданный момент времени. Частично устранить эту проблему помогает использование мьютексов только на тех участках кода функции, которые работают с разделяемыми переменными (так называемые критические участки).

Мьютексы позволяют сделать потокобезопасной большинство функций, но это негативно сказывается на производительности, поскольку открытие и закрытие мьютексов требует определенных ресурсов. Вместо этого потоковую безопасность можно обеспечить за счет реентерабельных функций, избегая задействования глобальных и статических переменных.

Большинство функций, описанных в стандарте SUSv3, должны быть потокобезопасными. Этот стандарт также ограничивает круг функций, на которые это требование не распространяется. Такие функции обычно используют статическое хранилище для возвращения информации вызывающему потоку или сохраняют данные между последовательными вызовами. По умолчанию они не являются реентерабельными, и для обеспечения их потоков безопасности нельзя применять мьютексы. Мы рассмотрели две примерно равнозначные методики, которые позволяют сделать функцию потокобезопасной, не изменяя при этом ее интерфейс — данные уровня потока и локальное хранилище потока. Оба этих подхода позволяют функции выделять в каждом потоке отдельное постоянное хранилище.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 29.10.

31.6. Упражнения

31.1. Реализуйте функцию `one_time_init(control, init)`, которая делает то же самое, что и вызов `pthread_once()`. Аргумент `control` должен быть указателем на статически выделенную структуру, содержащую булеву переменную и мьютекс. Булева переменная сигнализирует, вызывалась ли функция `init` ранее, а мьютекс защищает доступ к переменной. Чтобы не усложнять реализацию, вы можете игнорировать случаи, когда `init()` завершается неудачей или отменяется при первом вызове из потока (то есть в таких случаях вам не обязательно предусматривать механизм, благодаря которому следующий поток, вызывающий `one_time_init()`, делает повторную попытку вызова `init()`).

31.2. Напишите потокобезопасные версии функций `dirname()` и `basename()` (см. раздел 18.14) на основе данных уровня потока.

32 Потоки выполнения: отмена потока

Обычно разные потоки работают параллельно, выполняя свою задачу до тех пор, пока не решат завершиться с помощью вызова `pthread_exit()` или вернуться из своей начальной функции.

Но иногда возникает необходимость в *отмене* потока; это выглядит как передача потоку запроса с просьбой немедленно закончить работу. Это может пригодиться в ситуации, когда, например, группа потоков выполняет какие-то вычисления и один из них обнаруживает ошибку, из-за которой нужно завершить все остальные потоки. Или, к примеру, графическое приложение может иметь кнопку отмены, с помощью которой пользователь может завершить задачу, выполняемую потоком в фоновом режиме; в этом случае главный поток (который управляет графическим интерфейсом) должен попросить фоновый поток завершиться.

В этой главе мы рассмотрим механизм отмены POSIX-потоков.

32.1. Отмена потока

Функция `pthread_cancel()` отправляет заданному потоку `thread` запрос отмены.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Выполнив запрос отмены, функция `pthread_cancel()` немедленно возвращается; то есть она не ждет завершения заданного потока.

Что именно происходит с заданным потоком и в какой момент, зависит от его состояния и типа отмены (о чем пойдет речь в следующем разделе).

32.2. Состояние и тип отмены

Функции `pthread_setcancelstate()` и `pthread_setcanceltype()` устанавливают флаги, которые позволяют управлять реакцией потока на запрос отмены.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Обе функции возвращают `0` при успешном завершении или положительное число, если произошла ошибка

Функция pthread_setcancelstate() приводит состояние отмены вызывающего потока к одному из двух значений, поддерживаемых аргументом state.

- PTHREAD_CANCEL_DISABLE — поток нельзя отменить. Если получен запрос отмены, он откладывается до момента, когда возможность отмены будет включена.
- PTHREAD_CANCEL_ENABLE — поток можно отменить. Это состояние используется по умолчанию для создаваемых потоков.

Предыдущее состояние отмены возвращается по адресу, на который указывает аргумент oldstate.

> Если нас не интересует предыдущее состояние отмены, Linux позволяет присвоить oldstate значение NULL. Это также относится ко многим другим системам; однако стандарт SUSv3 не предусматривает данной возможности, поэтому переносимые приложения не могут на нее полагаться. Для аргумента oldstate всегда нужно указывать ненулевое значение.

Временное отключение возможности отмены (PTHREAD_CANCEL_DISABLE) может пригодиться, если поток находится на участке кода, который должен быть выполнен *целиком*.

Если поток можно отменить (PTHREAD_CANCEL_ENABLE), реакция на запрос отмены определяется типом отмены потока, который указывается в аргументе type при вызове pthread_setcanceltype(). Этот аргумент может принимать одно из следующих значений:

- PTHREAD_CANCEL_ASYNCHRONOUS — поток может быть отменен в любой момент (в некоторых случаях немедленно). Возможность асинхронной отмены редко бывает полезной; мы отложим ее обсуждение до раздела 32.6;
- PTHREAD_CANCEL_DEFERRED — процедура отмены задерживается до определенного момента (см. следующий раздел). Этот тип отмены используется по умолчанию в создаваемых потоках. О задержке отмены пойдет речь в следующих разделах.

Предыдущий тип отмены потока возвращается по адресу, на который указывает аргумент oldtype.

> Как и в случае с аргумент oldstate для функции pthread_setcancelstate(), многие системы, в том числе и Linux, позволяют присвоить oldtype значение NULL, если нас не интересует предыдущий тип отмены. Но опять же стандарт SUSv3 не предусматривает этой возможности, поэтому переносимые приложения не должны на нее полагаться. Для аргумента oldtype всегда нужно указывать ненулевое значение.

При вызове fork() потомок наследует тип и состояние отмены вызывающего потока. При вызове exec() тип и состояние отмены главного потока новой программы сбрасываются к значениям соответственно PTHREAD_CANCEL_ENABLE и PTHREAD_CANCEL_DEFERRED.

32.3. Точки отмены

Если возможность отмены включена и отложена, запрос выполняется, тогда поток достигает следующей *точки отмены*, то есть вызова одной из функций, определенных системой.

Согласно стандарту SUSv3 функции, представленные в табл. 32.1, *должны* быть точками отмены, если они поддерживаются системой. Большинство из них способно заблокировать поток на неопределенное время.

Таблица 32.1. Функции, которые должны быть точками отмены согласно стандарту SUSv3

accept()	nanosleep()	sem_timedwait()
aio_suspend()	open()	sem_wait()
clock_nanosleep()	pause()	send()
close()	poll()	sendmsg()
connect()	pread()	sendto()
creat()	pselect()	sigpause()
fcntl(F_SETLKW)	pthread_cond_timedwait()	sigsuspend()
fsync()	pthread_cond_wait()	sigtimedwait()
fdatasync()	pthread_join()	sigwait()
getmsg()	pthread_testcancel()	sigwaitinfo()
getpmsg()	putmsg()	sleep()
lockf(F_LOCK)	putpmsg()	system()
mq_receive()	pwrite()	tcdrain()
mq_send()	read()	usleep()
mq_timedreceive()	readv()	wait()
mq_timedsend()	recv()	waitid()
msgrcv()	recvfrom()	waitpid()
msgsnd()	recvmsg()	write()
msync()	select()	writev()

Помимо содержимого табл. 32.1, стандарт SUSv3 описывает еще более масштабный набор функций, которые *могут* быть точками отмены в той или иной системе. Это касается функций стандартного ввода/вывода, программных интерфейсов `dlopen` и `syslog`, вызовов `nftw()`, `popen()`, `semop()` и `unlink()`, а также различных функций для извлечения информации из системных файлов, таких как `utmp`. Переносимые программы должны учитывать возможность того, что поток может быть отменен при вызове этих функций.

Стандарт SUSv3 гласит, что, помимо двух вышеупомянутых наборов функций, которые должны и могут быть точками отмены, ни одна другая функция, являющаяся частью стандарта, не может привести к отмене потока (то есть переносимым программам не нужно заботиться о том, что она может повести себя как точка отмены).

Стандарт SUSv4 добавляет `openat()` в список функций, которые должны быть точками отмены, и удаляет из него `sigpause()` (переходит в список функций, которые *могут* быть точками отмены) и `usleep()` (полностью исключается из стандарта).

> Каждая система может свободно маркировать дополнительные функции, которые не входят в стандартный список точек отмены. Вероятным кандидатом на попадание в этот список может оказаться любая функция, которая способна блокировать выполнение (например, во время доступа к файлу). Множество нестандартных функций в библиотеке glibc обозначены как точки отмены именно по этой причине.

Во время получения запроса отмены поток, который можно отменять с задержкой, завершается при достижении следующей точки отмены. Если он не был отсоединен,

какой-то другой поток в процессе может его присоединить, чтобы не дать ему превратиться в «зомби». Во время присоединения отмененного потока значение, возвращенное во втором аргументе функции `pthread_join()`, равно `PTHREAD_CANCELED` (специальное значение, возвращаемое потоком).

Пример программы

В листинге 32.1 показан простой пример использования функции `pthread_cancel()`. Главная программа создает поток, выполняющий бесконечный цикл, который останавливается на секунду и выводит значение счетчика цикла (этот поток завершится, только если ему послать запрос отмены или если процесс завершит свою работу). В то же время главная программа останавливается на три секунды, после чего отправляет запрос отмены потоку, который она создала. Запустив данную программу, мы увидим следующее:

```
$ ./t_pthread_cancel
New thread started
Loop 1
Loop 2
Loop 3
Thread was canceled
```

Листинг 32.1. Отмена потока с помощью pthread_cancel()

threads/thread_cancel.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    int j;
    printf("New thread started\n"); /* Может быть точкой отмены */
    for (j = 1; ; j++) {
        printf("Loop %d\n", j);     /* Может быть точкой отмены */
        sleep(1);                   /* Точка отмены */
    }

    /* NOTREACHED */
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;

    s = pthread_create(&thr, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    sleep(3); /* Позволяем новому потоку проработать некоторое время */

    s = pthread_cancel(thr);
    if (s != 0)
        errExitEN(s, "pthread_cancel");
```

```
    s = pthread_join(thr, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    if (res == PTHREAD_CANCELED)
        printf("Thread was canceled\n");
    else
        printf("Thread was not canceled (should not happen!)\n");

    exit(EXIT_SUCCESS);
}
```

threads/thread_cancel.c

32.4. Проверка возможности отмены потока

В листинге 32.1 поток, созданный функцией `main()`, принял запрос отмены, поскольку он выполнял функцию, которая является точкой отмены (`sleep()` является ею наверняка, а `printf()` может быть таковой). Но представьте, что поток выполняет цикл, который не содержит точек отмены (например, состоящий из сплошных вычислений). В этом случае запрос отмены никогда не будет удовлетворен.

Единственное назначение функции `pthread_testcancel()` — быть точкой отмены. Если отмена отложена во время вызова данной функции, это означает, что вызывающий поток завершен.

```
#include <pthread.h>

void pthread_testcancel(void);
```

Поток, который не содержит точек отмены, может время от времени вызывать `pthread_testcancel()`, чтобы обеспечить своевременный ответ на запрос отмены, отправляемый другим потоком.

32.5. Обработчики, освобождающие ресурсы

Если поток с отложенной отменой просто завершается при достижении точки отмены, разделяемые переменные и объекты библиотеки Pthreads (например, мьютексы) могут остаться в несогласованном состоянии, что может привести к получению некорректных результатов, взаимным блокировкам или аварийному завершению оставшихся потоков. Чтобы обойти эту проблему, поток может установить один или несколько *обработчиков для освобождения ресурсов* — это функции, которые автоматически выполняются при отмене потока. Эти обработчики могут выполнять перед завершением потока такие задачи, как изменение значений глобальных переменных и открытие мьютексов.

Каждый поток может иметь стек обработчиков для очистки ресурсов. При отмене потока эти обработчики выполняются снизу вверх; то есть обработчик, установленный позже других, вызывается первым. Когда все обработчики выполнились, поток завершается.

Функции `pthread_cleanup_push()` и `pthread_cleanup_pop()` соответственно добавляют и удаляют обработчики в стеке вызывающего потока.

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void*), void *arg);
void pthread_cleanup_pop(int execute);
```

Вызов `pthread_cleanup_push()` добавляет функцию, чей адрес указан в аргументе `routine`, на вершину стека обработчиков для очистки ресурсов потока. Аргумент `routine` указывает на функцию следующего вида:

```
void
routine(void *arg)
{
    /* Код для очистки ресурсов */
}
```

Значение `arg`, переданное в `pthread_cleanup_push()`, предоставляется в виде аргумента вызываемого обработчика. Тип этого аргумента — `void *`, но его можно привести и к другому типу данных.

Обычно очистка требуется, только если поток был отменен во время выполнения определенного участка кода. Если поток проходит этот участок без отмены, очищать ресурсы больше не нужно. В связи с этим `pthread_cleanup_push()` имеет сопровождающий вызов, `pthread_cleanup_pop()`, который удаляет функцию на вершине стека обработчиков для очистки ресурсов. Если аргумент `execute` не равен 0, обработчик тоже выполняется. Это может пригодиться в ситуациях, когда нам нужно очистить ресурсы, даже если поток не был отменен.

Здесь мы описываем `pthread_cleanup_push()` и `pthread_cleanup_pop()` как функции, но стандарт SUSv3 позволяет реализовывать их в виде макросов, которые разворачиваются в цепочки инструкций, включающих соответственно открывающую (`{`) и закрывающую (`}`) скобки. Так происходит в Linux и многих других системах, хотя не все реализации UNIX используют данный подход. Это означает, что в одном и том же лексическом блоке кода каждому `pthread_cleanup_push()` должен соответствовать `pthread_cleanup_pop()` (в системах, которые реализованы таким образом, переменные, объявленные между вызовами `pthread_cleanup_push()` и `pthread_cleanup_pop()`, будут ограничены этой лексической областью). Например, следующий код является некорректным:

```
pthread_cleanup_push(func, arg);
...
if (cond) {
    pthread_cleanup_pop(0);
}
```

В качестве дополнительного удобства все обработчики для очистки ресурсов, которые не сработали, выполняются автоматически, если поток завершается с помощью вызова `pthread_exit()` (это не касается ситуаций, когда для завершения используется инструкция `return`).

Пример программы

В листинге 32.2 представлен простой пример использования обработчика для освобождения ресурсов. Главная программа создает поток ❽, который первым делом выделяет блок памяти ❸ (его адрес будет храниться в переменной `buf`) и затем закрывает мьютекс `mtx` ❹. Поскольку поток может быть отменен, он применяет функцию `pthread_cleanup_push()` ❺ для установки обработчика очистки ресурсов, который вызывается по адресу, хранящемуся внутри `buf`. Если выполнить этот обработчик, он очистит освободившуюся память ❶ и откроет мьютекс ❷.

Затем поток входит в цикл и ждет, когда условная переменная `cond` получит уведомление ❻. Этот цикл будет завершен одним из двух способов в зависимости от того, запускалась ли программа с аргументом командной строки.

- Если аргумент командной строки отсутствует, поток отменяется функцией `main()` ❾. В этом случае отмена произойдет в момент вызова функции `pthread_cond_wait()` ❻, которая является точкой отмены (см. табл. 32.1). В ходе этой процедуры автоматически выполняются обработчики очистки ресурсов, установленные с помощью `pthread_cleanup_push()`.
- Если аргумент командной строки был предоставлен, условная переменная получает уведомление ❿ после того, как связанной с ней глобальной переменной `glob` будет впервые присвоено ненулевое значение. В этом случае поток проходит через выполнение функции `pthread_cleanup_pop()` ❼, которая, если ей передать ненулевой аргумент, тоже приводит к вызову обработчика очистки ресурсов.

Главная программа присоединяет завершенный поток ⓫ и сообщает, был ли он отменен или завершился в штатном режиме.

Листинг 32.2. Использование обработчиков для освобождения ресурсов

threads/thread_cleanup.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int glob = 0; /* Предикат */

static void          /* Освобождаем память по адресу 'arg' и открываем мьютекс */
cleanupHandler(void *arg)
{
    int s;

    printf("cleanup: freeing block at %p\n", arg);
❶   free(arg);

    printf("cleanup: unlocking mutex\n");
❷   s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}

static void *
threadFunc(void *arg)
{
    int s;
    void *buf = NULL; /* Буфер, выделенный потоком */

❸   buf = malloc(0x10000); /* Не является точкой отмены */
    printf("thread: allocated memory at %p\n", buf);

❹   s = pthread_mutex_lock(&mtx); /* Не является точкой отмены */
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

❺   pthread_cleanup_push(cleanupHandler, buf);

    while (glob == 0) {
❻       s = pthread_cond_wait(&cond, &mtx); /* Точка отмены */
        if (s != 0)
```

```
            errExitEN(s, "pthread_cond_wait");
    }

    printf("thread: condition wait loop completed\n");
❼   pthread_cleanup_pop(1); /* Выполняет обработчик очистки ресурсов */
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    void *res;
    int s;

❽   s = pthread_create(&thr, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");
    sleep(2); /* Даем потоку шанс начать работу */

    if (argc == 1) { /* Отменяем поток */
        printf("main: about to cancel thread\n");
❾       s = pthread_cancel(thr);
    if (s != 0)
        errExitEN(s, "pthread_cancel");

    } else { /* Уведомляем условную переменную */
        printf("main: about to signal condition variable\n");
        glob = 1;
❿       s = pthread_cond_signal(&cond);
        if (s != 0)
            errExitEN(s, "pthread_cond_signal");
    }

⓫   s = pthread_join(thr, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");
    if (res == PTHREAD_CANCELED)
        printf("main: thread was canceled\n");
    else
        printf("main: thread terminated normally\n");
    exit(EXIT_SUCCESS);
}
```

threads/thread_cleanup.c

Если запустить программу из листинга 32.2 без аргументов командной строки, функция `main()` сделает вызов `pthread_cancel()`, после чего автоматически выполнится обработчик очистки ресурсов, и мы увидим следующее:

```
$ ./thread_cleanup
thread: allocated memory at 0x804b050
main: about to cancel thread
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main: thread was canceled
```

Если запустить программу с аргументом командной строки, функция `main()` присвоит `glob` значение `1` и уведомит об этом условную переменную, после чего вызов `pthread_cleanup_pop()` приведет к выполнению обработчика очистки ресурсов, и мы увидим следующее:

```
$ ./thread_cleanup s
thread: allocated memory at 0x804b050
main: about to signal condition variable
thread: condition wait loop completed
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main: thread terminated normally
```

32.6. Асинхронная отмена

Если сделать поток асинхронно отменяемым (тип отмены `PTHREAD_CANCEL_ASYNCHRONOUS`), его можно будет отменить в любой момент (то есть в любом машинном коде); это процедура не задерживается до достижения потоком следующей точки отмены.

Проблема с асинхронной отменой заключается в том, что, хотя обработчики очистки ресурсов по-прежнему запускаются, они не могут определить состояние потока. В программе из листинга 32.2, в которой применяется отложенная отмена, поток может быть отменен только при вызове `pthread_cond_wait()` — единственной точки отмены. На этом этапе мы уже знаем, что переменная `buf` уже была инициализирована с помощью адреса блока выделенной памяти и что мьютекс `mtx` был закрыт. Но асинхронная отмена может произойти в любой момент; например, между вызовом `malloc()` и закрытием мьютекса или до/после этих событий. Обработчик очистки ресурсов не может узнать, где именно произошла отмена или какие именно ресурсы нужно освободить. Более того, поток может быть отменен даже *во время* вызова `malloc()`, что, скорее всего, приведет к непредвиденным результатам (см. подраздел 7.1.3).

В целом, асинхронно отменяемые потоки не могут выделять никаких ресурсов или владеть мьютексами, семафорами или блокировками. Это делает невозможным использование целого ряда библиотечных функций, включая большую часть содержимого библиотеки Pthreads (для функций `pthread_cancel()`, `pthread_setcancelstate()` и `pthread_setcanceltype()` в стандарте SUSv3 сделано исключение, так как они обязаны быть безопасными в условиях асинхронной отмены; то есть они должны быть реализованы так, чтобы их можно было безопасно вызывать из асинхронно отменяемого потока). Иными словами, существует несколько случаев, когда асинхронная отмена может оказаться полезной. Это, например, относится к отмене потока, находящегося в вычислительном цикле.

32.7. Резюме

Функция `pthread_cancel()` позволяет потокам отправлять друг другу запрос отмены, в результате которого поток должен быть завершен.

Реакцию потока на полученный запрос определяет его состояние и тип отмены. Если поток нельзя отменить, запрос будет отложен до тех пор, пока состояние потока не изменится. Если поток является отменяемым, реакция на запрос определяется типом его отмены. Если отмена отложенная, поток будет отменен при достижении одной из функций, которые определены стандартом SUSv3 в качестве точек отмены. Если отмена асинхронная, поток может быть отменен в любой момент (что редко бывает полезным).

Поток может иметь стек обработчиков для освобождения ресурсов; это функции, объявляемые программистом, которые в случае отмены потока автоматически вызываются для проведения очистки (то есть для восстановления состояния разделяемых переменных или открытия мьютексов).

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 29.10.

33 Потоки выполнения: дальнейшие подробности

В этой главе продолжается рассмотрение различных аспектов POSIX-потоков. Мы обсудим их взаимодействие в контексте традиционных программных интерфейсов UNIX — в частности, с помощью сигналов и средств управления процессами (`fork()`, `exec()` и `_exit()`). Здесь также будет представлен краткий обзор двух реализаций POSIX-потоков, доступных в Linux, — LinuxThreads и NPTL и их расхождения со спецификацией Pthreads, описанной в стандарте SUSv3.

33.1. Стеки потоков

Каждый поток имеет свой собственный стек фиксированного размера, который определяется при его создании. На платформе Linux/x86-32 размер стека во всех потоках, кроме главного, по умолчанию равен 2 Мбайт (в некоторых 64-битных архитектурах стандартный размер выше; например, в системах IA-64 он достигает 32 Мбайт). Главному потоку под стек выделяется намного больше места (см. рис. 29.1).

Иногда бывает полезно изменить размер стека, принадлежащего потоку. Это делается с помощью функции `pthread_attr_setstacksize()`, которая устанавливает атрибут потока (см. раздел 29.8), определяющий размер стека. Связанная с ней функция `pthread_attr_setstack()` позволяет управлять как размером, так и местоположением стека, хотя изменение последнего может негативно сказаться на переносимости приложения. Подробности об этих функциях ищите на их справочных страницах.

Изменение размера стека, принадлежащего потоку, может понадобиться в случаях, когда потоку нужно выделять большие автоматические переменные или выполнять вложенные вызовы функций большой глубины (возможно, в результате рекурсии). Приложение также может уменьшить размер стека, чтобы иметь возможность создать большее количество потоков в рамках одного процесса. Например, на платформе x86-32 виртуальное адресное пространство, доступное пользователю, ограничено размером 3 Гбайт, а стандартный объем стека равен 2 Мбайт; следовательно, мы можем создать около 1500 потоков (точное число зависит от того, сколько виртуальной памяти выделено на сегменты с текстом и данными, на разделяемые библиотеки и т. д.). Минимальный размер стека, доступный в текущей системе, можно узнать с помощью вызова `sysconf(_SC_THREAD_STACK_MIN)`. Для библиотеки NPTL на платформе Linux/x86-32 это значение равно `16384`.

> Если в потоковой библиотеке NPTL размер стека (RLIMIT_STACK) не является неограниченным, он используется по умолчанию при создании новых потоков. Это значение устанавливается до запуска программы, обычно с помощью встроенной команды ulimit -s (которая ограничивает размер стека в командной оболочке C). Для задания ограничения недостаточно вызвать функцию setrlimit() внутри главной программы, поскольку библиотека NPTL определяет размер стека во время инициализации, которая происходит до запуска функции main().

33.2. Потоки и сигналы

Модель сигналов в UNIX была спроектирована с учетом особенностей процессов в этой системе и опередила на два десятилетия появление программного интерфейса Pthreads. В итоге между сигналами и потоками возникают заметные конфликты. В основе этих конфликтов, как правило, лежит необходимость сохранения традиционной для однопоточных процессов семантики (то есть интерфейс Pthreads не должен менять семантику сигналов старых программ) в сочетании с разработкой такой модели сигналов, которая была бы уместной в условиях многопоточности.

Эти расхождения между моделями сигналов и потоков приводят к тому, что их совместное использование сопряжено с трудностями и по возможности его следует избегать. Но иногда нам все же приходится работать с сигналами в многопоточных программах. В этом разделе мы обсудим взаимодействие между потоками и сигналами и опишем различные функции, которые могут при этом пригодиться.

33.2.1. Как модель сигналов в UNIX соотносится с потоками

Чтобы понять, как UNIX-сигналы соотносятся с моделью Pthreads, необходимо понимать, какие из аспектов сигнальной модели распространяются на весь процесс (то есть являются общими для всех потоков в процессе), а какие относятся к отдельным потокам. В следующем списке собраны ключевые моменты.

- Назначение сигналов распространяется на весь процесс. Если какой-либо необрабатываемый сигнал с действием *остановки* или *завершения* доставляется любому потоку, остановке или завершению подлежат все потоки в этом процессе.
- Действия сигналов распространяются на весь процесс; все потоки в процессе разделяют одно и то же действие для каждого сигнала. Если один поток использует вызов `sigaction()`, чтобы установить обработчик для какого-нибудь сигнала (например, `SIGINT`), этот обработчик можно будет вызвать из любого потока, которому доставлен данный сигнал. Аналогично, если один поток решит сделать действие сигнала *игнорируемым*, этот сигнал будет игнорироваться всеми потоками.
- Сигнал может быть направлен как процессу в целом, так и отдельному потоку. Сигнал направлен в поток, если:
  - он сгенерирован в качестве непосредственного результата выполнения определенной аппаратной инструкции в рамках контекста потока (то есть аппаратных исключений `SIGBUS`, `SIGFPE`, `SIGILL` и `SIGSEGV`; подробности — в разделе 22.4);
  - это сигнал `SIGPIPE`, сгенерированный в момент, когда поток пытался выполнить запись в поврежденный конвейер;
  - он был отправлен с помощью функций `pthread_kill()` или `pthread_sigqueue()` (описаны в подразделе 33.2.3), которые позволяют потокам одного процесса обмениваться между собой сигналами.
- Все сигналы, сгенерированные другими механизмами, направлены на весь процесс. Например, это могут быть сигналы, отправленные другим процессом с помощью вызовов `kill()` или `sigqueue()`; или такие сигналы, как `SIGINT` и `SIGTSTP`, генерируемые в момент, когда пользователь нажимает соответствующее сочетание клавиш; или сигналы, сгенерированные для таких программных событий, как изменение размера окна терминала (`SIGWINCH`) или истечение срока действия таймера (например, `SIGALRM`).

- Когда сигнал доставляется многопоточному процессу, у которого есть подходящий обработчик, ядро наугад выбирает один поток, чтобы доставить ему этот сигнал и дать возможность его обработать. Такой подход соответствует традиционной семантике сигналов. Обработка одного и того же сигнала несколько раз не имела бы никакого смысла.
- Маска сигнала относится к каждому отдельному потоку (понятия глобальной маски, распространяемой на все потоки в многопоточном процессе, не существует). Потоки могут независимо друг от друга блокировать и разблокировать разные сигналы, используя новую функцию `pthread_sigmask()`, которая входит в состав программного интерфейса Pthreads. С помощью индивидуальных масок сигнала приложение может контролировать то, какие потоки должны получить сигнал, направленный всему процессу.
- Ядро ведет учет сигналов, которые ожидают доставки как в процесс целиком, так и в каждый отдельный его поток. Вызов `sigpending()` возвращает объединенный список сигналов, ожидающих доставки в процесс и вызывающий поток. Сразу после создания потока список сигналов, которые должны быть ему доставлены, является пустым. Сигнал, направленный в поток, может быть доставлен только этому потоку. Если сигнал блокируется потоком, он будет оставаться в состоянии ожидания до тех пор, пока поток его не разблокирует (или не завершится).
- Если вызов `pthread_mutex_lock()` прерывается обработчиком сигнала, он всегда автоматически перезапускается. Вызов `pthread_mutex_lock()`, прерванный обработчиком сигнала, либо автоматически перезапускается (как это делается в Linux), либо возвращает 0, что свидетельствует о ложном срабатывании (в этом случае хорошо спроектированное приложение должно повторно проверить соответствующий предикат и перезапустить вызов, как описано в подразделе 30.2.3). Поведение, описанное выше, предусмотрено стандартом SUSv3.
- Альтернативный стек сигналов относится к каждому отдельному потоку (см. описание вызова `sigaltstack()` в разделе 21.3). Новый поток не наследует альтернативный стек сигналов от своего создателя.

  Если быть более точным, стандарт SUSv3 гласит, что отдельный альтернативный стек сигналов выделяется для каждой единицы планирования ядра (Kernel Scheduling Entity или KSE). В системах, в которых потоки реализованы в соответствии с моделью 1:1 (как, например, в Linux), каждому потоку соответствует один экземпляр KSE.

33.2.2. Изменение масок сигналов потока

Новый поток наследует от своего создателя копию сигнальной маски. Изменить и/или извлечь эту маску он может с помощью вызова `pthread_sigmask()`.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Если не принимать во внимание тот факт, что функция `pthread_sigmask()` работает с сигнальной маской потока, ее использование ничем не отличается от вызова `sigprocmask()` (см. раздел 20.10).

В стандарте SUSv3 отмечается, что применение вызова sigprocmask() внутри многопоточной программы является неопределенным. Этот вызов нельзя использовать в переносимых многопоточных приложениях. Хотя на практике sigprocmask() и pthread_sigmask() идентичны во многих системах, включая Linux.

33.2.3. Отправка сигнала потоку

Функция `pthread_kill()` отправляет сигнал `sig` другому потоку в том же процессе. Целевой поток определяется аргументом `thread`.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

Возвращает `0` при успешном завершении или положительное число, если произошла ошибка

Поскольку идентификатор потока уникален только в рамках одного процесса (см. раздел 29.5), мы не можем использовать функцию `pthread_kill()` для отправки сигнала потоку из другого процесса.

Функция pthread_kill() реализована с применением системного вызова tgkill(tgid, tid, sig), который доступен только в Linux. Этот вызов отправляет сигнал sig потоку, определяемому по аргументу tid (идентификатору потока ядра, который имеет такой же тип, как тот, что возвращается вызовом gettid()) в рамках группы потоков с идентификатором tgid. Больше подробностей ищите на справочной странице tgkill(2).

Функция `pthread_sigqueue()`, доступная только в Linux, сочетает в себе возможности `pthread_kill()` и `sigqueue()` (см. подраздел 22.8.1): она отправляет сигнал с сопутствующими данными другому потоку в том же процессе.

```
#define _GNU_SOURCE
#include <signal.h>

int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

Возвращает `0` при успешном завершении или положительное число, если случилась ошибка

Как и в случае с `pthread_kill()`, аргумент `sig` определяет сигнал, который нужно отправить, а `thread` — поток-адресат. Аргумент `value` содержит данные, которые сопровождают сигнал; он используется так же, как одноименный аргумент в вызове `sigqueue()`.

Функция pthread_sigqueue() была добавлена в библиотеку glibc в версии 2.11 и требует поддержки на уровне ядра. Эта поддержка предоставляется за счет системного вызова rt_tgsigqueueinfo(), который появился в Linux 2.6.31.

33.2.4. Разумная обработка асинхронных сигналов

В главах 20–22 мы обсуждали различные факторы, которые способны затруднить работу с асинхронно генерируемыми сигналами посредством обработчиков — например, проблемы с реентерабельностью, необходимость перезапуска прерванных системных

вызовов и предотвращение состояния гонки. К этому можно добавить, что ни одну функцию из состава Pthreads нельзя безопасно вызвать внутри обработчика сигнала (см. подраздел 21.1.2). В связи с этим многопоточные программы, сталкивающиеся с асинхронно генерируемыми сигналами, обычно не должны оповещать о доставке сигналов с помощью их обработчиков. Рекомендуется использовать следующий подход.

- Все потоки блокируют любые асинхронные сигналы, которые может получить их процесс. Проще всего это сделать путем блокирования сигналов в главной программе до создания других потоков. Каждый поток, создаваемый после этого, унаследует копию сигнальной маски главной программы.
- Создается отдельный поток, который принимает входящие сигналы с помощью вызовов `sigwaitinfo()`, `sigtimedwait()` или `sigwait()`. Первые два вызова описаны в разделе 22.10. С `sigwait()` мы познакомимся чуть ниже.

Преимуществом данного подхода является то, что асинхронно сгенерированные сигналы принимаются синхронно. По мере получения входящих сигналов выделенный поток может безопасно изменять разделяемые переменные (под защитой мьютекса) и вызывать функции, не адаптированные к работе с асинхронными сигналами. Он также может уведомлять условные переменные и использовать другие механизмы синхронизации и взаимодействия между процессами и потоками.

Функция `sigwait()` ожидает доставки сигнала, входящего в набор, на который указывает аргумент `set`, и затем возвращает его внутри `sig`.

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

Возвращает `0` при успешном завершении
или положительное число при ошибке

Вызов `sigwait()` ведет себя так же, как `sigwaitinfo()`, за исключением следующих моментов.

- Вместо возвращения структуры `siginfo_t`, описывающей сигнал, `sigwait()` возвращает только его номер.
- Возвращаемое значение соответствует другим функциям, связанным с потоками (вместо того чтобы использовать традиционную схему для системных вызовов в UNIX — значения `0` и `-1`).

Если с помощью `sigwait()` сигнал ожидается сразу несколькими потоками, он сможет быть доставлен только одному из них; при этом невозможно предсказать, какому именно.

33.3. Потоки и управление процессами

Как и сигнальный механизм, вызовы `exec()`, `fork()` и `exit()` появились раньше программного интерфейса Pthreads. В следующих подразделах мы рассмотрим некоторые моменты, касающиеся применения этих системных вызовов в многопоточных программах.

Потоки и вызов exec()

Когда любой поток делает вызов одной из функций семейства `exec()`, происходит полная замена вызывающей программы. Все потоки, кроме того, что сделал вызов `exec()`, немедленно исчезают. Ни один из них не выполняет деструкторы для своих данных или

обработчики для освобождения ресурсов. Все мьютексы (уровня процесса) и условные переменные, принадлежащие процессу, тоже перестают существовать. После вызова `exec()` идентификатор оставшегося потока становится неопределенным.

Потоки и вызов fork()

Когда многопоточная программа делает вызов `fork()`, в дочерний процесс копируется только вызывающий поток (его идентификатор в новом процессе будет таким же, как у потока родителя, вызвавшего `fork()`). Остальные потоки дочернего процесса исчезают; деструкторы данных уровня потока и обработчики для очистки ресурсов, принадлежавшие им, игнорируются. Это может привести к различным проблемам.

- Несмотря на то что потомок получает только копию вызывающего потока, состояния глобальных переменных, а также объекты Pthreads, такие как мьютексы и условные переменные, сохраняются (дело в том, что объекты Pthreads находятся внутри родительской памяти, копию которой получает потомок). Это может привести к путанице. Представьте, к примеру, что во время вызова `fork()` другой поток закрыл мьютекс и начал обновлять глобальную структуру данных. В этом случае поток дочернего процесса не сможет открыть мьютекс (поскольку он им не владеет) и заблокируется, если попытается получить к нему доступ. Более того, копия глобальной структуры данных, принадлежащая дочернему процессу, скорее всего, будет находиться в фрагментированном состоянии, потому что поток, который начал ее обновлять, исчез прямо во время этой процедуры.
- Поскольку деструкторы для данных уровня потока и обработчики для освобождения ресурсов не вызываются, в многопоточной программе `fork()` может привести к утечкам памяти в дочернем процессе. Кроме того, данные уровня потока, созданные другими потоками, скорее всего, будут недоступными в дочернем процессе, поскольку оставшийся поток не будет иметь соответствующих указателей.

Ввиду этих проблем вызов `fork()` в многопоточных программах обычно рекомендуется использовать только в том случае, если после него сразу же следует вызов `exec()`. Благодаря `exec()` все объекты Pthreads в дочернем процессе исчезают, поскольку новая программа перезаписывает его память.

На случай, когда приложение должно выполнить какие-то действия между вызовами `fork()` и `exec()`, программный интерфейс Pthreads предоставляет механизм для объявления обработчиков создания нового процесса. Эти обработчики устанавливаются с помощью вызова `pthread_atfork()` следующего вида:

```
pthread_atfork(prepare_func, parent_func, child_func);
```

Каждый такой вызов добавляет `prepare_func` в список функций, которые будут автоматически выполнены (в порядке, обратном их регистрации) перед созданием нового процесса с помощью `fork()`. Аналогично `parent_func` и `child_func` добавляются в список функций, которые будут автоматически вызваны (в порядке их регистрации) в родителе и потомке соответственно прямо перед возвращением `fork()`.

Иногда обработчики создания нового процесса могут пригодиться в библиотечном коде, в котором используются потоки. Без них библиотеки не могли бы работать с приложениями, которые вызывают `fork()` и не учитывают того, что библиотечные функции могут создавать новые потоки.

Потомок, созданный вызовом `fork()`, наследует обработчики создания нового процесса от вызывающего потока. Во время выполнения `exec()` данные обработчики не сохраняются (это невозможно сделать, поскольку `exec()` перезаписывает их код).

Дальнейшие подробности об обработчиках создания нового процесса и примеры их использования можно найти в книге [Butenhof, 1996].

В Linux обработчики создания нового процесса не вызываются, если программа вызывает vfork() из библиотеки NPTL. Это не относится к программам, которые используют реализацию LinuxThreads.

Потоки и вызов exit()

Если какой-либо поток вызовет `exit()` или главная программа выполнит инструкцию `return` (что то же самое), все потоки будут сразу же уничтожены; при этом не будут выполнены деструкторы данных уровня потока и обработчики для освобождения ресурсов.

33.4. Модели реализации потоков

В этом разделе мы обратимся к теории и бегло рассмотрим три разные модели реализации программного интерфейса потоков. Это станет хорошим подспорьем для раздела 33.5, в котором описываются реализации многопоточности в Linux. Разница между упомянутыми выше моделями заключается в способе привязки потоков к *единицам планирования ядра* (KSE), которые используются системой для выделения процессорного времени и других ресурсов (в традиционных реализациях UNIX, которые разрабатывались задолго до появления потоков, термин *«единица планирования ядра»* является синонимом *процесса*).

Реализации потоков уровня пользователя (M:1, или «многие к одному»)

В реализации многопоточности вида M:1 все процедуры, связанные с созданием потока, планированием и синхронизацией (закрытие мьютексов, ожидание условных переменных и т. д.), полностью выполняются внутри процесса пользовательской библиотекой. Ядру ничего не известно о существовании в процессе разных потоков.

Реализации типа M:1 имеют несколько преимуществ. Главное из них заключается в том, что многие операции с потоками, такие как создание, завершение, переключение контекста и работа с мьютексами и условными переменными, имеют высокую производительность, так как не требуют переключения в режим ядра. Кроме того, реализацию типа M:1 можно относительно легко переносить из системы в систему, поскольку библиотека, реализующая потоки, не требует поддержки ядра.

Тем не менее данная модель обладает некоторыми серьезными недостатками.

- Когда поток делает системный вызов, такой как `read()`, поток выполнения переходит от библиотеки уровня пользователя к ядру. Это означает, что в случае блокирования вызова `read()` заблокированными окажутся все потоки в процессе.
- Поскольку ядру ничего не известно об отдельных потоках внутри процесса, оно не может планировать их выполнение, распределяя их между разными процессорами. Также невозможно назначить потоку повышенный приоритет по сравнению с потоками в других процессах, потому что отдельный процесс полностью берет на себя все планирование.

Реализация потоков на уровне ядра (1:1, или «один к одному»)

В реализации многопоточности вида 1:1 каждый поток привязывается к отдельному экземпляру KSE. Ядро планирует выполнение каждого потока отдельно. Процедура синхронизации потоков реализована внутри ядра в виде системных вызовов.

Модель 1:1 исключает недостатки, характерные для реализаций типа M:1. Блокирующий системный вызов не приводит к блокированию всех потоков в процессе, а ядро способно планировать выполнение потоков, распределяя их между разными процессорами.

Однако такие операции, как создание потоков, переключение контекста и синхронизация, работают медленней, поскольку им требуется переключение в режим ядра. Более того, дополнительные ресурсы, необходимые на выделение отдельных экземпляров KSE для каждого отдельного потока в приложении, могут существенно загрузить планировщик ядра, что негативно скажется на общей производительности системы.

Но, несмотря на эти недостатки, модель типа 1:1 обычно является более предпочтительной по сравнению с M:1. Именно она применяется в обеих реализациях многопоточности в Linux — LinuxThreads и NPTL

Во время разработки библиотеки NPTL существенное внимание уделялось переписыванию планировщика ядра и проектированию такой модели многопоточности, которая бы позволила эффективно выполнять процессы, состоящие из тысяч отдельных потоков. Последующее тестирование показало, что данная цель была достигнута.

Двухуровневая реализация потоков (M:N, или «многие ко многим»)

Реализация многопоточности вида M:N призвана совместить в себе преимущества моделей 1:1 и M:1 и в то же время устранить их недостатки.

В модели M:N каждый процесс может содержать несколько экземпляров KSE, к каждому из которых может быть привязано несколько потоков. Такая структура позволяет ядру распределять потоки приложения между процессорами, избавляясь от потенциальных проблем с масштабированием, которые присущи приложениям с большим количеством потоков.

Главным недостатком модели типа M:N является ее сложность. Задача планирования потоков ложится как на ядро, так и на библиотеку, работающую на уровне пользователя; эти две сущности должны взаимодействовать между собой и обмениваться информацией. Управление сигналами в соответствии с требованиями стандарта SUSv3 тоже является непростой задачей, если задействовать реализацию вида M:N.

Реализация вида M:N изначально рассматривалась в качестве основной для библиотеки NPTL, но была отклонена, так как требовала внесения слишком объемных и, вероятно, излишних изменений в ядро, особенно учитывая способность Linux хорошо масштабироваться при работе с большим количеством экземпляров KSE.

33.5. Разные реализации POSIX-потоков в Linux

Linux предоставляет две основные реализации программного интерфейса Pthreads.

- *LinuxThreads*. Это первая реализация многопоточности в Linux, разработанная Хавьером Лероем.
- *NPTL (Native POSIX Threads Library)*. Это современная реализация многопоточности в Linux, разработанная Ульрихом Дреппером и Инго Молнаром в качестве преемника LinuxThreads. NPTL отличается повышенной производительностью по сравнению с LinuxThreads и более строго следует спецификации Pthreads, описанной в стандарте SUSv3. Поддержка библиотеки NPTL требовала внесения в ядро изменений, которые появились в версии Linux 2.6.

Некоторое время в качестве преемника библиотеки LinuxThreads рассматривали другую реализацию под названием NGPT (Next Generation POSIX Threads — POSIX-потоки следующего поколения), разработанную в компании IBM. Библиотека NGPT использовала модель M:N и демонстрировала существенный прирост в производительности по сравнению с LinuxThreads. Однако разработчики NPTL решили заняться новой реализацией. И это

решение себя оправдало, так как модель вида 1:1, на которой основана библиотека NPTL, показывала лучшие результаты, чем NGPT. С выпуском NPTL разработка проекта NGPT была прекращена.

В следующих разделах мы подробно рассмотрим эти две реализации и выделим аспекты, в которых они расходятся с требованиями стандарта SUSv3 для спецификации Pthreads.

На данном этапе стоит отметить, что библиотека LinuxThreads уже считается устаревшей; она не поддерживается в `glibc` версии 2.4 и выше. Все нововведения, касающиеся потоков, добавляются в библиотеку NPTL.

33.5.1. LinuxThreads

На протяжении многих лет библиотека LinuxThreads была основной реализацией многопоточности в Linux, и ее хватало для написания различных многопоточных приложений. Ниже перечислены ее главные особенности.

- Потоки создаются с помощью вызова `clone()`, для которого указываются следующие флаги:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

- Это означает, что потоки в LinuxThreads разделяют виртуальную память, файловые дескрипторы, атрибуты файловой системы (umask, корневой и текущий каталог) и действия сигналов. При этом идентификаторы текущего и родительского процессов не разделяются.
- Помимо потоков, создаваемых самим приложением, LinuxThreads создает дополнительный «управляющий» поток, который отвечает за создание и завершение потоков.
- Внутренняя работа LinuxThreads основана на сигналах. Если ядро поддерживает сигналы реального времени (Linux 2.2 и выше), эта библиотека использует первые три из них. В более старых ядрах применяются сигналы `SIGUSR1` и `SIGUSR2`. Сигналы этого вида недоступны на уровне приложения (такой подход приводит к значительным задержкам при выполнении различных операций синхронизации потоков).

Расхождения библиотеки LinuxThreads со спецификацией

LinuxThreads не соответствует спецификации Pthreads из стандарта SUSv3 в целом ряде аспектов (реализация LinuxThreads была ограничена возможностями ядра, доступными на момент ее разработки; она была совместима настолько, насколько это было возможно в тех условиях). Расхождения со спецификацией перечислены в следующем списке.

- Вызов `getpid()` возвращает уникальное значение для каждого потока в процессе. Эта функция показывает, что все потоки, кроме главного, создаются управляющим потоком процесса (то есть `getppid()` возвращает идентификатор управляющего потока). Если вызвать `getppid()` в других потоках, она вернет то же значение, которое можно было бы получить в главном потоке.
- Если один поток создает потомка с помощью вызова `fork()`, остальные потоки должны иметь возможность получить код завершения этого потомка, используя вызов `wait()` (или аналогичный ему). Но на практике это не так; только поток, создавший дочерний процесс, может ожидать его завершения.
- Стандарт SUSv3 требует, чтобы при вызове потоком `exec()` все остальные потоки завершали свою работу. Но если вызов `exec()` — из любого потока, кроме главного, итоговый процесс будет иметь такой же идентификатор, как и вызывающий поток, —

то есть идентификатор этого процесса будет отличаться от аналогичного идентификатора главного потока. Согласно стандарту SUSv3 они должны совпадать.

- Потоки не разделяют учетные данные (идентификаторы пользователя и группы). Когда многопоточный процесс выполняет программу, которая устанавливает идентификатор пользователя, может случиться так, что потоки утратят возможность обмениваться сигналами с помощью вызова `pthread_kill()`; причина может быть в том, что учетные данные потоков были изменены, в результате чего один поток больше не имеет привилегий для отправки сигнала другому потоку (см. рис. 20.2). Кроме того, библиотека LinuxThreads использует сигналы для внутренней работы, поэтому при изменении учетных данных потока различные операции программного интерфейса Pthreads могут завершиться неудачей или зависнуть.
- Не соблюдаются различные аспекты спецификации SUSv3, касающиеся взаимодействия между потоками и сигналами.
  - Сигнал, отправленный процессу с помощью вызовов `kill()` или `sigqueue()`, должен быть доставлен произвольному потоку в этом процессе, который обработает его неблокирующим способом. Однако в библиотеке LinuxThreads потоки имеют разные идентификаторы процессов, поэтому сигнал можно направить только какому-то определенному потоку. И даже если сигнал блокируется только одним этим потоком, он все равно остается в режиме ожидания.
  - LinuxThreads не поддерживает сигналы, которые ожидают весь процесс целиком; ожидающие сигналы доступны только на уровне отдельных потоков.
  - Если сигнал направлен группе процессов, которая содержит многопоточное приложение, он будет обработан всеми потоками этого приложения (то есть теми из них, которые установили соответствующий обработчик), а не каким-то одним из них (выбранным произвольным образом). Такой сигнал, например, можно сгенерировать в терминале с помощью комбинации клавиш, которая предназначена для управления активной группой процессов.
  - Атрибуты альтернативного стека сигналов (которые устанавливаются с помощью `sigaltstack()`) относятся к отдельным потокам. Но, поскольку новый поток ошибочно наследует эти параметры от потока, вызвавшего `pthread_create()`, альтернативный стек сигналов становится общим для них обоих. Стандарт SUSv3 требует, чтобы новый поток по умолчанию не имел альтернативного стека сигналов. Это несоответствие, допущенное библиотекой LinuxThreads, может привести к непредвиденным результатам (например, может произойти сбой в программе), если оба потока одновременно станут обрабатывать разные сигналы в общем альтернативном стеке. Часто эта проблема плохо поддается воспроизведению и отладке, так как ее появление зависит от редкого события, когда два сигнала обрабатываются в один и тот же момент.

  В программе, которая использует LinuxThreads, новый поток может сделать вызов sigaltstack(), чтобы проверить, отличается ли его альтернативный стек сигналов от стека его создателя (как вариант, стек может и вовсе отсутствовать). Однако переносимые приложения (и библиотечные функции для создания потоков) не будут знать о таком подходе, поскольку в других системах это не является обязательным требованием. Но даже если мы воспользуемся данной методикой, это не избавит нас от возможности возникновения состояния гонки: новый поток может получить и обработать сигнал из альтернативного стека до того, как у него появится возможность вызвать sigaltstack().

- Потоки не разделяют общие идентификаторы сессии и группы процессов. Вызовы `setsid()` и `setpgid()` не могут использоваться для изменения сессии или принадлежности к группе процессов в многопоточном приложении.

- Блокировки записей, устанавливаемые вызовом `fcntl()`, не разделяются. Пересекающиеся запросы блокировки того же типа не объединяются.
- Потоки не разделяют ограничения на ресурсы. Стандарт SUSv3 гласит, что ограничения на ресурсы являются атрибутами уровня процесса.
- Процессорное время и сведения о потреблении ресурсов, возвращаемые вызовами `times()` и `getrusage()` соответственно, относятся к отдельным потокам. Итоговые значения, возвращаемые этими системными вызовами, должны относиться ко всему процессу целиком.
- Некоторые версии команды `ps(1)` показывают все потоки процесса (включая управляющий поток) в виде отдельных элементов со своими собственными идентификаторами.
- Потоки не разделяют атрибут `nice`, устанавливаемый вызовом `setpriority()`.
- Интервальные таймеры, создаваемые вызовом `setitimer()`, не разделяются между потоками.
- Потоки не разделяют значения отмены семафоров (`semadj`) в System V.

Другие проблемы, присущие библиотеке LinuxThreads

Помимо расхождений со стандартом SUSv3, перечисленных выше, библиотека LinuxThreads имеет следующие недостатки.

- Если управляющий поток завершился, оставшиеся потоки придется утилизировать вручную.
- Дамп памяти многопоточной программы может содержать не все потоки процесса (или всего один поток, который спровоцировал создание дампа памяти).
- Нестандартная операция `ioctl() TIOCNOTTY` может убрать привязку процесса к управляющему терминалу, только если она вызвана из главного потока.

33.5.2. Библиотека NPTL

Библиотека NPTL была создана с целью устранения большинства недостатков, присущих LinuxThreads. В частности, тех, которые перечислены ниже.

- NPTL более строго следует спецификации Pthreads из стандарта SUSv3.
- По сравнению с LinuxThreads NPTL обеспечивает лучшее масштабирование приложений, использующих большое количество потоков.

  Библиотека NPTL позволяет приложениям создавать большое число потоков, которое может достигать 100 000. В LinuxThreads ограничение на количество потоков обычно составляет несколько тысяч (следует признать, что столько потоков требуется очень немногим программам).

Работа над реализацией библиотеки NPTL началась в 2002 году и продолжалась около года. Параллельно с этим в ядро Linux вносились различные изменения, чтобы удовлетворить требования NPTL. Среди этих изменений, появившихся в Linux 2.6, можно выделить следующие:

- улучшения в реализации групп потоков (см. раздел 28.2.1);
- добавление фьютексов в качестве средства синхронизации (это универсальный механизм, который разрабатывался не только для NPTL);
- добавление новых системных вызовов (`get_thread_area()` и `set_thread_area()`) для поддержки локального хранилища на уровне потока;
- поддержку множественных потоков в дампе памяти и отладку многопоточных процессов;

- изменения для поддержки управления сигналами в соответствии с моделью Pthreads;
- добавление нового системного вызова `exit_group()` для завершения всех потоков в процессе (начиная с glibc 2.3, вызов `_exit()`, а значит, и библиотечная функция `exit()` превратились в обертку вокруг `exit_group()`, в то время как функция `pthread_exit()` на самом деле выполняет в ядре системный вызов `_exit()`, что приводит к завершению только вызывающего потока);
- для эффективного планирования большого количества (то есть тысяч) экземпляров KSE переписан планировщик ядра;
- улучшена производительность кода ядра, который отвечает за завершение процессов;
- расширен системный вызов `clone()` (см. раздел 28.2).

Основные особенности внутреннего устройства библиотеки NPTL таковы.

- Потоки создаются с помощью вызова `clone()`, для которого указаны следующие флаги:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD |
CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

  NPTL-потоки разделяют всю ту же информацию, что и потоки в LinuxThreads, и не только. Флаг `CLONE_THREAD` означает, что поток помещается в ту же группу, что и его создатель, и имеет с ним общие идентификаторы текущего и родительского процессов. Благодаря флагу `CLONE_SYSVSEM` поток разделяет со своим создателем значения отмены семафоров в System V.

  Если воспользоваться командой ps(1) для вывода многопоточного процесса, основанного на NPTL, результатом будет всего одна строчка. Чтобы увидеть информацию о потоках внутри процесса, можно указать параметр ps -L.

- Внутри библиотеки используются первые два сигнала реального времени, недоступные обычным приложениям.

  Один из этих сигналов отвечает за отмену потоков. Другой применяется в механизме, который гарантирует наличие одних и тех же идентификаторов пользователя и группы у всех потоков внутри одного процесса. Необходимость в этом механизме возникает из-за того, что на уровне ядра потоки имеют разные учетные данные. Следовательно, библиотека NPTL проделывает некоторую работу в функции-обертке для каждого системного вызова, который изменяет идентификаторы пользователя и группы (setuid(), setresuid() и аналогичные вызовы для группы), что приводит к изменению идентификаторов для всех потоков в процессе.

- В отличие от LinuxThreads библиотека NPTL не применяет управляющие потоки.

Соответствие библиотеки NPTL стандартам

Благодаря этим изменениям библиотека NPTL намного строже следует стандарту SUSv3, чем LinuxThreads. На момент написания данной книги все еще остается следующее расхождение.

- Потоки не разделяют значение `nice`.

Дополнительные расхождения имелись в ядрах, предшествовавших версии 2.6.*x*:

- В ядрах версии ниже 2.6.16 альтернативный стек сигналов относился к отдельным потокам, однако новый поток ошибочно наследовал его атрибуты (которые устанавливаются с помощью вызова `sigaltstack()`) от своего создателя, вызвавшего `pthread_create()`; в результате оба потока имели общий альтернативный стек сигналов.
- В ядрах версии ниже 2.6.16 только лидирующий (то есть главный) поток мог начать новую сессию с помощью вызова `setsid()`.

- В ядрах версии ниже 2.6.16 только лидирующий поток мог использовать `setpgid()`, чтобы сделать свой процесс лидером в группе.
- В ядрах версии ниже 2.6.12 интервальные таймеры, созданные с помощью вызова `setitimer()`, не разделялись между потоками процесса.
- В ядрах версии ниже 2.6.10 потоки процесса не разделяли параметры ограничения ресурсов.
- В ядрах версии ниже 2.6.9 процессорное время и сведения о потреблении ресурсов, возвращаемые вызовами `times()` и `getrusage()` соответственно, относились к отдельным потокам.

Библиотека NPTL проектировалась так, чтобы быть совместимой с LinuxThreads на уровне двоичного интерфейса (Application Binary Interface, ABI). Это означает, что программы, скомпонованные с библиотекой GNU C, которая поддерживала LinuxThreads, не нужно заново компоновать для использования NPTL. Однако с переходом на NPTL поведение программы может измениться — в основном из-за более строгого соблюдения спецификации Pthreads из стандарта SUSv3.

33.5.3. Выбор между разными реализациями многопоточности

Некоторые дистрибутивы Linux поставляются вместе с библиотекой GNU C, которая предоставляет как LinuxThreads, так и NPTL; реализация по умолчанию определяется динамическим компоновщиком в соответствии с текущим ядром системы (такие дистрибутивы уже устарели, потому что glibc, начиная с версии 2.4, больше не предоставляет LinuxThreads). Следовательно, у нас может возникнуть необходимость ответить на следующие вопросы.

- Какая реализация многопоточности доступна в текущем дистрибутиве Linux?
- Какая реализация используется по умолчанию и каким образом нам самим выбрать подходящую библиотеку для нашей программы, если дистрибутив предоставляет как LinuxThreads, так и NPTL?

Определение доступной реализации многопоточности

Мы можем воспользоваться несколькими методиками для обнаружения доступной реализации многопоточности или, если их сразу две, реализации, которая используется по умолчанию при запуске программы.

В системе с glibc версии 2.3.2 и выше это можно сделать с помощью следующей команды:

```
$ getconf GNU_LIBPTHREAD_VERSION
```

Если библиотека NPTL является единственной реализацией в системе или используется по умолчанию, мы получим следующую строку в качестве результата:

```
NPTL 2.3.4
```

> Начиная с glibc 2.3.2, программа может получить похожие сведения с помощью функции confstr(3), которая возвращает значение переменной _CS_GNU_LIBPTHREAD_VERSION (устанавливаемое самой библиотекой glibc).

В системах с более ранними версиями библиотек GNU C необходимо проделать немного дополнительной работы. Во-первых, с помощью следующей команды можно

узнать путь к библиотеке GNU C, который использовался при запуске программы (здесь в качестве примера мы взяли стандартную команду `ls`, полный путь к которой — `/bin/ls`):

```
$ ldd /bin/ls | grep libc.so
libc.so.6 => /lib/tls/libc.so.6 (0x40050000)
```

> Чуть больше о программе ldd (list dynamic dependencies — «список динамических зависимостей») можно узнать в разделе 41.5.

Путь к библиотеке GNU C указан после `=>`. Если выполнить его в качестве команды, glibc сама выведет целый набор информации. Мы можем отфильтровать полученные сведения с помощью команды `egrep`, чтобы выбрать строки с данными о реализации многопоточности:

```
$ /lib/tls/libc.so.6 | egrep -i 'threads|nptl'
Native POSIX Threads Library by Ulrich Drepper et al
```

Мы указали для команды `egrep` регулярное выражение `nptl`, поскольку некоторые версии glibc, поддерживающие NPTL, выводят информацию следующего вида:

```
NPTL 0.61 by Ulrich Drepper
```

Поскольку путь к библиотеке glibc может варьироваться в зависимости от дистрибутива, мы можем воспользоваться консольной командой, которая точно так же выведет сведения о текущей реализации многопоточности в любой Linux-системе:

```
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
Native POSIX Threads Library by Ulrich Drepper et al
```

Выбор реализации многопоточности, которая будет использоваться в программе

Если Linux-система предоставляет как NPTL, так и LinuxThreads, иногда возникает необходимость в непосредственном выборе реализации многопоточности. Наиболее распространенным примером этого может служить ситуация, когда у нас есть старая программа, работа которой зависит от неких (вероятно, нестандартных) аспектов библиотеки LinuxThreads, поэтому мы хотим заставить ее работать именно с этой реализаций вместо NPTL.

Для этой задачи можно воспользоваться специальной переменной среды, которую учитывает динамический компоновщик: `LD_ASSUME_KERNEL`. Как видно по ее имени, она делает так, чтобы компоновщик выполнялся в режиме совместимости с определенным ядром Linux. Если указать версию ядра, которая не поддерживает NPTL (например, 2.2.5), это гарантированно приведет к задействованию LinuxThreads. Таким образом, мы можем запустить многопоточную программу на основе LinuxThreads с помощью следующей команды:

```
$ LD_ASSUME_KERNEL=2.2.5 ./prog
```

Совместив установку этой переменной среды с командой для вывода текущей реализации многопоточности, описанной выше, мы получим следующий результат:

```
$ export LD_ASSUME_KERNEL=2.2.5
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
linuxthreads-0.10 by Xavier Leroy
```

Диапазон версий ядра, которые можно указать переменной LD_ASSUME_KERNEL, имеет определенные ограничения. В некоторых популярных дистрибутивах, предоставляющих как NPTL, так и LinuxThreads, для выбора последней достаточно указать номер версии 2.2.5. Больше информации о применении этой переменной среды можно найти по адресу people.redhat.com/drepper/assumekernel.html.

33.6. Продвинутые возможности программного интерфейса Pthreads

Среди продвинутых возможностей программного интерфейса Pthreads можно выделить следующие.

- *Планирование в режиме реального времени.* Мы можем устанавливать политику и приоритеты планирования в режиме реального времени. Это аналогично системным вызовам для планирования процессов, описанным в разделе 35.3.
- *разделяемые мьютексы и условные переменные процесса.* Стандарт SUSv3 предусматривает параметр, который позволяет сделать мьютексы и условные переменные общими для нескольких процессов (а не просто для разных потоков одного процесса). В этом случае условная переменная или мьютекс должны быть выделены на участке памяти, разделяемом процессами. Данная возможность поддерживается библиотекой NPTL.
- *Дополнительные средства синхронизации потоков.* Речь идет о барьерах, блокировках для чтения/записи и циклических блокировках.

Подробные сведения обо всех этих возможностях можно найти в книге [Butenhof, 1996].

33.7. Резюме

Потоки плохо сочетаются с асинхронными сигналами; при проектировании многопоточных приложений использования сигналов следует избегать любыми путями. Но если это сделать не удается, наиболее безопасным подходом является блокирование сигналов во всех потоках и выделение отдельного потока специально для приема сигналов с помощью вызова `sigwait()` (или аналогичного). Этот поток впоследствии сможет безопасно выполнять такие действия как изменение разделяемых переменных (с применением мьютексов) и вызова функций, не рассчитанных на работу с асинхронными сигналами.

Наиболее распространенными реализациями многопоточности в Linux являются библиотеки LinuxThreads и NPTL. Первая была создана много лет назад, но из-за ряда расхождений с требованиями стандарта SUSv3 считается устаревшей. Современная реализация, NPTL, более строго следует стандарту SUSv3 и демонстрирует намного лучшую производительность; именно она предоставляется по умолчанию в современных дистрибутивах Linux.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 29.10.

Автор LinuxThreads разместил документацию к своей библиотеке на веб-странице pauillac.inria.fr/~xleroy/linuxthreads/. Реализация NPTL описана ее разработчиками в (уже несколько устаревшем) документе, электронная версия которого доступна по адресу people.redhat.com/drepper/nptl-design.pdf.

33.8. Упражнения

33.1. Напишите программу для демонстрации того, что разные потоки в одном и том же процессе могут иметь разные наборы ожидающих сигналов, которые возвращаются вызовом `sigpending()`. Вы можете воспользоваться функцией `pthread_kill()`, чтобы отправить разные сигналы двум разным потокам; эти потоки их заблокируют, и затем каждый из них вызовет `sigpending()` и выведет информацию об ожидающих сигналах (вам могут пригодиться функции из листинга 20.4).

33.2. Представьте, что поток создает потомка с помощью вызова `fork()`. Гарантируется ли доставка итогового сигнала `SIGCHLD` потоку, вызвавшему `fork()` (об остальных потоках внутри процесса речь не идет), если потомок завершится?

34 Группы процессов, сессии и управление заданиями

Группы процессов и сессии составляют двухуровневые иерархические отношения между процессами: первые являются наборами связанных между собой процессов, а вторые — групп. По мере чтения данной главы вы получите более четкое представление о том, что именно означают эти связи в каждом из случаев.

Группы процессов и сессии — это абстракции, созданные для поддержки управления заданиями в командной оболочке; это позволяет выполнять команды как в интерактивном, так и в фоновом режиме. Понятие *«задание»* часто используется в качестве синонима *группы процессов*.

Группы процессов, сессии и управление заданиями будут рассмотрены в этой главе.

34.1. Краткий обзор

Группа процессов — это набор из одного или нескольких процессов с одним и тем же идентификатором PGID (process group identifier — *идентификатор группы процессов*). Этот идентификатор представляет собой число того же типа (`pid_t`), что и идентификатор процесса. В каждой группе есть свой *лидер* — процесс, который создал эту группу и чей идентификатор становится ее PGID. Новый процесс наследует PGID своего родителя.

Группа процессов имеет *жизненный цикл* — период времени, который начинается с создания группы лидером и заканчивается, когда группу покидает ее последний участник. Чтобы покинуть группу, процесс должен либо завершиться, либо присоединиться к другой группе. Лидер группы процессов может не быть ее последним участником.

Сессия — это набор групп процессов. Членство процесса в сессии определяется идентификатором SID (session identifier — *идентификатор сессии*), который по аналогии с PGID является числом типа `pid_t`. *Лидером сессии* является процесс, который ее создал, и чей идентификатор используется в качестве SID. Новый процесс наследует идентификатор SID своего родителя.

Все процессы в сессии разделяют один и тот же *контролирующий терминал*, который устанавливается при первом открытии устройства терминала лидером сессии. Любой контролирующий терминал может быть связан не более чем с одной сессией.

В любой момент времени одна из групп процессов в сессии является *активной*, а остальные — *фоновыми*. Только процессы из активной группы могут считывать ввод из контролирующего терминала. Сигнал, сгенерированный пользователем в терминале с помощью комбинации клавиш, передается всем участникам активной группы процессов. В число этих комбинаций входят *прерывание* (обычно Ctrl+C), которое генерирует сигнал `SIGINT`, *выход* (обычно Ctrl+\), генерирующий сигнал `SIGQUIT`, и *приостановка* (обычно Ctrl+Z), передающая сигнал `SIGTSTP`.

В результате установления соединения с контролирующим терминалом (то есть открытия) лидер сессии становится контролирующим процессом этого терминала. Главной особенностью такого процесса является то, что при утрате соединения с терминалом ядро отправляет ему сигнал `SIGHUP`.

В Linux идентификаторы PGID и SID любого процесса можно определить с помощью файлов вида /proc/PID/stat. Мы также можем определить идентификатор устройства контролирующего терминала (имеющий вид десятичного числа, состоящего из основного и дополнительного идентификаторов) и идентификатор его процесса. Больше подробностей ищите на справочной странице proc(5).

Основное назначение групп процессов и сессий связано с управлением заданиями в командной оболочке. Чтобы прояснить понимание данных концепций, рассмотрим пример из этой области. Когда пользователь входит в систему в интерактивном режиме, он задействует контролирующий терминал. Командная оболочка, задействованная при этом, становится лидером сессии, контролирующим процессом для терминала, а также единственным участником своей собственной группы процессов. Любые команды или конвейеры команд, запущенных из этой оболочки, становятся новыми процессами и помещаются в новую группу (изначально они являются единственными ее участниками, однако в нее также попадают потомки этих процессов). Команды или конвейеры, после которых указывается амперсанд (**&**), попадают в фоновую группу процессов. В противном случае их группа будет активной. Все процессы, созданные во время терминальной сессии, являются ее частью.

В оконной среде контролирующим является псевдотерминал, и каждое его окно имеет отдельную сессию, лидером и контролирующим процессом которой становится его командная оболочка.

Группы процессов иногда можно применять не только для управления заданиями. Они имеют два полезных свойства: родительский процесс может ожидать любого из своих потомков в определенной группе (см. подраздел 26.1.2), и сигнал может быть отправлен сразу всем участникам группы (см. раздел 20.5).

На рис. 34.1 показана группа процессов и отношения между различными процессами в рамках одной сессии, которые являются результатом выполнения следующих команд:

```
$ echo $$                  Выводим идентификатор процесса командной оболочки
400
$ find / 2> /dev/null | wc -l &     Результатом будут два процесса в фоновой группе
[1] 659
$ sort < longlist | uniq –c         Результатом будут два процесса в активной группе
```

На этом этапе, помимо командной оболочки (bash), запущены программы `find`, `wc`, `sort` и `uniq`.

34.2. Группы процессов

Каждый процесс имеет числовой идентификатор PGID, который определяет его принадлежность к той или иной группе. Новый процесс наследует PGID своего родителя. Для получения текущего идентификатора группы используется вызов `getpgrp()`.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Всегда успешно возвращает идентификатор группы вызывающего процесса

Значение, возвращенное `getpgrp()`, совпадает с идентификатором вызывающего процесса, этот процесс является лидером своей группы.

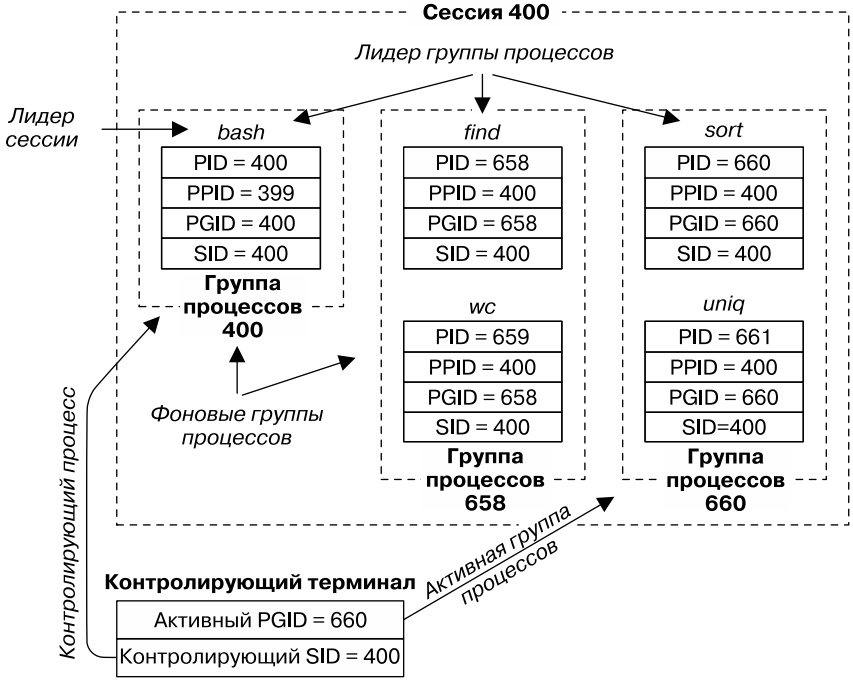


Рис. 34.1. Отношения между группой процессов, сессиями и контролирующим терминалом

Системный вызов `setpgid()` позволяет сменить группу процесса с идентификатором `pid` на значение, указанное в аргументе `pgid`.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Если в качестве `pid` указать `0`, изменится идентификатор PGID вызывающего процесса. Если присвоить `0` аргументу `pgid`, PGID процесса, указанного с помощью `pid`, станет таким же, как идентификатор этого процесса. Таким образом, следующие вызовы `setpgid()` являются эквивалентными.

```
setpgid(0, 0);
setpgid(getpid(), 0);
setpgid(getpid(), getpid());
```

Если аргументы `pid` и `pgid` указывают на один и тот же процесс (то есть если `pgid` равен 0 или идентификатору процесса, заданному с помощью `pid`), создается новая группа процессов и ее лидером становится заданный процесс (то есть PGID и идентификатор процесса совпадают). Если в этих аргументах указаны разные значения (то есть когда `pgid` не равен 0 и не совпадает с идентификатором процесса, заданного с помощью `pid`), вызов `setpgid()` используется для перемещения процессов между группами. Обычно вызовы `setpgid()` и `setsid()` (описан в разделе 34.3), выполняют такие программы, как командная оболочка или `login(1)`. В разделе 37.2 вы увидите,

что вызов `setsid()` используется на одном из этапов превращения процесса в процесс-демон.

Вызов `setpgid()` имеет несколько ограничений.

- Аргумент `pid` может указывать только на вызывающий процесс или его потомка. Нарушение этого правила приводит к ошибке `ESRCH`.
- При перемещении процесса между группами вызывающий процесс, процесс, указанный с помощью `pid` (которые могут совпадать), и текущая группа должны принадлежать к одной и той же сессии. Нарушение этого правила приводит к ошибке `EPERM`.
- Процесс, на который указывает аргумент `pid`, не должен быть лидером сессии. Нарушение этого правила приводит к ошибке `EPERM`.
- Процесс не должен менять идентификатор PGID своего потомка, если тот уже вызвал `exec()`. Нарушение этого правила приводит к ошибке `EACCES`. Данное ограничение связано с тем, что программа может быть введена в заблуждение, если изменить идентификатор группы процессов уже после ее запуска.

Использование setpgid() для управления заданиями в командной оболочке

Тот факт, что процессу запрещено менять идентификатор PGID своего потомка, если тот успел выполнить вызов `exec()`, влияет на архитектуру командных оболочек, управляющих заданиями, которые, в свою очередь, имеют следующие требования.

- Все процессы в задании (то есть в команде или конвейере) должны быть помещены в одну и ту же группу (желаемый результат можно увидеть на примере двух групп процессов, созданных оболочкой `bash` на рис. 34.1). На этом этапе командная оболочка может использовать вызов `killpg()` (или, как вариант, `kill()` с отрицательным аргументом `pid`) для одновременной отправки всем членам группы сигналов, управляющих заданием. Естественно, этот этап необходимо выполнить до отправки сигналов подобного рода.
- Все дочерние процессы должны быть доставлены в группу до запуска программы, поскольку сама программа ничего не знает о манипуляциях с идентификатором группы процессов.

Родитель или поток любого процесса в задании могут использовать вызов `setpgid()` для изменения PGID дочернего процесса. Но поскольку невозможно предсказать, в каком порядке будут выполняться родитель и потомок после вызова `fork()` (см. раздел 24.4), мы не можем полагаться на то, что родитель изменит PGID до того, как потомок выполнит `exec()`; мы также не можем быть уверены в том, что потомок изменит свой PGID до того, как родитель попытается передать ему какой-нибудь сигнал для управления заданием (зависимость от любого из этих сценариев приведет к состоянию гонки). Следовательно, командные оболочки, управляющие заданиями, написаны так, чтобы сразу после вызова `fork()` родитель и его дочерний процесс изменяли идентификатор потомка PGID с помощью `setpgid()`. При этом родитель игнорирует любые ошибки типа `EACCES`, возвращаемые вызовом `setpgid()`. Иными словами, в любой командной оболочке, управляющей заданиями, можно найти код, похожий на содержимое листинга 34.1.

Листинг 34.1. Как командные оболочки, управляющие заданиями, устанавливают идентификатор PGID для дочернего процесса

```
pid_t childPid;
pid_t pipelinePgid;    /* PGID, к которому будет назначен процесс в конвейере */

/* Другой код */
```

```
childPid = fork();
switch (childPid) {
case -1: /* fork() завершается неудачей */
    /* Обрабатываем ошибку */

case 0: /* Потомок */
    if (setpgid(0, pipelinePgid) == -1)
        /* Обрабатываем ошибку */
    /* Потомок продолжает работу, запуская заданную программу */

default: /* Родитель (командная оболочка) */
    if (setpgid(childPid, pipelinePgid) == -1 && errno != EACCES)
        /* Обрабатываем ошибку */
    /* Родитель продолжает работу, выполняя другие задания */
}
```

На самом деле все немного сложнее, чем показано на рис. 34.1, поскольку при создании процесса для конвейера родительская командная оболочка записывает идентификатор первого процесса в цепочке команд и использует его в качестве PGID (`pipelinePgid`) для всех процессов в группе.

Другие (устаревшие) интерфейсы для извлечения и изменения идентификаторов PGID

Разница в суффиксах у системных вызовов `getpgrp()` и `setpgid()` заслуживает отдельного объяснения.

Изначально система 4.2BSD предоставляла системный вызов `getprgp(pid)`, который возвращал идентификатор PGID для процесса, заданного с помощью аргумента `pid`. На практике в качестве этого аргумента всегда указывали вызывающий процесс. По этой причине комитет, стоящий за разработкой стандарта POSIX, решил, что данный вызов является более сложным, чем следует, и заменил его вызовом `getpgrp()` из System V, который не принимает никаких аргументов и возвращает PGID для текущего процесса.

Для изменения идентификатора PGID система предоставляла вызов `setpgrp(pid, pgid)`, который принципом своей работы напоминал `setpgid()`. Ключевое различие было в том, что с помощью вызова `setpgrp()` из состава BSD можно было поменять идентификатор PGID на любое значение (ранее отмечалось, что `setpgid()` не может переместить процесс в группу, которая находится в другой сессии). Это приводило к некоторым проблемам с безопасностью и обеспечивало гибкость, излишнюю для реализации управления заданиями. В результате комитет, разрабатывающий стандарт POSIX, остановился на более строгой функции, назвав ее `setpgid()`.

Но и это еще не все. Стандарт SUSv3 содержит функцию `getpgid(pid)` с такой же семантикой, как у старого вызова `getpgrp()` из состава BSD. В нем же ненавязчиво предлагается альтернатива — версия вызова `setpgrp()`, берущая свое начало в System V; она не принимает никаких аргументов и практически является эквивалентом вызова `setpgid(0, 0)`.

Системные вызовы `setpgid()` и `getpgrp()`, описанные выше, удовлетворяют нужды по реализации управления заданиями в командной оболочке, однако в дополнение к ним Linux, как и большинство других UNIX-систем, предоставляет вызовы `getpgid(pid)` и `setpgrp(void)`. Из соображений обратной совместимости многие BSD-системы продолжают поддерживать вызов `setprgp(pid, pgid)` в качестве эквивалента `setpgid(pid, pgid)`.

Если при компиляции программы явно объявить макрос `_BSD_SOURCE` для проверки доступных возможностей, библиотеке `glibc` предоставит вместо стандартных вызовов `setpgrp()` и `getpgrp()` их версии из состава BSD.

34.3. Сессии

Сессия — это набор групп процессов. Принадлежность процесса к сессии определяется числовым идентификатором SID (session ID). Новый процесс наследует SID своего родителя. Системный вызов `getsid()` возвращает идентификатор SID процесса, указанного с помощью аргумента `pid`.

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Возвращает SID заданного процесса или `-1` (`pid_t`), если случилась ошибка

Если аргумент `pid` равен `0`, `getsid()` возвращает идентификатор сессии вызывающего процесса.

В некоторых реализациях UNIX (например, в HP-UX 11) вызов getsid() можно использовать для получения идентификатора SID только того процесса, который находится в одной сессии с вызывающим (данная возможность оговорена в стандарте SUSv3). Иными словами, исходя из успешного или неудачного (ошибка EPERM) выполнения этого вызова, можно понять, принадлежат ли заданный и вызывающий процессы к одной и той же сессии. Это ограничение не действует в Linux и большинстве других систем.

Если вызывающий процесс не является лидером своей группы, вызов `setsid()` создает новую сессию.

```
#include <unistd.h>

pid_t setsid(void);
```

Возвращает идентификатор новой сессии или `-1` (`pid_t`), если случилась ошибка

Создание новой сессии системным вызовом `setsid()` происходит следующим образом.

- Вызывающий процесс становится лидером новой сессии и новой группы процессов внутри нее. Идентификаторы PGID и SID нового процесса получают то же значение, что и сам процесс.
- Вызывающий процесс не имеет контролирующего терминала. Любое соединение с контролирующим терминалом, установленное ранее, разрывается.

Если вызывающий процесс является лидером своей группы, вызов `setsid()` завершается ошибкой `EPERM`. Чтобы этого избежать, проще всего выполнить `fork()` и дать родителю завершиться, после чего вызвать `setsid()` из потомка. Поскольку дочерний процесс наследует идентификатор PGID родителя и получает свой собственный идентификатор, он не может оказаться лидером группы.

Ограничение относительно возможности лидера группы вызывать `setsid()` является необходимым, иначе лидер смог бы переместить себя в другую (новую) сессию, оставляя остальные процессы в исходной сессии (при этом не была бы создана новая группа процессов, поскольку PGID лидера группы по определению совпадает с его собственным идентификатором). Это бы нарушило строгую двухуровневую иерархию сессий и групп процессов, так как все члены группы должны быть частью одной и той же сессии.

При создании нового процесса с помощью вызова fork() ядро выдает ему уникальный идентификатор и проверяет, чтобы этот идентификатор не совпадал с PGID или SID любого существующего процесса. То есть, даже если лидер группы или сессии уже завершился, новый процесс не может повторно использовать его идентификатор, чтобы случайно не стать лидером имеющейся сессии или группы.

В листинге 34.2 демонстрируется использование вызова `setsid()` для создания новой сессии. Чтобы убедиться в том, что данная программа больше не имеет контролирующего терминала, мы пытаемся открыть специальный файл `/dev/tty` (описанный в следующем разделе). Запустив программу, мы увидим следующее:

```
$ ps -p $$ -o 'pid pgid sid command'  $$ – это ID процесса командной оболочки
PID PGID SID COMMAND
12243 12243 12243 bash                Идентификатор процесса оболочки, ее PGID и SID
$ ./t_setsid
$ PID=12352, PGID=12352, SID=12352
ERROR [ENXIO Device not configured] open /dev/tty
```

Как можно видеть по результату, процесс успешно помещает себя в новой группе в рамках новой сессии. Поскольку эта сессия не имеет контролирующего терминала, вызов `open()` завершается неудачей (в предпоследней строке мы видим, что приглашение командной строки смешивается с программным выводом, потому что командная оболочка узнает о том, что родительский процесс завершился после вызова `fork()`, и выводит свое следующее приглашение до завершения потомка).

Листинг 34.2. Создание новой сессии

pgsjc/t_setsid.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (fork() != 0)        /* Выходим в случае ошибки или если это родитель */
    _exit(EXIT_SUCCESS);

    if (setsid() == -1)
        errExit("setsid");

    printf("PID=%ld, PGID=%ld, SID=%ld\n", (long) getpid(),
           (long) getpgrp(), (long) getsid(0));

    if (open("/dev/tty", O_RDWR) == -1)
        errExit("open /dev/tty");
    exit(EXIT_SUCCESS);
}
```

pgsjc/t_setsid.c

34.4. Контролирующие терминалы и контролирующие процессы

Все процессы в сессии могут иметь (единый) контролирующий терминал, который отсутствует на момент создания сессии, но устанавливается, когда ее лидер впервые

открывает терминал, который не является для нее контролирующим (если только при вызове open() не был указан флаг O_NOCTTY). Терминал может быть контролирующим максимум для одной сессии.

> Стандарт SUSv3 содержит функцию tcgetsid(int fd) (объявленную в заголовочном файле <termios.h>), которая возвращает идентификатор сессии, связанной с контролирующим терминалом fd. Эта функция предоставляется библиотекой glibc (и реализована с помощью операции TIOCGSID в вызове ioctl()).

Контролирующий терминал наследуется потомком, созданным с помощью fork(), и сохраняется на протяжении работы вызова exec().

Когда лидер сессии открывает контролирующий терминал, он сразу же становится его контролирующим процессом. Если впоследствии произойдет отключение терминала, ядро уведомит об этом контролирующий процесс, отправив ему сигнал SIGHUP. Более подробно об этом моменте мы поговорим в подразделе 34.6.2.

Если процесс имеет контролирующий терминал, он может получить его файловый дескриптор, открыв специальный файл /dev/tty. Это может пригодиться в ситуации, когда стандартные ввод и вывод перенаправлены, а программа хочет удостовериться, что она соединена с контролирующим терминалом. Данный подход, к примеру, используется в функции getpass(), описанной в разделе 8.5. Если у процесса нет контролирующего терминала, открытие файла /dev/tty завершается ошибкой ENXIO.

Отключение процесса от контролирующего терминала

С помощью операции ioctl(fd, TIOCNOTTY) можно отключить процесс от его контролирующего терминала, указанного в виде файлового дескриптора fd. После этого вызова все попытки открыть файл /dev/tty будут завершаться неудачей (операция TIOCNOTTY не входит в стандарт SUSv3, однако поддерживается в большинстве UNIX-систем).

Если вызывающий процесс является контролирующим для терминала, во время его завершения (см. подраздел 34.6.2) происходит следующее.

1. Все процессы в сессии теряют соединение с контролирующим терминалом.
2. Контролирующий терминал теряет соединение с сессией и впоследствии может стать контролирующим процессом для другой сессии.
3. Ядро шлет всем участникам активной группы процессов сигнал SIGHUP (и SIGCONT), чтобы проинформировать их о потере контролирующего терминала.

Получение пути к контролирующему терминалу: ctermid()

Функция ctermid() возвращает путь к контролирующему терминалу.

```
#include <stdio.h>        /* Объявляет константу L_ctermid */

char *ctermid(char *ttyname);
```

Возвращает указатель на строку, содержащую путь к контролирующему терминалу, или NULL, если путь не удается определить

Функция ctermid() возвращает путь к контролирующему терминалу двумя разными способами: через результат своего выполнения и посредством буфера, на который указывает ttyname.

Если аргумент ttyname не равен NULL, то он должен быть буфером размера как минимум L_ctermid и содержать копию пути. В этом случае значение, возвращаемое функцией,

тоже является указателем на этот буфер. Если аргумент `ttyname` равен `NULL`, то `ctermid()` возвращает указатель на статически выделенный массив, содержащий путь; при этом функция `ctermid()` не является реентерабельной.

В Linux и других реализациях UNIX `ctermid()` обычно возвращает строку **/dev/tty**. Назначение этой функции заключается в упрощении переносимости программы на системы, отличные от UNIX.

34.5. Активные и фоновые группы процессов

Контролирующий терминал несет в себе понятие активной группы процессов. В любой момент времени только один процесс в рамках сессии может быть активным; остальные группы процессов в этой сессии являются фоновыми. Только активная группа процессов может свободно выполнять чтение и запись данных в контролирующем терминале. Сигнал, который генерируется в результате нажатия подходящей комбинации клавиш, передается драйвером терминала участникам активной группы процессов. Мы остановимся на этом подробнее в разделе 34.7.

> Теоретически может возникнуть ситуация, в которой сессия не имеет активной группы процессов. Это может случиться, к примеру, если все процессы в активной группе завершатся и ни один другой процесс этого не заметит и не станет активным. Однако на практике такие ситуации крайне маловероятны. Обычно отслеживанием состояния активной группы занимается командная оболочка; заметив, что такая группа завершила свою работу (через вызов wait()), она сама становится активной.

Функции `tcgetpgrp()` и `tcsetpgrp()` используются соответственно для получения и изменения группы процессов терминала. Они в основном применяются командными оболочками для управления заданиями.

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
```

Возвращает идентификатор активной группы процессов для терминала или `-1`, если случилась ошибка

```
int tcsetpgrp(int fd, pid_t pgid);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

Функция `tcgetpgrp()` возвращает идентификатор активной группы процессов для терминала, заданного с помощью файлового дескриптора `fd`, который должен быть управляющим по отношению к вызывающему процессу.

> Если у текущего терминала нет активной группы процессов, tcgetpgrp() возвращает значение больше 1, которое не совпадает с идентификаторами любой существующей группы (такое поведение описано в стандарте SUSv3).

Функция `tcsetpgrp()` меняет активную группу процессов терминала. Если вызывающий процесс имеет контролирующий терминал, на который указывает файловый дескриптор `fd`, то `tcsetpgrp()` назначает активной группе процессов значение `pgid`, которое должно совпадать с идентификатором PGID одного из процессов, входящих в текущую сессию.

Функции `tcgetpgrp()` и `tcsetpgrp()` являются частью стандарта SUSv3. В Linux, как и во многих других UNIX-системах, они реализованы с помощью двух нестандартных операций для вызова `ioctl()`: `TIOCGPGRP` и `TIOCSPGRP`.

34.6. Сигнал SIGHUP

Когда контролирующий процесс теряет соединение с терминалом, ядро информирует его об этом факте, отправляя ему сигнал `SIGHUP` (кроме того, чтобы обеспечить перезапуск процесса в случае, если тот был ранее остановлен по сигналу, отправляется сигнал `SIGCONT`). Обычно это происходит в одной из двух ситуаций.

- Когда «отключение» обнаруживается драйвером терминала и указывает на потерю сигнала в модеме или терминальном соединении.
- Когда окно терминала закрывается на клиентской стороне. Это случается в результате закрытия последнего открытого дескриптора на серверной стороне псевдотерминала, связанного с его окном.

По умолчанию сигнал `SIGHUP` завершает процесс терминала. Но если вместо этого контролирующий процесс обрабатывает или игнорирует данный сигнал, дальнейшие попытки прочитать информацию из терминала будут возвращать символ конца файла (EOF).

> Стандарт SUSv3 гласит, что если теряется соединение с терминалом и вместе с этим выполняется одно из условий, приводящих к возникновению ошибки EIO во время работы функции read(), то невозможно сказать, какой результат вернет эта функция — символ конца файла или ошибку EIO. Переносимые программы должны учитывать обе возможности. Ситуация, когда read() завершается ошибкой EIO, будет рассмотрена в подразделах 34.7.2 и 34.7.4.

Доставка сигнала `SIGHUP` контролирующему процессу может запустить своеобразную цепную реакцию, в результате которой этот сигнал получают множество других процессов. Это может случиться двумя разными способами.

- В качестве контролирующего процесса обычно выступает командная оболочка. Она устанавливает обработчик сигнала `SIGHUP`, чтобы перед завершением ретранслировать этот сигнал всем заданиям, которые она создала. По умолчанию это приводит к завершению этих заданий, но, перехватив сигнал `SIGHUP`, они будут знать о том, что их оболочка завершила свою работу.
- Во время завершения контролирующего процесса ядро отсоединяет от связанного с ним терминала все процессы из его сессии, которая в итоге тоже отсоединяется от этого терминала (что позволяет использовать его уже в другой сессии). Кроме того, ядро информирует активную группу процессов о потере соединения с управляющим терминалом, отправляя ей сигнал `SIGHUP`.
  Каждый из этих случаев будет подробно рассмотрен в следующих разделах.

> Сигнал SIGHUP имеет и другие применения. В подразделе 34.7.4 вы увидите, что он генерируется, когда группа процессов остается без родителя. Кроме того, SIGHUP принято отправлять вручную для того, чтобы заставить процесс-демон заново себя инициализировать или еще раз прочитать свой конфигурационный файл (демон по определению не имеет контролирующего терминала, поэтому получить сигнал SIGHUP от ядра). Применение сигнала SIGHUP в контексте процессов-демонов будет описано в разделе 37.4.

34.6.1. Обработка сигнала SIGHUP командной оболочкой

В контексте сессии входа в систему командная оболочка обычно является управляющим процессом для терминала. Большинство оболочек написаны таким образом, что в интерактивном режиме они устанавливают обработчик сигнала SIGHUP. Этот обработчик завершает работу командной оболочки, но перед этим отправляет SIGHUP каждой группе процессов (активной и фоновой), созданной этой оболочкой (в зависимости от вида оболочки и от того, остановлено или нет ее задание, за SIGHUP может последовать сигнал SIGCONT). То, как процессы в этих группах отреагируют на сигнал SIGHUP, зависит от конкретного приложения; если не было предусмотрено никаких особых действий, они просто завершаются.

> Некоторые командные оболочки, управляющие заданиями, при завершении (то есть при выходе из системы или при нажатии Ctrl+D в окне терминала) отправляют сигнал SIGHUP еще и фоновым процессам. Так поступают bash и Korn Shell (после того как выводят сообщение при первой попытке выхода).
>
> Чтобы сделать процесс нечувствительным к SIGHUP, можно воспользоваться командой nohup(1); во время запуска процесса она устанавливает для SIGHUP действие SIG_IGN. Команда disown, встроенная в bash, имеет похожее назначение: она удаляет задание из списка заданий командной оболочки, чтобы, когда та завершит свою работу, сигнал SIGHUP доставлен не был.

Воспользуемся программой из листинга 34.3, чтобы показать, что командная оболочка, получив сигнал SIGHUP, передает его всем заданиям, которые она создала. Главная цель этой программы — создать дочерний процесс, после чего приостановить работу родителя и потомка, чтобы перехватить сигнал SIGHUP и, если это удалось, вывести соответствующее сообщение. Если указать для программы необязательный аргумент командной строки (который может представлять собой набор символов), потомок разместится в другой группе процессов (в рамках той же сессии). Это позволит показать, что командная оболочка не передает сигнал SIGHUP группам процессов, которые она не создавала, даже если они находятся внутри одной сессии (поскольку завершающий цикл for является бесконечным, мы используем вызов alarm(), чтобы установить таймер для доставки сигнала SIGALRM; появление этого сигнала при отсутствии обработчика всегда приводит к завершению процесса, если он не был завершен по какой-то другой причине).

Листинг 34.3. Перехватывание сигнала SIGHUP

pgsjc/catch_SIGHUP.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
handler(int sig)
{
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    struct sigaction sa;
```

```
    setbuf(stdout, NULL);  /* Отключаем буферизацию стандартного вывода */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");

    childPid = fork();
    if (childPid == -1)
        errExit("fork");

    if (childPid == 0 && argc > 1)
        if (setpgid(0, 0) == -1)
        /* Перемещаемся в новую группу процессов */
            errExit("setpgid");

    printf("PID=%ld; PPID=%ld; PGID=%ld; SID=%ld\n", (long) getpid(),
           (long) getppid(), (long) getpgrp(), (long) getsid(0));

    alarm(60);     /* Необработанный сигнал SIGALRM всегда приводит
                      к завершению процесса, если он не был завершен
                      по другой причине */

    for(;;) {      /* Ждем появления сигнала */
        pause();
        printf("%ld: caught SIGHUP\n", (long) getpid());
    }
}
```

pgsjc/catch_SIGHUP.c

Представим, что мы ввели следующие команды, чтобы запустить два экземпляра программы из листинга 34.3, и затем закрыли окно терминала:

```
$ echo $$          Идентификаторы оболочки и сессии совпадают
5533
$ ./catch_SIGHUP > samegroup.log 2>&1 &
$ ./catch_SIGHUP x > diffgroup.log 2>&1
```

Первая команда приводит к созданию двух процессов, которые остаются в группе, созданной командной оболочкой. Вторая команда создает дочерний процесс, который помещается в отдельную группу.

Просмотрев файл `samegroup.log`, мы увидим следующий вывод, указывающий на то, что оба участника группы процессов получили сигналы от командной оболочки:

```
$ cat samegroup.log
PID=5612; PPID=5611; PGID=5611; SID=5533    Потомок
PID=5611; PPID=5533; PGID=5611; SID=5533    Родитель
5611: caught SIGHUP
5612: caught SIGHUP
```

Если взглянуть на содержимое файла `diffgroup.log`, станет ясно, что оболочка, получив сигнал `SIGHUP`, не отправила его группе процессов, которую она не создавала:

```
$ cat diffgroup.log
PID=5614; PPID=5613; PGID=5614; SID=5533    Потомок
PID=5613; PPID=5533; PGID=5613; SID=5533    Родитель
5613: caught SIGHUP        Сигнал пришел к родителю, но не к потомку
```

34.6.2. Сигнал SIGHUP и завершение контролирующего процесса

Если сигнал SIGHUP, передающийся контролирующему процессу в результате отключения терминала, приводит к завершению этого процесса, он отправляется всем участникам активной группы (см. раздел 25.2). Это является следствием завершения контролирующего процесса и не связано непосредственно с сигналом SIGHUP (так как он передается вне зависимости от причины завершения).

> В Linux вслед за SIGHUP передается сигнал SIGCONT; благодаря этому группа процессов может возобновить свою работу, если ранее она была приостановлена по сигналу. Однако такое поведение не предусмотрено стандартом SUSv3, и большинство других UNIX-систем не отправляют сигнал SIGCONT в этой ситуации.

На примере программы из листинга 34.4 можно продемонстрировать, что в результате завершения контролирующего процесса сигнал SIGHUP передается всем участникам активной группы терминала. Эта программа создает по одному дочернему процессу на каждый аргумент командной строки (2). Если аргумент равен d, потомок помещается в отдельную (другую) группу (3); в противном случае он остается в одной группе с родителем (для обозначения этого действия мы используем букву s, хотя для этого подойдет любой символ, кроме d). Затем каждый потомок устанавливает обработчик для SIGHUP (4). Чтобы обеспечить завершение родителя и потомка, даже если оно не было инициировано извне, в каждом из них предусмотрен вызов alarm(), который устанавливает таймер для доставки сигнала SIGALRM через 60 секунд (5). В конце все процессы (включая родителя) выводят свои идентификаторы и идентификаторы своей группы (6), после чего входят в цикл и ждут появления сигнала (7). Когда сигнал получен, обработчик выводит его номер вместе с идентификатором процесса (1).

Листинг 34.4. Перехватывание сигнала SIGHUP при отключении от терминала

pgsjc/disc_SIGHUP.c

```
#define _GNU_SOURCE        /* Получаем объявление strsignal() из <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void                    /* Обработчик сигнала SIGHUP */
handler(int sig)
{
(1) printf("PID %ld: caught signal %2d (%s)\n", (long) getpid(),
    sig, strsignal(sig));
    /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */
}

int
main(int argc, char *argv[])
{
    pid_t parentPid, childPid;
    int j;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s {d|s}... [ > sig.log 2>&1 ]\n", argv[0]);
    setbuf(stdout, NULL);   /* Отключаем буферизацию стандартного вывода */
```

```
    parentPid = getpid();
    printf("PID of parent process is:       %ld\n", (long) parentPid);
    printf("Foreground process group ID is: %ld\n", (long) tcgetpgrp(STDIN_FILENO));

❷   for (j = 1; j < argc; j++) {    /* Создаем дочерний процесс */
        childPid = fork();
        if (childPid == -1)
            errExit("fork");

        if (childPid == 0) {          /* Если это потомок... */
❸           if (argv[j][0] == 'd')    /* 'd' означает перемещение в другую группу */
                if (setpgid(0, 0) == -1)
                    errExit("setpgid");

                sigemptyset(&sa.sa_mask);
                sa.sa_flags = 0;
                sa.sa_handler = handler;
❹               if (sigaction(SIGHUP, &sa, NULL) == -1)
                    errExit("sigaction");
                break;        /* Потомок выходит из цикла */
        }
    }

    /* Все процессы доходят сюда */

❺   alarm(60);     /* Гарантируем, что все процессы в итоге завершатся */

❻   printf("PID=%ld PGID=%ld\n", (long) getpid(), (long) getpgrp());
    for (;;)
❼       pause();        /* Ждем сигнала */
}
```

pgsjc/disc_SIGHUP.c

Представьте, что мы запустили программу из листинга 34.4 в окне терминала с помощью следующей команды:

```
$ exec ./disc_SIGHUP d s s > sig.log 2>&1
```

Встроенная команда `exec` заставляет командную оболочку выполнить вызов `exec()` и заменить себя заданной программой. Поскольку командная оболочка была контролирующим процессом для терминала, наша программа теперь сама становится контролирующим процессом и при закрытии терминала получит сигнал `SIGHUP`; когда это случится, в файле `sig.log` можно будет найти следующие строки:

```
PID of parent process is: 12733
Foreground process group ID is: 12733
PID=12755 PGID=12755   Первый потомок попадает в другую группу процессов
PID=12756 PGID=12733   Остальные потомки попадают в одну группу с родителем
PID=12757 PGID=12733
PID=12733 PGID=12733   Это родительский процесс
PID 12756: caught signal 1 (Hangup)
PID 12757: caught signal 1 (Hangup)
```

Закрытие окна терминала привело к отправке сигнала `SIGHUP` контролирующему процессу (родителю), который в ответ на это завершил свою работу. Как видно, оба потомка, находившиеся в одной группе с родителем (то есть в активной группе процессов текущего терминала) тоже получили сигнал `SIGHUP`. Однако этот сигнал не был отправлен потомку, который был в отдельной (фоновой) группе.

34.7. Управление заданиями

Возможность управления заданиями впервые появилась в командной оболочке `csh` в системе BSD; это случилось в 1980 году. Она позволяет пользователю командной строки выполнять одновременно несколько программ (заданий): одну в активном режиме, а все остальные — в фоновом. Задания можно останавливать, возобновлять и переключать между фоновым и активным режимами, как будет показано в следующих подразделах.

> Изначально управление заданиями в стандарте POSIX.1 было опциональным. В дальнейших стандартах, относящихся к UNIX, эта возможность стала обязательной.

Во времена примитивных текстовых терминалов (физических устройств, которые были способны выводить только символы в кодировке ASCII), многие пользователи умели применять команды для управления заданиями. Но с появлением графических мониторов и оконной системы X (X Window System) подобные навыки стали менее распространенными. Тем не менее управление заданиями все так же является полезной возможностью, которая упрощает и ускоряет работу с одновременно выполняющимися программами, если сравнивать это с переключением между разными окнами. Для читателей, которые не знакомы с управлением заданиями, подготовлено короткое руководство по использованию этой возможности. Закончив с этим, мы рассмотрим несколько моментов, касающихся ее реализации, и то, как она может отразиться на архитектуре вашего приложения.

34.7.1. Управление заданиями в рамках командной оболочки

Если после команды указать знак амперсанда (`&`), она будет запущена в качестве фнового задания. Это проиллюстрировано в следующих примерах:

```
$ grep -r SIGHUP /usr/src/linux >x &
[1] 18932   Задание 1: процесс, выполняющий grep, имеет идентификатор 18932
$ sleep 60 &
[2] 18934   Задание 2: процесс, выполняющий sleep, имеет идентификатор 18934
```

Каждому заданию, помещенному в фон, командная оболочка присваивает уникальный номер. Этот номер выводится в квадратных скобках после запуска задания и при управлении им с помощью различных команд. Число, которое выводится после него, является идентификатором процесса для выполнения запущенной программы или идентификатором последнего процесса в конвейере. Команды, описанные в следующих подразделах, позволяют обращаться к заданиям в формате `%num`, где `num` — это номер, назначенный заданию командной оболочкой.

> Во многих случаях аргумент %num можно опустить; при этом по умолчанию будет использоваться текущее задание. Текущим является последнее задание, остановленное в активном режиме (с помощью сочетания остановки, описанного ниже); если такового не существует, берется последнее задание, запущенное в фоне (разные командные оболочки могут поразному определять, какое фоновое задание является текущим). Кроме того, на текущее задание указывает запись %% (или %+), а для выбора предыдущего текущего задания используется запись %. В выводе команды `jobs` (которую мы обсудим ниже) текущее задание и то, которое было текущим до него, обозначаются соответственно плюсом (+) и минусом (-).

Встроенная команда `jobs` выводит список всех фоновых заданий:

```
$ jobs
[1]- Running            grep -r SIGHUP /usr/src/linux >x &
[2]+ Running            sleep 60 &
```

На этом этапе командная оболочка является активным процессом. Поскольку только активные процессы могут считывать ввод из контролирующего терминала и получать сгенерированные им сигналы, в некоторых случаях фоновое задание приходится делать активным. Для этого используется встроенная команда `fg`:

```
$ fg %1
grep -r SIGHUP /usr/src/linux >x
```

Как показано в этом примере, командная строка задания выводится при каждом его переключении между фоновым и активным режимами. Ниже вы увидите, что то же самое происходит при каждом изменении состояния задания в фоне.

Если задание является активным, его можно приостановить с помощью соответствующего сочетания клавиш (обычно Ctrl+Z), которое отправляет активной группе процессов терминала сигнал `SIGTSTP`:

```
Нажмите Ctrl+Z
[1]+ Stopped            grep -r SIGHUP /usr/src/linux >x
```

После нажатия Ctrl+Z командная оболочка выведет программу, которая была остановлена в фоне. При желании мы можем воспользоваться командами `fg` и `bg`, чтобы возобновить задание соответственно в активном и фоновом режимах. В обоих случаях оболочка возобновляет выполнение задания, отправляя ему сигнал `SIGCONT`.

```
$ bg %1
[1]+ grep -r SIGHUP /usr/src/linux >x &
```

Мы можем остановить фоновое задание, отправив ему сигнал `SIGSTOP`:

```
$ kill -STOP %1
[1]+ Stopped        grep -r SIGHUP /usr/src/linux >x
$ jobs
[1]+ Stopped        grep -r SIGHUP /usr/src/linux >x
[2]- Running        sleep 60 &
$ bg %1                 Перезапускаем задание в фоновом режиме
[1]+ grep -r SIGHUP /usr/src/linux >x &
```

Оболочки ksh и csh предоставляют команду stop, которая является сокращением для kill -stop.

Когда фоновое задание в какой-то момент завершается, оболочка, прежде чем отобразить следующее приглашение командной строки, выводит сообщение:

```
Чтобы увидеть следующее приглашение командной строки, нажмите Enter
[1]- Done            grep -r SIGHUP /usr/src/linux >x
[2]+ Done            sleep 60
$
```

Только процесс, принадлежащий к активному заданию, может считывать данные из контролирующего терминала. Это ограничение позволяет избежать конкуренции за терминальный ввод со стороны нескольких заданий. Если фоновое задание попытается прочитать данные из терминала, оно получит сигнал `SIGTTIN`, который по умолчанию приводит к его остановке:

```
$ cat > x.txt &
[1] 18947
$
Еще раз нажмите Enter, чтобы вывести изменения в состоянии задания перед отображением
```

```
следующего приглашения командной строки
[1]+ Stopped cat >x.txt
$
```

Иногда для вывода изменений состояния не обязательно нажимать клавишу Enter; это касается как предыдущего, так и некоторых последующих примеров. В зависимости от решений, принимаемых планировщиком, командная оболочка может получить уведомление об изменении состояния фонового задания до вывода следующего приглашения командной строки.

На этом этапе мы должны сделать задание активным (`fg`) и предоставить необходимый ввод. После этого мы можем продолжить выполнение задания в фоновом режиме; для этого его нужно сначала приостановить, а затем возобновить с помощью команды `bg` (конечно, в этом конкретном примере утилита `cat` снова остановила бы свою работу, так как это была бы еще одна попытка чтения с терминала).

Фоновым заданиям по умолчанию разрешено направлять вывод в контролирующий терминал. Но если для терминала установлен флаг `TOSTOP` (*terminal output stop* — остановка вывода терминала, см. раздел 58.5), попытки фоновой задачи вывести данные приведут к отправке сигнала `SIGTTOU` (флаг `TOSTOP` можно установить с помощью команды `stty`, описанной в разделе 58.3). Как и `SIGTTIN`, `SIGTTOU` останавливает задание.

```
$ stty tostop              Устанавливаем для этого терминала флаг TOSTOP
$ date &
[1] 19023
$
Еще раз нажмите Enter, чтобы вывести изменения в состоянии задания перед отображением
следующего приглашения командной строки
[1]+ Stopped        date
```

После этого мы можем получить вывод задания, сделав его активным:

```
$ fg
date
Tue Dec 28 16:20:51 CEST 2010
```

На рис. 34.2 показано, как команды оболочки и сочетания клавиш терминала (вместе с сопутствующими сигналами) применяются для переключения заданий между разными состояниями. Там же вы можете заметить условное *завершенное* состояние. Чтобы его достичь, заданию можно отправлять различные сигналы, такие как `SIGINT` и `SIGQUIT`, используя соответствующие сочетания клавиш.

34.7.2. Реализация управления заданиями

В этом разделе мы рассмотрим различные аспекты реализации управления заданиями, подытожив примером программы, которая позволяет сделать этот процесс более прозрачным.

Исходный стандарт POSIX.1 описывал реализацию управления заданиями как опциональную, однако уже в последующих стандартах, включая SUSv3, она стала обязательной. Ее поддержка связана со следующими условиями.

- Реализация должна предоставлять определенные сигналы для управления заданиями: `SIGTSTP`, `SIGSTOP`, `SIGCONT`, `SIGTTOU` и `SIGTTIN`. Также необходим сигнал `SIGCHLD` (см. раздел 26.3), который уведомляет командную оболочку (родителя всех заданий) о завершении или остановке какого-либо из ее дочерних процессов.
- Драйвер терминала должен поддерживать генерирование вышеописанных сигналов (как показано на рис. 34.2), чтобы их можно было отправить подходящей группе процессов при использовании определенного сочетания клавиш или выполнении из

фоновой задачи терминального ввода/вывода или других операций, связанных с терминалом (и описанных ниже). Для обеспечения выполнения этих действий драйвер терминала также должен записывать идентификатор сессии (контролирующего процесса) и активной группы процессов, связанной с терминалом (см. рис. 34.1).

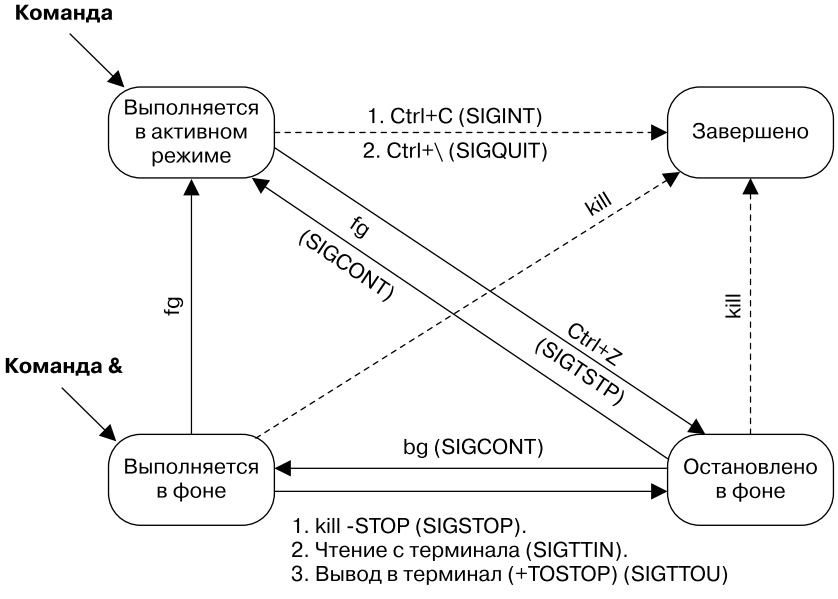


Рис. 34.2. Состояния задания

- Управление заданиями должно поддерживаться командной оболочкой (в наиболее современных из них эта поддержка присутствует). Это делается посредством команд, описанных выше. Они позволяют переключать задания между активным и фоновым режимами, а также отслеживать их состояние. Некоторые из этих команд шлют заданию сигналы (как показано на рис. 34.2). Кроме того, при выполнении операций, которые переключают задание из *активного состояния* в любое другое, командная оболочка выполняет вызовы `tcsetpgrp()`, чтобы синхронизировать запись драйвера терминала с активной группой процессов.

  В разделе 20.5 мы рассматривали сигналы, которые можно отправлять процессу, только если настоящий или действующий пользовательский идентификатор отправителя совпадает с действующим или установленным пользовательским идентификатором получателя. Однако сигнал SIGCONT является исключением из этого правила. Ядро позволяет программе (например, командной оболочке) отправлять сигнал SIGCONT любому процессу в той же сессии, независимо от учетных данных этого процесса. Это послабление требуется для того, чтобы мы по-прежнему могли возобновить работу остановленной программы с помощью сигнала SIGCONT, даже если она изменила свои учетные данные (в частности, настоящий идентификатор пользователя).

Сигналы SIGTTIN и SIGTTOU

Стандарт SUSv3 предусматривает (а Linux реализует) некоторые специальные сценарии, касающиеся отправки фоновым заданиям сигналов `SIGTTIN` и `SIGTTOU`.

- Сигнал SIGTTIN не отправляется, если в этот момент процесс его блокирует или игнорирует. Вместо этого вызов read(), выполненный из контролирующего терминала, завершается неудачей и присваивает аргументу errno значение EIO. Это объясняется тем, что в противном случае процесс не смог бы узнать, что операция read() не была разрешена.
- Даже если для терминала установлен флаг TOSTOP, сигнал SIGTTOU не отправляется, если в этот момент процесс его блокирует или игнорирует. Вместо этого разрешается выполнение записи в контролирующий терминал (то есть флаг TOSTOP игнорируется).
- Вне зависимости от наличия флага TOSTOP определенные функции, изменяющие структуры данных терминального драйвера, приводят к отправке фоновому процессу сигнала SIGTTOU, если тот пытается применить их к своему контролирующему терминалу. Это относится к функциям tcsetpgrp(), tcsetattr(), tcflush(), tcflow(), tcsendbreak() и tcdrain() (они будут описаны в главе 58). Их выполнение проходит успешно, если SIGTTOU блокируется или игнорируется.

Пример программы: демонстрация управления заданиями

Программа, представленная в листинге 34.5, показывает, каким образом командная оболочка организует составляющие конвейера в задание (группу процессов). Она также позволяет отслеживать отправку определенных сигналов и изменения состояния активной группы процессов терминала, вносимые в ходе управления заданиями. Программа написана таким образом, чтобы сразу несколько ее экземпляров можно было запускать в конвейере, как показано в следующем примере:

```
$ ./job_mon | ./job_mon | ./job_mon
```

Программа из листинга 34.5 выполняет следующие шаги.

- При запуске программа устанавливает единый обработчик для сигналов SIGINT, SIGTSTP и SIGCONT (4). Этот обработчик выполняет такие действия.
  - Отображает активную группу процессов терминала (1). Чтобы избежать вывода нескольких одинаковых строк, это делается только лидером группы.
  - Отображает идентификатор процесса, его место в конвейере и полученный сигнал (2).
  - В случае перехвата SIGTSTP обработчик должен проделать дополнительную работу, поскольку этот сигнал не останавливает процесс. В связи с этим обработчик генерирует SIGSTOP (3), который всегда приводит к остановке процесса (в подразделе 34.7.3 будет предложен улучшенный способ обработки SIGTSTP).
- Если программа является начальным процессом конвейера, она отображает заголовки для вывода, сгенерированного всеми его участниками (6). Чтобы узнать, является ли процесс начальным (или завершающим), мы используем функцию isatty() (описанную в разделе 58.10), которая проверяет, связан ли стандартный ввод (или вывод) с терминалом (5). Если заданный файл указывает на элемент конвейера, isatty() возвращает отрицательный ответ (0).
- Программа формирует сообщение, которое будет передано следующему звену конвейера. Этим сообщением является целое число, обозначающее местоположение процесса в конвейере. Таким образом, исходное сообщение инициализируется значением 0; дальше начальный процесс передает сообщение 1. Если процесс не является первым в цепочке, он считывает сообщение от своего предшественника (7) и инкрементирует его, прежде чем выполнять последующие шаги (8).
- Вне зависимости от своего положения в конвейере программа выводит строку со своим порядковым номером, идентификаторами своего и родительского процессов, а также идентификаторами группы и сессии (9).

- Если программа не является последней в цепочке, она записывает сообщение с числом для следующего процесса ⑩.
- В конце программа входит в бесконечный цикл и использует вызов `pause()` в ожидании сигналов ⑪.

Листинг 34.5. Как управление заданиями отражается на работе процессов

pgsjc/job_mon.c

```
#define _GNU_SOURCE     /* Получаем объявление strsignal() из <string.h> */
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static int cmdNum;      /* Наше положение в конвейере */

static void             /* Обработчик для различных сигналов */
handler(int sig)
{
    /* НЕБЕЗОПАСНО: этот обработчик использует функции, не рассчитанные на безопасную
       работу с асинхронными сигналами (fprintf(), strsignal(); см. подраздел 21.1.2) */

① if (getpid() == getpgrp())   /* Если процесс является лидером группы */
②     fprintf(stderr, "Terminal FG process group: %ld\n",
              (long) tcgetpgrp(STDERR_FILENO));
    fprintf(stderr, "Process %ld (%d) received signal %d (%s)\n",
              (long) getpid(), cmdNum, sig, strsignal(sig));

    /* Если мы перехватим сигнал SIGTSTP, он нас не остановит.
       Следовательно, чтобы остановиться, мы генерируем SIGSTOP. */

③ if (sig == SIGTSTP)
      raise(SIGSTOP);
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = handler;
④ if (sigaction(SIGINT, &sa, NULL) == -1)
      errExit("sigaction");
    if (sigaction(SIGTSTP, &sa, NULL) == -1)
      errExit("sigaction");
    if (sigaction(SIGCONT, &sa, NULL) == -1)
      errExit("sigaction");

  /* Если стандартный ввод связан с терминалом, то это первый процесс в конвейере:
     выводим заголовки и инициализируем сообщение, которое отправится дальше по цепочке */

⑤ if (isatty(STDIN_FILENO)) {
      fprintf(stderr, "Terminal FG process group: %ld\n",
          (long) tcgetpgrp(STDIN_FILENO));
⑥     fprintf(stderr, "Command     PID   PPID   PGRP   SID\n");
      cmdNum = 0;
```

```
    } else {    /* Это не первый процесс, значит, считываем сообщение из конвейера */
7       if (read(STDIN_FILENO, &cmdNum, sizeof(cmdNum)) <= 0)
            fatal("read got EOF or error");
    }

8   cmdNum++;
9   fprintf(stderr, "%4d    %5ld %5ld %5ld %5ld\n", cmdNum,
        (long) getpid(), (long) getppid(),
        (long) getpgrp(), (long) getsid(0));

    /* Если процесс не последний, передаем сообщение дальше */

    if (!isatty(STDOUT_FILENO))  /* Если нет файла tty, то это не конец конвейера */
10      if (write(STDOUT_FILENO, &cmdNum, sizeof(cmdNum)) == -1)
            errMsg("write");

11  for(;;)         /* Ждем сигнала */
    pause();
}
```

pgsjc/job_mon.c

Использование программы из листинга 34.5 продемонстрировано в следующей сессии командной строки. Сначала мы выводим идентификатор процесса, принадлежащего оболочке (это лидер сессии и группы процессов, для которой он является единственным участником), после чего создаем фоновое задание, состоящее из двух процессов:

```
$ echo $$                    Выводим идентификатор сессии оболочки
1204
$ ./job_mon | ./job_mon &    Запускаем задание, состоящее из двух процессов
[1] 1227
Terminal FG process group: 1204
Command   PID    PPID    PGRP    SID
   1     1226    1204    1226   1204
   2     1227    1204    1226   1204
```

По выводу, представленному выше, понятно, что командная оболочка остается активным процессом для терминала. Мы также можем видеть, что новое задание находится в одной сессии с командной оболочкой и что все процессы собраны в одной группе. Судя по идентификаторам, процессы в задании были созданы в том же порядке, в каком были перечислены программы в командной строке (так происходит в большинстве случаев, однако некоторые командные оболочки создают процессы в другом порядке).

Теперь создадим второе фоновое задание, состоящее из трех процессов:

```
$ ./job_mon | ./job_mon | ./job_mon &
[2] 1230
Terminal FG process group: 1204
Command   PID    PPID    PGRP    SID
   1     1228    1204    1228   1204
   2     1229    1204    1228   1204
   3     1230    1204    1228   1204
```

Как видите, командная оболочка по-прежнему является активной группой процессов по отношению к терминалу. Мы также видим, что процессы для нового задания находятся в одной сессии с командной оболочкой, но в другой группе по сравнению с первым заданием. Теперь переключим второе задание в активный режим и отправим ему сигнал `SIGINT`:

```
$ fg
./job_mon | ./job_mon | ./job_mon
```

```
Нажмите Ctrl+C, чтобы сгенерировать сигнал SIGINT (signal 2)
Process 1230 (3) received signal 2 (Interrupt)
Process 1229 (2) received signal 2 (Interrupt)
Terminal FG process group: 1228
Process 1228 (1) received signal 2 (Interrupt)
```

На основе вывода, приведенного выше, можно заключить, что сигнал SIGINT был доставлен всем процессам в активной группе. Мы также видим, что данное задание теперь представляет собой активную группу процессов по отношению к терминалу. Теперь пошлем ему сигнал SIGTSTP:

```
Нажмите Ctrl+Z, чтобы сгенерировать сигнал SIGTSTP (signal 20 в Linux/x86-32)
Process 1230 (3) received signal 20 (Stopped)
Process 1229 (2) received signal 20 (Stopped)
Terminal FG process group: 1228
Process 1228 (1) received signal 20 (Stopped)

[2]+ Stopped ./job_mon | ./job_mon | ./job_mon
```

Теперь все участники группы процессов остановлены. Вывод говорит о том, что группа процессов 1228 являлась активным заданием. После остановки ее место заняла командная оболочка, хотя по листингу сессии этого не видно.

Продолжим, перезапустив задание с помощью команды bg, которая доставляет процессу в задании сигнал SIGCONT:

```
$ bg                                  Возобновляем задание в фоновом режиме
[2]+ ./job_mon | ./job_mon | ./job_mon &
Process 1230 (3) received signal 18 (Continued)
Process 1229 (2) received signal 18 (Continued)
Terminal FG process group: 1204       Оболочка становится активной
Process 1228 (1) received signal 18 (Continued)
$ kill %1 %2                          Закончив, освобождаем ресурсы
[1]- Terminated ./job_mon | ./job_mon
[2]+ Terminated ./job_mon | ./job_mon | ./job_mon
```

34.7.3. Обрабатываем сигналы, связанные с управлением заданиями

Поскольку в большинстве приложений процедура управления заданиями является прозрачной, для работы с сигналами, которые с ней связаны, не нужно предпринимать дополнительных действий. Исключение составляют программы, которые работают с экраном, — например, vi или less. Они отвечают за точное позиционирование текста в терминале, изменяя различные его параметры, в том числе параметр, который позволяет терминалу считывать отдельные символы (вместо целых строк). Этот и другие параметры будут описаны в главе 58.

Программы, работающие с экраном, должны обрабатывать сигнал остановки, отправляемый терминалом (SIGTSTP). Обработчик должен переключить терминал в исходный режим ввода (построчный) и поместить курсор в нижний левый угол. При возобновлении работы программа снова переключает терминал в нужный ей режим, проверяет размер окна терминала (который за это время мог измениться) и заново перерисовывает содержимое экрана.

Если приостановить или завершить программу, работающую с терминалом (такую как vi в xterm или другом эмуляторе терминала), терминал, как правило, перерисовывается с помощью текста, который был видим до запуска этой программы. Это достигается за счет перехватывания двух символьных последовательностей, которые программа, использующая пакеты terminfo

или termcap, обязана вывести при получении и возврате контроля над содержимым экрана. Первая из этих последовательностей, которую называют smcup (обычно это клавиша Esc в сочетании с символами [?1049h), заставляет эмулятор терминала переключиться на «альтернативный» экран. Вторая последовательность, rmcup (обычно это клавиша Esc в сочетании с символами [?1049l), переключает терминал обратно на стандартный экран, что приводит к возвращению текста, который отображался до того, как программа захватила контроль над терминалом.

При обработке сигнала SIGTSTP нужно учитывать определенные тонкости. Впервые это отмечалось в подразделе 34.7.2: если сигнал SIGTSTP перехвачен, он не выполняет своего стандартного действия — то есть не останавливает процесс. В листинге 34.5 мы решали эту проблему, генерируя в обработчике SIGTSTP сигнал SIGSTOP. SIGSTOP нельзя перехватить, заблокировать или проигнорировать, и он гарантированно приводит к немедленной остановке процесса. Однако такой подход является не совсем корректным. В подразделе 26.1.3 мы видели, что родительский процесс может использовать статус ожидания, возвращенный вызовами wait() или waitpid(), чтобы определить, какой сигнал вызвал остановку его потомка. Если сгенерировать сигнал SIGSTOP в обработчике SIGTSTP, родитель (ошибочно) посчитает, что причиной остановки потомка стал именно этот сигнал.

Правильный подход в этой ситуации — сгенерировать в обработчике SIGTSTP еще один сигнал SIGTSTP, который и остановит процесс. Это делается следующим образом.

1. Обработчик сбрасывает действие сигнала SIGTSTP к значению по умолчанию (SIG_DFL).
2. Обработчик генерирует SIGTSTP.
3. Поскольку сигнал SIGTSTP был заблокирован на входе в обработчик (если только не был указан флаг SA_NODEFER), тот его разблокирует. После этого ожидающий сигнал SIGTSTP, сгенерированный в предыдущем шаге, выполняет свое стандартное действие: немедленно приостанавливает работу процесса.
4. Позже, при получении сигнала SIGCONT, процесс возобновит свою работу. На этом этапе выполнение обработчика продолжается.
5. Перед началом работы обработчик заново блокирует сигнал SIGTSTP и готовится перехватить следующий его экземпляр.

Повторное блокирование сигнала SIGTSTP нужно для того, чтобы избежать рекурсивного вызова обработчика, если после его установки (но до возвращения) был доставлен еще один экземпляр SIGTSTP. Как отмечалось в разделе 22.7, рекурсивные вызовы обработчиков сигналов могут привести к переполнению стека, если сигналы доставляются слишком быстро. Для обхода этой проблемы также можно прибегнуть к блокированию сигнала, если между установкой и возвращением обработчику больше не нужно выполнять никаких других действий (например, сохранение или восстановление значений из глобальных переменных).

Пример программы

Обработчик из листинга 34.6 обеспечивает корректное обращение с сигналом SIGTSTP, выполняя шаги, описанные выше (еще один пример обработки этого сигнала будет представлен в листинге 58.4). Установив обработчик SIGTSTP, главная функция этой программы входит в цикл и ждет появления сигнала. Пример того, что можно увидеть при запуске данного кода, показан ниже:

```
$ ./handling_SIGTSTP
Нажмите Ctrl+Z, чтобы послать сигнал SIGTSTP
Caught SIGTSTP            Это сообщение выводится обработчиком SIGTSTP

[1]+ Stopped ./handling_SIGTSTP
```

```
$ fg                        Отправляет сигнал SIGTSTP
./handling_SIGTSTP
Exiting SIGTSTP handler     Выполнение обработчика продолжается; обработчик возвращается
Main                        Вызов pause() в главной функции был прерван обработчиком
Нажмите Ctrl+C, чтобы завершить программу
```

В программах, которые управляют экраном (таких как `vi`), вызов `printf()` внутри обработчика сигнала из листинга 34.6 был бы заменен кодом, переключающим режим терминала и перерисовывающим его окно, как было отмечено выше. Чтобы избежать использования функций, которые не рассчитаны на безопасную работу с асинхронными сигналами (см. подраздел 21.1.2), обработчик должен установить флаг, информируя главную программу о необходимости перерисовать экран.

Стоит отметить, что обработчик `SIGTSTP` может прервать работу некоторых блокирующих системных вызовов (описанных в разделе 21.5). В программном выводе, приведенном выше, этот момент проиллюстрирован тем, что после прерывания вызова `pause()` главная программа выводит сообщение `Main`.

Листинг 34.6. Обработка сигнала SIGTSTP

pgsjc/handling_SIGTSTP.c

```
#include <signal.h>
#include "tlpi_hdr.h"

static void            /* Обработчик для сигнала SIGTSTP */
tstpHandler(int sig)
{
    sigset_t tstpMask, prevMask;
    int savedErrno;
    struct sigaction sa;

    savedErrno = errno;          /* Если мы изменим здесь 'errno' */

    printf("Caught SIGTSTP\n");  /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */

    if (signal(SIGTSTP, SIG_DFL) == SIG_ERR)
        errExit("signal");       /* Используем действие по умолчанию */

    raise(SIGTSTP);              /* Генерируем еще один сигнал SIGTSTP */

    /* Разблокируем SIGTSTP; ожидающий сигнал SIGTSTP сразу приостанавливает программу */

    sigemptyset(&tstpMask);
    sigaddset(&tstpMask, SIGTSTP);
    if (sigprocmask(SIG_UNBLOCK, &tstpMask, &prevMask) == -1)
        errExit("sigprocmask");

    /* Выполнение продолжается после получения SIGCONT */

    if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
        errExit("sigprocmask");   /* Еще раз блокируем SIGTSTP */

    sigemptyset(&sa.sa_mask);     /* Снова устанавливаем обработчик */
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = tstpHandler;
    if (sigaction(SIGTSTP, &sa, NULL) == -1)
        errExit("sigaction");
```

```
    printf("Exiting SIGTSTP handler\n");
    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;

    /* Устанавливаем обработчик для сигнала SIGTSTP, только если он не игнорируется */
    if (sigaction(SIGTSTP, NULL, &sa) == -1)
        errExit("sigaction");

    if (sa.sa_handler != SIG_IGN) {
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;
        sa.sa_handler = tstpHandler;
        if (sigaction(SIGTSTP, &sa, NULL) == -1)
            errExit("sigaction");
    }

    for (;;) {            /* Ждем сигнала */
        pause();
        printf("Main\n");
    }
}
```

pgsjc/handling_SIGTSTP.c

Работа с игнорируемыми сигналами, сгенерированными терминалом или предназначенными для управления заданиями

Программа, представленная в листинге 34.6, устанавливает обработчик для сигнала `SIGTSTP`, только если тот не игнорируется. Это частный случай более общего правила, согласно которому приложения должны обрабатывать сигналы, сгенерированные терминалом или предназначенные для управления заданиями, только если они не были проигнорированы ранее. В случае с сигналами для управления заданиями (`SIGTSTP`, `SIGTTIN` и `SIGTTOU`) это помогает избежать их обработки, если программа была запущена из командной оболочки, которая не поддерживает эту возможность (такой как `bash`). В таких оболочках действия этих сигналов имеют значение `SIG_IGN`; значение `SIG_DFL` устанавливается только в том случае, если командная оболочка поддерживает управление заданиями.

Похожая ситуация и с другими сигналами, которые могут быть сгенерированы терминалом: `SIGINT`, `SIGQUIT` и `SIGHUP`. В случае с первыми двумя причина заключается в том, что при выполнении программы в фоновом режиме в рамках командной оболочки, не поддерживающей управление заданиями, итоговый процесс не размещается в отдельной группе. Вместо этого он остается в одной группе с оболочкой, которая перед выполнением программы делает действия сигналов `SIGINT` и `SIGQUIT` игнорируемыми. Благодаря этому процесс не завершится, если пользователь нажмет в терминале сочетания клавиш для *прерывания* или *выхода* (которые должны касаться только лишь тех заданий, что номинально являются активными). Если впоследствии процесс восстановит стандартные действия для этих сигналов, измененные командной оболочкой, он снова сможет их получить.

Сигнал `SIGHUP` игнорируется, если программа выполняется посредством утилиты `nohup(1)`. Это защищает ее от преждевременного завершения в случае зависания терминала. Таким образом, приложение не должно пытаться изменить действие этого сигнала, если он игнорируется.

34.7.4. Осиротевшие группы процессов (и новый взгляд на сигнал SIGHUP)

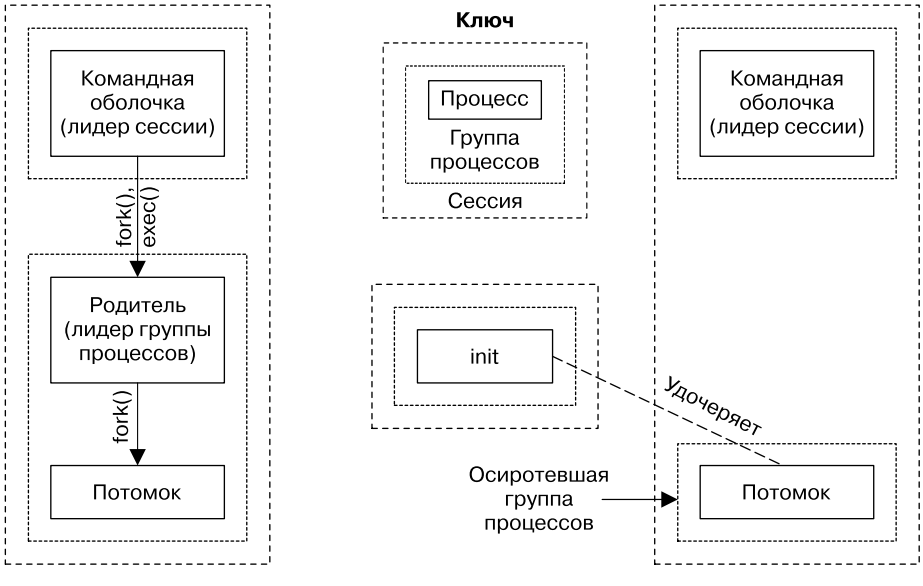
В разделе 26.2 вы узнали, что «осиротевшим» считается процесс, который после завершения своего родителя «удочеряется» программой `init` (с идентификатором процесса 1). Внутри программы процесс-«сироту» можно создать с помощью следующего кода:

```
if (fork() != 0) /* Выходим, если это родитель (или в случае ошибки) */
exit(EXIT_SUCCESS);
```

Представьте, что этот код является частью программы, запущенной из командной оболочки. На рис. 34.3 показано состояние процессов до и после завершения родителя.

После завершения родителя дочерний процесс не просто становится «сиротой», но и входит в *«осиротевшую» группу*. Согласно стандарту SUSv3 группа процессов считается «осиротевшей», «если родитель каждого ее участника либо сам входит в эту группу, либо не является членом ее сессии». Иными словами, группа процессов не является осиротевшей, если родитель хотя бы одного ее участника находится в той же сессии, но в другой группе. На рис. 34.3 группа, содержащая потомка, является «осиротевшей», поскольку потомок находится в группе процессов сам по себе, а его родитель (`init`) выполняется в другой сессии.

Лидер сессии по определению находится в «осиротевшей» группе процессов. Это вытекает из того факта, что вызов setsid() создает новую группу в новой сессии, отличной от той, в которой размещен родитель лидера.



А. Создание родительского и дочернего процессов

Б. Удочерение потомка процессом init после завершения родителя

Рис. 34.3. Этапы создания «осиротевшей» группы процессов

Чтобы понять, почему «осиротевшие» группы процессов имеют большое значение, необходимо взглянуть на вещи с точки зрения управления заданиями. Рассмотрим следующую ситуацию, основанную на рис. 34.3.

1. Перед завершением родительского процесса потомок был остановлен (возможно, родитель послал ему соответствующий сигнал).
2. Когда родительский процесс завершается, командная оболочка удаляет его группу из своего списка заданий. Потомок удочеряется процессом `init` и становится фоновым по отношению к терминалу. Группа процессов, содержащая потомка, становится «сиротой».
3. На данном этапе не существует процесса, который отслеживает с помощью вызова `wait()` состояние остановленного потомка.

Поскольку командная оболочка не создавала дочерний процесс, она не знает о его существовании и о том, что он входит в ту же группу, что и его завершившийся родитель. Более того, процесс `init` обратит внимание на потомка только после того, как тот завершится (чтобы освободить процесс-«зомби»). Следовательно, остановленный потомок может находиться в этом состоянии вечно, так как ни один другой процесс не знает, что его работу нужно возобновить с помощью сигнала `SIGCONT`.

Но даже если остановленный процесс в «осиротевшей» группе имеет выполняющегося родителя, этот родитель не всегда будет иметь возможность послать ему сигнал, если он находится в другой сессии. Он сможет послать сигнал `SIGCONT` любому другому процессу в одной сессии, но если сессия потомка не совпадает с его собственной, в силу вступают обычные правила для отправки сигналов (см. раздел 20.5); это означает, что родитель не сможет послать сигнал потомку, если тот является привилегированным процессом, изменившим свои учетные данные.

Чтобы избежать ситуаций, подобных вышеописанной, стандарт SUSv3 гласит, что если группа процессов становится осиротевшей и содержит остановленных участников, всем входящим в нее процессам отправляется два сигнала: сначала `SIGHUP`, который уведомляет участников о том, что они были отключены от своей сессии, и затем `SIGCONT`, который возобновляет их выполнение. Если осиротевшая группа процессов не имеет остановленных участников, сигналы не отправляются.

Группа процессов может стать «сиротой» по двум причинам: либо потому, что завершился последний родитель в той же сессии, но в другой группе, либо в результате завершения последнего процесса, который имел родителя из другой группы (вторая ситуация проиллюстрирована на рис. 34.3). Но какой бы ни была причина, к «осиротевшей» группе процессов, содержащей остановленных потомков, применяется одна и та же процедура.

> Отправка сигналов SIGHUP и SIGCONT осиротевшей группе процессов, некоторые участники которой остановлены, делается для того, чтобы закрыть определенную лазейку в системе управления заданиями. Нет ничего, что бы помешало процессу (с подходящими привилегиями) отправить сигнал остановки участникам уже «осиротевшей» группы. В этом случае процесс останавливается до тех пор, пока какой-то другой процесс (опять же с подходящими привилегиями) не отправит ему SIGCONT.
>
> Функция tcsetpgrp() (см. раздел 34.5), если она вызвана участником осиротевшей группы процессов, возвращает ошибку ENOTTY; функции tcsetattr(), tcflush(), tcflow(), tcsendbreak() и tcdrain() (все описаны в главе 58) в этой же ситуации завершаются ошибкой EIO.

Пример программы

Обращение с процессами-«сиротами», описанное выше, показано на примере программы из листинга 34.7. После установки обработчиков для сигналов `SIGHUP` и `SIGCONT` (2) она создает по одному дочернему процессу для каждого аргумента командной строки (3). Затем каждый потомок останавливается (генерируя сигнал `SIGSTOP`) (4) или ждет сигнала (с помощью вызова `pause()`) (5). Поведение потомка зависит от того, начинается ли соответствующий ему аргумент командной строки с буквы `s` (от слова *stop*). Для обозначения иного действия (то есть вызова `pause()`) мы используем букву `p`, хотя здесь подойдет любой символ, отличный от `s`.

После создания всех потомков родитель останавливается на несколько секунд, чтобы дать им возможность запуститься ❻ (как отмечалось в разделе 24.2, использование вызова `sleep()` для этой цели — не лучший вариант, но иногда это позволяет получить желаемый результат). Затем родитель завершается ❼, делая тем самым группу процессов со своими потомками «сиротой». Если в результате этого какой-либо из потомков получит сигнал, запустится обработчик, который выведет идентификатор дочернего процесса и номер сигнала ❶.

Листинг 34.7. Сигнал SIGHUP и «осиротевшая» группа процессов

pgsjc/orphaned_pgrp_SIGHUP.c

```
#define _GNU_SOURCE     /* Получаем объявление strsignal() из <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void             /* Обработчик сигнала */
handler(int sig)
{
❶   printf("PID=%ld: caught signal %d (%s)\n", (long) getpid(),
    sig, strsignal(sig));   /* НЕБЕЗОПАСНО (см. подраздел 21.1.2) */
}

int
main(int argc, char *argv[])
{
    int j;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s {s|p} ...\n", argv[0]);

    setbuf(stdout, NULL);  /* Отключаем буферизацию стандартного вывода */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
❷   if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGCONT, &sa, NULL) == -1)
        errExit("sigaction");

    printf("parent: PID=%ld, PPID=%ld, PGID=%ld, SID=%ld\n", (long) getpid(),
           (long) getppid(), (long) getpgrp(), (long) getsid(0));

    /* Создаем по одному потомку для каждого аргумента командной строки */
❸   for (j = 1; j < argc; j++) {
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:         /* Потомок */
            printf("child:    PID=%ld, PPID=%ld, PGID=%ld, SID=%ld\n",
                (long) getpid(), (long) getppid(),
                (long) getpgrp(), (long) getsid(0));

            if (argv[j][0] == 's') {    /* Останавливаем по сигналу */
                printf("PID=%ld stopping\n", (long) getpid());
❹               raise(SIGSTOP);
```

```
            } else {       /* Ждем сигнала */
                alarm(60); /* Завершаемся, если не получили сигнал SIGHUP */
                printf("PID=%ld pausing\n", (long) getpid());
❺               pause();
            }
            _exit(EXIT_SUCCESS);

        default:        /* Родитель идет дальше по циклу */
            break;
        }
    }

    /* Родитель переходит сюда после создания всех потомков */
❻   sleep(3);        /* Даем потомкам шанс запуститься */
    printf("parent exiting\n");
❼   exit(EXIT_SUCCESS);    /* Делаем потомков и их группу «сиротами» */
}
```

pgsjc/orphaned_pgrp_SIGHUP.c

Следующая сессия командной строки демонстрирует результаты двух запусков программы из листинга 34.7:

```
$ echo $$    Выводим ID командной оболочки, совпадающий с ID сессии
4785
$ ./orphaned_pgrp_SIGHUP s p
parent: PID=4827, PPID=4785, PGID=4827, SID=4785
child: PID=4828, PPID=4827, PGID=4827, SID=4785
PID=4828 stopping
child: PID=4829, PPID=4827, PGID=4827, SID=4785
PID=4829 pausing
parent exiting
$ PID=4828: caught signal 18 (Continued)
PID=4828: caught signal 1 (Hangup)
PID=4829: caught signal 18 (Continued)
PID=4829: caught signal 1 (Hangup)
Нажмите Enter, чтобы получить еще одно приглашение командной строки
$ ./orphaned_pgrp_SIGHUP p p
parent: PID=4830, PPID=4785, PGID=4830, SID=4785 child: PID=4831, PPID=4830,
PGID=4830, SID=4785
PID=4831 pausing
child: PID=4832, PPID=4830, PGID=4830, SID=4785
PID=4832 pausing
parent exiting
```

При первом запуске в группе, которая должна стать «сиротой», создается два потомка: один сам себя останавливает, второй приостанавливается (в этом случае приглашение командной строки отображается посреди вывода потомков, поскольку командная оболочка замечает, что родитель уже завершился). Как можно заметить, после завершения родителя оба дочерних процесса получают сигналы `SIGCONT` и `SIGHUP`. При втором запуске снова создается два потомка, но на этот раз ни один из них не останавливается; благодаря этому завершение родителя не приводит к отправке сигналов.

«Осиротевшие» группы процессов и сигналы SIGTSTP, SIGTTIN и SIGTTOU

«Осиротевшие» группы сигналов также влияют на семантику доставки сигналов `SIGTSTP`, `SIGTTIN` и `SIGTTOU`.

В подразделе 34.7.1 мы видели, что сигналы `SIGTTIN` и `SIGTTOU` передаются фоновому процессу, если тот пытается соответственно прочитать или записать данные в контролирующем терминале (в последнем случае должен быть установлен флаг `TOSTOP`). Однако в передаче их «осиротевшей» группе процессов нет никакого смысла, поскольку, остановившись, она уже никогда не возобновит работу. По этой причине вместо отправки сигналов `SIGTTIN` и `SIGTTOU` ядро завершает операции чтения и записи ошибкой `EIO`.

Аналогично, если сигналы `SIGTSTP`, `SIGTTIN` и `SIGTTOU` должны привести к остановке «осиротевшей» группы процессов, их доставка просто отменяется (если же сигнал имеет соответствующий обработчик, он доставляется процессу). Это происходит вне зависимости от способа отправки — то есть сигнал может быть сгенерирован драйвером терминала или передан посредством явного вызова `kill()`.

34.8. Резюме

Сессии и группы процессов (известные также как задания) составляют двухуровневую иерархию: сессия представляет собой набор групп, а группы — набор процессов. Лидером сессии является процесс, который создал ее с помощью вызова `setsid()`. Аналогично лидером группы является процесс, который создал ее с помощью вызова `setpgid()`. Все участники группы процессов имеют один и тот же идентификатор PGID (который совпадает с PGID ее лидера), а все процессы из всех групп, составляющих сессию, имеют один и тот же идентификатор SID (который совпадает с SID ее лидера). Любая сессия может иметь контролирующий терминал, который устанавливается при открытии лидером сессии устройства `/dev/tty`. Открытие контролирующего терминала также приводит к тому, что лидер сессии становится контролирующим процессом этого терминала.

Сессии и группы процессов были созданы для поддержки управления заданиями (хотя иногда их можно применять и для других целей). В этом контексте командная оболочка является лидером сессии и контролирующим процессом для терминала, в котором она выполняется. Каждое задание (обычная команда или конвейер), запускаемое командной оболочкой, выполняется в отдельной группе процессов и может иметь три состояния (переключаться между которыми можно с помощью встроенных в оболочку команд): активная работа, работа в фоне и остановка в фоновом режиме.

Для поддержки управления заданиями драйвер терминала хранит запись для активной группы процессов (задания) контролирующего терминала. Он также доставляет сигналы активному заданию при нажатии определенного сочетания клавиш. Эти сигналы либо завершают, либо приостанавливают активную группу процессов.

Понятие активного задания терминала также используется для распределения запросов ввода/вывода. Только процессы из активной группы могут считывать данные из контролирующего терминала. Фоновым заданиям при попытке чтения передается сигнал `SIGTTIN`, стандартным действием которого является остановка всей группы процессов. Если в контролирующем терминале установлен флаг `TOSTOP`, при попытке записи в него фоновые задания получают сигнал `SIGTTOU`, действие по умолчанию которого тоже приводит к остановке группы процессов.

Когда сессия отключается от терминала, ядро передает контролирующему процессу сигнал `SIGHUP`, уведомляя его об этом факте. Это событие может запустить цепную реакцию, доставляя сигнал `SIGHUP` множеству других процессов. Во-первых, если командная оболочка является контролирующим процессом (что является обычной ситуацией), то при завершении она отправляет `SIGHUP` каждому процессу в группе, которую она создала. Во-вторых, если `SIGHUP` приводит к завершению контролирующего процесса, ядро передает его всем участникам активной группы контролирующего терминала.

В целом, приложениям не обязательно знать о сигналах управления заданиями. Исключением являются программы, работающие с экраном. Такие программы должны корректно обрабатывать сигнал SIGTSTP, сбрасывая атрибуты терминала к подходящим значениям перед остановкой и восстанавливая их при возобновлении работы после получения сигнала SIGCONT.

Группа процессов считается «сиротой», если ни один из ее участников не имеет родителя в другой группе в рамках той же сессии. Это важно понимать, поскольку за пределами группы нет такого процесса, который мог бы как отслеживать состояние остановленных процессов внутри группы, так и перезапускать их с помощью сигнала SIGCONT. Это могло бы привести к тому, что остановленные процессы оставались бы в системе навсегда. Чтобы этого избежать, все участники группы процессов, которая становится осиротевшей, получают два последовательных сигнала — SIGHUP и SIGCONT, которые уведомляют их о том, что они стали «сиротами», и обеспечивают их перезапуск.

Дополнительная информация

В главе 9 книги [Stevens & Rago, 2005] представлен материал, похожий на тот, что был рассмотрен нами здесь, включая поэтапное описание установления сессии во время входа в систему. Справочное руководство к библиотеке glibc содержит пространное описание функций, связанных с системой управления заданиями и ее реализацией в рамках командной оболочки. В стандарте SUSv3 можно найти подробное обсуждение сессий, групп процессов и управления заданиями.

34.9. Упражнения

34.1. Представьте, что родительский процесс выполняет следующие шаги:

```
/* Вызываем fork(), чтобы ряд дочерних процессов, каждый
   из которых остается в одной группе с родителем */
/* Чуть позже... */
signal(SIGUSR1, SIG_IGN);   /* Родитель становится нечувствительным к SIGUSR1 */
killpg(getpgrp(), SIGUSR1); /* Отправляем сигнал потомкам, созданным ранее */
```

К какой проблеме может привести данный подход (вспомните конвейеры в командной оболочке)? Как ее можно избежать?

34.2. Напишите программу, которая подтверждает, что родительский процесс может изменить идентификатор PGID одного из своих потомков до того, как тот выполнит вызов exec(), но не после.

34.3. Напишите программу, которая подтверждает, что вызов setsid(), сделанный из лидера группы процессов, завершается неудачей.

34.4. Модифицируйте программу из листинга 3.4 (disc_SIGHUP.c), чтобы проверить, что ядро не отправляет сигнал SIGHUP участнику активной группы, если тот не приводит к завершению контролирующего процесса.

34.5. Представьте, что код, который разблокирует сигнал SIGTSTP в листинге 34.6, был перемещен в начало обработчика. К какому состоянию гонки это может привести?

34.6. Напишите программу, которая подтверждает, что операция read() возвращает ошибку EIO, если попытаться ее выполнить внутри процесса из осиротевшей группы в контролирующем терминале.

34.7. Напишите программу, которая подтверждает, что сигналы SIGTTIN, SIGTTOU или SIGTSTP, если отправить их члену осиротевшей группы процессов, отменяются (то есть не имеют никаких последствий), при условии, что они могли бы привести к остановке процесса (то есть если они имеют действие SIG_DFL); однако, если они имеют соответствующий обработчик, их доставка проходит успешно.

35 Приоритеты процессов и их планирование

Эта глава посвящена различным системным вызовам и атрибутам, которые определяют, какой процесс в какой момент времени получит доступ к процессору (-ам). Мы начнем с описания значения `nice` — свойства процесса, которое влияет на количество процессорного времени, выделяемого ему планировщиком ядра. После этого мы перейдем к программному интерфейсу POSIX для планирования в режиме реального времени. Этот программный интерфейс позволяет устанавливать политику и приоритет планирования процессов, предоставляя более строгий контроль над выделением ресурсов процессора. В конце мы уделим внимание системным вызовам для задания маски родственного процессора, определяющей набор процессоров, на которых должен выполняться процесс.

35.1. Приоритеты процессов (значение nice)

В Linux, как и в большинстве других реализаций UNIX, для распределения ресурсов процессора по умолчанию применяется *циклическое разделение времени*. Согласно этой модели, каждому процессу по очереди выделяется небольшой период времени (*отрезок*, или *квант, времени*), на протяжении которого он может использовать ресурсы центрального процессора. Циклическое разделение времени удовлетворяет двум следующим важным требованиям, предъявляемым к интерактивным многозадачным системам.

- *Справедливость*: каждый процесс получает свою долю ресурсов процессора
- *Отзывчивость*: процесс не должен долго ждать, прежде чем иметь возможность воспользоваться процессором.

Алгоритм циклического разделения времени не дает процессам прямого контроля над тем, когда и как долго они смогут задействовать центральный процессор. По умолчанию каждый процесс поочередно получает доступ к процессору, пока не истечет его квант времени или пока он сам не освободит ресурсы (например, путем переключения в спящий режим или чтения с диска). Если все процессы пытаются использовать ЦПУ с максимальной интенсивностью (то есть когда ни один из них не переходит в режим сна или блокируется в результате операций ввода/вывода), они получат примерно одинаковый объем ресурсов процессора.

Однако существует атрибут, значение `nice`, который позволяет процессу опосредованно влиять на алгоритм планирования ядра. Каждый процесс имеет значение `nice` в диапазоне от –20 (высокий приоритет) до +19 (низкий приоритет); по умолчанию оно равно `0` (рис. 35.1). В традиционных реализациях UNIX только привилегированные процессы могут назначать себе (или другим процессам) отрицательный (высокий) приоритет (в подразделе 35.3.2 будут описаны некоторые особенности, характерные для Linux). Непривилегированные процессы могут только понижать свой приоритет, делая значение `nice` больше нуля. Делая это, они ведут себя «хорошо» (от англ. *nice*) по отношению к другим процессам, от чего и происходит название данного атрибута.

Значение `nice` наследуется потомком, созданным с помощью `fork()`, и сохраняется на протяжении работы вызова `exec()`.

Вместо непосредственного значения nice системный вызов getpriority() возвращает число в диапазоне от 1 (низкий приоритет) до 40 (высокий приоритет), вычисленное по формуле unice = 20 – knice. Это делается для того, чтобы избежать возвращения отрицательных значений, которые используются в системных вызовах для обозначения ошибок (описание работы системных вызовов ищите в разделе 3.1). Приложения не знают о том, что результатом работы системного вызова getpriority() является измененное значение, так как одноименная функция-обертка из библиотеки Си делает обратное вычисление, возвращается значение 20 – unice.

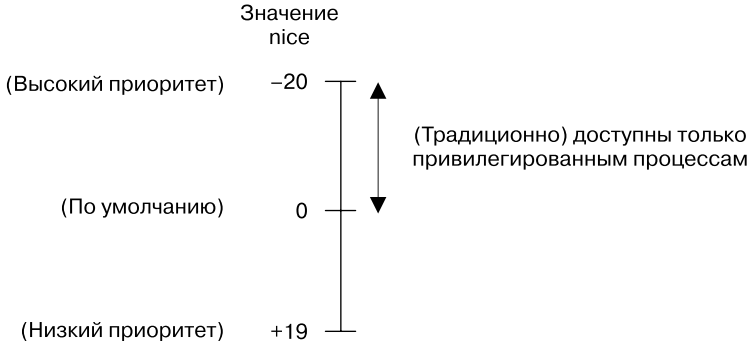


Рис. 35.1. Диапазон и интерпретация значений nice

Эффект от значения nice

Значение `nice` не устанавливает четкой иерархии планирования процессов; вместо этого она играет роль весового коэффициента, на основе которого планировщик ядра выдает повышенные приоритеты. Процесс с низким приоритетом (то есть с высоким значением `nice`) не лишится процессорного времени полностью, но будет получать его в относительно меньших объемах. Степень влияния значения `nice` на планирование процессов зависит от версии ядра Linux или от конкретной UNIX-системы.

Начиная с версии 2.6.23, в ядре используется новый алгоритм планирования, согласно которому разница в значениях nice имеет большее влияние, чем раньше. В результате этого процессы с низкими и высокими значениям nice получают соответственно меньше и больше процессорного времени по сравнению с предыдущими версиями.

Получение и изменение приоритетов

Системные вызовы `getpriority()` и `setpriority()` позволяют процессу получать и изменять свое собственное или чужое значение `nice`.

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
```

Возвращает (возможно, отрицательное) значение `nice` заданного процесса или `-1`, если случилась ошибка

```
int setpriority(int which, id_t who, int prio);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

Оба системных вызова принимают аргументы `which` и `who`, определяющие процесс (-ы), приоритет которого (-ых) нужно получить или изменить. Первый влияет на то, как будет интерпретирован второй. Аргумент `which` может иметь одно из следующих значений.

- `PRIO_PROCESS` — работать с процессом, чей идентификатор равен `who`. Если `who` равен `0`, используется идентификатор вызывающего процесса.
- `PRIO_PGRP` — работать со всеми участниками группы процессов, чей идентификатор PGID равен `who`. Если `who` равен `0`, используется группа вызывающего процесса.
- `PRIO_USER` — работать со всеми процессами, реальный пользовательский идентификатор которых равен `who`. Если `who` равен `0`, берется пользовательский идентификатор вызывающего процесса.

Тип данных `id_t`, который применяется для аргумента `who`, представляет собой целое число размера, достаточного для того, чтобы вместить идентификаторы процессов и пользователей.

Системный вызов `getpriority()` возвращает значение `nice` процесса, указанного с помощью аргументов `which` и `who`. Если заданным критериям отвечает сразу несколько процессов (это бывает, когда аргумент `which` равен `PRIO_PGRP` или `PRIO_USER`), берется процесс с самым высоким приоритетом. Поскольку `getpriority()` при успешном выполнении может вернуть `-1`, перед вызовом следует присваивать переменной `errno` значение `0` и затем, если мы получили `-1`, проверять, чему равна переменная `errno` после вызова.

Системный вызов `setpriority()` устанавливает значение `nice(prio)` для процесса (-ов), заданного (-ых) с помощью аргументов `which` и `who`. Если попытаться установить число, которое выходит за допустимые рамки (от –20 до +19), оно будет автоматически подогнано под корректное значение.

> Изначально для изменения значений nice использовался вызов nice(incr), который добавлял incr к текущему значению. Эта функция все еще доступна, но ей на смену пришел более универсальный вызов, setpriority().
>
> В командной строке аналогами вызова setpriority() являются утилиты nice(1) и renice(8); первая позволяет запускать программы с пониженным или повышенным приоритетом (во втором случае необходимы особые привилегии), а с помощью второй администратор может изменять значения nice для уже запущенных процессов.

Привилегированный процесс (`CAP_SYS_NICE`) может изменить приоритет любой программы. Обычный процесс может изменить свой собственный приоритет (передав аргументам `which` и `who` значения `PRIO_PROCESS` и `0`) или приоритет другой (целевой) программы, если она имеет такой же действующий идентификатор пользователя. В Linux система прав для вызова `setpriority()` отличается от той, что описана в стандарте SUSv3, согласно которой для изменения приоритета непривилегированным процессом ему достаточно иметь такой же действующий *или* реальный идентификатор пользователя, что и целевая программа. В этом аспекте разные UNIX-системы демонстрируют определенные отличия. Некоторые из них следуют правилам стандарта SUSv3, а другие (в частности, BSD) ведут себя так, как Linux.

Начиная с версии 2.6.12, ядро Linux предоставляет ограничение на ресурсы `RLIMIT_NICE`, который позволяет непривилегированным процессам увеличивать значения `nice`. Чтобы максимально повысить собственный приоритет, такой процесс должен руководствоваться формулой `20 – rlim_cur`, где `rlim_cur` — текущее мягкое ограничение на ресурсы. Например, мягкое ограничение `RLIMIT_NICE` процесса равно 25, то его значение `nice` можно поднять до –5. Исходя из этой формулы и того факта, что значение `nice` должно находиться в диапазоне от +19 (низкое) до –20 (высокое), можно прийти к выводу, что фактический полезный диапазон ограничения `RLIMIT_NICE` заключен между 1 (низкий) и 40 (высокий). `RLIMIT_NICE` не использует диапазон от +19 до –20, поскольку некоторые отрицатель-

ные значения ограничений несут в себе особую информацию — например, ограничение RLIM_INFINITY представлено в виде числа –1).

С помощью вызова setpriority() непривилегированный процесс может изменить значение nice другого (целевого) процесса, если действующий пользовательский идентификатор первого совпадает с действующим или реальным пользовательским идентификатором второго, и если это изменение соответствует ограничению RLIMIT_NICE целевого процесса.

Программа из листинга 35.1 задействует вызов setpriority() для изменения значения nice процесса(ов), заданного с помощью аргументов командной строки (которые соответствуют аргументам самого вызова setpriority()), и затем вызывает getpriority(), чтобы проверить результат.

Листинг 35.1. Изменение и получение значения nice процесса

procpri/t_setpriority.c

```
#include <sys/time.h>
#include <sys/resource.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int which, prio;
    id_t who;

    if (argc != 4 || strchr("pgu", argv[1][0]) == NULL)
        usageErr("%s {p|g|u} who priority\n"
    "    set priority of: p=process; g=process group; "
    "u=processes for user\n", argv[0]);

    /* Устанавливаем значение nice в соответствии с аргументами командной строки */
    which = (argv[1][0] == 'p') ? PRIO_PROCESS :
        (argv[1][0] == 'g') ? PRIO_PGRP : PRIO_USER;
    who = getLong(argv[2], 0, "who");
    prio = getInt(argv[3], 0, "prio");

    if (setpriority(which, who, prio) == -1)
        errExit("setpriority");

    /* Получаем значение nice, чтобы проверить изменение */

    errno = 0;        /* Потому что успешный вызов может вернуть -1 */
    prio = getpriority(which, who);
    if (prio == -1 && errno != 0)
        errExit("getpriority");
    printf("Nice value = %d\n", prio);

    exit(EXIT_SUCCESS);
}
```

procpri/t_setpriority.c

35.2. Обзор планирования в режиме реального времени

Стандартный алгоритм планирования ядра, как правило, обеспечивает достаточную производительность и отзывчивость для смеси интерактивных и фоновых процессов, которые

обычно выполняются в системе. Однако приложения, работающие в режиме реального времени, имеют более высокие требования к планировщику.

- Приложение реального времени должно гарантировать некое максимальное время реакции на внешний ввод. Во многих ситуациях это время должно быть довольно небольшим (например, порядка нескольких долей секунды). Например, медленная реакция навигационной системы автомобиля может привести к катастрофе. Чтобы удовлетворить это требование, ядро обязано предоставить высокоприоритетным процессам возможность своевременно получать ресурсы процессора, упреждая любые программы, которые могут работать в этот момент.

  Приложениям, чувствительным к задержкам, иногда приходится предпринимать дополнительные меры, чтобы обеспечить своевременное выполнение. Например, чтобы избежать задержек, связанных с отказами страницы, оно может запретить перенос своего виртуального адресного пространства за пределы оперативной памяти, используя вызовы mlock() или mlockall().

- Высокоприоритетному процессу должен быть предоставлен эксклюзивный доступ к процессору, пока он не закончит работу или добровольно не освободит ресурсы.
- Приложение реального времени должно иметь возможность контролировать порядок, в котором выполняются его отдельные процессы.

Стандарт SUSv3 описывает программный интерфейс для планирования процессов в режиме реального времени (изначально входил в POSIX.1b), который частично отвечает описанным выше требованиям. Этот интерфейс предоставляет две политики планирования: `SCHED_RR` и `SCHED_FIFO`. Процессы, работающие в их рамках, всегда имеют приоритет перед процессами, планируемыми в соответствии с алгоритмом циклического разделения времени, описанного в разделе 35.1 (определяется константой `SCHED_OTHER`).

Каждая из этих политик предоставляет диапазон отдельных приоритетов, количество которых согласно стандарту SUSv3 должно быть не меньше 32. В рамках любой политики планирования процесс с более высоким приоритетом всегда «обгоняет» менее приоритетные процессы в соперничестве за процессорное время.

Утверждение о том, что процесс с повышенным приоритетом всегда обгоняет менее приоритетные процессы, относится к многопроцессорным (в том числе и гиперпоточным) Linux-системам. Каждый процессор в таких системах имеет свою отдельную очередь выполнения (что обеспечивает лучшую производительность по сравнению с единой системной очередью), в рамках которой и расставляются приоритеты. Представьте, что на двухпроцессорном компьютере выполняется три процесса; процесс А с приоритетом реального времени 20 может попасть в очередь ожидания процессора 0, который в этот момент выполняет процесс Б с приоритетом 30, хотя в то же самое время на процессоре 1 работает процесс В с приоритетом 10.

Приложения реального времени, состоящие из нескольких процессов (или потоков), могут использовать программный интерфейс для выбора родственного процессора, описанный в разделе 35.4, чтобы избежать проблем, связанных с планированием. Например, в четырехпроцессорном компьютере все некритичные процессы могут быть изолированы на едином ЦПУ, оставляя три других процессора доступными для вашего приложения.

Linux предоставляет 99 приоритетов реального времени: от 1 (низшего) до 99 (высшего); этот диапазон распространяется на обе политики планирования (`SCHED_RR` и `SCHED_FIFO`). Приоритеты в каждой из политик являются эквивалентными. Это означает, что любой из двух процессов с одинаковыми приоритетами, но разными политиками может в одинаковой степени претендовать на получение процессорного времени, в зависимости от порядка, в котором они были запланированы. В сущности,

каждый приоритет хранит очередь выполняющихся процессов, и процесс, который будет выполнен следующим, берется из начала (непустой) очереди с самым высоким приоритетом.

Жесткий режим реального времени по сравнению с режимом POSIX

Режим реального времени, который удовлетворяет всем требованиям, перечисленным в начале этого раздела, иногда называют *жестким*. Однако программный интерфейс POSIX для планирования процессов в режиме реального времени поддерживает не все из этих возможностей. В частности, он не позволяет приложениям гарантировать время реакции на ввод. Для обеспечения таких гарантий нужны инструменты, которые не входят в основную ветку ядра Linux (как и ни в одну другую стандартную операционную систему). Программный интерфейс POSIX предоставляет всего лишь так называемый *мягкий* режим реального времени, который позволяет управлять планированием выполнения процессов.

Добавление поддержки жесткого режима реального времени сложно достичь без затраты дополнительных ресурсов; это не позволило бы достичь производительности, необходимой для приложений, планируемых по алгоритму разделения времени (коих большинство как в настольных, так и в серверных системах). Именно поэтому большинство ядер в UNIX-системах, включая Linux, изначально не имели стандартной поддержки приложений реального времени. Тем не менее, начиная с версии 2.6.18, в ядре Linux стали появляться возможности, которые в конечном счете должны обеспечить полную поддержку жесткого режима реального времени без вышеупомянутой затраты ресурсов для операций разделения времени.

35.2.1. Политика SCHED_RR

В рамках политики `SCHED_RR` (циклического разделения) процессы с одинаковым приоритетом выполняются в соответствии с циклическим разделением времени. При каждом использовании ЦПУ процесс получает квант времени фиксированной длины и выполняется до тех пор, пока он не:

- достигнет конца своего временного отрезка;
- добровольно освободит ресурсы процессора — либо выполнив блокирующую системную операцию, либо сделав вызов `sched_yield()` (описанный в подразделе 35.3.3);
- завершится;
- будет вытеснен процессом с более высоким приоритетом.

В случае с первыми двумя событиями, описанными выше, процесс, который выполняется с политикой `SCHED_RR` и теряет доступ к ЦПУ, возвращается обратно в очередь в соответствии со своим приоритетом. В последнем случае, когда высокоприоритетный процесс завершил выполнение, вытесненный процесс продолжает работу, потребляя оставшуюся часть выделенного ему отрезка времени (оставаясь при этом во главе очереди для своего приоритета).

Согласно политикам `SCHED_RR` и `SCHED_FIFO` выполняющийся процесс может быть вытеснен по одной из следующих причин:

- процесс с более высоким приоритетом, который был блокирован ранее, разблокировался (например, в результате завершения операции ввода/вывода, которую он ожидал);
- приоритет другого процесса был повышен до уровня, превышающего приоритет выполняющегося процесса;
- приоритет текущего процесса был понижен до уровня, уступающего приоритету другого выполняющегося процесса.

Политика SCHED_RR похожа на стандартный алгоритм планирования с циклическим разделением времени (SCHED_OTHER) в том смысле, что она позволяет группе процессов с одинаковым приоритетом иметь общий доступ к процессору. Главной же отличительной чертой является наличие четких приоритетов, которые однозначно определяют очередность выполнения процессов. Малое значение nice (то есть высокий приоритет), напротив, не гарантирует процессу эксклюзивного доступа к ЦПУ; оно просто играет роль весового коэффициента при планировании. Как было отмечено в разделе 35.1, процесс с низким приоритетом (то есть большим значением nice) всегда получает какое-то процессорное время. Еще одно важное отличие состоит в том, что политика SCHED_RR дает возможность определять точный порядок выполнения процессов.

35.2.2. Политика SCHED_FIFO

Политика SCHED_FIFO («первым пришел — первым ушел») похожа на политику SCHED_RR. Главное, что ее отличает, — это отсутствие кванта времени. Получив доступ к ЦПУ, процесс продолжает выполнение до тех пор, пока:

- добровольно не освободит ресурсы процессора (тем же способом, как это было описано выше в случае с политикой SCHED_RR);
- не завершится;
- не будет вытеснен процессом с более высоким приоритетом (при тех же обстоятельствах, что были описаны выше в случае с политикой SCHED_RR).

В первом случае процесс возвращается обратно в очередь в соответствии со своим приоритетом. В последнем случае, когда процесс с более высоким приоритетом прекратил выполнение (завершившись или заблокировавшись), вытесненный процесс продолжает свою работу (оставаясь во главе очереди в соответствии со своим приоритетом).

35.2.3. Политики SCHED_BATCH и SCHED_IDLE

В ветку ядра Linux 2.6 были добавлены две нестандартные политики: SCHED_BATCH и SCHED_IDLE. Несмотря на то что они входят в состав программного интерфейса реального времени POSIX, они на самом деле не обеспечивают режим реального времени.

Политика SCHED_BATCH появилась в версии Linux 2.6.16 и напоминает политику SCHED_OTHER, которая используется по умолчанию. Разница заключается в том, что SCHED_BATCH реже выделяет процессорное время заданиям, которые часто возобновляют свою работу. Эта политика нацелена на пакетное выполнение процессов.

Политика SCHED_IDLE, которая была добавлена в ядро версии 2.6.23, тоже похожа на SCHED_OTHER, но обеспечивает поведение, эквивалентное очень маленькому значению nice (то есть ниже +19). Само значение nice процесса не играет в контексте этой политики никакой роли. Данная политика нацелена на выполнение заданий, которые получают существенную долю процессорного времени только в том случае, если ни одно другое задание в системе с ними не конкурирует.

35.3. Программный интерфейс планирования в режиме реального времени

В этом разделе будут рассмотрены различные системные вызовы, составляющие программный интерфейс для планирования в реальном времени. С помощью этих вызовов мы сможем управлять политиками планирования и приоритетами.

Несмотря на то что планирование в режиме реального времени поддерживается в ядре Linux, начиная с версии 2.0, на протяжении долгого времени его реализации были присущи различные проблемы. В ядрах ветки 2.2 (и даже в начальных версиях 2.4) целый ряд возможностей оставался в нерабочем состоянии. Большинство из этих проблем было исправлено примерно с выходом ядра 2.4.20.

35.3.1. Диапазон приоритетов реального времени

Системные вызовы `sched_get_priority_min()` и `sched_get_priority_max()` возвращают диапазон, доступный для политики планирования.

```
#include <sched.h>

int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Оба вызова возвращают либо неотрицательный целочисленный приоритет, либо `-1`, если произошла ошибка

В обоих системных вызовах аргумент `policy` обозначает политику планирования, информацию о которой нужно получить; он может принимать значения `SCHED_RR` и `SCHED_FIFO`. Системный вызов `sched_get_priority_min()` возвращает минимальный приоритет для заданной политики, а `sched_get_priority_max()` — максимальный. В Linux эти вызовы возвращают соответственно `1` и `99` как для `SCHED_RR`, так и для `SCHED_FIFO`. Иными словами, диапазоны приоритетов для обеих политик планирования в реальном времени полностью совпадают, и процессы, которые используют любую из них и имеют одинаковый приоритет, в равной степени претендуют на процессорное время (то, какой из них окажется первым, зависит от порядка их размещения в очереди с соответствующим приоритетом).

Диапазон приоритетов реального времени зависит от конкретной реализации UNIX. Следовательно, вместо использования заранее подготовленных приоритетов мы должны делать поправку на значения, возвращаемые одной из этих функций. Например, самый низкий приоритет для политики `SCHED_RR` равен `sched_get_priority_min(SCHED_FIFO)`, следующий за ним — `sched_get_priority_min(SCHED_FIFO) + 1` и т. д.

Стандарт SUSv3 не требует использования одинаковых диапазонов приоритетов для политик SCHED_RR и SCHED_FIFO, однако этот подход применяется в большинстве реализаций UNIX. Например, в Solaris 8 приоритеты этих политик находятся в диапазоне от 0 до 59, а в FreeBSD 6.1 — от 0 до 31.

35.3.2. Изменение и получение политик и приоритетов

В этом разделе мы рассмотрим системные вызовы, позволяющие изменять и получать данные о политиках планирования и приоритетах.

Изменение политик планирования и приоритетов

Системный вызов `sched_setscheduler()` изменяет политику планирования и приоритет процесса с идентификатором `pid`. Если аргумент `pid` равен 0, изменяются атрибуты вызывающего процесса.

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
```

Возвращает 0 при успешном завершении или –1, если произошла ошибка

Аргумент param является указателем на структуру следующего вида:

```
struct sched_param {
    int sched_priority;     /* Приоритет планирования */
};
```

В стандарте SUSv3 аргумент param описывается как структура, позволяющая указывать нестандартные поля, которые могут пригодиться, если система предоставляет дополнительные политики планирования. Но, как и большинство UNIX-систем, Linux предоставляет только поле sched_priority, которое описывает приоритет планирования. В случае с политиками SCHED_RR и SCHED_FIFO его значение должно входить в диапазон, возвращаемый вызовами sched_get_priority_min() и sched_get_priority_max(); для других политик приоритет должен быть равен 0.

Аргумент policy определяет политику планирования процесса. Значения, которые он может принимать, перечислены в табл. 35.1.

Таблица 35.1. Политики планирования в Linux: обычные и в режиме реального времени

Политика	Описание	SUSv3
SCHED_FIFO	Режим реального времени («первым пришел — первым ушел»)	*
SCHED_RR	Циклическое планирование реального времени	*
SCHED_OTHER	Стандартное циклическое разделение времени	*
SCHED_BATCH	Похожа на SCHED_OTHER, но нацелена на пакетное выполнение (начиная с Linux 2.6.16)	
SCHED_IDLE	Похожа на SCHED_OTHER, но с приоритетом, даже ниже чем значение nice +19 (начиная с Linux 2.6.23)	

При успешном выполнении вызов sched_setscheduler() перемещает процесс с идентификатором pid в конец очереди для соответствующего приоритета.

В стандарте SUSv3 сказано, что при успешном выполнении вызов sched_setscheduler() должен вернуть предыдущую политику планирования. Однако Linux в этом аспекте отклоняется от стандарта и в случае успеха возвращает 0. Переносимые приложения должны определять успешность вызова, проверяя, не вернул ли он –1.

Политика планирования и приоритет наследуются потомком, созданным с помощью fork(), и сохраняются на протяжении работы вызова exec().

Системный вызов sched_setparam() является частичным аналогом sched_setscheduler(). Он изменяет приоритет планирования процесса, не затрагивая его политику.

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```

Возвращает 0 при успешном завершении или –1, если случилась ошибка

Аргументы `pid` и `param` аналогичны тем, что используются в вызове `sched_setscheduler()`.

В случае успеха вызов `sched_setparam()` перемещает процесс с идентификатором `pid` в конец очереди для соответствующего приоритета.

Программа, представленная в листинге 35.2, использует вызов `sched_setscheduler()`, чтобы установить политику и приоритет процессов, указанных в виде аргументов командной строки. Первый аргумент — это буква, определяющая политику планирования, а второй — целочисленный приоритет; остальные аргументы представляют собой список идентификаторов процессов, атрибуты планирования которых нужно изменить.

Листинг 35.2. Изменение политики планирования и приоритета процессов

procpri/sched_set.c

```
#include <sched.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j, pol;
    struct sched_param sp;

    if (argc < 3 || strchr("rfobi", argv[1][0]) == NULL)
        usageErr("%s policy priority [pid...]\n"
                 "    policy is 'r' (RR), 'f' (FIFO), "
#ifdef SCHED_BATCH        /* Относится только к Linux */
                 "'b' (BATCH), "
#endif
#ifdef SCHED_IDLE         /* Относится только к Linux */
                 "'i' (IDLE), "
#endif
                 "or 'o' (OTHER)\n",
                 argv[0]);

    pol = (argv[1][0] == 'r') ? SCHED_RR :
                 (argv[1][0] == 'f') ? SCHED_FIFO :
#ifdef SCHED_BATCH
                 (argv[1][0] == 'b') ? SCHED_BATCH :
#endif
#ifdef SCHED_IDLE
                 (argv[1][0] == 'i') ? SCHED_IDLE :
#endif
                 SCHED_OTHER;
    sp.sched_priority = getInt(argv[2], 0, "priority");
    for (j = 3; j < argc; j++)
        if (sched_setscheduler(getLong(argv[j], 0, "pid"), pol, &sp) == -1)
            errExit("sched_setscheduler");

    exit(EXIT_SUCCESS);
}
```

procpri/sched_set.c

Привилегии и ограничения на ресурсы, которые влияют на изменение параметров планирования

В ядрах версии ниже 2.6.12 менять политику планирования и приоритет обычно позволено только привилегированным процессам (`CAP_SYS_NICE`) . Единственное исключение

к этому требованию состоит в том, что непривилегированный процесс может изменять политику планирования на SCHED_OTHER, если его действующий пользовательский идентификатор совпадает с таковым у целевого процесса.

В версии ядра 2.6.12 появился новое, нестандартное ограничение ресурсов под названием RLIMIT_RTPRIO, которое изменило вышеописанные правила. Как и в старых ядрах, привилегированный (CAP_SYS_NICE) процесс может вносить любые изменения в политику и приоритет любой программы. Однако теперь то же самое может делать и обычный процесс. Для этого должны выполняться следующие условия.

- Если процесс имеет мягкое ненулевое ограничение RLIMIT_RTPRIO, он может вносить произвольные изменения в свою политику планирования и приоритет. При этом учитывая, что верхнее ограничение для приоритета реального времени является максимальным значением его текущего приоритета (если процесс в этот момент работает в рамках политики реального времени) и значением RLIMIT_RTPRIO.
- Если мягкое ограничение процесса RLIMIT_RTPRIO равно 0, все, что он может сделать, — понизить свой приоритет планирования или переключиться с политики реального времени на обычную.
- Политика SCHED_IDLE является особенной. Процесс, который работает под ее управлением, не может менять свою политику, каким бы ни было значение ограничения RLIMIT_RTPRIO.
- Изменения политики и приоритета можно также выполнять из другого непривилегированного процесса, если его действующий пользовательский идентификатор совпадает с реальным или действующим пользовательским идентификатором (именем пользователя) целевого процесса.
- Мягкое ограничение процесса RLIMIT_RTPRIO определяет только те изменения, которые он сам или другой привилегированный процесс может вносить в политику планирования и приоритет. Ненулевое ограничение не дает непривилегированному процессу право изменять политику планирования и приоритет других процессов.

Получение данных о политиках планирования и приоритетах

Системные вызовы sched_getscheduler() и sched_getparam() позволяют получить сведения о политике планирования и приоритете процесса.

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

Возвращает политику планирования или –1, если произошла ошибка

```
int sched_getparam(pid_t pid, struct sched_param *param);
```

Возвращает 0 при успешном завершении или –1, если произошла ошибка

В обоих этих системных вызовах аргумент pid обозначает идентификатор процесса, информацию о котором нужно получить. Если он равен 0, возвращаются данные о вызывающем процессе. Оба вызова могут быть использованы непривилегированными процессами для получения информации о любом другом процессе вне зависимости от привилегий.

Системный вызов sched_getparam() возвращает политику планирования реального времени в поле sched_priority внутри структуры sched_param, на которую указывает аргумент param.

В случае успешного выполнения `sched_getscheduler()` возвращает одну из политик, перечисленных ранее в табл. 35.1.

Программа, представленная в листинге 35.3, задействует вызовы `sched_getscheduler()` и `sched_getparam()` для получения сведений о политике и приоритете всех процессов, чьи идентификаторы переданы в качестве аргументов командной строки. Использование этой программы, а также программы из листинга 35.2, демонстрируется на примере следующей сессии командной оболочки:

```
$ su                        Получаем привилегии, чтобы иметь возможность
                            устанавливать политику реального времени
Password:
# sleep 100 &               Создаем процесс
[1] 2006
# ./sched_view 2006         Выводим исходную политику процесса sleep
2006: OTHER 0
# ./sched_set f 25 2006     Назначаем процессу политику SCHED_FIFO и приоритет 25
# ./sched_view 2006         Выводим изменения
2006: FIFO 25
```

Листинг 35.3. Получение сведений о политике планирования и приоритете

procpri/sched_view.c

```
#include <sched.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j, pol;
    struct sched_param sp;

    for (j = 1; j < argc; j++) {
        pol = sched_getscheduler(getLong(argv[j], 0, "pid"));
        if (pol == -1)
            errExit("sched_getscheduler");

        if (sched_getparam(getLong(argv[j], 0, "pid"), &sp) == -1)
            errExit("sched_getparam");

        printf("%s: %-5s ", argv[j],
            (pol == SCHED_OTHER) ? "OTHER" :
            (pol == SCHED_RR) ? "RR" :
            (pol == SCHED_FIFO) ? "FIFO" :
#ifdef SCHED_BATCH        /* Относится только к Linux */
            (pol == SCHED_BATCH) ? "BATCH" :
#endif
#ifdef SCHED_IDLE        /* Относится только к Linux */
            (pol == SCHED_IDLE) ? "IDLE" :
#endif
            "???");
        printf("%2d\n", sp.sched_priority);
    }

    exit(EXIT_SUCCESS);
}
```

procpri/sched_view.c

Предотвращение блокирования системы процессами реального времени

Поскольку процессы SCHED_RR и SCHED_FIFO опережают любую низкоприоритетную программу (например, командную оболочку, в которой эти процессы запущены), при разработке приложений, использующих эти политики, необходимо учитывать возможность того, что слишком активный процесс, работающий в режиме реального времени, может занять все ресурсы ЦПУ и заблокировать систему. Существует несколько программных способов этого избежать.

- Задействуя вызов setrlimit(), установить мягкое ограничение для процессорного времени (RLIMIT_CPU, описанный в разделе 36.3) на достаточно низком уровне. Если процесс потребляет слишком много ресурсов ЦПУ, ему будет послан сигнал SIGXCPU, который по умолчанию должен привести к его завершению.
- Установить таймер с помощью вызова alarm(). Если процесс продолжает работать на протяжении времени, которое превышает количество секунд, заданное в таймере, он будет завершен по сигналу SIGALRM.
- Создать сторожевой процесс, который работает с высоким приоритетом реального времени. Он может постоянно находиться в цикле, переходя в режим сна на определенное время, и затем проверяя состояние других процессов. Такой мониторинг может включать в себя измерение процессорного времени, потребляемого процессами (см. обсуждение функции clock_getcpuclockid() в подразделе 23.5.3), а также проверку их политики планирования и приоритета с помощью вызовов sched_getscheduler() и sched_getparam(). Если какой-то процесс ведет себя неправильно, сторожевой поток может понизить его приоритет или послать ему сигнал, который его остановит или завершит.
- Начиная с версии 2.6.25, ядро Linux предоставляет нестандартное ограничение на ресурсы, RLIMIT_RTTIME, позволяющее контролировать объем процессорного времени, которое может быть потреблено во время одного всплеска активности процесса, работающего в режиме реального времени. Ограничение RLIMIT_RTTIME указывается в микросекундах и ограничивает количество процессорного времени, которое процесс может потребить без выполнения блокирующих системных вызовов. При выполнении такого вызова насчитанное время сбрасывается к нулю. Однако этого не происходит, если процесс опережается другой, более приоритетной программой, если истекает временной отрезок, выделенный ему планировщиком (при использовании политики SCHED_RR), или в случае выполнения вызовов sched_yield() (см. подраздел 35.3.3). Когда процесс достигает установленного ограничения, то, как и в случае с ограничением RLIMIT_CPU, ему передается сигнал SIGXCPU, который по умолчанию приводит к его завершению.

  Это нововведение, появившееся в ядре версии 2.6.25, также помогает предотвратить блокирование системы слишком активными процессами. Подробности ищите в исходном файле ядра Documentation/scheduler/sched-rt-group.txt.

Предотвращение наследования привилегированной политики планирования дочерними процессами

В Linux 2.6.32 было добавлено значение SCHED_RESET_ON_FORK, которое можно указать для аргумента policy при вызове sched_setscheduler(). Это флаг, который с помощью логического ИЛИ используется совместно с политиками, перечисленными в табл. 35.1. Если его установить, потомок, созданный процессом с помощью вызова fork(), не наследует его привилегированную политику планирования и приоритет. При этом действуют следующие правила.

- Если процесс имеет политику планирования реального времени (SCHED_RR или SCHED_FIFO), политика потомка сбрасывается к стандартному циклическому алгоритму разделения времени SCHED_OTHER.
- Если процесс имеет отрицательное (то есть высокое) значение nice, это значения для потомка сбрасывается к нулю.

Флаг SCHED_RESET_ON_FORK был создан для использования в приложениях, воспроизводящих мультимедийные данные. Он позволяет создавать отдельные процессы, от которых нельзя наследовать политику выполнения реального времени. Применение этого флага помогает избежать так называемых fork-бомб, которые пытаются обойти ограничения, наложенные ограничением RLIMIT_RTTIME, создавая множество потомков, работающих в режиме реального времени.

Если флаг SCHED_RESET_ON_FORK уже был установлен, отменить его может только привилегированный процесс (CAP_SYS_NICE). Когда создается потомок, флаг SCHED_RESET_ON_FORK для него автоматически отменяется.

35.3.3. Освобождение ресурсов процессора

Процессы реального времени могут добровольно освободить ресурсы ЦПУ двумя способами: выполнив блокирующий системный вызов (например, чтение из терминала) или воспользовавшись вызовом sched_yield().

```
#include <sched.h>

int sched_yield(void);
```

Возвращает 0 при успешном завершении или –1, если произошла ошибка

Принцип работы sched_yield() достаточно простой. Если процесс находится на одном уровне приоритетов с другими процессами, ожидающими выполнения, он перемещается в конец очереди, а процесс, находящийся в ее начале, получает процессорное время. Если в очереди с этим приоритетом нет ни одного другого процесса, вызов sched_yield() ничего не делает; вызывающий процесс просто продолжает свою работу.

Стандарт SUSv3 допускает возвращение ошибок вызовом sched_yield(), однако в Linux, как и во многих других реализациях UNIX, он всегда завершается успешно. Тем не менее переносимые приложения обязательно должны делать проверку на ошибки.

Использование вызова sched_yield() вне политики реального времени не определено.

35.3.4. Временной отрезок в политике SCHED_RR

Системный вызов sched_rr_get_interval() позволяет определять длину каждого временного отрезка, на протяжении которого процесс с политикой SCHED_RR может использовать ресурсы ЦПУ.

```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

Возвращает 0 при успешном завершении или –1, если произошла ошибка

Как и в случае с другими системными вызовами, предназначенными для планирования, аргумент `pid` обозначает процесс, информацию о котором мы хотим получить, а чтобы задать вызывающий процесс, используется значение 0. Временной отрезок возвращается внутри структуры `timespec`, на которую указывает аргумент `tp`:

```
struct timespec {
    time_t tv_sec;      /* Секунды */
    long tv_nsec;       /* Наносекунды */
};
```

В последних ядрах ветки 2.6 циклический отрезок реального времени равен 0,1 секунды.

35.4. Привязка к процессору

Когда планирование происходит в многопроцессорной системе, работа процесса может быть запланирована не на том ЦПУ, на котором он выполнялся до этого. Обычно причина заключается в том, что тот процессор уже занят.

Смена процессоров отражается на производительности: чтобы загрузить линию данных процесса в кэш нового ЦПУ, они сначала должно быть освобождены (удалены, если их никто не успел изменить, или просто сброшены в главную память) из кэша предыдущего процессора (чтобы избежать несогласованности кэшей, многопроцессорные архитектуры позволяют хранить данные в кэше только одного ЦПУ). Эта процедура увеличивает время выполнения. По этой причине ядро Linux (начиная с 2.6) пытается выполнять *мягкую* привязку процесса к ЦПУ, то есть по возможности работа процесса планируется на одном и том же процессоре.

Кэш-линия является аналогом страницы в системе управления виртуальной памятью. Она имеет размер, который используется для передачи данных между кэшем процессора и главной памятью. Обычно размер линии варьируется в пределах от 32 до 128 байт. Больше информации можно найти в источниках [Schimmel, 1994] и [Drepper, 2007].

Одно из полей файла /proc/PID/stat (доступного только в Linux) содержит количество ЦПУ, на которых выполняется или ранее выполнялся процесс. Подробности ищите на справочной странице proc(5).

В некоторых случаях предпочтительной является *жесткая* привязка процесса к ЦПУ, которая позволяет ему выполняться только на одном доступном процессоре (или подгруппе процессоров). Среди причин, по которым это может понадобиться, можно выделить следующие.

- Мы можем избежать понижения производительности, вызванного удалением данных из кэша.
- Если несколько потоков (или процессов) работают с одними и теми же данными, мы можем повысить производительность, привязав их к одному и тому же процессору; так они не будут соперничать за общие данные и смогут избежать потери кэша.
- Если приложение чувствительно к времени выполнения, ему можно выделить отдельный процессор, привязав другие процессы к остальным ЦПУ.

Linux 2.6 предоставляет два нестандартных системных вызова для изменения и получения данных о жесткой привязке процесса к ЦПУ: `sched_setaffinity()` и `sched_getaffinity()`.

Во многих других UNIX-системах тоже имеются интерфейсы для управления привязкой к процессорам. Например, HP-UX и Solaris предоставляют системный вызов pset_bind().

Системный вызов `sched_setaffinity()` привязывает к ЦПУ процесс, заданный с помощью аргумента `pid`. Если `pid` равен 0, изменения будут применены к вызывающему процессу.

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

ЦПУ, который будет привязан к процессу, описывается с помощью структуры `cpu_set_t`, на которую указывает аргумент `set`.

Привязка к процессору на самом деле выполняется на уровне отдельных потоков и может быть изменена отдельно для каждого потока в группе. Если нам нужно изменить привязку для определенного потока, мы можем указать для аргумента pid значение, которое возвращает вызов gettid(). Если аргумент pid равен 0, берется вызывающий поток.

Тип данных `cpu_set_t` реализован в виде битовой маски, однако обращаться с ним следует как с непрозрачной структурой. Все изменения в содержимое этой структуры нужно выполнять с помощью макросов `CPU_ZERO()`, `CPU_SET()`, `CPU_CLR()` и `CPU_ISSET()`.

```
#define _GNU_SOURCE
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);
```

Возвращает истину (`1`), если `cpu` входит в `set`, и ложь (`0`), если это не так

Эти макросы вносят такие изменения в набор процессоров, на который указывает `set`:

- `CPU_ZERO()` инициализирует набор `set`, делая его пустым;
- `CPU_SET()` добавляет процессор `cpu` в набор `set`;
- `CPU_CLR()` удаляет процессор `cpu` из набора `set`;
- `CPU_ISSET()` возвращает `1`, если процессор `cpu` входит в набор `set`.

Библиотека GNU C также предоставляет целый ряд других макросов для работы с наборами процессоров. Подробности ищите на справочной странице CPU_SET(3).

Процессоры в наборе пронумерованы, начиная с нуля. В заголовочном файле `<sched.h>` находится объявление константы `CPU_SETSIZE`, значение которой на один больше, чем количество процессоров, которое можно уместить в переменной `cpu_set_t (1024)`.

Аргумент `len`, который передается вызову `sched_setaffinity()`, обозначает число байтов в аргументе `set` (то есть `sizeof(cpu_set_t)`).

Следующий код позволяет процессу, указанному с помощью аргумента `pid`, выполняться четырехпроцессорной системе на любом ЦПУ, кроме первого:

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(1, &set);
CPU_SET(2, &set);
CPU_SET(3, &set);
sched_setaffinity(pid, CPU_SETSIZE, &set);
```

Если процессоры, указанные в аргументе `set`, не соответствуют ни одному ЦПУ в системе, вызов `sched_setaffinity()` завершается ошибкой `EINVAL`.

Если в наборе не указан ЦПУ, на котором процесс выполняется в текущий момент, процесс мигрирует на один из ЦПУ из заданного набора.

Непривилегированный процесс может привязать другую программу к определенному ЦПУ только в том случае, только если его действующий пользовательский идентификатор совпадает с действующим или реальным пользовательским идентификатором этой программы. Привилегированный процесс (`CAP_SYS_NICE`) может устанавливать привязку к ЦПУ для любой программы.

Системный вызов `sched_getaffinity()` получает маску родственного процессора для процесса, указанного с помощью аргумента `pid`. Если `pid` равен 0, возвращается маска вызывающего процесса.

```
#define _GNU_SOURCE
#include <sched.h>

int sched_getaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

Маска родственного процессора возвращается внутри структуры `cpu_set_t`, на которую указывает аргумент `set`. Аргументу `len` нужно присвоить число байтов в структуре (то есть `sizeof(cpu_set_t)`). С помощью макроса `CPU_ISSET()` можно определить, какие процессоры находятся в возвращенном значении `set`.

Если маска родственного процессора заданного процесса больше не была никаким образом изменена, вызов `sched_getaffinity()` возвращает набор, состоящий из всех процессоров в системе.

Вызов `sched_getaffinity()` не проверяет привилегии; непривилегированный процесс может получить маску родственного процессора для любого процесса в системе.

Дочерний процесс, созданный с помощью вызова `fork()`, наследует маску родственного процессора своего родителя и сохраняет ее на протяжении всего выполнения функции `exec()`.

Системные вызовы `sched_setaffinity()` и `sched_getaffinity()` доступны только в Linux.

Программы t_sched_setaffinity.c и t_sched_getaffinity.c, которые можно найти в подкаталоги procpri внутри архива с исходными кодами, прилагающегося к этой книге, демонстрируют применение вызовов sched_setaffinity() и sched_getaffinity().

35.5. Резюме

Алгоритм планирования, который по умолчанию применяется в ядре, использует политику циклического разделения времени. Согласно ей все процессы имеют одинаковые

шансы получить доступ к ресурсам ЦПУ; однако мы можем присвоить значению nice число в диапазоне от –20 (высокий приоритет) до +19 (низкий приоритет), чтобы сделать процесс более или менее предпочтительным при планировании. Но даже процессы с наименьшим приоритетом не могут быть полностью лишены процессорного времени.

Linux также реализует политики планирования реального времени, входящие в состав стандарта POSIX. Они позволяют приложению строго контролировать процедуру выделения ресурсов ЦПУ. Процессы, работающие в рамках этих двух политик (SCHED_RR — циклическая и SCHED_FIFO — «первым пришел — первым ушел») всегда имеют повышенный приоритет по сравнению со всеми другими процессами. Приоритет процессов реального времени варьируется от 1 (низкий) до 99 (высокий). Выполняясь, высокоприоритетный процесс полностью блокирует выдачу ресурсов ЦПУ процессам с низким приоритетом. Процесс, работающий в рамках политики SCHED_FIFO, сохраняет эксклюзивный доступ к процессору до тех пор, пока не завершится или добровольно не освободит ресурсы или пока его не вытеснит более приоритетный процесс. Похожие правила применимы и к политике SCHED_RR, с той лишь разницей, что выполнение процессов с одинаковым приоритетом регулируется циклическим алгоритмом.

Маска родственного процессора позволяет привязать выполнение процесса к определенному набору доступных ЦПУ (если речь идет о многопроцессорной системе). Это может улучшить производительность некоторых видов приложений.

Дополнительная информация

Книга [Love, 2010] описывает подробности о приоритетах и планировании процессов в Linux в исторической перспективе. Дополнительную информацию о программном интерфейсе планирования в реальном времени, входящем в состав стандарта POSIX, можно найти в книге [Gallmeister, 1995]. Книга [Butenhof, 1996] посвящена POSIX-потокам, однако существенная ее часть покрывает программный интерфейс планирования в реальном времени и может значительно помочь при чтении этой главы.

Дополнительные сведения о привязке к процессорам и управлении выделением ресурсов ЦПУ и памяти для потоков в многопроцессорных системах ищите в исходном файле ядра Documentation/cpusets.txt, а также на справочных страницах mbind(2), set_mempolicy(2) и cpuset(7).

35.6. Упражнения

35.1. Реализуйте команду nice(1).

35.2. Напишите программу, которая устанавливает идентификатор администратора и является аналогом команды nice(1), работающим в режиме реального времени. Ее интерфейс командной строки должен быть следующим:

```
# ./rtsched policy priority command arg...
```

Аргумент policy может принимать два значения: r для SCHED_RR или f для SCHED_FIFO. Перед выходом данная программа должна сбрасывать свой привилегированный идентификатор (причины описаны в подразделе 9.7.1 и разделе 38.3).

35.3. Напишите программу, которая устанавливает для себя политику планирования SCHED_FIFO, после чего создает дочерний процесс. Оба процесса должны выполнить функцию, которая приводит к максимальному потреблению ресурсов процессора на протяжении 3 секунд (это можно сделать с помощью цикла, внутри которого делается системный вызов times(), определяющий количество процессорного времени, потребленного на текущий момент). После каждой четверти секунды потраченного

процессорного времени функция должна выводить сообщение с идентификатором процесса и количеством потребленных им ресурсов ЦПУ. После первой секунды потраченного процессорного времени функция должна вызывать `sched_yield()`, чтобы освободить ЦПУ для других процессов (как вариант, процессы могут повышать приоритеты друг друга с помощью `sched_setparam()`). Вывод программы должен показать, что оба процесса поочередно выполнялись на протяжении одной секунды (отнеситесь серьезно к совету о недопустимости чрезмерного потребления ресурсов ЦПУ процессом реального времени, который был дан в подразделе 35.3.2).

35.4. Если в многопроцессорной системе два процесса обмениваются большими объемами данных через конвейер, их взаимодействие должно ускориться, если выполнять их на одном и том же ЦПУ. Дело в том, что при использовании одного процессора скорость доступа к данным конвейера возрастает, поскольку эти данные могут оставаться в кэше. Если применяются разные процессоры, преимущества хранения данных в кэше утрачиваются. Если у вас есть возможность работать на многопроцессорном компьютере, напишите программу, которая демонстрирует этот эффект с помощью вызова `sched_setaffinity()`, привязывая процессы сначала к одному, а потом к разным ЦПУ (применение конвейеров описано в главе 44).

Преимущества от выполнения процессов на одном и том же ЦПУ не распространяются на гиперпоточные и некоторые современные многопроцессорные архитекторы, в которых процессоры используют общий кэш. В таких системах процессы работают быстрее на разных ЦПУ. Сведения о структуре многопроцессорной системы можно получить, изучив содержимое доступного только в Linux файла /proc/cpuinfo.

36 Ресурсы процессов

Каждый процесс потребляет системные ресурсы, такие как память и процессорное время. Эта глава посвящена системным вызовам, связанным с подобной информацией. Мы начнем с вызова `getrusage()`, который позволяет процессу следить за ресурсами, потребленными им или его потомками. Затем будут рассмотрены вызовы `setrlimit()` и `getrlimit()`, которые позволяют изменять и получать данные об установленных для вызывающего процесса ограничениях на различные ресурсы.

36.1. Ресурсы, использующиеся процессом

Системный вызов `getrusage()` возвращает статистику, которая касается различных ресурсов системы, потребленных самим вызывающим процессом или всеми его потомками.

```
#include <sys/resource.h>

int getrusage(int who, struct rusage *res_usage);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

Аргумент `who` обозначает процесс (-ы), для которого (-ых) будет извлекаться информация о потреблении ресурсов. Он может принимать одно из следующих значений.

- `RUSAGE_SELF` — вызывает сведения о вызывающем процессе.
- `RUSAGE_CHILDREN` — возвращает сведения обо всех потомках вызывающего процесса, которые были завершены и которых он ожидал.
- `RUSAGE_THREAD` (начиная с Linux 2.6.26) — возвращает сведения о вызывающем потоке. Поддерживается только в Linux.

Аргумент `res_usage` представляет собой указатель на структуру типа `rusage`, объявление которой показано в листинге 36.1.

Листинг 36.1. Объявление структуры rusage

```
struct rusage {
    struct timeval  ru_utime;    /* Процессорное время, потребленное пользователем */
    struct timeval  ru_stime;    /* Процессорное время, потребленное системой */
    long            ru_maxrss;   /* Размер страницы памяти, выделенной процессу
                                    (в килобайтах) [используется с Linux 2.6.32] */
    long            ru_ixrss;    /* Интегральный объем (разделяемой) текстовой памяти
                                    (килобайты в секунду) [не используется] */
    long            ru_idrss;    /* Интегральный объем (неразделяемого)
                                    сегмента памяти с данными (килобайты
                                    в секунду) [не используется] */
    long            ru_isrss;    /* Интегральный объем (неразделяемого)
                                    стека (килобайты в секунду) [не используется] */
    long            ru_minflt;   /* Мягкий сбой страницы памяти
                                    (ввод/вывод необязателен) */
```

```
    long            ru_majflt;   /* Жесткий сбой страницы памяти
                                    (ввод/вывод обязателен) */
    long            ru_nswap;    /* Количество сбросов физической
                                    памяти на диск [не используется] */
    long            ru_inblock;  /* Блочные операции ввода в файловой
                                    системе [используется с Linux 2.6.22] */
    long            ru_oublock;  /* Блочные операции вывода в файловой
                                    системе [используется с Linux 2.6.22] */
    long            ru_msgsnd;   /* Количество отправленных IPC сообщений
                                    [не используется] */
    long            ru_msgrcv;   /* Количество полученных IPC сообщений
                                    [не используется] */
    long            ru_nsignals; /* Количество полученных сигналов
                                    [не используется] */
    long            ru_nvcsw;    /* Добровольные переключения контекста (процесс
                                    освободил ЦПУ до истечения выделенного ему
                                    времени) [используется, начиная с Linux 2.6] */
    long            ru_nivcsw;   /* Принудительные переключения контекста (выделенное
                                    время истекло, или начал работу более приоритетный
                                    процесс) [используется, начиная с Linux 2.6] */
};
```

Согласно комментариям в листинге 36.1 в Linux далеко не все поля структуры `rusage` заполняются вызовом `getrusage()` (или `wait3()` и `wait4()`), а если и заполняются, то только в относительно новых версиях ядра. Некоторые поля, игнорирующиеся в Linux, используются другими реализациями UNIX. В Linux они присутствуют на случай, если их реализуют в будущем; это позволит избежать изменений структуры `rusage`, которые могут сломать бинарную совместимость с уже существующими приложениями.

> Вызов getrusage() поддерживается в большинстве UNIX-систем, однако стандарт SUSv3 описывает его довольно слабо, упоминая лишь поля ru_utime и ru_stime. В какой-то мере причиной этого является тот факт, что большая часть информации в структуре rusage зависит от конкретной реализации.

Поля `ru_utime` и `ru_stime` представляют собой структуры типа `timeval` (см. раздел 10.1), которые возвращают количество секунд и микросекунд процессорного времени, потребленное процессом в режимах соответственно пользователя и ядра (похожую информацию можно извлечь с помощью системного вызова `times()`, описанного в разделе 10.7).

> Файл /proc/PID/stat, доступный только в Linux, предоставляет некоторые сведения о потреблении ресурсов (процессорное время и сбои страниц памяти) для любых процессов в системе. Подробности ищите на справочной странице proc(5).

Структура `rusage`, возвращаемая операцией `RUSAGE_CHILDREN` для вызова `getrusage()`, включает в себя статистику о потреблении ресурсов для всех потомков вызывающего процесса. Представьте, к примеру, что у нас есть три процесса с иерархией «родитель → сын → внук»; когда сын делает вызов `wait()` для внука, сведения о потреблении ресурсов внуком добавляются к значениям `RUSAGE_CHILDREN` сына; когда вызов `wait()` выполняет родитель, к его значениям `RUSAGE_CHILDREN` добавляются сведения о потреблении ресурсов как сыном, так и внуком. И наоборот, если сын не выполняет `wait()` для внука, статистика использования ресурсов, принадлежащая внуку, не учитывается в значениях `RUSAGE_CHILDREN` родителя.

Для операции `RUSAGE_CHILDREN` поле `ru_maxrss` максимальный размер страницы памяти во всех потомках вызывающего процесса (вместо суммарного объема).

В стандарте SUSv3 говорится о том, что если сигнал SIGCHLD игнорируется (чтобы дочерние процессы не превратились в «зомби», для которых можно сделать вызов wait()), статистика о потомке не должна добавляться в значения, возвращаемые операцией RUSAGE_CHILDREN. Но, как отмечалось в подразделе 26.3.3, до версии 2.6.9 ядра Linux не следовали этому требованию — если сигнал SIGCHLD игнорируется, сведения о потреблении ресурсов завершенными потомками попадают в итоговый результат операции RUSAGE_CHILDREN.

36.2. Ограничения на ресурсы для отдельных процессов

Каждый процесс обладает набором ограничений, с помощью которых можно ограничить объем тех или иных системных ресурсов, которые тот может потребить. Например, если мы боимся, что программа может потребовать слишком много ресурсов, перед ее запуском для нее можно установить определенные ограничения. Это можно сделать с помощью команды `ulimit`, встроенной в оболочку (или `limit`, если это `csh`). Эти ограничения наследуются процессами, которые создаются командной оболочкой для выполнения пользовательских программ.

Начиная с ядра 2.6.24 к файлу /proc/PID/limits, доступный только в Linux, можно обращаться для просмотра всех ограничений на ресурсы для любого процесса. Этому файлу назначен реальный идентификатор пользователя соответствующего процесса, и читать его может только этот пользователь (или привилегированный процесс).

Сведения о ограничениях на ресурсы процесса можно получать и изменять с помощью системных вызовов `getrlimit()` и `setrlimit()`.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

Возвращает `0` при успешном завершении или `-1`, если случилась ошибка

Аргумент `resource` обозначает ограничение на ресурсы, которое нужно получить или изменить. Аргумент `rlim` используется либо для возвращения существующих значений ограничения (в `getrlimit()`), либо для задания новых (в `setrlimit()`); он представляет собой структуру, состоящую из двух полей:

```
struct rlimit {
    rlim_t rlim_cur;   /* Мягкое ограничение (настоящее ограничение для процесса) */
    rlim_t rlim_max;   /* Жесткое ограничение (максимальное значение rlim_cur) */
};
```

Эти поля имеют целочисленный тип `rlim_t` и соответствуют двум ограничениям на один ресурс: *мягкому* (`rlim_cur`) и *жесткому* (`rlim_max`). Мягкое ограничение регулирует количество ресурсов, которые могут быть использованы процессом. Процесс может присваивать ему любое значение от 0 до жесткого ограничения. В большинстве ресурсов жесткое ограничение служит исключительно для предоставления этого максимума. Привилегированный процесс (`CAP_SYS_RESOURCE`) может изменять его в любую сторону (до тех пор пока он превышает мягкое ограничение), однако обычные процессы могут его только уменьшать (без возможности отменить изменения). Значение `RLIM_INFINITY`, если его

присвоить или извлечь из поля `rlim_cur` или `rlim_max` с помощью вызовов `getrlimit()` или `setrlimit()`, делает ограничение бесконечным.

В большинстве случаев ограничения на ресурсы действуют как для привилегированных, так и для обычных процессов. Они наследуются потомками через вызов `fork()` и сохраняются на протяжении выполнения функции `exec()`.

Значения, которые можно указать в аргументе `resource` для вызовов `getrlimit()` и `setrlimit()`, перечислены в табл. 36.1 и подробно рассматриваются в разделе 36.3.

Ограничения на ресурсы относятся к отдельным процессам, однако в некоторых случаях они измеряются для всех процессов с одним и тем же реальным идентификатором пользователя. Хорошим примером, объясняющим данный подход, является ограничение `RLIMIT_NPROC`, которое устанавливает максимальное количество доступных для создания процессов. Если применить его только к потомкам, создаваемым самим процессом, это принесет мало пользы, так как каждый потомок тоже может создавать свои собственные дочерние процессы, а те, в свою очередь, могут делать то же самое и т. д. Поэтому данное ограничение относится ко всем процессам, которые имеют один и тот же реальный пользовательский идентификатор. Однако стоит отметить, что ограничение на ресурсы проверяется только в процессе, в котором оно было установлено (то есть в самом процессе и в его потомках, которые унаследовали ограничение). Если процесс с тем же реальным пользовательским идентификатором его не устанавливал (то есть он у него бесконечный) или имеет другое ограничение, количество потомков, доступных для создания, будет ограничено его собственным ограничением.

При описании каждого ограничения на ресурсы мы будем отмечать те из них, которые распространяются на все процессы с одним и тем же реальным идентификатором пользователя. Если такого примечания нет, это означает, что ограничение касается потребления ресурсов только самим процессом.

> Имейте в виду, что часто единицы измерения, которые применяются в консольных командах для получения и установления ограничений на ресурсы (ulimit в bash и ksh или limit в csh), отличаются от тех, что действуют в вызовах getrlimit() и setrlimit(). Например, для выражения ограничений на размер различных сегментов памяти консольные команды обычно используют килобайты.

Таблица 36.1. Значения ресурсов для вызовов getrlimit() и setrlimit()

resource	Ограничение на	SUSv3
RLIMIT_AS	Размер виртуальной памяти процесса (байты)	*
RLIMIT_CORE	Размер дампа памяти (байты)	*
RLIMIT_CPU	Процессорное время (секунды)	*
RLIMIT_DATA	Сегмент с данными процесса (байты)	*
RLIMIT_FSIZE	Размер файла (байты)	*
RLIMIT_MEMLOCK	Заблокированную память (байты)	
RLIMIT_MSGQUEUE	Количество байтов, выделенных под очереди POSIX-сообщений для реального пользовательского идентификатора (начиная с Linux 2.6.8)	
RLIMIT_NICE	Значение nice (начиная с Linux 2.6.12)	
RLIMIT_NOFILE	Максимальное количество файловых дескрипторов плюс один	*

Продолжение ⇗

Таблица 36.1 *(продолжение)*

resource	Ограничение на	SUSv3
RLIMIT_NPROC	Количество процессов с реальным идентификатором пользователя	
RLIMIT_RSS	Размер страницы памяти (байты; не реализован)	
RLIMIT_RTPRIO	Приоритет планирования в реальном времени (начиная с Linux 2.6.12)	
RLIMIT_RTTIME	Реальное процессорное время (микросекунды; начиная с Linux 2.6.25)	
RLIMIT_SIGPENDING	Количество сигналов в очереди для реального идентификатора пользователя (начиная с Linux 2.6.8)	
RLIMIT_STACK	Размер стека (байты)	*

Пример программы

Прежде чем переходить к подробностям каждого ограничения, рассмотрим простой пример их использования. В листинге 36.2 определяется функция `printRlimit()`, которая выводит сообщение, а также мягкое и жесткое ограничения для заданного ресурса.

> Тип данных rlim_t обычно имеет тот же вид, что и off_t; он применяется для представления ограничения на размер файла, RLIMIT_FSIZE. По этой причине при выводе значений rlim_t мы приводим их к типу long long и указываем в функции printf() спецификатор %lld (подробней об этом в разделе 5.10).

Программа, представленная в листинге 36.3, вызывает `setrlimit()`, чтобы установить мягкое и жесткое ограничения на количество процессов, которые пользователь может создать (`RLIMIT_NPROC`); для вывода значения ограничений до изменения и после применяется функция `printRlimit()`. В конце программа пытается создать как можно больше дочерних процессов. Если запустить ее с аргументами 30 (для мягкого ограничения) и 100 (для жесткого ограничения), мы увидим следующее:

```
$ ./rlimit_nproc 30 100
Initial maximum process limits:    soft=1024; hard=1024
New maximum process limits:        soft=30; hard=100
Child 1 (PID=15674) started
Child 2 (PID=15675) started
Child 3 (PID=15676) started
Child 4 (PID=15677) started
ERROR [EAGAIN Resource temporarily unavailable] fork
```

В этом примере программа сумела создать всего четыре новых процесса, поскольку 26 процессов уже выполнялось от имени текущего пользователя.

Листинг 36.2. Вывод ограничений на ресурсы процесса

procres/print_rlimit.c

```
#include <sys/resource.h>
#include "print_rlimit.h"   /* Объявление функции, которая здесь определяется */
#include "tlpi_hdr.h"

int                         /* Выводим 'msg' и ограничение для 'resource' */
printRlimit(const char *msg, int resource)
{
```

```
    struct rlimit rlim;

    if (getrlimit(resource, &rlim) == -1)
        return -1;

    printf("%s soft=", msg);
    if (rlim.rlim_cur == RLIM_INFINITY)
        printf("infinite");
#ifdef RLIM_SAVED_CUR    /* Не определено в некоторых системах */
    else if (rlim.rlim_cur == RLIM_SAVED_CUR)
        printf("unrepresentable");
#endif
    else
        printf("%lld", (long long) rlim.rlim_cur);

    printf("; hard=");
    if (rlim.rlim_max == RLIM_INFINITY)
        printf("infinite\n");
#ifdef RLIM_SAVED_MAX    /* Не определено в некоторых системах */
    else if (rlim.rlim_max == RLIM_SAVED_MAX)
        printf("unrepresentable");
#endif
    else
        printf("%lld\n", (long long) rlim.rlim_max);

    return 0;
}
```

procres/print_rlimit.c

Листинг 36.3. Установление ограничения RLIMIT_NPROC

procres/rlimit_nproc.c

```
#include <sys/resource.h>
#include "print_rlimit.h"    /* Объявление функции printRlimit() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct rlimit rl;
    int j;
    pid_t childPid;

    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s soft-limit [hard-limit]\n", argv[0]);

    printRlimit("Initial maximum process limits: ", RLIMIT_NPROC);

    /* Устанавливаем новые ограничения для процесса (по умолчанию
       жесткое и мягкое совпадают) */

    rl.rlim_cur = (argv[1][0] == 'i') ? RLIM_INFINITY :
        getInt(argv[1], 0, "soft-limit");
    rl.rlim_max = (argc == 2) ? rl.rlim_cur :
         (argv[2][0] == 'i') ? RLIM_INFINITY :
        getInt(argv[2], 0, "hard-limit");
    if (setrlimit(RLIMIT_NPROC, &rl) == -1)
        errExit("setrlimit");
```

```
    printRlimit("New maximum process limits:    ", RLIMIT_NPROC);

    /* Создаем как можно больше дочерних процессов */

    for (j = 1; ; j++) {
        switch (childPid = fork()) {
        case -1: errExit("fork");

        case 0: _exit(EXIT_SUCCESS);        /* Потомок */

        default:     /* Родитель выводит сообщение о каждом новом потомке
                        и позволяет учитывать процессы-"зомби" */
            printf("Child %d (PID=%ld) started\n", j, (long) childPid);
            break;
        }
    }
}
```

procres/rlimit_nproc.c

Значения ограничений, которые не имеют представления

В некоторых средах программирования типа данных `rlim_t` может быть недостаточно для представления всего диапазона значений, предусмотренных для определенного ограничения на ресурсы. Это случается в системах, которые предоставляют несколько сред программирования с разными размерами типа `rlim_t`. Эта проблема обычно возникает, когда среда компиляции с поддержкой больших файлов, в которой структура `off_t` занимает 64 бита, интегрируется в систему, где значения `off_t` традиционно 32-битные (в обеих средах размеры `rlim_t` и `off_t` совпадают). Это приводит к ситуации, когда процесс с 32-битным типом данных `rlim_t`, если он запущен программой с 64-битной структурой `off_t`, может наследовать ограничение на ресурсы (например, ограничение на размер файла), которое превышает максимальное значение `rlim_t`.

Чтобы помочь переносимым приложениям справляться со случаями, когда ограничение на ресурсы выходит за пределы допустимого диапазона, стандарт SUSv3 предоставляет две константы, обозначающие значения ограничений, которые не могут быть представлены: `RLIM_SAVED_CUR` и `RLIM_SAVED_MAX`. Если мягкое ограничение нельзя представить с помощью типа данных `rlim_t`, вызов `getrlimit()` возвращает в поле `rlim_cur` значение `RLIM_SAVED_CUR`. Точно так же константа `RLIM_SAVED_MAX` используется для жесткого ограничения, возвращаемого в поле `rlim_max`.

Если тип `rlim_t` позволяет представить все возможные значения ограничения на ресурсы, то стандарт SUSv3 разрешает объявить константы `RLIM_SAVED_CUR` и `RLIM_SAVED_MAX` равными `RLIM_INFINITY`. Именно так сделано в Linux; это подразумевает, что `rlim_t` может вместить любые значения ограничений. Однако это не относится к 32-битным архитектурам, таким как x86-32. В таких системах в среде компиляции больших файлов (то есть когда макросу `_FILE_OFFSET_BITS`, который проверяет наличие тех или иных возможностей, присваивается значение `64`, как описано в разделе 5.10) библиотека `glibc` содержит 64-битную структуру `rlim_t`, однако в ядре для представления ограничений на ресурсы используется тип `unsigned long`, размер которого равен 32 битам. Современные версии `glibc` решают эту проблему следующим образом: если программа, скомпилированная с макросом `_FILE_OFFSET_BITS=64`, пытается установить ограничение на ресурсы, которое не вмещается в 32-битный тип `unsigned long`, тогда обертка для вызова `setrlimit()` из состава `glibc` автоматически меняет это значение на `RLIM_INFINITY`. Иными словами, запрашиваемое ограничение на ресурсы не устанавливается.

Во многих дистрибутивах с архитектурой x86-32 утилиты для работы с файлами обычно скомпилированы с макросом _FILE_OFFSET_BITS=64, поэтому невозможность установить ограничение, которое нельзя представить 32-битным значением, является проблемой, затрагивающей не только программистов, но и конечных пользователей.

С одной стороны, было бы лучше, если бы обертка setrlimit() из состава glibc возвращала ошибку, когда запрашиваемое ограничение не помещается в 32-битный тип unsigned long. Однако в основе проблемы лежит ограничение ядра, и для решения этой проблемы разработчиками glibc был выбран подход, описанный выше.

36.3. Подробности об отдельных ограничениях на ресурсы

В этом разделе мы подробно рассмотрим все ограничения на ресурсы, доступные в Linux, и отдельно будут отмечены те из них, которые присутствуют только в этой системе.

- `RLIMIT_AS` — обозначает максимальный размер виртуальной памяти процесса (адресное пространство) в байтах. Попытки его превысить (с помощью вызовов `brk()`, `sbrk()`, `mmap()`, `mremap()` и `shmat()`) приводят к ошибке `ENOMEM`. На практике наиболее вероятным местом, где можно встретить это ограничение, являются функции из состава пакета `malloc`, которые используют вызовы `sbrk()` и `mmap()`. Во время приближения к этому ограничению можно также столкнуться с переполнением стека (см. `RLIMIT_STACK` ниже).
- `RLIMIT_CORE` — обозначает максимальный размер (в байтах) файлов с дампами памяти, которые генерируются, когда процесс завершается по определенным сигналам (см. раздел 22.1). При достижении этого ограничения создание дампа памяти будет остановлено. Чтобы предотвратить создание дампов памяти, можно указать значение `0`; это может быть полезно, поскольку такие файлы иногда достигают больших размеров, а конечные пользователи, как правило, не знают, что с ними делать. Еще одна причина отключения дампов памяти связана с безопасностью — это позволяет избежать сбрасывания памяти приложения на диск. Если значение `RLIMIT_FSIZE` меньше этого ограничения, файлы дампов памяти ограничиваются `RLIMIT_FSIZE` байтами.
- `RLIMIT_CPU` — обозначает максимальное количество секунд процессорного времени (как в режиме ядра, так и в режиме пользователя), которые могут быть задействованы процессом. Стандарт SUSv3 требует, чтобы при исчерпании этого ограничения процессу был послан сигнал `SIGXCPU`, однако никаких уточнений больше не дается (действием по умолчанию сигнала `SIGXCPU` является завершение процесса со сбрасыванием дампа памяти). Для этого сигнала можно установить обработчик, который будет выполнять нужные процессу операции и возвращать в конце контроль за выполнением главной программе. После этого (в Linux) сигнал `SIGXCPU` отправляется каждую секунду потребленного процессорного времени. Если процесс продолжает работу и достигает жесткого ограничения, ядро отправляет ему сигнал `SIGKILL`, который гарантированно его завершит.

  Реакция на поведение процесса, который продолжает потреблять ресурсы ЦПУ после обработки сигнала `SIGXCPU`, зависит от конкретной реализации UNIX. Большинство систем продолжает периодически посылать сигнал `SIGXCPU`. Разрабатывая переносимое приложение, вы должны спроектировать его таким образом, чтобы при первом получении `SIGXCPU` оно выполняло все необходимые операции по освобождению ресурсов и завершалось (как вариант, после получения этого сигнала программа может изменить ограничение на ресурсы).

- RLIMIT_DATA — обозначает максимальный размер (в байтах) сегмента с данными, принадлежащего процессу (сочетание сегментов с инициализированными/неинициализированными данными и кучей, описанное в разделе 6.3). Попытки расширить сегмент с данными за пределы допустимого диапазона (с помощью вызовов sbrk() и brk()) заканчиваются ошибкой ENOMEM. По аналогии с RLIMIT_AS, встретить это ограничение чаще всего можно при вызове функций из пакета malloc.
- RLIMIT_FSIZE — обозначает максимальный размер файлов (в байтах), которые может создавать процесс. Если попытаться выйти за пределы мягкого ограничения, процессу будет послан сигнал SIGXFSZ, а системный вызов (например, write() или truncate()) завершится ошибкой EFBIG. Действием по умолчанию для сигнала SIGXFSZ является завершение процесса и сбрасывание дампа памяти. Вы можете его перехватить и вернуть управление главной программе, однако учитывайте, что дальнейшие попытки расширить файл приведут к отправке того же сигнала и получению той же ошибки.
- RLIMIT_MEMLOCK — ограничение (происходит из систем BSD; не входит в стандарт SUSv3 и присутствует только в системах Linux и BSD) обозначает максимальный объем виртуальной памяти (в байтах), который процесс может удерживать от сброса на диск. Это ограничение затрагивает системные вызовы mlock() и mlockall(), а также параметры блокирования для вызовов mmap() и shmctl(). Подробности будут описаны в разделе 46.2.

  Если при вызове mlockall() указать флаг MCL_FUTURE, ограничение RLIMIT_MEMLOCK может также привести к сбоям в последующих вызовах brk(), sbrk(), mmap() или mremap().
- RLIMIT_MSGQUEUE — ограничение (доступно только в Linux; реализовано в версии ядра 2.6.8) обозначает максимальный объем памяти (в байтах), который может быть выделен под очереди POSIX-сообщений для пользователя с реальным идентификатором или вызывающего процесса. Когда вызов mq_open() создает очередь POSIX-сообщений, ее размер вычисляется по следующей формуле:

  ```
  bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
          attr.mq_maxmsg * attr.mq_msgsize;
  ```

  Команда attr является структурой mq_attr, которая передается в качестве четвертого аргумента для вызова mq_open(). Слагаемое sizeof(struct msg_msg *) гарантирует, что пользователь не сможет поместить в очередь неограниченное количество сообщений нулевой длины (тип данных msg_msg применяется внутри ядра). Это необходимо, поскольку сообщения нулевой длины, несмотря на то что они не содержат никаких данных, все равно расходуют некоторый объем памяти на свои внутренние нужды.

  Это ограничение влияет только на вызывающий процесс. Другие процессы, принадлежащие тому же пользователю, затрагиваются только в том случае, если они установили или наследовали данное ограничение.
- RLIMIT_NICE — ограничение (доступно только в Linux; реализовано в версии ядра 2.6.12) обозначает максимальное значение nice, которое можно установить для процесса с помощью вызовов sched_setscheduler() и nice(). Это ограничение вычисляется по формуле 20 - rlim_cur, где rlim_cur — это текущее мягкое ограничение RLIMIT_NICE. Дальнейшие подробности можно найти в разделе 35.1.
- RLIMIT_NOFILE — обозначает число, превышающее на единицу максимальное количество файловых дескрипторов, которые можно выделить процессу. Попытки выйти за этот предел (с помощью вызовов pen(), pipe(), socket(), accept(), shm_open(), dup(), dup2(), fcntl(F_DUPFD) и epoll_create()) завершаются неудачей. В большинстве ситуаций возвращается ошибка EMFILE, но вы также можете получить ошибку

EBADF (в случае с dup2(fd, newfd)) или EINVAL, если аргумент newfd вызова fcntl(fd, F_DUPFD, newfd) превышает ограничение или равен ему.
Изменение ограничения RLIMIT_NOFILE отражается на значении, возвращаемом вызовом sysconf(_SC_OPEN_MAX). Стандарт SUSv3 разрешает, но не требует от этого вызова возвращать разные значения до и после изменения данного ограничения; другие системы в этом отношении могут вести себя иначе, чем Linux.

Согласно стандарту SUSv3, если мягкое или жесткое ограничение RLIMIT_NOFILE, устанавливаемое приложением, не превышает номер последнего файлового дескриптора, открытого процессом, это может привести к непредсказуемым последствиям.

Linux позволяет получить информацию о файловых дескрипторах, которые процесс держит открытыми; для этого предусмотрен вызов readdir(), который сканирует содержимое каталога /proc/PID/fd, хранящего символьные ссылки на каждый файловый дескриптор, открытый процессом в данный момент.

Ядро устанавливает граничное значение, выше которого ограничение RLIMIT_NOFILE поднять нельзя. В ядрах до версии 2.6.25 это значение было явно указано в коде и определялось константой NR_OPEN, которая равна 1048576 (чтобы его повысить, приходилось перекомпилировать ядро). Но, начиная с версии 2.6.25, оно определено в файле /proc/sys/fs/nr_open (доступном только в Linux). По умолчанию оно по-прежнему равно 1048576, но теперь администратор может его изменить. Попытки установить мягкое или жесткое ограничение RLIMIT_NOFILE выше граничного значения приводят к ошибке EPERM.
Существует также системное ограничение на общее количество файлов, которые могут быть открыты всеми процессами, вместе взятыми. Это ограничение доступно для чтения и редактирования посредством файла /proc/sys/fs/file-max (только в Linux). В разделе 5.4 упоминалось, что значение file-max более точно можно описать как общесистемное ограничение на число открытых файловых дескрипторов. Превысить это ограничение могут только привилегированные процессы (CAP_SYS_ADMIN). Если процесс без особых привилегий приблизится к ограничению file-max, он завершится ошибкой ENFILE.

- RLIMIT_NPROC — ограничение (происходит из систем BSD; не входит в стандарт SUSv3 и присутствует только в системах Linux и BSD) обозначает максимальное количество процессов, которые могут быть созданы для реального идентификатора пользователя, принадлежащего вызывающему процессу. Попытки превысить это ограничение (с помощью вызовов fork(), vfork() и clone()) приводят к ошибке EAGAIN.
  Ограничение RLIMIT_NPROC затрагивает только вызывающий процесс, не влияя на другие процессы, принадлежащие тому же пользователю (если только они не установили или наследовали это ограничение). Оно не распространяется на привилегированные процессы (CAP_SYS_ADMIN или CAP_SYS_RESOURCE).

  В Linux также действует общесистемное ограничение на количество процессов, созданных всеми пользователями. Для получения и редактирования этого значения можно обратиться к файлу /proc/sys/kernel/threads-max (начиная с Linux 2.4).

  Если быть точным, ограничение RLIMIT_NPROC и файл threads-max ограничивают количество потоков, а не процессов.

  То, как устанавливается значение по умолчанию для ограничения RLIMIT_NPROC, зависит от версии ядра. В Linux 2.2 оно рассчитывалось по определенной формуле. Начиная с Linux 2.4, оно вычисляется в зависимости от объема доступной физической памяти.

Ограничение на ресурсы RLIMIT_NPROC не входит в стандарт SUSv3. Стандартным средством извлечения (но не изменения) максимального количества процессов, которые позволено создавать пользователю, является вызов sysconf(_SC_CHILD_MAX). Он поддерживается в Linux, но в версиях ядра ниже 2.6.23 он всегда возвращает 999, что не отвечает реальности. Начиная с версии 2.6.23 (и с glibc 2.4), этот вызов выдает корректное ограничение (проверяя значение RLIMIT_NPROC).

Не существует переносимого способа получения информации о количестве процессов, которые уже созданы пользователем с определенным идентификатором. Мы можем попытаться просканировать все файлы /proc/PID/status в системе, проверяя значения в поле Uid (которые содержат четыре пользовательских идентификатора в следующем порядке: действующий, реальный, сохраненный набор и файловая система), чтобы оценить, сколькими процессами владеет пользователь на текущий момент. Однако нужно понимать, что при завершении такого сканирования эта информация уже может поменяться.

- `RLIMIT_RSS` — ограничение (происходит из систем BSD; отсутствует в стандарте SUSv3, но широко распространено) обозначает максимальное количество страниц в резидентной памяти процесса, то есть общее количество страниц виртуальной памяти, которые в данный момент находятся в ОЗУ. Это ограничение предоставляется в Linux, но пока еще ни на что не влияет.
- `RLIMIT_RTPRIO` — ограничение (доступно только в Linux версии 2.6.12 и выше) обозначает граничное значение приоритета реального времени, который может быть установлен для текущего процесса с помощью вызовов `sched_setscheduler()` и `sched_setparam()`. Больше подробностей ищите в подразделе 35.3.2.
- `RLIMIT_RTTIME` — ограничение (доступно только в Linux версии 2.6.25 и выше) обозначает максимальное количество процессорного времени (в микросекундах), которое может быть потреблено процессом, выполняющимся в реальном времени, без остановок (то есть без вызова блокирующих системных операций). Поведение этого ограничения похоже на `RLIMIT_CPU`: если процесс достигает мягкого ограничения, ему отправляется сигнал `SIGXCPU`, который будет повторно посылаться при потреблении каждой следующей секунды процессорного времени. При достижении жесткого ограничения процесс получает сигнал `SIGKILL`. Больше подробностей можно найти в подразделе 35.3.2.
- `RLIMIT_SIGPENDING` — ограничение (доступно только в Linux версии 2.6.8 и выше) обозначает максимальное количество сигналов, которые могут попасть в очередь для пользователя с реальным идентификатором. Попытки его превысить (с помощью вызова `sigqueue()`) приводят к ошибке `EAGAIN`.
  Это ограничение влияет только на вызывающий процесс. Другие процессы, принадлежащие тому же пользователю, затрагиваются только в том случае, если они сами установили или унаследовали данное ограничение.
  В исходной реализации значение ограничения `RLIMIT_SIGPENDING` по умолчанию было равно `1024`. Однако в ядре 2.6.12 оно было изменено и теперь равняется значению `RLIMIT_NPROC`.
  Ограничение `RLIMIT_SIGPENDING` учитывает в очереди как стандартные сигналы, так и сигналы реального времени (стандартные сигналы могут быть отложены только один раз для каждого отдельного процесса). Но это ограничение действует только для вызова `sigqueue()`. Даже если в очередь к процессам, принадлежащим пользователю с заданным реальным идентификатором, попало максимальное количество сигналов, вызов `kill()` позволяет поместить в очередь по одному экземпляру каждого из сигналов (включая те, что выполняются в режиме реального времени), которые еще не были отложены для процесса.

Начиная с ядра 2.6.12, поле SigQ файла /proc/PID/status (доступного только в Linux) хранит текущие и максимальные номера отложенных сигналов для реального пользовательского идентификатора процесса.

- RLIMIT_STACK — обозначает максимальный размер стека процессов (в байтах). Попытки увеличить стек за пределы этого ограничения приводят к отправке процессу сигнала SIGSEGV. Поскольку стек исчерпан, единственным способом перехватить этот сигнал является установление альтернативного стека сигналов, как было описано в разделе 21.3. Начиная с Linux 2.6.23 это ограничение также определяет объем памяти, доступной для хранения аргументов командной строки и переменных среды процесса. См. справочную страницу execve(2).

36.4. Резюме

Процессы потребляют различные системные ресурсы. Системный вызов getrusage() позволяет отслеживать некоторые из них, относящиеся как к самому процессу, так и к его потомкам.

Системные вызовы setrlimit() и getrlimit() позволяют процессу устанавливать и получать ограничения на потребление им разных ресурсов. Каждое ограничение состоит из двух значений: мягкого, действие которого обеспечивается ядром, и жесткого, которое является граничным значением для мягкого ограничения. Непривилегированный процесс может устанавливать мягкие ограничения на ресурсы в диапазоне от 0 до жесткого ограничения (которое нельзя превысить). Привилегированный процесс может вносить любые изменения в ограничения обоих типов, но при этом мягкое ограничение всегда должно быть меньше жесткого или равно ему. Достигая мягкого ограничения, процесс обычно уведомляется об этом факте либо с помощью сигнала, либо неудачным завершением системного вызова, который пытается превысить ограничение.

36.5. Упражнения

36.1. Напишите программу для демонстрации того, что флаг RUSAGE_CHILDREN вызова getrusage() получает информацию только о потомках, для которых была выполнена операция wait() (пусть программа создаст дочерний процесс, потребляющий некоторое процессорное время, затем родитель должен сделать вызов getrusage() до и после wait()).

36.2. Напишите программу, которая выполняет команду и затем показывает, сколько она потребила ресурсов. Это аналогично действию команды time(1). Запускаться она должна следующим образом:

```
$ ./rusage command arg...
```

36.3. Напишите программу, чтобы определить, что происходит при исчерпании процессом различных ограничений, которые задаются с помощью вызова setrlimit().

37 Демоны

В этой главе рассматриваются свойства процессов-демонов и процедуры, которые требуются для получения демона из обычного процесса. Вы также узнаете, как записывать в системный журнал сообщения, отправляемые демоном, используя систему `syslog`.

37.1. Краткий обзор

Демон (англ. *daemon*) — это процесс, обладающий следующими свойствами.

- Имеет длинный жизненный цикл. Часто демоны создаются во время загрузки системы и работают до момента ее выключения.
- Выполняется в фоновом режиме и не имеет контролирующего терминала. Последняя особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких как `SIGINT`, `SIGTSTP` и `SIGHUP`).

Демоны создаются для выполнения специфических задач. Например:

- `cron` — демон, который выполняет команды в запланированное время;
- `sshd` — демон защищенной командной оболочки, который позволяет входить в систему с удаленных компьютеров, используя безопасный протокол;
- `httpd` — демон HTTP-сервера (Apache), который обслуживает веб-страницы;
- `inetd` — демон IP-служб (описан в разделе 56.5), который ожидает входящих сетевых подключений на заданных портах TCP/IP и запускает соответствующие серверные программы для их обслуживания.

Многие стандартные демоны работают в качестве привилегированных процессов (то есть их действующий пользовательский идентификатор равен 0), поэтому при их написании следует руководствоваться рекомендациями, перечисленными в главе 38.

Названия демонов принято заканчивать буквой `d` (хотя это не является обязательным правилом).

> В Linux некоторые демоны выполняются в качестве потоков ядра. Код таких процессов является частью ядра и обычно запускается во время загрузки системы. Команда ps(1) выводит их названия в квадратных скобках ([]). В качестве примера можно привести демон pdflush, который периодически сбрасывает «грязные» страницы памяти (например, страницы из кэша буфера) на диск.

37.2. Создание демона

Для того чтобы стать демоном, программа должна выполнить следующие шаги.

1. Сделать вызов `fork()`, после которого родитель завершается, а потомок продолжает работать (в результате этого демон становится потомком процесса `init`). Этот шаг делается по двум следующим причинам.

- Исходя из того, что демон был запущен в командной строке, завершение родителя будет обнаружено командной оболочкой, которая вслед за этим выведет новое приглашение и позволит потомку выполняться в фоновом режиме.
- Потомок гарантированно не станет лидером группы процессов, поскольку он наследует PGID от своего родителя и получает свой уникальный идентификатор, который отличается от унаследованного PGID. Это необходимо для успешного выполнения следующего шага.

2. Дочерний процесс вызывает `setsid()` (см. раздел 34.3), чтобы начать новую сессию и разорвать любые связи с контролирующим терминалом.
3. Если после этого демон больше не открывает никаких терминальных устройств, мы можем не волноваться о том, что он восстановит соединение с контролирующим терминалом. В противном случае нам необходимо сделать так, чтобы терминальное устройство не стало контролирующим. Это можно сделать двумя нижеописанными способами.
   - Указывать флаг `O_NOCTTY` для любых вызовов `open()`, которые могут открыть терминальное устройство.
   - Есть более простой вариант: после `setsid()` можно еще раз сделать вызов `fork()`, опять позволив родителю завершиться, а потомку (правнуку) — продолжить работу. Это гарантирует, что потомок не станет лидером сессии, что делает невозможным повторное соединение с контролирующим терминалом (это соответствует процедуре получения контролирующего терминала, принятой в System V, — см. раздел 34.4).

   В реализациях, которые соблюдают правила, принятые в BSD-системах, процесс может получить контролирующий терминал только с помощью явного выполнения операции TIOCSCTTY в вызове ioctl(); в этом случае второй вызов fork() не влияет на соединение с контролирующим терминалом, хотя и не причиняет никакого вреда.

4. Очистить атрибут `umask` процесса (см. подраздел 15.4.6), чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа.
5. Поменять текущий рабочий каталог процесса (обычно на корневой — `/`). Это необходимо, поскольку демон обычно выполняется вплоть до выключения системы. Если файловая система, на которой находится его текущий рабочий каталог, не является корневой, она не может быть отключена (см. подраздел 14.8.2). Как вариант, в качестве рабочего каталога демон может задействовать то место, где он выполняет свою работу, или воспользоваться значением в конфигурационном файле; главное, чтобы файловая система, в которой находится этот каталог, никогда не нуждалась в отключении. Например, `cron` применяет для этого `/var/spool/cron`.
6. Закрыть все открытые файловые дескрипторы, которые демон унаследовал от своего родителя (возможно, некоторые из них необходимо оставить открытыми, поэтому данный шаг является необязательным и может быть откорректирован). Это делается по целому ряду причин. Поскольку демон потерял свой контролирующий терминал и работает в фоновом режиме, ему больше не нужно хранить дескрипторы с номерами 0, 1 и 2 (если они ссылаются на терминал). Кроме того, мы не можем отключить файловую систему, на которой долгоживущий демон удерживает открытыми какие-либо файлы. И, следуя обычным правилам, мы должны закрывать неиспользуемые файловые дескрипторы, поскольку их число ограничено.

   Некоторые UNIX-системы (такие как Solaris 9 и новые версии BSD) предоставляют функцию closefrom(n) (или похожую), которая закрывает все файловые дескрипторы, номера которых больше или равны n. В Linux она недоступна.

7. Закрыв дескрипторы с номерами 0, 1 и 2, демон обычно перенаправляет их в предварительно открытый файл `/dev/null`, используя вызов `dup2()` (или похожий). Это делается по двум причинам.
   - Это позволяет избежать ошибки при вызове библиотечных функций, которые выполняют операции ввода/вывода с этими дескрипторами.
   - Это исключает возможность повторного открытия демоном файлов с помощью дескрипторов 1 или 2, так как библиотечные функции, которые записывают в них данные, ожидают, что эти дескрипторы указывают на потоки `stdout` (стандартный вывод) и `stderr` (стандартный вывод ошибок).

   /dev/null — это виртуальное устройство, которое всегда отклоняет записываемые в него данные. Мы можем перенаправлять в него ненужные нам ошибки или вывод консольных команд. Чтение из этого устройства всегда приводит к возвращению символа конца файла (EOF).

Ниже представлена реализация функции `becomeDaemon()`, которая превращает вызывающий процесс в демона, выполняя вышеописанные шаги.

```
#include <syslog.h>

int becomeDaemon(int flags);
```

Возвращает `0` при успешном завершении или `-1`, если произошла ошибка

В качестве аргумента `flags` функция `becomeDaemon()` принимает битовую маску, которая позволяет вызывающему процессу выборочно пропускать некоторые шаги (см. комментарии в заголовочном файле в листинге 37.1).

Листинг 37.1. Заголовочный файл для программы become_daemon.c

daemons/become_daemon.h

```
#ifndef BECOME_DAEMON_H    /* Предотвращаем двойное подключение */
#define BECOME_DAEMON_H

/* Значения битовой маски аргумента 'flags' из функции becomeDaemon() */

#define BD_NO_CHDIR            01  /* Не выполнять chdir("/") */
#define BD_NO_CLOSE_FILES      02  /* Не закрывать все открытые файлы */
#define BD_NO_REOPEN_STD_FDS   04  /* Не перенаправлять потоки stdin,
                                      stdout и stderr в /dev/null */
#define BD_NO_UMASK0          010  /* Не выполнять umask(0) */

#define BD_MAX_CLOSE         8192  /* Максимальное количество закрываемых
                                      файловых дескрипторов, если операция
                                      sysconf(_SC_OPEN_MAX) не определена */

int becomeDaemon(int flags);

#endif
```

daemons/become_daemon.h

Реализация функции `becomeDaemon()` приводится в листинге 37.2.

Библиотека GNU C предоставляет нестандартную функцию daemon(), которая превращает вызывающий процесс в демона. Она не поддерживает ничего похожего на аргумент flags функции becomeDaemon().

Листинг 37.2. Создание процесса-демона

daemons/become_daemon.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "become_daemon.h"
#include "tlpi_hdr.h"

int        /* Возвращает 0 в случае успеха или –1, если случилась ошибка */
becomeDaemon(int flags)
{
    int maxfd, fd;

    switch (fork()) {              /* Превращение в фоновый процесс */
    case –1: return –1;
    case 0:    break;              /* Потомок проходит этот этап... */
    default: _exit(EXIT_SUCCESS); /* ...а родитель завершается */
    }

    if (setsid() == –1)            /* Процесс становится лидером новой сессии */
        return –1;

    switch (fork()) {              /* Делаем так, чтобы процесс
                                      не стал лидером сессии */
    case –1: return –1;
    case 0:    break;
    default: _exit(EXIT_SUCCESS);
    }

    if (!(flags & BD_NO_UMASK0))
        umask(0);          /* Сбрасываем маску режима создания файлов */

    if (!(flags & BD_NO_CHDIR))
        chdir("/");        /* Переходим в корневой каталог */

    if (!(flags & BD_NO_CLOSE_FILES)) { /* Закрываем все открытые файлы */
        maxfd = sysconf(_SC_OPEN_MAX);
        if (maxfd == –1)           /* Ограничение не определено... */
            maxfd = BD_MAX_CLOSE;  /* ...поэтому устанавливаем его наугад */

        for (fd = 0; fd < maxfd; fd++)
            close(fd);
    }

    if (!(flags & BD_NO_REOPEN_STD_FDS)) {
        close(STDIN_FILENO);        /* Перенаправляем стандартные потоки
                                       данных в /dev/null */

        fd = open("/dev/null", O_RDWR);

        if (fd != STDIN_FILENO)    /* Значение 'fd' должно быть больше 0 */
            return –1;
        if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
```

```
            return -1;
        if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
            return -1;
    }

    return 0;
}
```

daemons/become_daemon.c

Написав программу, которая вызывает функцию becomeDaemon(0), и затем на какое-то время останавливается, мы сможем рассмотреть некоторые атрибуты итогового процесса, воспользовавшись командой ps(1):

```
$ ./test_become_daemon
$ ps -C test_become_daemon -o "pid ppid pgid sid tty command"
  PID     PPID      PGID        SID      TT        COMMAND
24731        1     24730      24730      ?        ./test_become_daemon
```

Мы не приводим здесь исходный код программы daemons/test_become_daemon.c, так как он достаточно тривиален; вы можете найти его в архиве с исходными файлами, который прилагается к этой книге.

В выводе команды ps знак ? в столбце TT указывает на то, что процесс не привязан к контролирующему терминалу. Из того факта, что идентификаторы процесса и сессии (SID) не совпадают, можно сделать вывод, что процесс не является лидером сессии и не сможет установить соединение с контролирующим терминалом при открытии соответствующего устройства. Именно так и должен вести себя демон.

37.3. Рекомендации по написанию демонов

Как уже отмечалось выше, процесс-демон обычно завершается во время выключения системы. Для многих стандартных демонов предусмотрены специальные скрипты, которые выполняются, когда система завершает работу. Остальные демоны просто получают сигнал SIGTERM, который при выключении компьютера отправляется процессом init всем своим потомкам. По умолчанию этот сигнал приводит к завершению процесса. Если демону перед этим необходимо освободить какие-либо ресурсы, он должен делать это в обработчике данного сигнала. Эту процедуру следует выполнять как можно быстрее, поскольку через 5 секунд после SIGTERM процесс init отправляет сигнал SIGKILL (это вовсе не означает, что у демона есть 5 секунд процессорного времени на освобождение ресурсов; init шлет эти сигналы всем процессам в системе одновременно, поэтому процедуру очистки в этот момент может выполнять каждый из них).

Так как демоны имеют длинный жизненный цикл, нам следует особенно тщательно следить не только за потенциальными утечками памяти (см. подраздел 7.1.3), но и за файловыми дескрипторами (когда приложению не удается закрыть все файловые дескрипторы, которые оно открыло). Для временного исправления подобных ошибок демон приходится перезапускать заново.

Часто демону необходимо убедиться в том, что только один его экземпляр активен в любой заданный момент времени. Например, запуск двух копий демона cron для выполнения одних и тех же заданий не имел бы никакого смысла. Методики, позволяющие этого достичь, будут рассмотрены в разделе 51.6.

37.4. Использование сигнала SIGHUP для повторной инициализации демона

Из того факта, что демоны должны выполняться непрерывно, вытекает две проблемы.

- Обычно при запуске демон считывает параметры из соответствующего конфигурационного файла. Но иногда возникает необходимость изменить эти параметры на лету, без остановки или перезапуска самого демона.
- Некоторые демоны генерируют журнальные файлы. Если эти файлы никогда не закрывать, они могут бесконечно увеличиваться в размере и в какой-то момент исчерпают свободное пространство в системе (в разделе 18.3 отмечалось, что файл, если процесс удерживает его открытым, продолжает существовать даже после переименования). Нам нужен какой-то способ сообщить демону о том, что он должен закрыть текущий журнальный файл и открыть новый, чтобы при необходимости можно было выполнять их ротацию.

Решением обеих этих проблем является создание обработчика сигнала `SIGHUP` и выполнение внутри него всех необходимых шагов. В разделе 34.4 отмечалось, что сигнал `SIGHUP` генерируется для контролирующего процесса при отключении его от контролирующего терминала. Поскольку демоны лишены соединения с терминалом, они никогда не получают этот сигнал и могут использовать его в целях, описанных в данном разделе.

> Для автоматической ротации журнальных файлов демона можно применить программу logrotate. Подробности ищите на справочной странице logrotate(8).

В листинге 37.3 приводится пример того, каким образом демон может использовать сигнал `SIGHUP`. Программа устанавливает обработчик `SIGHUP` ❷, становится демоном ❸, открывает журнальный файл ❹ и считывает файл конфигурации ❺. Обработчик ❶ всего лишь присваивает значение глобальной переменной `hupReceived`, которое проверяется главной функцией. Главная функция выполняет цикл, записывая сообщение в журнальный файл каждые 15 секунд ❽. Вызовы `sleep()` ❻ в этом цикле должны имитировать некие вычисления, которые могли бы проводиться настоящим приложением. После возвращения вызова `sleep()` программа проверяет, было ли установлено значение переменной `hupReceived` ❼; если ответ положительный, она заново открывает журнальный файл, повторно считывает конфигурацию и сбрасывает значение `hupReceived`.

Для лаконичности функции `logOpen()`, `logClose()`, `logMessage()` и `readConfigFile()` не вошли в листинг 37.3, но вы можете найти их в архиве с исходным кодом, который прилагается к этой книге. О назначении первых трех из них можно догадаться по их именам, а функция `readConfigFile()` просто считывает строчку из конфигурационного файла и записывает ее в журнал.

> Некоторые демоны используют альтернативный метод повторной инициализации при получении сигнала SIGHUP, закрывая все файлы и перезапускаясь с помощью вызова exec().

Ниже представлен пример работы программы из листинга 37.3. Сначала мы создаем фиктивный конфигурационный файл, после чего запускается демон:

```
$ echo START > /tmp/ds.conf
$ ./daemon_SIGHUP
```

```
$ cat /tmp/ds.log                          Выводим журнальный файл
2011-01-17 11:18:34: Opened log file
2011-01-17 11:18:34: Read config file: START
```

Теперь, прежде чем отправить демону сигнал SIGHUP, изменим конфигурационный файл и переименуем журнал:

```
$ echo CHANGED > /tmp/ds.conf
$ date +'%F %X'; mv /tmp/ds.log /tmp/old_ds.log
2011-01-17 11:19:03 AM
$ date +'%F %X'; killall -HUP daemon_SIGHUP
2011-01-17 11:19:23 AM
$ ls /tmp/*ds.log            Журнальный файл был повторно открыт
/tmp/ds.log /tmp/old_ds.log
$ cat /tmp/old_ds.log        Выводим старый журнальный файл
2011-01-17 11:18:34: Opened log file
2011-01-17 11:18:34: Read config file: START
2011-01-17 11:18:49: Main: 1
2011-01-17 11:19:04: Main: 2
2011-01-17 11:19:19: Main: 3
2011-01-17 11:19:23: Closing log file
```

Вывод команды ls показывает наличие как старого, так и нового журнального файла. Просматривая старый журнал с помощью утилиты cat, мы можем увидеть, что даже после ввода команды mv для переименования файла демон продолжает записывать в него сообщения. На этом этапе старый журнальный файл можно удалить, если он нам больше не нужен. Содержимое нового журнала указывает на то, что конфигурационный файл был прочитан заново:

```
$ cat /tmp/ds.log
2011-01-17 11:19:23: Opened log file
2011-01-17 11:19:23: Read config file: CHANGED
2011-01-17 11:19:34: Main: 4
$ killall daemon_SIGHUP            Завершаем работу демона
```

Обратите внимание на то, что журнальный и конфигурационный файлы демонов обычно размещаются в стандартных каталогах, а не в /tmp, как это сделано в программе из листинга 37.3. Конфигурацию принято хранить в каталоге /etc или в одном из его подкаталогов, а журнал часто находится в /var/log. Программы-демоны обычно предоставляют параметры командной строки, которые позволяют указать альтернативные пути вместо стандартных.

Листинг 37.3. Использование сигнала SIGHUP для повторной инициализации демона
daemons/daemon_SIGHUP.c

```
#include <sys/stat.h>
#include <signal.h>
#include "become_daemon.h"
#include "tlpi_hdr.h"

static const char *LOG_FILE = "/tmp/ds.log";
static const char *CONFIG_FILE = "/tmp/ds.conf";

/* Определения функций logMessage(), logOpen(), logClose()
   и readConfigFile() в этом листинге опущены */

static volatile sig_atomic_t hupReceived = 0;
       /* При получении SIGHUP устанавливаем ненулевое значение */
```

```
static void
sighupHandler(int sig)
{
1   hupReceived = 1;
}

int
main(int argc, char *argv[])
{
    const int SLEEP_TIME = 15;    /* Период бездействия между сообщениями */
    int count = 0;        /* Количество завершенных интервалов SLEEP_TIME */
    int unslept;          /* Время, оставшееся до завершения периода бездействия */
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = sighupHandler;
2   if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");

3   if (becomeDaemon(0) == -1)
    errExit("becomeDaemon");

4   logOpen(LOG_FILE);
5   readConfigFile(CONFIG_FILE);

    unslept = SLEEP_TIME;

    for (;;) {
6       unslept = sleep(unslept);  /* В случае прерывания возвращает значение > 0 */

7       if (hupReceived) {         /* При получении сигнала SIGHUP... */
            logClose();
            logOpen(LOG_FILE);
            readConfigFile(CONFIG_FILE);
            hupReceived = 0;       /* Готовимся принять следующий сигнал SIGHUP */
        }

        if (unslept == 0) {        /* Когда интервал исчерпан */
            count++;
8           logMessage("Main: %d", count);
            unslept = SLEEP_TIME; /* Сбрасываем интервал */
        }
    }
}
```

daemons/daemon_SIGHUP.c

37.5. Запись в журнал сообщений и ошибок с помощью системы syslog

При написании демона одной из проблем является вывод сообщений об ошибках. Поскольку демон выполняется в фоновом режиме, он не может выводить информацию в терминале, как это делают другие программы. В качестве альтернативы сообщения можно записывать в отдельный журнальный файл программы (см. листинг 37.3). Основной недостаток такого

подхода (с точки зрения системного администратора) заключается в сложности управления журнальными файлами разных приложений и их мониторинге. Для решения этой проблемы была разработана система `syslog`.

37.5.1. Краткий обзор

Система `syslog` предоставляет единый централизованный механизм, который позволяет любому приложению на компьютере записывать свои сообщения в журнал. Общая структура `syslog` показана на рис. 37.1.

Система `syslog` состоит из двух основных компонентов: демона `syslogd` и библиотечной функции `syslog(3)`.

Демон *System Log*, `syslogd`, принимает журнальные сообщения из двух разных источников: сокета UNIX-домена (`/dev/log`), который хранит сообщения, сгенерированные локально, и (если включено в конфигурации) сокет интернет-домена (UDP-порт 514), которые хранит сообщения, отправленные по сети TCP/IP (в некоторых системах UNIX сокет `syslog` находится в каталоге `/var/run/log`).

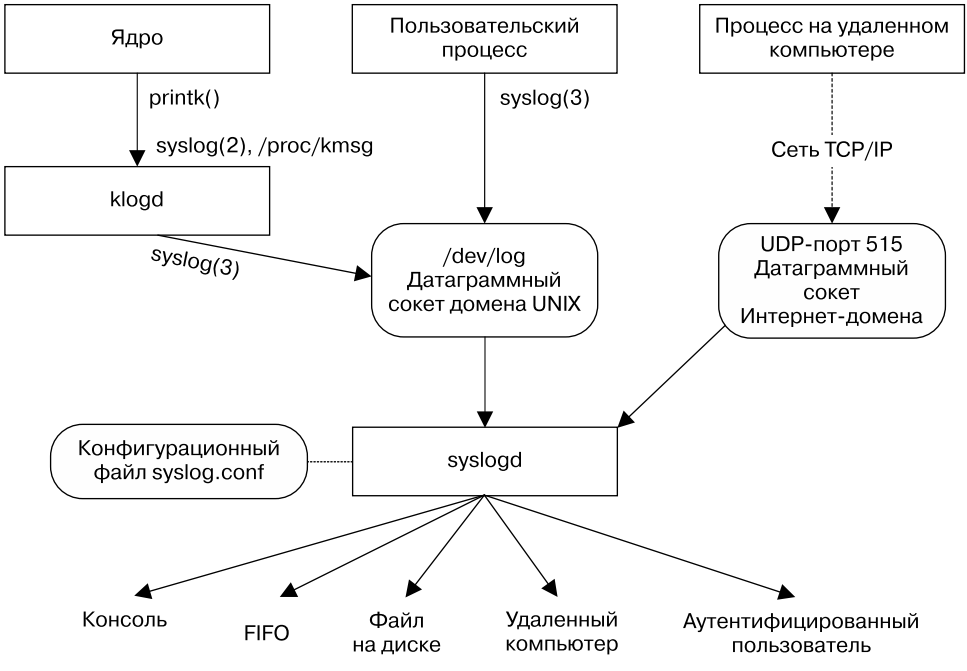


Рис. 37.1. Принцип работы системного журнала

Каждое сообщение, обработанное `syslogd`, имеет набор атрибутов, включая *категорию*, которая определяет тип программы, сгенерировавшей сообщение, и *уровень*, обозначающий значимость (приоритет) этого сообщения. Демон `syslogd` проверяет категорию и уровень всех сообщений, передавая их в несколько возможных мест, которые определяются соответствующим конфигурационным файлом, `/etc/syslog.conf`. Среди адресатов могут быть терминал или виртуальная консоль, файл на диске, очередь FIFO, один или несколько пользователей в системе (или сразу все) или процесс (обычно другой демон `syslogd`) на другом компьютере, подключенный по сети TCP/IP (передача сообщений

процессу в другой системе используется для их сбора на едином компьютере, что приводит к упрощению администрирования). Одно сообщение можно послать нескольким адресатам (или вовсе никому), а сообщения с разными *категориями* и *уровнями* могут быть направлены разным получателям или их экземплярам (то есть разным консолям, файлам на диске и т. д.).

> Передача syslog-сообщений другому компьютеру по протоколу TCP/IP может также помочь обнаружить несанкционированное проникновение в систему. Взломщики часто оставляют следы в системном журнале, и обычно они пытаются их стереть, чтобы скрыть свою деятельность. В случае с удаленным журналированием злоумышленнику пришлось бы для этого проникнуть еще и на другой компьютер.

Для записи сообщений в журнал любой процесс может воспользоваться библиотечной функцией `syslog(3)` (мы подробно опишем ее чуть ниже). На основе переданных ей аргументов она создает сообщение стандартного вида и помещает его в сокет `/dev/log`, где оно будет доступно для `syslogd`.

Альтернативной системой сбора сообщений является демон `klogd` *(Kernel Log)*, который собирает журнальные записи ядра (ядро генерирует их с помощью своей функции `printk()`). Эти записи передаются через один из двух равнозначных интерфейсов (которые существуют только в Linux) — файл `/proc/kmsg` или системный вызов `syslog(2)` — после чего помещаются в сокет `/dev/log` с помощью библиотечной функции `syslog(3)`.

> Несмотря на одинаковые имена, вызовы syslog(2) и syslog(3) выполняют совсем разные задачи. Интерфейс syslog(2) предоставляется библиотекой glibc под именем klogctl(). При упоминании в этом разделе вызова syslog() обычно имеется в виду syslog(3) (если явно не указано противоположное).

Впервые средства `syslog` были представлены в системе 4.2BSD, но теперь они доступны в большинстве реализаций UNIX. Стандарт SUSv3 включает в себя вызов `syslog(3)` и другие связанные с ним функции, оставляя без внимания реализацию и поведение демона `syslogd`, а также формат файла `syslog.conf`. Версия `syslogd`, использующаяся в Linux, отличается от оригинальной тем, что предусматривает некоторые расширения правил обработки сообщений, которые могут быть указаны в файле `syslog.conf`.

37.5.2. Программный интерфейс syslog

Программный интерфейс `syslog` состоит из трех основных функций.

- Функция `openlog()` устанавливает настройки, которые по умолчанию применяются ко всем последующим вызовам `syslog()`. Она не является обязательной. Если ею не воспользоваться, соединение с системой ведения журнала устанавливается при первом вызове `syslog()` на основе стандартных настроек.
- Функция `syslog()` записывает сообщения в журнал.
- Функция `closelog()` вызывается после окончания записи сообщений, чтобы разорвать соединение с журналом.

Ни одна из этих функций не возвращает значение статуса. Частично это продиктовано тем, что системное журналирование должно быть всегда доступным (если оно перестанет работать, системный администратор должен быстро это заметить). Кроме того, если при ведении журнала произошла ошибка, приложение обычно мало что может сделать, чтобы об этом сообщить.

Библиотека GNU C также предоставляет функцию void vsyslog(int priority, const char *format, va_list args), которая делает то же, что и syslog(), но принимает список аргументов, предварительно обработанных интерфейсом stdarg(3) (таким образом, vsyslog() и syslog() соотносятся как vprintf() и printf()). Функция vsyslog() не входит в стандарт SUSv3 и доступна не во всех UNIX-системах.

Установление соединения с системным журналом

Функция `openlog()` при необходимости устанавливает соединение с системным средством ведения журнала и задает настройки, которые будут применяться по умолчанию ко всем последующим вызовам `syslog()`.

```
#include <syslog.h>

void openlog(const char *ident, int log_options, int facility);
```

Аргумент `ident` является указателем на строку, которая добавляется в каждое сообщение, записываемое с помощью `syslog()`; обычно это название программы. Стоит отметить, что `openlog()` всего лишь копирует значение этого указателя. Продолжая использовать вызовы `syslog()`, приложение должно следить за тем, чтобы строка, на которую ссылается данный аргумент, не изменилась.

Если в качестве аргумента ident указать NULL, интерфейс syslog из состава glibc, как и некоторые другие реализации, будет автоматически подставлять вместо него название программы. Однако такое поведение не предусмотрено стандартом SUSv3 и не выполняется в некоторых системах, поэтому переносимые приложения не должны на него полагаться.

Аргумент `log_options` для вызова `openlog()` представляет собой битовую маску, состоящую из любых комбинаций следующих констант, к которым применяется побитовое ИЛИ.

- `LOG_CONS` — если в системный журнал приходит ошибка, она записывается в системную консоль (`/dev/console`).
- `LOG_NDELAY` — соединение с системой ведения журнала (то есть с сокетом домена UNIX, `/dev/log`) устанавливается немедленно. По умолчанию (`LOG_ODELAY`) это происходит, только когда (и если) первое сообщение попадает в журнал с помощью вызова `syslog()`. Флаг `O_NDELAY` может пригодиться в программах, которым нужно контролировать момент выделения файлового дескриптора для `/dev/log`. Например, это может быть приложение, которое вызывает `chroot()`; после этого вызова путь `/dev/log` перестает быть доступным, поэтому, если вы вызываете функцию `openlog()` с флагом `LOG_NDELAY`, это нужно делать до `chroot()`. Примером программы, которая использует флаг `LOG_NDELAY` таким образом, может служить демон `tftpd` *(Trivial File Transfer)*.
- `LOG_NOWAIT` — вызов `syslog()` не ждет дочерний процесс, который мог быть создан для записи сообщения в журнал. Этот флаг нужен в приложениях, в которых для записи сообщений используются отдельные дочерние процессы. Он позволяет вызову `syslog()` избежать ожидания потомков, которые уже были утилизированы родителем, который тоже их ожидал. В Linux флаг `LOG_NOWAIT` ни на что не влияет, так как в это системе при записи сообщений в журнал дочерние процессы не создаются.
- `LOG_ODELAY` — противоположность флагу `LOG_NDELAY`. Соединение с системой ведения журнала откладывается до тех пор, пока не будет записано первое сообщение. Этот флаг используется по умолчанию и его не нужно указывать отдельно.

- `LOG_PERROR` — сообщения попадают не только в системный журнал, но и в стандартный поток ошибок. Обычно демоны закрывают этот поток или перенаправляют его в `/dev/null`; в этом случае флаг `LOG_PERROR` ни на что не влияет.
- `LOG_PID` — вместе с каждым сообщением записывается идентификатор вызывающего процесса. Использование этого флага в сервере, который создает множество потомков, позволяет понять, какой из процессов записал то или иное сообщение.

Все константы, приведенные выше, входят в стандарт SUSv3. Исключение составляет константа `LOG_PERROR`, которая поддерживается во многих (но не всех) системах UNIX.

Аргумент `facility` в функции `openlog()` обозначает *категорию*, которая по умолчанию будет использоваться во всех последующих вызовах `syslog()`. Возможные значения этого аргумента перечислены в табл. 37.1.

Большинство категорий из табл. 37.1 входят в стандарт SUSv3 (о чем указано в столбце *SUSv3*). Исключением являются константы `LOG_AUTHPRIV` и `LOG_FTP` (которые встречаются в некоторых UNIX-системах), а также `LOG_SYSLOG` (которая поддерживается в большинстве реализаций). Значение `LOG_AUTHPRIV` может пригодиться в ситуациях, когда сообщения, содержащие пароли или другую деликатную информацию, записывают не в `LOG_AUTH`, а в какое-то другое место.

Категория `LOG_KERN` применяется для сообщений ядра. Сообщения этого вида не могут быть сгенерированы программами, работающими в пользовательском режиме. Константа `LOG_KERN` имеет значение 0. В сочетании с вызовом `syslog()` она приводит к задействованию «уровня по умолчанию».

Таблица 37.1. Значения категорий для openlog() и аргумент priority для syslog()

Значение	Описание	SUSv3
LOG_AUTH	Сообщения, связанные с безопасностью и авторизацией (например, su)	*
LOG_AUTHPRIV	Сообщения, связанные с частной безопасностью и авторизацией	
LOG_CRON	Сообщения, посланные демонами cron и at	*
LOG_DAEMON	Сообщения от других системных демонов	*
LOG_FTP	Сообщения от демона ftp (ftpd)	
LOG_KERN	Сообщения ядра (не могут быть сгенерированы пользовательским процессом)	*
LOG_LOCAL0	Зарезервировано для локального использования (то же самое для значений с LOG_LOCAL1 по LOG_LOCAL7)	*
LOG_LPR	Сообщения от системы построчной печати (lpr, lpd, lpc)	*
LOG_MAIL	Сообщения от почтовой системы	*
LOG_NEWS	Сообщения, связанные с новостной сетью Usenet	*
LOG_SYSLOG	Внутренние сообщения от демона syslogd	
LOG_USER	Сообщения, сгенерированные пользовательскими процессами (используется по умолчанию)	*
LOG_UUCP	Сообщения от системы UUCP	*

Запись сообщения в журнал

Для записи сообщения в журнал нужно вызвать функцию `syslog()`.

```
#include <syslog.h>

void syslog(int priority, const char *format, ...);
```

Аргумент `priority` создается в связке со значениями `facility` и `level`, к которым применяется оператор побитового ИЛИ. `facility` обозначает основную категорию приложения, которое записывает сообщение, и может принимать одно из значений, перечисленных в табл. 37.1. Если этот аргумент опустить, по умолчанию будет использоваться значение, указанное во время предыдущего вызова `openlog()`, или `LOG_USER`, если этот вызов не был выполнен. Аргумент `level` обозначает степень важности сообщения и может принимать одно из значений, перечисленных в табл. 37.2 (все они входят в стандарт SUSv3).

Таблица 37.2. Значения level аргумента priority для вызова syslog() (от самого высокого до самого низкого уровня)

Значение	Описание
LOG_EMERG	Аварийная ситуация или критическая ошибка (отказ системы)
LOG_ALERT	Ситуация, требующая принятия немедленных мер (например, повреждение системной базы данных)
LOG_CRIT	Критическая ситуация (например, ошибка на дисковом устройстве)
LOG_ERR	Ошибка общего характера
LOG_WARNING	Сообщение с предупреждением
LOG_NOTICE	Штатная ситуация, которая может потребовать особого обращения
LOG_INFO	Информационное сообщение
LOG_DEBUG	Отладочное сообщение

Оставшиеся аргументы вызова `syslog()` представляют собой строку форматирования и сопутствующие параметры в стиле функции `printf()`. Отличие от `printf()` заключается в том, что строка форматирования может не содержать в конце символ перевода строки. В этой строке также можно указать двухсимвольную последовательность `%m`, вместо которой подставляется ошибка, соответствующая текущему значению `errno` (то есть эквивалент вызова `strerror(errno)`).

Следующий код демонстрирует использование функций `openlog()` и `syslog()`:

```
openlog(argv[0], LOG_PID | LOG_CONS | LOG_NOWAIT, LOG_LOCAL0);
syslog(LOG_ERROR, "Bad argument: %s", argv[1]);
syslog(LOG_USER | LOG_INFO, "Exiting");
```

Поскольку категория (`facility`) не была указана при первом вызове `syslog()`, по умолчанию используется значение из вызова `openlog()` (`LOG_LOCAL0`). Во втором вызове `syslog()` явно заданная константа `LOG_USER` переопределяет значение по умолчанию, установленное вызовом `openlog()`.

Для добавления записей в системный журнал можно воспользоваться утилитой командной строки logger(1). Эта программа позволяет указать для записываемых сообщений значения level (приоритет) и ident (метка). Больше подробностей содержится на справочной странице logger(1). Команда logger присутствует в большинстве реализаций UNIX, хотя в стандарте SUSv3 она упоминается лишь вскользь.

Нельзя передавать вызову `syslog()` произвольные строки, переданные пользователем:

```
syslog(priority, user_supplied_string);
```

Проблема этого кода состоит в том, что он делает приложение уязвимым к так называемым *атакам строки форматирования*. Если строка, которую предоставил пользователь, содержит спецификаторы форматирования (например, `%s`), результаты могут быть непредсказуемыми и потенциально опасными (эти замечания относятся и к применению стандартной функции `printf()`). Вызов, представленный выше, нужно переписать следующим образом:

```
syslog(priority, "%s", user_supplied_string);
```

Закрытие журнала

Закончив добавлять сообщения в журнал, мы можем вызвать функцию `closelog()`, чтобы освободить файловый дескриптор, связанный с сокетом `/dev/log`.

```
#include <syslog.h>

void closelog(void);
```

Поскольку демон обычно удерживает соединение с системным журналом на протяжении всей своей работы, вызов `closelog()` в основном не применяется.

Фильтрация журнальных записей

Функция `setlogmask()` устанавливает маску, которая фильтрует сообщения, записанные с помощью `syslog()`.

```
#include <syslog.h>

int setlogmask(int mask_priority);
```

Возвращает предыдущую маску приоритетов журнала

Любое сообщение, чей приоритет (`level`) не указан в текущей маске, отбрасывается. Маска по умолчанию позволяет записывать сообщения любой важности.

Макрос `LOG_MASK()` (объявленный в заголовочном файле `<syslog.h>`) приводит значения аргумента `levels` из табл. 37.2 к битовому виду, совместимому с вызовом `setlogmask()`. Например, чтобы разрешить запись сообщений с приоритетом не ниже `LOG_ERR`, нужно сделать следующий вызов:

```
setlogmask(LOG_MASK(LOG_EMERG) | LOG_MASK(LOG_ALERT |
           LOG_MASK(LOG_CRIT) | LOG_MASK(LOG_ERR));
```

Макрос LOG_MASK() описан в стандарте SUSv3. Большинство реализаций UNIX (включая Linux) предоставляют также нестандартный макрос LOG_UPTO(), который создает битовую маску, фильтрующую все сообщения, начиная с заданного приоритета (и выше). С его помощью можно упростить предыдущий вызов setlogmask():

```
setlogmask(LOG_UPTO(LOG_ERR));
```

37.5.3. Файл /etc/syslog.conf

Конфигурационный файл /etc/syslog.conf определяет поведение демона syslogd. Он состоит из правил и комментариев (последние начинаются с символа #). Правила в общем случае имеют следующий вид:

```
категория.приоритет        действие
```

Сочетание *категории* и *приоритета* называют *селектором*, поскольку они позволяют выбрать сообщения, к которым применяется правило. Эти поля представляют собой строки, соответствующие значениям из табл. 37.1 и 37.2. Под *действием* подразумевается место назначения сообщений, которые соответствуют *селектору*. *Селектор* и *действие* разделены пробельными символами. Ниже показан пример нескольких правил:

```
*.err                          /dev/tty10
auth.notice                    root
*.debug;mail.none;news.none    -/var/log/messages
```

Согласно первому правилу сообщения всех категорий (*) с приоритетом err (LOG_ERR) или выше должны передаваться консольному устройству /dev/tty10. Второе правило делает так, что сообщения, связанные с авторизацией (LOG_AUTH) и имеющие приоритет notice (LOG_NOTICE) или выше, должны отправляться во все консоли или терминалы, в которых работает пользователь root. Это, например, позволит администратору немедленно получать все сообщения о неудачных попытках повышения привилегий (su).

В последней строчке демонстрируются некоторые продвинутые аспекты синтаксиса для описания правил. В ней перечислено сразу несколько селекторов, разделенных точкой с запятой. Первый селектор относится к сообщениям любой категории (*) с приоритетом debug (самым низким) и выше; то есть это затрагивает *все* сообщения (в Linux, как и в большинстве других UNIX-систем, вместо debug можно указать символ *, который будет иметь то же значение; однако данная возможность поддерживается не всеми реализациями syslog). Обычно, если правило содержит несколько селекторов, оно охватывает сообщения, соответствующие любому из них, но если в качестве приоритета указать значение none, сообщения, принадлежащие к заданной категории, будут *отбрасываться*. Таким образом, это правило передает все сообщения (кроме тех, что имеют категории mail и news) в файл /var/log/messages. Символ «тильда» (~) перед именем этого файла говорит о том, что сбрасывание данных на диск будет происходить не при каждой передаче сообщения (см. раздел 13.3). Это приводит к увеличению скорости записи, но в случае сбоя системы сообщения, пришедшие недавно, могут быть утеряны.

При каждом изменении файла syslog.conf демону следует отправлять сигнал, чтобы он смог заново себя инициализировать:

```
$ killall -HUP syslogd              Отправляем сигнал SIGHUP демону syslogd
```

Синтаксис файла syslog.conf позволяет создавать куда более сложные правила, чем те, что были показаны. Все подробности можно найти на справочной странице syslog.conf(5).

37.6. Резюме

Демон — это долгоживущий процесс, не связанный с контролирующим терминалом (то есть работающий в фоновом режиме). Он выполняет специфичные задачи, такие как предоставление удаленного входа в систему или обслуживание веб-страниц. Чтобы стать демоном, программа должна выполнить стандартную последовательность шагов, включая выполнение вызовов `fork()` и `setsid()`.

При необходимости демоны должны корректно обрабатывать появление сигналов `SIGTERM` и `SIGHUP`; первый должен приводить к штатному завершению программы, тогда как второй должен служить уведомлением о том, что демону следует заново себя инициализировать, еще раз прочитать конфигурационный файл и повторно открыть любые журнальные файлы, которые он может использовать.

Система `syslog` предоставляет демонам (и другим приложениям) удобный способ записывать ошибки и прочие сообщения в единый журнал. Эти сообщения обрабатываются демоном `syslogd`, который распределяет их в соответствии с содержимым конфигурационного файла `syslogd.conf`. Они могут быть направлены разным адресатам, таким как терминалы, файлы на диске, пользователи, находящиеся в системе, и другие процессы на удаленных компьютерах (которые обычно являются другими демонами `syslogd`), связь с которыми поддерживается по сети TCP/IP.

Дополнительная информация

Вероятно, лучшим источником информации о написании демонов являются их исходные коды.

37.7. Упражнение

37.1. Напишите программу (похожую на `logger(1)`), которая записывает произвольные сообщения в системный журнал, используя вызов `syslog(3)`. Помимо аргумента командной строки, содержащего само сообщение, эта программа должна позволять указывать его приоритет (`level`).

38 Написание безопасных программ с повышенными привилегиями

Привилегированные программы имеют доступ к возможностям и ресурсам (файлам, устройствам и т. д.), недоступным обычным пользователям. Программа может получить повышенные привилегии двумя основными способами.

- Программа была запущена от имени привилегированного пользователя. К этой категории относятся многие демоны и сетевые серверы, которые обычно выполняются от имени администратора.
- Программе был установлен бит доступа с пользовательским или групповым идентификатором. При запуске такие программы меняют действующий идентификатор пользователя (группы) на тот, который имеет владелец (группа) исполняемого файла (впервые программы, устанавливающие идентификаторы пользователя и группы были описаны в разделе 9.3). В этой главе мы рассмотрим программы, которые устанавливают идентификатор администратора (`root`) и выдают процессу повышенные привилегии.

Если привилегированная программа содержит ошибки или может быть использована злоумышленником, это означает, что безопасность системы или приложения может быть нарушена. Учитывая это, мы должны писать программы таким образом, чтобы минимизировать вероятность потенциального взлома и ущерб, если он все же произойдет. Именно на этих темах мы и сосредоточимся в данной главе. Вы получите целый ряд рекомендаций по написанию безопасного кода и узнаете о различных подводных камнях, которых следует избегать при создании привилегированных программ.

38.1. Нужно ли программе устанавливать идентификаторы пользователя или группы?

Один из лучших советов касательно программ, устанавливающих идентификаторы пользователя (UID) или группы (GID), заключается в том, что их лучше не писать. Если существует возможность выполнить задачу без получения повышенных привилегий, мы должны рассматривать ее в первую очередь, поскольку это позволяет избежать потенциальной угрозы безопасности.

Иногда функциональность, требующую повышенных привилегий, можно изолировать и вынести в отдельную программу, которая выполняет одно единственное действие и при необходимости может быть запущена в дочернем процессе. Такой подход может быть особенно полезен в случае с библиотеками. Одним из примеров является программа `pt_chown`, описанная в подразделе 60.2.2.

Даже в случаях, когда установка идентификаторов пользователя или группы является необходимой, не всегда обязательно выдавать процессу привилегии администратора. Если программе подходят учетные данные с меньшими возможностями, вы должны отдавать предпочтение именно им, поскольку наличие прав администатора потенциально чревато нарушением безопасности.

Возьмем для примера приложение, которое должно предоставить пользователям возможность обновить файл, прав на запись которого они не имеют. Наиболее безопасным способом это сделать является создание отдельной групповой учетной записи (идентификатора группы) специально для этой программы и назначение этой группы соответствующему файлу (так, чтобы участники группы могли его записывать); после этого можно написать программу, которая будет устанавливать действующий идентификатор этой группы. Поскольку данная группа не имеет никаких других привилегий, это ограничивает потенциальный ущерб на случай, если программа содержит ошибки или может быть использована злоумышленником.

38.2. Работайте с минимальными привилегиями

Программе, устанавливающей идентификатор пользователя (или группы), привилегии обычно требуются для выполнения какой-то определенной операции. В остальное время эти привилегии должны быть отключены (особенно если они принадлежат администратору). Если вы знаете, что они больше не потребуются, от них следует полностью избавиться. Иными словами, программа всегда должна работать с минимальными привилегиями, достаточными для выполнения текущей задачи. Для этих целей была предусмотрена возможность сохранять устанавливаемый идентификатор (см. раздел 9.4).

Сохраняйте привилегии, только если они необходимы

Программа, устанавливающая идентификатор пользователя, может временно отказаться от привилегий и восстановить их позже, применив такую последовательность вызовов `seteuid()`:

```
uid_t orig_euid;

orig_euid = geteuid();
if (seteuid(getuid()) == -1)     /* Отказываемся от привилегий */
    errExit("seteuid");

/* Непривилегированная работа */

if (seteuid(orig_euid) == -1)    /* Восстанавливаем привилегии */
    errExit("seteuid");


/* Привилегированная работа */
```

Первый вызов делает действующий пользовательский идентификатор процесса таким же, как его реальный идентификатор. Второй вызов восстанавливает действующий UID, сохраненный ранее.

Программы, устанавливающие идентификатор группы, сохраняют исходный действующий GID, а вызов `setegid()` позволяет отказаться от привилегий или получить их заново. Функции `seteuid()`, `setegid()` и похожие системные вызовы, которые упоминаются ниже, описаны в главе 9 и собраны в табл. 9.1.

Наиболее безопасным подходом является отказ от привилегий сразу же при запуске программы и временное их восстановление при дальнейшей работе. Если в какой-то момент станет ясно, что привилегии вам больше не понадобятся, вы должны отказаться от них навсегда; для этого нужно изменить сохраненный пользовательский идентификатор. Это позволяет исключить восстановление привилегий обманным путем — например, с помощью повреждения стека, описанного в разделе 38.9.

Если привилегии больше не понадобятся, отказывайтесь от них навсегда

Если программа, устанавливающая UID или GID, завершила все задачи, требующие особых прав, она должна навсегда отказаться от повышенных привилегий, чтобы устранить любые риски безопасности, связанные с ошибками или непредвиденным поведением. Полный отказ от привилегий достигается за счет сбрасывания пользовательских (групповых) идентификаторов процесса к значению реального UID (GID).

Если действующий пользовательский ID программы равен 0 (то есть она имеет привилегии администратора), можуj сбросить для нее все UID с помощью такого кода:

```
if (setuid(getuid()) == -1)
    errExit("setuid");
```

Однако код, приведенный выше, *не* сбрасывает сохраненный пользовательский идентификатор, если в данный момент действующий UID вызывающего процесса не равен 0. Вызов `setuid()`, если его выполнить из программы с ненулевым действующим пользовательским идентификатором, изменяет только действующий UID (см. подраздел 9.7.1). Иными словами, в программе с правами администратора следующая цепочка вызовов не сбрасывает идентификатор пользователя к 0 навсегда:

```
/* Исходные UID: реальный=1000, действующий=0, сохраненный=0 */

/* 1. Вызов который обычно приводит к временному сбрасыванию привилегий */

orig_euid = geteuid();
if (seteuid(getuid() == -1)
    errExit("seteuid");

/* UID изменились, теперь реальный=1000, действующий=1000, сохраненный=0 */

/* 2. Похоже на правильный способ окончательного сбрасывания привилегий (НЕПРАВИЛЬНО) */

if (setuid(getuid() == -1)
    errExit("setuid");

/* UID изменились: реальный=1000, действующий=1000, сохраненный=0 */
```

Вместо этого перед окончательным отказом от привилегий мы снова должны их получить, вставив следующий вызов между первым и вторым шагом:

```
if (seteuid(orig_euid) == -1)
    errExit("seteuid");
```

С другой стороны, если программой владеет любой пользователь, кроме администратора, для окончательного сбрасывания привилегированного ID следует применить один из двух вызовов — `setreuid()` или `setresuid()`, так как `setuid()` для этого оказывается недостаточно. Например, можно достичь желаемого результата с помощью `setreuid()`, как показано ниже:

```
if (setreuid(getuid(), getuid()) == -1)
errExit("setreuid");
```

Этот код полагается на особенность реализации вызова `setreuid()` в Linux: если первый аргумент (`ruid`) не равен `-1`, значит, сохраненному пользовательскому идентификатору присваивается значение (нового) действующего UID. Этот нюанс не описан в стандарте SUSv3, но присутствует во многих других системах.

Аналогично для окончательного отказа от привилегированного идентификатора группы необходимо применить вызов `setregid()` или `setresgid()`, поскольку `setgid()`, если действующий UID программы не равен 0, изменяет действующий GID только для вызывающего процесса.

Общие замечания об изменении прав доступа для процессов

На предыдущих страницах были описаны методики временного и постоянного отказа от привилегий. Теперь мы добавим несколько общих замечаний касательно их использования.

- Семантика некоторых системных вызовов, которые изменяют права доступа для процессов, варьируется в зависимости от системы. Кроме того, эта семантика также зависит от того, имел ли вызывающий процесс повышенные привилегии (действующий пользовательский идентификатор равный 0). Подробности ищите в главе 9, особенно в подразделе 9.7.4. Ввиду этих расхождений [Tsafrir et al., 2008] для изменения привилегий процесса рекомендуется использовать *нестандартные* системные вызовы, так как во многих случаях они имеют более простую и последовательную семантику, чем их стандартные аналоги. В Linux это означает применение вызовов `setresuid()` и `setresgid()`, которые изменяют пользовательские и групповые привилегии. И хотя они поддерживаются не во всех системах, их применение, скорее всего, будет менее подвержено ошибкам (в книге [Tsafrir et al., 2008] предлагается библиотека функций, которая выполняет изменение привилегий с помощью наиболее подходящих интерфейсов, доступных на каждой платформе).
- В Linux, даже если действующий пользовательский идентификатор вызывающего процесса равен 0, системные вызовы, изменяющие привилегии, могут демонстрировать неожиданное поведение, если программа явно ограничила свои возможности. Например, в результате отключения возможности `CAP_SETUID` попытки изменения UID процесса завершаются неудачей (или, что еще хуже, приводят к изменению только некоторых из запрашиваемых идентификаторов).
- Учитывая риски, описанные в двух предыдущих пунктах, крайне рекомендуется (см. [Tsafrir et al., 2008]) не только проверять успешность системного вызова для изменения прав доступа, но и следить за тем, чтобы внесенные изменения действительно вступили в силу. Например, если мы временно отказываемся от идентификатора привилегированного пользователя или заново его получаем, применяя `seteuid()`, нам следует после этого сделать вызов `geteuid()`, чтобы узнать, соответствует ли действующий UID нашим ожиданиям. Аналогично после окончательного отказа от привилегий следует проверить, принадлежат ли реальный, действующий и сохраненный идентификаторы непривилегированному пользователю. К сожалению, стандартные вызовы позволяют получить только реальный и действующий UID, но не сохраненный. В Linux и нескольких других реализациях для этого предусмотрены вызовы `getresuid()` и `getresgid()`; в остальных системах могут понадобиться нетривиальные методики — например, разбор информации в файлах `/proc`.
- Некоторые права доступа могут быть изменены только процессом, чей действующий UID равен 0. Следовательно, при изменении нескольких идентификаторов (GID, дополнительного GID и UID) в последнюю очередь следует отказываться от действующего UID привилегированного пользователя. И наоборот — при получении привилегий действующий UID привилегированного пользователя должен быть первым.

38.3. Будьте осторожны при выполнении программы

Следует соблюдать осторожность при запуске внешней программы из своего приложения — либо напрямую, с помощью `exec()`, либо опосредованно, применяя вызовы `system()`, `popen()` или аналогичные.

Полностью отказывайтесь от привилегий перед запуском другой программы

Если приложение, устанавливающее пользовательский или групповой идентификатор, запускает другую программу, оно должно сбросить все свои UID и GID к значению реального идентификатора пользователя (группы), чтобы новая программа не унаследовала повышенные привилегии и не смогла их самостоятельно получить. Один из способов, как этого можно достичь, заключается в сбрасывании всех идентификаторов до выполнения `exec()`, применяя методики, описанные в разделе 38.2.

Тот же результат достигается путем вызова `setuid(getuid())`, предшествующего `exec()`. И хотя `setuid()` изменяет действующий идентификатор пользователя только в процессе, чей UID не равен 0, привилегии все равно сбрасываются, потому что (как было описано в разделе 9.4) успешный вызов `exec()` делает сохраненный UID таким же, как и действующий (если же `exec()` завершается неудачей, сохраненный идентификатор остается без изменений; это может пригодиться в ситуациях, когда после неудачного вызова `exec()` программе приходится выполнять другие привилегированные операции).

Аналогичный подход (то есть вызов `setgid(getgid())`) можно применить к программам, устанавливающим идентификатор группы, поскольку вызов `exec()` делает сохраненный GID таким же, как и действующий.

Представьте, к примеру, что у нас есть программа, владелец которой имеет идентификатор 200. Если ее запустит пользователь с UID 1000, пользовательские идентификаторы итогового процесса будут выглядеть так:

```
реальный=1000 действующий=200 сохраненный=200
```

Если данная программа впоследствии выполнит вызов `setuid(getuid())`, пользовательские идентификаторы процесса изменятся следующим образом:

```
реальный=1000 действующий=1000 сохраненный=200
```

Когда процесс выполняет непривилегированную программу, действующий UID процесса становится сохраненным, в результате чего значения пользовательских идентификаторов будут иметь такой вид:

```
реальный=1000 действующий=200 сохраненный=1000
```

Избегайте выполнения командной оболочки (или другого интерпретатора) с повышенными привилегиями

Привилегированные программы, запущенные пользователем, никогда не должны запускать командную оболочку (напрямую или опосредованно, с помощью библиотечных функций `system()`, `popen()`, `execlp()`, `execvp()` и им подобных). Сложность и обширные возможности командных оболочек (и других многофункциональных интерпретаторов, таких как `awk`) делают практически невозможным устранение всех лазеек, связанных с безопасностью, даже если отключить интерактивный режим. Это, в свою очередь, дает возможность выполнять произвольные консольные команды от имени пользователя с действующим идентификатором процесса. Если вам необходимо запустить командную оболочку, заранее откажитесь от повышенных привилегий.

> Пример бреши в безопасности, которая может возникнуть в результате запуска командной оболочки, приводится при описании функции system() в разделе 27.6.

Немногочисленные реализации UNIX учитывают биты с идентификаторами пользователя и группы, установленные для интерпретируемых скриптов (см. раздел 27.3); благодаря этому процесс, запускающий скрипт, исходит из того, что тот будет выполняться от имени какого-то другого (привилегированного) пользователя. Но ввиду рисков

безопасности, описанных выше, Linux, как и некоторые другие UNIX-системы, просто игнорирует эти биты при выполнении скриптов. Но даже если система поддерживает установку UID и GID для скриптов, этой возможностью следует пренебрегать.

Закрывайте все ненужные файловые дескрипторы перед выполнением exec()

В разделе 27.4 отмечалось, что файловые дескрипторы по умолчанию остаются открытыми на протяжении всей работы вызова `exec()`. Привилегированная программа может открыть файл, недоступный обычным процессам. Полученный файловый дескриптор представляет привилегированный ресурс. Перед выполнением `exec()` он должен быть закрыт, чтобы запускаемая программа не смогла получить доступ к связанному с ним файлу. Для этого можно либо вручную закрыть файловый дескриптор, либо установить флаг `FD_CLOEXEC` (см. раздел 27.4).

38.4. Избегайте раскрытия деликатной информации

Если программа считывает деликатную информацию (например, пароль), она должна удалить ее из памяти сразу же после выполнения всех необходимых задач (пример этого продемонстрирован в разделе 8.5). В противном случае могут возникнуть риски безопасности, проистекающие из следующих причин.

- Страница виртуальной памяти, содержащая данные, может быть сброшена на диск (если только она не была заблокирована с помощью вызова `mlock()` или аналогичного) и прочитана оттуда привилегированной программой.
- Процесс может сгенерировать файл с дампом памяти, получив соответствующий сигнал; этот файл может быть использован для получения данных.

Учитывая последний пункт, можно прийти к выводу, что безопасная программа в целом не должна генерировать дампы памяти, так как из них можно извлечь деликатную информацию. Для этого можно обнулить ограничение на ресурсы `RLIMIT_CORE`, воспользовавшись вызовом `setrlimit()` (см. раздел 36.3).

> По умолчанию Linux не разрешает программе, устанавливающей пользовательский идентификатор, генерировать дамп памяти в ответ на сигнал (см. раздел 22.1), даже если она отказалась от всех привилегий. Однако в других системах этот элемент безопасности может отсутствовать.

38.5. Изоляция процесса

В этом разделе мы рассмотрим различные способы изоляции программы, которые позволяют ограничить потенциальный ущерб в случае ее взлома.

Подумайте об использовании системы возможностей

В Linux существует система, позволяющая разделить традиционную для UNIX структуру привилегий («все или ничего») на отдельные элементы, которые называются *возможностями*. Возможности могут включаться и выключаться самим процессом в индивидуальном порядке. Выдавая программе нужные ей возможности, мы обеспечиваем ей менее широкие привилегии, чем если бы она выполнялась от имени администратора. Это снижает потенциал для нанесения вреда в случае ее взлома.

Кроме того, используя возможности в сочетании с *битами безопасности*, можно создать процесс, обладающий ограниченными функциями, однако не принадлежащий

администратору (то есть ни один из его пользовательских идентификаторов не равен 0). Такой процесс больше не сможет применить функцию `exec()` для возвращения себе полного набора возможностей. Мы вернемся к этой теме в главе 39.

Подумайте об использовании тюрьмы chroot

В некоторых ситуациях может пригодиться еще один элемент безопасности, *тюрьма* `chroot`, который позволяет ограничить набор каталогов и файлов, доступных программе (не забудьте сделать вызов `chdir()`, чтобы текущий каталог находился внутри тюрьмы). Однако имейте в виду, что этот механизм не способен изолировать программу, устанавливающую идентификатор администратора (см. раздел 18.12).

> Альтернативой тюрьме chroot является виртуальный сервер; это сервер, реализованный поверх виртуального ядра. Поскольку все виртуальные ядра, работающие на одном и том же компьютере, изолированы друг от друга, виртуальный сервер обеспечивает лучшие безопасность и гибкость по сравнению с тюрьмой chroot (некоторые современные операционные системы предоставляют собственные реализации виртуальных серверов). Впервые механизм виртуализации в Linux был реализован в виде системы User-Mode Linux (Linux пользовательского режима, или UML), которая является стандартной частью ядра версии 2.6. Подробности о UML можно найти по адресу user-mode-linux.sourceforge.net. Более современные проекты виртуальных ядер основаны на системах Xen (www.cl.cam.ac.uk/Research/SRG/netos/xen/) и KVM (kvm.qumranet.com).

38.6. Не забывайте о сигналах в состоянии гонки

Пользователи могут отправлять запущенным ими программам произвольные сигналы. Это может происходить в любое время и с любой частотой. Не следует забывать о том, что получение сигнала на *любом* этапе выполнения программы может привести к состоянию гонки. При необходимости, чтобы избежать потенциальных проблем с безопасностью, сигнал должен быть перехвачен, заблокирован или проигнорирован. Кроме того, структура обработчиков сигналов должна быть максимально простой, чтобы снизить риск случайного входа в состояние гонки.

Особенно остро эта проблема стоит в случае с сигналами, которые останавливают процесс (например, `SIGTSTP` и `SIGSTOP`). Возможен следующий сценарий.

1. Программа, устанавливающая пользовательский идентификатор, определяет некоторые сведения о среде выполнения.
2. Пользователю удается остановить процесс, выполняющий программу, и изменить некоторые свойства среды выполнения. Это может быть изменение прав доступа к файлу, замена целевого объекта символьной ссылки или удаление файла, от которого зависит программа.
3. Пользователь возобновляет процесс с помощью сигнала `SIGCONT`. На этом этапе программа продолжит работу, оперируя ложными данными о среде выполнения, что может привести к дыре в безопасности.

Описанная выше ситуация на самом деле является всего лишь частным случаем состояния гонки вида «время проверки ко времени использования» (англ. *time-of-check-to-time-of-use*, или TOCTTOU). Привилегированный процесс не должен выполнять операции, основываясь на ранее проведенных проверках, которые могли потерять свою актуальность (конкретный пример приводится в подразделе 15.4.4, в котором обсу-

ждается системный вызов access()). Эта рекомендация относится даже к тем случаям, когда пользователь не может отправлять процессу сигналы. Возможность остановить процесс просто позволяет увеличить интервал между моментом проверки и моментом использования.

> Остановить программу в момент между проверкой данных и их использованием не так-то просто, но злоумышленник может предпринять множество попыток, применяя параллельно скрипт командной оболочки, который постоянно шлет программе сигналы остановки и изменяет среду выполнения. Это значительно увеличивает его шансы на успешный взлом программы.

38.7. Подводные камни, связанные с файловыми операциями и вводом/выводом файлов

Если привилегированный процесс должен создать файл, мы должны позаботиться о правах доступа к этому файлу, чтобы исключить любую возможность вредоносных манипуляций с ним на любом, даже самом коротком отрезке времени. Для этого необходимо следовать нижеописанным рекомендациям.

- Атрибут umask процесса (см. подраздел 15.4.6) должен иметь значение, которое запрещает ему создавать файлы, открытые для публичной записи, поскольку они могут быть отредактированы злоумышленником.
- Так как владелец файла определяется на основе пользовательского идентификатора создающего процесса, для установления корректной информации о владельце созданного файла, возможно, придется временно изменить учетные данные с помощью вызова seteuid() или setreuid(). То же самое касается программ, устанавливающих GID, так как принадлежность к группе *может* определяться на основе действующего группового идентификатора (см. подраздел 15.3.1). Проблем можно избежать, если использовать соответствующие вызовы для работы с идентификатором группы (строго говоря, в Linux владелец нового файла определяется на основе UID файловой системы процесса, который обычно совпадает с действующим пользовательским идентификатором; см. раздел 9.5).
- Если программа, устанавливающая идентификатор администратора, должна изначально владеть файлом, который она создает, но потом может передать владение другому пользователю, этот файл с самого начала должен быть доступен для записи только администратору. Этого можно достичь с помощью подходящего значения для аргумента mode при вызове open() или путем задания атрибута umask для процесса перед этим вызовом. После этого программа при необходимости может воспользоваться вызовами fchown() и fchmod(), чтобы поменять соответственно данные о владении и права доступа. Ключевой момент заключается в том, что программа, устанавливающая UID, никогда не должна создавать файлы, которые принадлежат ее владельцу и которые могут в какой-либо момент быть изменены другими пользователями.
- Атрибуты файла следует проверять уже после получения открытого файлового дескриптора (например, когда за вызовом open() следует fstat()), вместо того чтобы сначала делать проверку по пути к файлу, а только потом его открывать (например, когда вызов open() выполняется после stat()). В противном случае возникает проблема вида TOCTTOU.
- Если программа должна убедиться в том, что она является создателем файла, при выполнении вызова open() следует использовать флаг O_EXCL.
- Привилегированная программа не должна создавать или задействовать файлы в публичных каталогах, доступных для записи (таких как /tmp), так как это позволяет

злоумышленнику создавать файлы с именами, на которые эта программа рассчитывает. Если вам действительно необходимо создать файл в каталоге, открытом для публичной записи, вы как минимум должны указать для него непредсказуемое имя, используя такие функции, как `mkstemp()` (см. раздел 5.12).

38.8. Не доверяйте внешнему вводу или среде выполнения

Привилегированные программы не должны действовать, исходя из своих предположений о предоставляемом им вводе или среде, в которой они выполняются.

Не доверяйте переменным среды

Программы, устанавливающие идентификаторы пользователя и группы, не должны полагаться на аутентичность переменных среды. Особенно это относится к двум переменным: `PATH` и `IFS`.

`PATH` определяет, где командная оболочка (а следовательно, и вызовы `system()` и `popen()`) и функции `execlp()` и `execvp()` будут искать программу. Злоумышленник может присвоить этой переменной значение, из-за которого вы по ошибке запустите в привилегированном режиме произвольную программу. Если вы используете вышеупомянутые функции, переменная `PATH` должна быть ограничена списком доверенных каталогов (хотя для запуска программ лучше указывать абсолютные пути). Однако, как уже было отмечено, перед выполнением командной оболочки или использованием функций семейства `exec()` лучше всего отказаться от привилегий.

`IFS` определяет символы, которые командная оболочка интерпретирует как разделители между словами в командной строке. Эта переменная должна быть равна пустой строке — это означает, что разделителями могут быть только пробельные символы. Некоторые оболочки используют ее с этой целью при загрузке (в разделе 27.6 описана одна уязвимость, которая связана с переменной среды `IFS` и проявляется в старых версиях интерпретатора `bash`).

В некоторых ситуациях наиболее надежным вариантом является полная очистка списка переменных среды (см. раздел 6.7) и восстановление только тех из них, которые точно имеют безопасные значения. Особенно это касается запуска внешних программ или вызов библиотек, на которые может влиять конфигурация переменных среды.

Осторожно обрабатывайте пользовательский ввод

Привилегированная программа должна тщательно проверять любые данные, поступающие не из доверенных источников, прежде чем предпринимать на их основе какие-либо действия. Проверка может включать в себя подтверждение того, что все числа попадают в заданный диапазон и что строки имеют приемлемую длину и состоят из разрешенных символов. Данные, требующие проверки, могут поступать в виде файлов, создаваемых пользователями, аргументов командной строки, интерактивного ввода, CGI-ввода, почтовых сообщений, переменных среды, каналов межпроцессного взаимодействия (очереди типа FIFO, разделяемая память и т. д.), доступных недоверенным пользователям, или сетевых пакетов.

Избегайте необоснованных предположений о среде выполнения процесса

Программы, устанавливающие пользовательский идентификатор, не должны делать необоснованные предположения относительно исходной среды, в которой они выполняются. Например, стандартные потоки ввода, вывода или ошибок могут быть закрыты

(это могла сделать программа, которая запустила ваше приложение). В этом случае при открытии файла можно случайно задействовать дескриптор с номером 1 (к примеру) и записывать данные в обычный файл вместо стандартного вывода.

Существует множество моментов, которые нужно учитывать. Например, процесс может исчерпать различные ограничения на ресурсы, такие как количество процессов, доступных для создания, процессорное время или размер файла. В результате этого могут завершиться неудачно разные системные вызовы, и вы можете получить разные сигналы. Злоумышленники, пытаясь взломать вашу программу, могут нарочно попытаться устроить такой сценарий.

38.9. Остерегайтесь переполнений буфера

Остерегайтесь ситуаций, когда входящее значение или скопированная строка превышают размер выделенного буфера. Никогда не используйте вызов `gets()` и будьте осторожны при работе с функциями `scanf()`, `sprintf()`, `strcpy()` и `strcat()` (например, применяйте инструкцию `if`, которая предотвращает переполнение буфера).

Переполнение буфера делает программу уязвимой к *повреждению стека* (англ. *stack crashing*) — это когда злоумышленник использует переполнение, чтобы разместить в блоке стека тщательно закодированные байты, принуждающие привилегированную программу выполнить произвольный код (эта методика описана на нескольких интернет-ресурсах; см. также [Erickson, 2008] и [Anley, 2007]). Переполнение буфера, вероятно, является наиболее распространенным источником проблем безопасности в компьютерных системах, судя по частоте их упоминаний в публикациях CERT (www.cert.org) и Bugtraq (www.securityfocus.com). Эта проблема является особенно опасной для сетевых серверов, так как она делает их уязвимыми к удаленным атакам из любой точки сети.

> Чтобы усложнить задачу повреждения стека (в частности, чтобы сделать эту процедуру более затратной по времени, когда она проводится с удаленными сетевыми серверами), ядро Linux, начиная с версии 2.6.12, поддерживает рандомизацию адресного пространства (Address Space Layout Randomization, или ASLR). Эта технология случайным образом меняет местоположение стека в верхнем сегменте виртуальной памяти размером 8 Мбайт. Также случайные адреса могут выбираться для отображения в памяти, если ограничение RLIMIT_STACK не бесконечно, а файл /proc/sys/vm/legacy_va_layout (поддерживается только в Linux) содержит значение 0.
>
> Более современные реализации архитектуры x86-32 предоставляют аппаратную поддержку маркировки таблиц памяти метками NX (no execute), которые предотвращают их выполнение. Эта технология не дает выполнять код программы в стеке, что делает повреждение стека еще более затруднительным.

Многие функции, упомянутые выше (например, `snprintf()`, `strncpy()` и `strncat()`), имеют безопасные альтернативы, позволяющие вызывающему процессу указать максимальное количество символов, которые должны быть скопированы. Они учитывают это ограничение и позволяют избежать переполнения заданного буфера. В целом эти альтернативы являются предпочтительными, но и их следует использовать осторожно. В частности, нужно учитывать следующие моменты.

- При достижении максимума большинство этих функций усекает исходную строку, помещенную в заданный буфер. Поскольку такое усечение может расходиться с семантикой программы, вызывающий процесс должен проверять, произошло ли оно (например, используя значение, возвращенное вызовом `snprintf()`), и при необходимости предпринимать соответствующие меры.

- Функция `strncpy()` может отрицательно сказаться на производительности. Например, если в вызове `strncpy(s1, s2, n)` длина строки, на которую указывает `s2`, меньше `n` байт, тогда недостающее место заполняется нулевыми символами, чтобы соблюсти указанную длину при записи.
- Если значения максимального размера, переданного функции `strncpy()`, не хватает для добавления в конце строки нулевого символа, итоговая строка *не* будет содержать в конце нулевой символ.

  Некоторые реализации UNIX предоставляют функцию strlcpy(), которая, если ей передать аргумент n, копирует в заданный буфер максимум n – 1 байт и всегда добавляет в конце нулевой символ. Однако эта функция не входит в стандарт SUSv3 и не реализована в библиотеке glibc. Кроме того, если вызывающий процесс не выполняет тщательную проверку длины строки, эта функция меняет одну проблему (переполнение буфера) на другую (неявное отклонение данных).

38.10. Остерегайтесь DoS-атак

С ростом количества интернет-услуг увеличивается и риск удаленных DoS-атак (англ. *denial-of-service* — отказ в обслуживании). Эти атаки направлены на то, чтобы сделать ресурс недоступным для обычных клиентов, и заключаются в отправке серверу искаженных данных, приводящих к сбоям, или перегрузке его фиктивными запросами.

DoS-атаки возможны и на локальном уровне. Наиболее известный пример — запуск простой fork-бомбы (программы, которая многократно выполняет вызов fork(), занимая все блоки, выделенные системой под создание процессов). Однако происхождение локальной DoS-атаки значительно легче определить, и они относительно легко предотвращаются с помощью надлежащих мер безопасности — как на физическом уровне, так и на уровне пароля.

Защита от искаженных запросов довольно тривиальна — сервер должен быть запрограммирован для тщательной проверки поступающего ввода и исключения переполнений буфера, описанных выше.

Сложнее защититься от атак, связанных с перегрузкой. Сервер не может контролировать поведение удаленных клиентов или частоту, с которой те отправляют свои запросы, поэтому такие атаки невозможно предотвратить (иногда сервер не может даже определить источник атаки, поскольку IP-адрес сетевого пакета может быть поддельным; кроме того, в распределенных атаках могут использоваться промежуточные сетевые узлы, захваченные для этой цели). Тем не менее вы можете снизить риск атак, связанных с перегрузкой, и минимизировать их последствия, предприняв следующие меры.

- Сервер должен регулировать загрузку и отклонять запросы, когда ее уровень превышает какое-то граничное значение. Следствием этого является отклонение некоторых нормальных запросов, однако данный подход помогает предотвратить перегрузку сервера и компьютера, на котором он работает. В ограничении чрезмерной загруженности также могут помочь ограничения на ресурсы и дисковые квоты (подробности о последних можно найти по адресу sourceforge.net/projects/linuxquota/).
- При взаимодействии с клиентом сервер должен иметь ограниченное время ожидания, чтобы не ждать бесконечно клиентов, которые перестают отвечать (возможно, намеренно).
- Сервер должен быть написан таким образом, чтобы неожиданная загрузка не могла привести к его сбою. Например, чтобы избыточный запрос не вышел за пределы структуры данных, необходимо тщательно проверять его на вхождение в допустимый диапазон.

- Структуры данных должны быть стойкими к *атакам на алгоритмическую сложность* (англ. *algorithmic-complexity attacks*). Например, при обычно загрузке двоичное дерево может быть сбалансировано и обеспечивать приемлемую производительность. Однако злоумышленник может сформировать последовательность входящих данных, которая способна привести к разбалансированию дерева (в худшем случае превращая его в эквивалент связного списка); от этого может пострадать скорость работы. Книга [Crosby & Wallach, 2003] подробно рассматривает суть таких атак и описывает методики написания структур данных, которые позволяют их избежать.

38.11. Проверяйте результаты выполнения и предусматривайте безопасное завершение в случае неудачи

Привилегированная программа должна проверять, завершились ли системные вызовы или библиотечные функции успешно и вернули ли они ожидаемый результат (конечно, это относится ко всем программам, но при наличии повышенных привилегий это имеет особенно большое значение). Различные системные вызовы могут завершиться неудачей, даже если вызывающая их программа выполняется от имени администратора. Например, к неудачному завершению вызова `fork()` может привести исчерпание общесистемного ограничения на количество процессов; в случае с вызовом `open()` это может быть попытка записи в файловой системе, предназначенной только для чтения; причиной неудачного вызова `chdir()` может стать отсутствие заданного каталога.

Иногда, даже если системный вызов завершился успешно, его результат все равно нужно проверить. Например, если это имеет значение, необходимо проверить, вернул ли успешный вызов `open()` один из стандартных файловых дескрипторов: 0, 1 или 2.

И наконец, если привилегированная программа сталкивается с непредвиденной ситуацией, обычно она должна сделать одно из двух: либо завершиться, либо, если это сервер, отклонить клиентский запрос. Попытки исправить неожиданную проблему обычно вынуждают делать предположения, которые в некоторых случаях могут быть необоснованными, что приводит к созданию лазеек в безопасности. В таких ситуациях более безопасными являются завершение программы или запись сообщения в журнал и отклонение запроса клиента.

38.12. Резюме

Привилегированные программы могут работать с системными ресурсами, недоступными обычным пользователям. В случае взлома такой программы безопасность системы может оказаться под ударом. В данной главе был дан целый ряд рекомендаций по написанию привилегированных приложений. Эти рекомендации имеют двойное назначение: во-первых, снизить вероятность взлома программы с повышенными привилегиями и, во-вторых, минимизировать ущерб в случае, если этого не удалось избежать.

Дополнительная информация

В [Viega & McGraw, 2002] рассматривается целый ряд тем, связанных с проектированием и реализацией безопасного программного обеспечения. Общие сведения о безопасности в UNIX-системах, а также целую главу, посвященную методикам безопасного

программирования, можно найти в книге [Garfinkel et al., 2003]. Тема компьютерной безопасности также затронута в [Bishop, 2005], хотя книга того же автора, [Bishop, 2003], дает более широкий охват. [Peikari & Chuvakin, 2004] делает акцент на различных способах атаки системы. В книгах [Erickson, 2008] и [Anley, 2007] дается исчерпывающий разбор вредоносных программ (экспоитов) и предоставляется достаточно информации, благодаря которой мудрый программист сможет от них защититься. В документе [Chen et al., 2002] описывается и анализируется модель установки пользовательских идентификаторов в UNIX. Документ [Tsafrir et al., 2008] пересматривает и дополняет обсуждение различных аспектов, изложенных в [Chen et al., 2002]. [Drepper, 2009] содержит богатый набор советов о безопасном и защитном программировании в Linux.

Несколько источников информации о написании безопасных программ можно найти в Сети.

- Мэтт Бишоп сделал целый ряд публикаций, связанных с безопасностью. Они доступны по адресу nob.cs.ucdavis.edu/~bishop/secprog. Наиболее интересная из них носит название «Как написать программу, устанавливающую UID» (англ. How to Write a Setuid Program; изначально опубликована в журнале ;login, № 12 (1), январь/февраль 1896 г.). Эта статья, несмотря на свой возраст, содержит огромное количество полезных советов.
- Руководство по безопасному программированию для Linux и Unix (англ. Secure Programming for Linux and Unix HOWTO) доступно по адресу www.dwheeler.com/secure-programs/.
- Перечень полезных рекомендаций по написанию программ, устанавливающих пользовательские идентификаторы, находится на сайте www.homeport.org/~adam/setuid.7.html.

38.13. Упражнения

38.1. Войдите в систему как обычный, непривилегированный пользователь, создайте исполняемый файл (или скопируйте уже существующий — например, `/bin/sleep`) и установите ему бит, позволяющий менять идентификатор пользователя (`chmod u+s`). Попробуйте отредактировать этот файл (например, `cat >> file`). Введите команду `ls -l` и посмотрите, что произошло с правами доступа. В чем причина?

38.2. Напишите программу, устанавливающую идентификатор администратора, похожую на команду `sudo(8)`. Она должна принимать параметры и аргументы командной строки следующего вида:

```
$ ./douser [-u user ] program-file arg1 arg2 ...
```

Программа `douser` запускает файл `program-file`, передавая ему аргументы так, как будто он был запущен пользователем `user` (если опустить параметр `-u`, пользователем по умолчанию должен быть администратор). Перед выполнением файла `program-file` программа `douser` должна запросить у пользователя пароль и сверить введенную строку с содержимым стандартного файла для хранения паролей (см. листинг 8.2). После этого она должна установить все идентификаторы UID и GID, принадлежащие заданному пользователю.

39 Система возможностей

Эта глава посвящена системе возможностей в Linux, которая разбивает традиционную для UNIX структуру привилегий («все или ничего») на отдельные сегменты, которые могут включаться и отключаться независимо друг от друга. Использование возможностей позволяет программе выполнять только строго определенные привилегированные операции.

39.1. Зачем нужна система возможностей

Традиционная для UNIX структура привилегий разделяет процессы на две категории: те, чей действующий пользовательский идентификатор равен 0 (то есть принадлежащие администратору) и которые обходят любые проверки прав доступа, и все остальные, права доступа которых проверяются в соответствии с их пользовательскими и групповыми идентификаторами.

Проблема этой структуры заключается в ее недостаточной гибкости. Если мы хотим разрешить процессу некоторые операции, доступные только администратору (например, изменять системное время), нам придется запускать его с действующим идентификатором равным 0 (если пользователю нужно выполнить такие операции, для этого обычно приходится устанавливать UID администратора). Однако этим мы позволяем процессу выполнять целый ряд других действий — например, обходить любые проверки прав доступа при работе с файлами. Тем самым мы рискуем создать дыру в безопасности, если программа начнет вести себя непредсказуемо (из-за непредвиденных обстоятельств или как следствие целенаправленных действий злоумышленника). Традиционный способ борьбы с этой проблемой был описан в предыдущей главе: мы отказываемся от действующих привилегий (то есть меняем действующий идентификатор пользователя, равный 0, на какой-то другой, сохраняя его на будущее) и временно их возвращаем, когда они нужны.

Система возможностей Linux позволяет решать эту задачу с большей точностью. Вместо того чтобы использовать сразу все привилегии администратора (пользователя с идентификатором 0), выполняя проверки безопасности на уровне ядра, мы разделяем их на отдельные категории, которые называются возможностями. Каждая привилегированная операция связывается с определенной возможностью и доступна для выполнения только в том случае, если эта возможность присутствует у текущего процесса (независимо от действующего UID). Иными словами, когда в этой книге говорится о привилегированном процессе, речь идет о наличии у него возможностей, подходящих для выполнения определенной операции.

Большую часть времени система возможностей Linux работает незаметно для пользователей. Дело в том, что в программе, которая рассчитывает получить нулевой пользовательский идентификатор, выдается полный набор возможностей, даже если она о них не подозревает.

Реализация возможностей в Linux основана на проекте стандарта POSIX 1003.1e (http://wt.tuxomania.net/publications/posix.1e/). Эта инициатива не была доведена до логического завершения, провалившись еще в конце 1990-х, но предварительное описание стандарта, которое она породила, стало фундаментом для различных систем управления

возможностями (некоторые возможности, перечисленные в табл. 39.1, являются частью проекта стандарта POSIX.1e, но в большинстве своем это расширения, появившиеся в Linux).

> Системы возможностей присутствуют и в нескольких других реализациях UNIX — например, в Solaris 10 компании Sun и в более ранних версиях Trusted Solaris, таких как Trusted Irix компании SGI и как часть проекта TrustedBSD для FreeBSD ([Watson, 2000]). Аналогичные механизмы существуют и в других операционных системах; например, похожим образом организованы привилегии в системе VMS компании Digital.

39.2. Система возможностей в Linux

В табл. 39.1 перечислены возможности, доступные в Linux, а также краткое (неполное) описание операций, к которым они относятся.

39.3. Возможности, связанные с процессами и файлами

Каждый процесс имеет три набора возможностей: *разрешенные*, *действующие* и *наследуемые*. Каждый из них может быть пустым или содержать возможности, перечисленные в табл. 39.1. То же самое относится и к файлам (по причинам, которые будут наглядно показаны, действующие возможности файла представляют собой один-единственный бит, который может быть либо включен, либо выключен). В следующих разделах мы подробно рассмотрим каждую из этих возможностей.

39.3.1. Возможности процесса

Для каждого процесса ядро предоставляет три набора возможностей (реализованных в виде битовых масок).

- *Разрешенные*. Это список возможностей, которые процесс *может* применить. Возможности, которые в него входят, потенциально могут стать действующими или наследуемыми. Возможность, исключенная из этого набора, уже никогда не может быть использована (разве что она принадлежит программе, которую запустил ваш процесс).
- *Действующие*. Это возможности, которые задействуются ядром для проверки привилегий процесса. Возможность, которая входит в этот набор, может быть временно из него исключена и внесена обратно.
- *Наследуемые*. Это возможности, которые могут быть переданы в разрешенный набор, когда программа выполняется текущим процессом.

Каждый из трех наборов возможностей для каждого процесса представлен в шестнадцатеричном виде в файле `/proc/PID/status` (поля `CapInh`, `CapPrm` и `CapEff`).

> Программа getpcap (которая входит в пакет libcap, описанный в разделе 39.7) позволяет вывести возможности процесса в удобном для чтения формате.

Дочерний процесс, созданный с помощью вызова `fork()`, наследует копии возможностей своего родителя. То, как наследуются возможности во время выполнения `exec()`, будет описано в разделе 39.5.

На самом деле возможности применяются и могут регулироваться для каждого отдельного потока в процессе. Возможности определенного потока внутри многопоточного процесса можно просмотреть в файле /proc/PID/task/TID/status. Файл /proc/PID/status показывает возможности только главного потока.

Раньше для представления возможностей в Linux использовалось 32 бита. Но в версии 2.6.25, в которую были добавлены новые возможности, был осуществлен переход на 64-битные наборы.

39.3.2. Возможности файлов

Если у файла есть наборы возможностей, они передаются процессу, который этот файл выполняет. У файла может быть три набора возможностей.

- *Разрешенные*. Это возможности, которые могут быть добавлены в разрешенный набор во время выполнения `exec()` и вне зависимости от текущих возможностей процесса.
- *Действующие*. Это всего лишь один бит. Если он включен, возможности, входящие в новый разрешенный набор процесса, могут также быть включены в его действующий набор во время выполнения `exec()`. Если этот бит выключен, новый набор действующих возможностей процесса после вызова `exec()` является пустым.
- *Наследуемые*. Это маска, которая применяется к набору наследуемых возможностей процесса, чтобы определить, какие из них должны быть включены в разрешенный набор после вызова `exec()`.

Подробности об использовании возможностей файла во время выполнения `exec()` приводятся в разделе 39.5.

Разрешенные и наследуемые возможности файла раньше назывались принудительными и допустимыми. Сейчас эти термины считаются устаревшими, но они по-прежнему имеют смысл. Разрешенные возможности принудительно включаются в действующий набор во время выполнения exec(), независимо от текущих возможностей процесса. Наследуемые возможности файла допускаются для включения в разрешенный набор во время выполнения exec(), если они также включены в наследуемом наборе процесса.

Возможности, принадлежащие файлу, хранятся в расширенном атрибуте безопасности (см. раздел 16.1) под названием security.capability. Для изменения этого атрибута требуется возможность CAP_SETFCAP.

Таблица 39.1. Операции, которые позволяет выполнять каждая из возможностей

Возможность	**Разрешает процессу**
CAP_AUDIT_CONTROL	(Начиная с Linux 2.6.11.) Включать и выключать запись результатов аудита ядра в журнал; изменять правила фильтрации для аудита; получать состояние аудита и правил фильтрации
CAP_AUDIT_WRITE	(Начиная с Linux 2.6.11.) Сохранять записи в журнал аудита ядра
CAP_CHOWN	Изменять идентификатор пользователя (владельца) файла или назначать файлу идентификатор группы, участником которой процесс не является (chown())
CAP_DAC_OVERRIDE	Обходить проверки прав доступа на чтение, запись и выполнение (аббревиатура DAC означает discretionary access control — избирательное управление доступом); читать содержимое символьных ссылок cwd, exe и root в файле /proc/PID

Продолжение ⇗

Таблица 39.1 *(продолжение)*

Возможность	Разрешает процессу
CAP_DAC_READ_SEARCH	Обходить проверку доступа на чтение файлов, а также проверку на просмотр и выполнение (поиск) каталогов
CAP_FOWNER	В целом игнорировать проверки прав доступа к операциям, которые обычно требуют, чтобы UID файловой системы процесса совпадал с UID файла (chmod(), utime()); устанавливать флаг i-node для произвольных файлов; устанавливать и изменять списки контроля доступа (ACL) для произвольных файлов; игнорировать действие бита закрепления (англ. sticky bit) каталога при удалении файлов (unlink(), rmdir(), rename()); назначать флаг O_NOATIME произвольным файлам при выполнении вызовов open() и fcntl(F_SETFL)
CAP_FSETID	Изменять файл, даже если ядро не выключило биты установки пользовательских и групповых идентификаторов (write(), truncate()); включать бит установки группового идентификатора для файла, чей GID не совпадает с групповым идентификатором файловой системы процесса или дополнительным GID (chmod())
CAP_IPC_LOCK	Переопределять ограничения на блокирование памяти (mlock(), mlockall(), shmctl(SHM_LOCK), shmctl(SHM_UNLOCK)); использовать флаги SHM_HUGETLB и MAP_HUGETLB для операций shmget() и mmap() соответственно
CAP_IPC_OWNER	Обходить проверки прав доступа к операциям с объектами System V IPC
CAP_KILL	Обходить проверки прав доступа к отправке сигналов (kill(), sigqueue())
CAP_LEASE	(Начиная с Linux 2.4.) Выполнять аренду произвольных файлов (fcntl(F_SETLEASE))
CAP_LINUX_IMMUTABLE	Устанавливать флаги append и i-node (последние должны быть постоянными)
CAP_MAC_ADMIN	(Начиная с Linux 2.6.25.) Конфигурировать или вносить изменения состояния обязательного контроля доступа (англ. mandatory access control, или MAC) (реализовано в Linux на уровне модулей безопасности)
CAP_MAC_OVERRIDE	(Начиная с Linux 2.6.25.) Переопределять MAC (реализовано в Linux на уровне модулей безопасности)
CAP_MKNOD	(Начиная с Linux 2.4.) Использовать вызов mknod() для создания устройств
CAP_NET_ADMIN	Выполнять различные сетевые операции (например, задавать параметры привилегированного сокета, включать многоадресную передачу, настраивать сетевые интерфейсы и изменять таблицы маршрутизации)
CAP_NET_BIND_SERVICE	Делать привязку к привилегированным портам сокетов
CAP_NET_BROADCAST	(Не используется.) Транслировать сокеты или принимать многоадресную передачу

Возможность	Разрешает процессу
CAP_NET_RAW	Задействовать сырые и пакетные сокеты
CAP_SETGID	Произвольным образом менять групповые идентификаторы процесса (setgid(), setegid(), setregid(), setresgid(), setfsgid(), setgroups(), initgroups()); применять фиктивные GID при передаче учетных данных через сокет домена UNIX (SCM_CREDENTIALS)
CAP_SETFCAP	(Начиная с Linux 2.6.24.) Устанавливать возможности файлов
CAP_SETPCAP	Выдавать или забирать у любого процесса (в том числе и текущего) возможности, входящие в разрешенный набор текущего процесса (если возможности файлов не поддерживаются); добавлять любые возможности из ограничивающего набора процесса в наследуемый, удалять возможности из ограничивающего набора, изменять флаги безопасности (если возможности файлов поддерживаются)
CAP_SETUID	Произвольным образом менять пользовательский идентификатор процесса (setuid(), seteuid(), setreuid(), setresuid(), setfsuid()); применять фиктивный UID при передаче учетных данных через сокет домена UNIX (SCM_CREDENTIALS)
CAP_SYS_ADMIN	Превышать ограничение /proc/sys/fs/file-max в системных вызовах, которые открывают файлы (например, open(), shm_open(), pipe(), socket(), accept(), exec(), acct(), epoll_create()); выполнять различные операции системного администрирования, включая quotactl() (управление дисковыми квотами), mount() и umount(), swapon() и swapoff(), pivot_root(), sethostname() и setdomainname(); выполнять разные операции с системным журналом (syslog(2)); переопределять ограничение на ресурсы RLIMIT_NPROC (fork()); вызывать lookup_dcookie(); устанавливать расширенные атрибуты trusted и security; выполнять операции IPC_SET и IPC_RMID с произвольными объектами System V IPC; использовать фиктивный UID при передаче учетных данных через сокет домена UNIX (SCM_CREDENTIALS); вызывать ioprio_set() для задания класса планирования IOPRIO_CLASS_RT; применять операцию TIOCCONS в вызове ioctl(); использовать флаг CLONE_NEWNS в вызовах clone() и unshare(); выполнять операции KEYCTL_CHOWN и KEYCTL_SETPERM в вызове keyctl(); администрировать устройство random(4); выполнять различные операции, связанные с устройствами
CAP_SYS_BOOT	Перезагружать систему с помощью вызова reboot(); вызывать kexec_load()
CAP_SYS_CHROOT	Устанавливать корневой каталог процесса с помощью вызова chroot()
CAP_SYS_MODULE	Загружать и выгружать модули ядра (init_module(), delete_module(), create_module())
CAP_SYS_NICE	Повышать значение nice (nice(), setpriority()); изменять значение nice для произвольных процессов (setpriority()); устанавливать для вызывающего процесса политики планирования

Продолжение ⇗

Таблица 39.1 *(продолжение)*

Возможность	Разрешает процессу
	реального времени SCHED_RR и SCHED_FIFO; сбрасывать флаг SCHED_RESET_ON_FORK; устанавливать политику планирования и приоритет для произвольных процессов (sched_setscheduler(), sched_setparam()); устанавливать класс планирования ввода/вывода и приоритет для произвольных процессов (ioprio_set()); привязывать произвольные процессы к определенному ЦПУ (sched_setaffinity()); применять вызов migrate_pages() для миграции произвольных процессов и позволять процессам мигрировать на произвольные узлы; применять вызов move_pages() к произвольным процессам; использовать флаг MPOL_MF_MOVE_ALL в сочетании с вызовами mbind() и move_pages()
CAP_SYS_PACCT	Включать или выключать учет процессов с помощью вызова acct()
CAP_SYS_PTRACE	Трассировать произвольные процессы с помощью вызова ptrace(); получать доступ к файлам /proc/PID/environ произвольных процессов; применять к произвольным процессам операцию get_robust_list()
CAP_SYS_RAWIO	Выполнять операции с портами ввода/вывода, используя вызовы iopl() и ioperm(); получать доступ к файлу /proc/kcore; открывать устройства /dev/mem и /dev/kmem
CAP_SYS_RESOURCE	Использовать зарезервированное пространство файловой системы; выполнять вызовы ioctl() для управления журналированием в ext3; переопределять дисковые квоты; увеличивать жесткое ограничение на ресурсы (setrlimit()); переопределять ограничение на ресурсы RLIMIT_NPROC (fork()); делать ограничение msg_qbytes для очереди сообщений System V выше значения, указанного в файле /proc/sys/kernel/msgmnb; обходить различные ограничения для очереди POSIX-сообщений, заданные в файле /proc/sys/fs/mqueue
CAP_SYS_TIME	Изменять системное время (settimeofday(), stime(), adjtime(), adjtimex()); устанавливать аппаратное время
CAP_SYS_TTY_CONFIG	Разрывать соединение с терминалом или псевдотерминалом, используя вызов vhangup()

39.3.3. Назначение разрешенных и действующих возможностей процесса

Набор *разрешенных возможностей процесса* определяет то, какие именно возможности процесс *может* задействовать. Набор *действующих возможностей процесса* определяет возможности, задействованные процессом на текущий момент — то есть набор, на основе которого ядро решает, обладает ли процесс достаточными привилегиями для выполнения конкретной операции.

Набор разрешенных возможностей устанавливает верхнюю границу действующего набора. Процесс может *расширить* действующий набор, но только если возможность является частью разрешенного набора.

Отношения между действующим и разрешенным наборами аналогичны тому, как соотносятся действующий и сохраненный пользовательские идентификаторы в программах, которые устанавливают UID администратора. Удаление возможности из действующего набора аналогично временному отказу от нулевого пользовательского идентификатора и сохранению его для будущего применения. Удаление возможности из обоих наборов аналогично полному отказу от привилегий администратора, когда действующему и сохраненному идентификаторам присваиваются ненулевые значения.

39.3.4. Для чего нужны разрешенные и действующие возможности файла

Набор *разрешенных возможностей файла* обеспечивает механизм, с помощью которого исполняемый файл может предоставить возможности процессу. Он содержит возможности, которые должны быть включены в разрешенный набор процесса во время выполнения `exec()`.

Набор *действующих возможностей файла* представляет собой один-единственный флаг (бит), который может быть либо включен, либо выключен. Чтобы понять, почему этот набор реализован таким образом, нужно рассмотреть две ситуации, возникающие при запуске программы.

- Программа может *не поддерживать возможности* (будучи написанной в традиционной манере с установкой UID администратора). Такая программа не будет знать о том, что ей нужно включить возможности в действующий набор, чтобы иметь возможность выполнять привилегированные операции. В этом случае в результате вызова `exec()` все разрешенные возможности процесса, которые он только что приобрел, должны быть автоматически включены в действующий набор. Это достигается путем установки действующего бита.
- Программа может *поддерживать возможности*; это означает, что она спроектирована с учетом системы возможностей и будет выполнять соответствующие системные вызовы (будут описаны далее) для включения и выключения различных функций в своем действующем наборе. Чтобы иметь как можно меньше привилегий, такие программы изначально запускаются с пустым действующим набором. Это достигается за счет выключения бита действующих возможностей для исполняемого файла.

39.3.5. Для чего нужны наследуемые возможности процесса и файла

На первый взгляд использование разрешенного и действующего наборов для процессов и файлов может показаться вполне достаточным для регулирования возможностей. Однако в некоторых случаях этого недостаточно. Представьте, к примеру, что процесс, выполняющий `exec()`, хочет сохранить некоторые из его текущих возможностей во время выполнения новой программы. Казалось бы, этого можно достичь, просто сохранив разрешенные возможности процесса на время выполнения `exec()`. Но такой подход не учитывает двух моментов.

- Выполнение `exec()` может потребовать определенных привилегий (например, `CAP_DAC_OVERRIDE`), которые мы не хотим сохранять на протяжении работы программы.
- Представьте, что мы явно отказались от некоторых разрешенных возможностей, которые мы не хотим сохранять на время выполнения программы, но затем вызов `exec()` завершился неудачей. В этом случае программе могут понадобиться некоторые разрешенные возможности, которые уже были (перманентно) выключены.

По этим причинам разрешенные возможности процесса не сохраняются во время работы вызова `exec()`. Для этих целей предусмотрен отдельный набор возможностей — *наследуемый*. Он предоставляет механизм, с помощью которого процесс может сохранить некоторые из своих возможностей при запуске программы.

Наследуемые возможности могут быть включены в разрешенный набор процесса во время выполнения `exec()`. Но перед этим к ним применяется маска *наследуемого набора файла* (путем побитового И), в результате чего определяется итоговая группа возможностей, которая попадает в разрешенный набор.

> Существует более глубокая, философская причина, по которой разрешенные возможности процесса не сохраняются на время выполнения exec(). Идея системы возможностей состоит в том, что все привилегии, выданные процессу, регулируются файлом, который тот выполняет. И хотя наследуемые возможности передаются исполняемой программе, к ним применяется маска наследуемого набора самого файла.

39.3.6. Назначение и просмотр возможностей файла из командной оболочки

Команды `setcap(8)` и `getcap(8)`, входящие в пакет `libcap` (и описанные в разделе 39.7), могут использоваться для работы с наборами возможностей файла. Их использование будет продемонстрировано с помощью короткого примера, в котором запускается стандартная команда `date(1)` (это одно из тех приложений, которые согласно определению из подраздела 39.3.4 не поддерживают систему возможностей). Если запустить ее с привилегиями, она способна изменить системное время. Программа `date` не устанавливает идентификатор администратора, поэтому, чтобы запустить ее в привилегированном режиме, необходимо войти в систему под именем `root`.

Сначала мы выведем текущее системное время и затем попытаемся изменить его от имени обычного пользователя:

```
$ date
Tue Dec 28 15:54:08 CET 2010
$ date -s '2018-02-01 21:39'
date: cannot set date: Operation not permitted
Thu Feb 1 21:39:00 CET 2018
```

Как видите, команда `date` не смогла изменить системное время, однако вывела переданный ей аргумент в стандартном формате.

Теперь получим права администратора, что позволит нам успешно изменить системное время:

```
$ sudo date -s '2018-02-01 21:39'
root's password:
Thu Feb    1 21:39:00 CET 2018
$ date
Thu Feb    1 21:39:02 CET 2018
```

Теперь создадим копию программы `date` и назначим ей необходимую возможность:

```
$ whereis -b date        Находим местоположение исполняемого файла date
date: /bin/date
$ cp /bin/date .
$ sudo setcap "cap_sys_time=pe" date
root's password:
$ getcap date
date = cap_sys_time+ep
```

Команда `setcap`, показанная выше, включает возможность `CAP_SYS_TIME` в разрешенный *(p)* и действующий *(e)* наборы исполняемого файла. Затем с помощью команды `getcap` мы проверяем, какие возможности были назначены файлу (синтаксис представления возможностей, который задействуется командами `getcap` и `setcap`, описан в справочной странице `cap_from_text(3)`, доступной в пакете `libcap`).

Возможности скопированного нами файла позволяют изменять системное время непривилегированным пользователям:

```
$ ./date -s '2010-12-28 15:55'
Tue Dec 28 15:55:00 CET 2010
$ date
Tue Dec 28 15:55:02 CET 2010
```

39.4. Современная реализация системы возможностей

Полная реализация системы возможностей требует выполнения следующих условий.

- При каждой привилегированной операции ядро должно проверять наличие у процесса соответствующей возможности, а не наличие нулевого действующего UID (или UID файловой системы).
- Ядро должно предоставлять системные вызовы, с помощью которых возможности процесса можно просматривать и изменять.
- Ядро должно поддерживать механизм привязки возможностей к исполняемому файлу, чтобы при его запуске процесс мог их получить. Это похоже на принцип работы бита для установки идентификатора пользователя, однако позволяет независимое назначение исполняемому файлу любых возможностей. Кроме того, файловая система должна предоставлять набор программных интерфейсов и команд для задания и просмотра возможностей, привязанных к исполняемому файлу.

Когда-то ядро Linux отвечало только первым двум критериям. Но, начиная с версии 2.6.24, ядро позволяет назначать возможности файлам. Чтобы сделать реализацию возможностей в Linux полноценной, в версиях 2.6.25 и 2.6.26 появилось множество других функций.

При обсуждении возможностей мы в основном сосредоточимся на современной реализации. В разделе 39.10 мы рассмотрим ее отличие от старых версий, которые не поддерживали назначение возможностей файлам. Кроме того, мы будем исходить из того, что система возможностей включена в ядре, хотя этот компонент является необязательным. Позже будут рассмотрены особенности работы в ситуациях, когда возможности файлов отключены (до некоторой степени это похоже на поведение ядра версий 2.6.23 и ниже, в которых возможности файлов не были реализованы).

В следующих разделах мы более подробно рассмотрим систему возможностей Linux.

39.5. Изменение возможностей процесса во время выполнения exec()

Во время выполнения вызова `exec()` ядро задает процессу новые возможности, основываясь на его текущих наборах и на наборах исполняемого файла. Итоговые возможности вычисляются согласно следующим правилам:

```
P'(permitted) = (P(inheritable) & F(inheritable)) |
    (F(permitted) & cap_bset)

P'(effective) = F(effective) ? P'(permitted) : 0

P'(inheritable) = P(inheritable)
```

P и P' обозначают наборы возможностей соответственно до и после выполнения exec(), а F — это набор возможностей файла. Идентификатор cap_bset обозначает ограничивающий набор. Стоит отметить, что вызов exec() оставляет наследуемый набор возможностей процесса без изменений.

39.5.1. Ограничивающий набор возможностей

Ограничивающий набор представляет собой механизм безопасности, с помощью которого задается диапазон возможностей, доступных процессу во время работы exec(). Этот набор используется следующим образом.

- Во время выполнения вызова exec() к ограничивающему набору применяется маска из разрешенных возможностей (с помощью побитового И); так определяются разрешенные возможности, которые будут выданы новой программе. Иными словами, если разрешенные возможности не входят в ограничивающий набор, они не могут быть доступны процессу.
- Ограничивающий набор является надмножеством возможностей, которые могут быть добавлены в наследуемый набор процесса. Это означает, что разрешенная возможность не может стать наследуемой, если она не входит в ограничивающий набор; то есть согласно первому правилу изменения возможностей, описанному выше, эта возможность не будет сохранена в разрешенном наборе, если она входит в наследуемый набор исполняемого файла.

Ограничивающий набор возможностей относится к каждому отдельному процессу, наследуется потомками, создаваемыми с помощью вызова fork(), и сохраняется на протяжении всей работы exec(). Если ядро поддерживает возможности файлов, процесс init (предок всех процессов) запускается с ограничивающим набором, состоящим из всех существующих возможностей.

Если процесс обладает возможностью CAP_SETPCAP, он может (перманентно) удалять элементы ограничивающего набора, используя операцию PR_CAPBSET_DROP для вызова prctl() (удаление возможностей из ограничивающего набора не влияет на разрешенный, действующий и наследуемый наборы процесса). Чтобы определить, входит ли возможность в ограничивающий набор, процесс может воспользоваться операцией PR_CAPBSET_READ для вызова prctl().

> Если быть точным, ограничивающий набор возможностей относится к отдельным потокам. Начиная с версии ядра 2.6.26, этот атрибут выводится в виде поля CapBnd в файле /proc/PID/task/TID/ (доступной только в Linux). Файл /proc/PID/status содержит ограничивающий набор главного потока процесса.

39.5.2. Сохранение традиционной для администратора семантики

Для того чтобы сохранить традиционную семантику, согласно которой администратор имеет неограниченные привилегии во время выполнения файла, любой набор возможностей, связанный с файлом, игнорируется. Вместо этого, чтобы не нарушать алгоритм,

показанный в разделе 39.5, наборы возможностей файла во время выполнения `exec()` номинально определяются следующим образом.

- Если выполняется программа, устанавливающая идентификатор администратора, или если реальный или действующий UID процесса, вызвавшего `exec()`, равен 0, наследуемый и разрешенный наборы имеют одинаковое содержимое.
- Если выполняется программа, устанавливающая идентификатор администратора, или если действующий UID процесса, вызвавшего `exec()`, равен 0, устанавливается действующий бит файла.

Исходя из того, что мы выполняем программу, которая устанавливает идентификатор администратора, эти формальные определения наборов возможностей файла означают, что вычисление новых разрешенных и действующих наборов процессов, описанное в разделе 39.5, сводится к следующим правилам:

```
P'(permitted) = P(inheritable) | cap_bset
P'(effective) = P'(permitted)
```

39.6. Как изменение пользовательского идентификатора влияет на возможности процесса

Чтобы сохранить совместимость с традиционным механизмом перехода между нулевым и ненулевым идентификатором, при изменении UID процесса (с помощью вызова `setuid()` и т. д.) ядро выполняет следующие действия.

1. Если изначально хотя бы один из пользовательских идентификаторов (реальный, действующий или сохраненный) был равен 0 и затем в результате изменения UID все они получили ненулевое значение, тогда содержимое действующего и разрешенного наборов удаляется (то есть перманентно сбрасываются все возможности).
2. Если действующий идентификатор, изначально будучи равным 0, получает ненулевое значение, удаляется содержимое действующего набора возможностей (то есть действующие возможности сбрасываются, но их все еще можно включить из разрешенного набора).
3. Если ненулевому идентификатору пользователя присваивается 0, содержимое разрешенного набора возможностей копируется в действующий набор (то есть задействуются все разрешенные возможности).
4. Если нулевому пользовательскому идентификатору файловой системы присваивается ненулевое значение, из действующего набора удаляются следующие возможности, связанные с файлами: `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_DAC_READ_SEARCH`, `CAP_FOWNER`, `CAP_FSETID`, `CAP_LINUX_IMMUTABLE` (начиная с Linux 2.6.30), `CAP_MAC_OVERRIDE` и `CAP_MKNOD` (начиная Linux 2.6.30). И наоборот — если ненулевое значение пользовательского идентификатора файловой системы меняется на 0, все эти возможности, если они входят в разрешенный набор, становятся действующими. Это делается для того, чтобы сохранить традиционную семантику работы с пользовательскими идентификаторами файловой системы, характерную для Linux.

39.7. Программное изменение возможностей процесса

Для добавления возможностей в наборы и исключения их оттуда процесс может использовать либо системный вызов `capset()`, либо программный интерфейс `libcap`, который будет описан чуть ниже. Процедура изменения возможностей процесса имеет следующие правила.

1. Если действующий набор процесса не содержит возможности CAP_SETPCAP, новый *наследуемый* набор должен быть подмножеством сочетания разрешенного и существующего наследуемого наборов.
2. Новый *наследуемый* набор должен быть подмножеством сочетания ограничивающего и существующего наследуемого наборов.
3. Новый *разрешенный* набор должен быть подмножеством существующего разрешенного набора. Иными словами, процесс не может выдать себе разрешенные возможности, которых у него нет. Это означает, что возможность, удаленная из разрешенного набора, не может быть возвращена.
4. Новый *действующий* набор может содержать только те возможности, которые находятся в новом разрешенном наборе.

Программный интерфейс libcap

Вплоть до этого момента не приводился прототип системного вызова capset() или его собрата capget(), который возвращает возможности процесса. Дело в том, что их использование является нежелательным. Вместо них следует применять функции из библиотеки libcap. Эти функции предоставляют интерфейс, который соответствует бывшему проекту стандарта POSIX 1003.1e и имеет несколько расширений, характерных для Linux:

Для экономии места программный интерфейс libcap не будет описан во всех подробностях. Нужно только отметить, что программы, которые его применяют, обычно выполняют следующие действия.

1. Применяют функцию cap_get_proc(), чтобы извлечь из ядра копию текущих наборов возможностей процесса и поместить ее в структуру, выделенную в пользовательском пространстве (как вариант, мы можем создать новую, пустую структуру, применив вызов cap_init()). В программном интерфейсе libcap тип данных cap_t представляет собой указатель на подобные структуры.
2. Используют функцию cap_set_flag() для изменения структуры, полученной на предыдущем шаге, чтобы включить (CAP_SET) или выключить (CAP_CLEAR) возможности в разрешенном, действующем и наследуемом наборах.
3. Используют функцию cap_set_proc(), чтобы вернуть ядру вышеупомянутую структуру и изменить тем самым возможности процесса.
4. Используют функцию cap_free(), чтобы удалить структуру, которая была создана интерфейсом libcap в самом начале.

> На момент написания этой книги создавалась новая, улучшенная библиотека для работы с возможностями под названием libcap-ng. Подробности можно найти на странице freshmeat.net/projects/libcap-ng.

Пример программы

В листинге 8.2 была представлена программа, которая принимает имя пользователя и пароль и затем выполняет аутентификацию на основе стандартной базы данных паролей. Мы отмечали, что файл с паролями (/etc/shadow) защищен от чтения со стороны обычных пользователей (которые не входят в группу shadow), поэтому для доступа к нему требуются повышенные привилегии. Традиционный механизм получения привилегий подразумевает выполнение этой программы от имени администратора или установки ей нулевого пользовательского идентификатора. Здесь мы рассмотрим измененную версию этого примера, в котором задействуются возможности и программный интерфейс libcap.

Чтобы прочитать файл /etc/shadow от имени обычного пользователя, нам необходимо обойти стандартные проверки прав доступа. Если просмотреть содержимое табл. 39.1,

можно увидеть, что подходящей возможностью в нашем случае является CAP_DAC_READ_SEARCH. Модифицированная версия нашей программы для аутентификации представлена в листинге 39.1. Прямо перед доступом к файлу с паролями она применяет интерфейс libcap, чтобы включить возможность CAP_DAC_READ_SEARCH в свой действующий набор; выполнив чтение, она немедленно сбрасывает эту возможность. Чтобы к этой программе смог обратиться обычный пользователь, мы должны установить эту возможность в разрешенном наборе ее исполняемого файла, как показано в следующем листинге сессии:

```
$ sudo setcap "cap_dac_read_search=p" check_password_caps
root's password:
$ getcap check_password_caps
check_password_caps = cap_dac_read_search+p
$ ./check_password_caps
Username: mtk
Password:
Successfully authenticated: UID=1000
```

Листинг 39.1. Программа для аутентификации пользователей с поддержкой системы возможностей
cap/check_password_caps.c

```
#define _BSD_SOURCE      /* Получаем объявление getpass()
                            из заголовочного файла <unistd.h> */
#define _XOPEN_SOURCE    /* Получаем объявление crypt()
                            из заголовочного файла <unistd.h> */
#include <sys/capability.h>
#include <unistd.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

/* Меняем параметры возможностей в действующем наборе вызывающего процесса */

static int
modifyCap(int capability, int setting)
{
    cap_t caps;
    cap_value_t capList[1];

    /* Получаем текущие возможности вызывающего процесса */
    caps = cap_get_proc();
    if (caps == NULL)
        return -1;

    /* Меняем параметры 'capability' в действующем наборе 'caps'.
       Третий аргумент, 1 — это количество элементов в массиве 'capList'. */
    capList[0] = capability;
    if (cap_set_flag(caps, CAP_EFFECTIVE, 1, capList, setting) == -1) {
        cap_free(caps);
        return -1;
    }

    /* Возвращаем ядру измененные наборы возможностей, чтобы они
       вступили в силу для вызывающего процесса */
    if (cap_set_proc(caps) == -1) {
        cap_free(caps);
        return -1;
    }
```

```
    /* Удаляем структуру, выделенную интерфейсом libcap */
    if (cap_free(caps) == -1)
        return -1;

    return 0;
}

static int      /* Включаем возможность в действующий набор вызывающего процесса */
raiseCap(int capability)
{
    return modifyCap(capability, CAP_SET);
}

/* Аналогичная функция dropCap() (не нужна в этой программе)
   может быть определена как modifyCap(capability, CAP_CLEAR); */

static int        /* Удаляем все возможности из всех наборов */
dropAllCaps(void)
{
    cap_t empty;
    int s;

    empty = cap_init();
    if (empty == NULL)
        return -1;
    s = cap_set_proc(empty);
    if (cap_free(empty) == -1)
        return -1;

    return s;
}

int
main(int argc, char *argv[])
{
    char *username, *password, *encrypted, *p;
    struct passwd *pwd;
    struct spwd *spwd;
    Boolean authOk;
    size_t len;
    long lnmax;

    lnmax = sysconf(_SC_LOGIN_NAME_MAX);
    if (lnmax == -1)        /* Если невозможно определить ограничение, */
        lnmax = 256;        /* берем значение наугад */

    username = malloc(lnmax);
    if (username == NULL)
        errExit("malloc");

    printf("Username: ");
    fflush(stdout);
    if (fgets(username, lnmax, stdin) == NULL)
        exit(EXIT_FAILURE);      /* Выходим при обнаружении символа EOF */
    len = strlen(username);
    if (username[len - 1] == '\n')
        username[len - 1] = '\0';    /* Удаляем символ '\n' в конце */

    pwd = getpwnam(username);
    if (pwd == NULL)
        fatal("couldn't get password record");
```

```
    /* Включаем возможность CAP_DAC_READ_SEARCH только на период, когда она нам нужна */
    if (raiseCap(CAP_DAC_READ_SEARCH) == -1)
        fatal("raiseCap() failed");

    spwd = getspnam(username);
    if (spwd == NULL && errno == EACCES)
        fatal("no permission to read shadow password file");

  /* На этом этапе нам больше не нужны никакие возможности, поэтому очищаем все наборы */
    if (dropAllCaps() == -1)
        fatal("dropAllCaps() failed");

    if (spwd != NULL)               /* Если запись в файле с паролями существует, */
        pwd->pw_passwd = spwd->sp_pwdp;    /* используем пароль */
    password = getpass("Password: ");

    /* Шифруем пароль и сразу же удаляем незашифрованную версию */
    encrypted = crypt(password, pwd->pw_passwd);
    for (p = password; *p != '\0'; )
        *p++ = '\0';

    if (encrypted == NULL)
        errExit("crypt");

    authOk = strcmp(encrypted, pwd->pw_passwd) == 0;
    if (!authOk) {
        printf("Incorrect password\n");
        exit(EXIT_FAILURE);
    }

    printf("Successfully authenticated: UID=%ld\n", (long) pwd->pw_uid);

    /* Теперь выполняем аутентификацию... */

    exit(EXIT_SUCCESS);
}
```

cap/check_password_caps.c

39.8. Создание среды, в которой возможности являются единственным механизмом повышения привилегий

Ранее были описаны различные механизмы, благодаря которым привилегии пользователя с идентификатором 0 (администратор) зависят от системы возможностей.

- Когда процесс с хотя бы одним нулевым пользовательским идентификатором присваивает всем своим UID ненулевое значение, его разрешенный и действующий наборы возможностей сбрасываются (см. раздел 39.6).
- Если поменять действующий идентификатор пользователя, который изначально был равен 0, на ненулевое значение, процесс потеряет все свои действующие возможности. При обратном действии разрешенные возможности копируются в действующий набор. Похожая процедура выполняется для подмножества возможностей, когда UID файловой системы переключается между 0 и ненулевым значением (см. раздел 39.6).

- Если процесс с реальным или действующим идентификатором администратора выполняет программу, или если программа, устанавливающая администраторский идентификатор, выполняется любым процессом, тогда наследуемый и разрешенный наборы файла будут иметь условно одинаковое содержимое. Если действующий UID процесса равен 0, или если он выполняет программу, устанавливающую идентификатор администратора, действующий бит файла будет условно равен 1 (см. подраздел 39.5.2). В обычных сценариях (то есть когда реальный и действующий идентификаторы пользователя равны 0 или когда выполняется программа, устанавливающая администраторский идентификатор) это означает, что процессы получают все возможности из своих разрешенных и действующих наборов.

В среде, которая основывается исключительно на системе возможностей, ядру не нужно было бы выполнять все эти процедуры, чтобы учесть особые права администратора. Программ, устанавливающих администраторский идентификатор, просто не существовало бы, а возможности файла использовались бы для выдачи минимально возможных привилегий, которые требуются приложению.

Существующие приложения не спроектированы для задействования возможностей файловой инфраструктуры, поэтому ядру приходится обеспечивать традиционный механизм работы с процессами, чей пользовательский идентификатор равен 0. Тем не менее у нас может возникнуть необходимость запустить программу в среде, основанной исключительно на системе возможностей, в которой администратор не требует к себе никакого особого обращения, описанного выше. Начиная с версии 2.6.26, Linux предоставляет механизм *флагов безопасности (securebits)*, управляющий набором индивидуальных флагов каждого процесса, которые включают или выключают каждый из трех уровней работы с пользователем `root` (если быть точным, флаги безопасности на самом деле относятся к каждому отдельному потоку). Стоит отметить, что для этого в ядре должна быть включена система возможностей.

Механизм `securebits` управляет флагами, перечисленными в табл. 39.2. Эти флаги связаны с *основными* (попарно) и *блокированными* (по одному) флагами. Каждый из основных флагов отвечает за один из уровней работы с пользователем `root`, описанных выше. Установка блокированного флага является необратимой и предотвращает дальнейшие изменения соответствующего основного флага — эту операцию нельзя отменить.

Параметры флагов безопасности наследуются дочерним процессом, созданным с помощью вызова `fork()`. Все они сохраняются на время работы `exec()`; исключение составляет параметр `SECBIT_KEEP_CAPS`, который в целях совместимости выключается с помощью флага `PR_SET_KEEPCAPS`, описанного ниже. Чтобы извлечь флаги безопасности, процесс может воспользоваться операцией `PR_GET_SECUREBITS` для вызова `prctl()`. Если процесс обладает возможностью `CAP_SETPCAP`, он может изменять флаги безопасности с помощью операции `PR_SET_SECUREBITS` для того же вызова. Приложение, которое полностью основано на системе возможностей, способно перманентно отключить особый уровень работы с привилегированным процессом и всеми его наследниками, выполнив такой вызов:

```
if (prctl(PR_SET_SECUREBITS,
          /* SECBIT_KEEP_CAPS отключен */
          SECBIT_NO_SETUID_FIXUP | SECBIT_NO_SETUID_FIXUP_LOCKED |
          SECBIT_NOROOT | SECBIT_NOROOT_LOCKED)
      == -1)
    errExit("prctl");
```

После этого процесс может получить возможности, только запустив исполняемый файл, который ими обладает.

Таблица 39.2. Флаги безопасности

Флаг	Значение, если установлен
SECBIT_KEEP_CAPS	Предотвращает сбрасывание возможностей, когда процесс с одним или несколькими нулевыми UID присваивает идентификатору пользователя ненулевое значение. Действует, только если не установлен флаг SECBIT_NO_SETUID_FIXUP. Сбрасывается при вызове exec()
SECBIT_NO_SETUID_FIXUP	Предотвращает изменение возможностей, когда действующий UID или UID файловой системы переключаются между 0 и ненулевыми значениями
SECBIT_NOROOT	Не выдает возможности процессу с реальным или действующим пользовательским идентификатором, равным 0, который выполняет exec() или запускает программу, устанавливающую UID администратора (если только возможности не были установлены для самого исполняемого файла)
SECBIT_KEEP_CAPS_LOCKED	Блокирует флаг SECBIT_KEEP_CAPS
SECBIT_NO_SETUID_FIXUP_LOCKED	Блокирует флаг SECBIT_NO_SETUID_FIXUP
SECBIT_NOROOT_LOCKED	Блокирует флаг SECBIT_NOROOT

Флаг SECBIT_KEEP_CAPS и операция PR_SET_KEEPCAPS для вызова prctl()

Флаг `SECBIT_KEEP_CAPS` не дает сбросить возможности, когда процесс с одним или несколькими нулевыми UID присваивает идентификатору пользователя ненулевое значение. Грубо говоря, это половина той работы, которую выполняет флаг `SECBIT_NO_SETUID_FIXUP` (как отмечено в табл. 39.2, `SECBIT_KEEP_CAPS` действует, только если не установлен флаг `SECBIT_NO_SETUID_FIXUP`). Этот флаг предоставляет параметры `securebits`, отражающие старшую операцию `PR_SET_KEEPCAPS` для вызова `prctl()`, которая отвечает за тот же атрибут (единственное отличие между этими двумя механизмами заключается в том, что для использования операции `PR_SET_KEEPCAPS` процессу не нужен флаг `CAP_SETPCAP`).

> Ранее мы уже отмечали, что во время выполнения exec() сохраняются все флаги безопасности, кроме SECBIT_KEEP_CAPS. Этот флаг имеет параметры, противоположные другим флагам securebits, благодаря чему он согласуется с обработкой атрибутов, установленных операцией PR_SET_KEEPCAPS.

Операция `PR_SET_KEEPCAPS` предназначена для использования программами, устанавливающими идентификатор администратора, которые работают на старых ядрах без поддержки системы возможностей. Такие программы могут улучшить свою безопасность, сбрасывая и приобретая возможности по мере необходимости (см. раздел 39.10).

Но даже если программа, устанавливающая идентификатор администратора, сбросит все возможности, кроме тех, что ей требуются, она все равно будет обладать двумя важными привилегиями: доступом к файлам, принадлежащим пользователю *root*, и способностью возвращать себе возможности путем выполнения программы (см. подраздел 39.5.2). Единственным способом сбросить эти привилегии навсегда является присваивание всем UID процесса ненулевых значений. Однако это обычно приводит к удалению содержимого разрешенного и действующего наборов (см. раздел 39.6, где приводится четыре замечания относительно того, как изменение пользовательского идентификатора влияет на возможности). Это сводит на нет нашу цель — полностью отказаться от нулевого UID, но сохранить при этом некоторые возможности. Чтобы обойти эту проблему, можно

применить операцию PR_SET_KEEPCAPS для вызова prctl(); она устанавливает атрибут процесса, который предотвращает удаление содержимого разрешенного набора, когда все пользовательские идентификаторы получают ненулевые значения (набор действующих возможностей всегда очищается, вне зависимости от установки данного атрибута).

39.9. Определение возможностей, которые требуются программе

Представьте, что у вас есть программа, написанная без поддержки системы возможностей, которая распространяется исключительно в двоичном виде, или же ее исходный код настолько объемный, что мы не в состоянии понять, какие именно возможности могут понадобиться для ее работы. Если эта программа нуждается в привилегиях, но не должна устанавливать идентификатор администратора, то как определить, какие возможности следует установить ее исполняемому файлу с помощью команды setcap(8)? Ответить на этот вопрос можно двумя способами.

- Мы можем воспользоваться утилитой strace(1), чтобы узнать, какие системные вызовы завершились ошибкой EPERM, которая указывает на отсутствие необходимых возможностей. Сверившись со справочной страницей системного вызова или с исходными текстами ядра, мы можем вычислить ту возможность, которая нам нужна. Данный подход неидеален, поскольку иногда у ошибки EPERM могут быть другие причины, не связанные с возможностями, необходимыми программе. Кроме того, иногда, если для выполнения определенной операции у программы не хватает привилегий, она может откорректировать свое поведение. Иногда сложно отличить такие «ложные срабатывания» от случаев, когда привилегии действительно нужны для работы программы.
- Можно попросить ядро выполнить проверку возможностей и изучить полученный ответ. Пример того, как это делается, приведен в статье [Hallyn, 2007], написанной одним из разработчиков системы возможностей файлов. Для каждого запроса проверки возможности в журнальных записях, прилагаемых к статье, указывается функция ядра, которая была вызвана, возможность, которая запрашивалась, и название программы, выполнившей запрос. Этот подход требует больше усилий, чем использование утилиты strace(1), однако он позволяет более точно определить возможности, которые требуются программе.

39.10. Резюме

Система возможностей в Linux разбивает привилегированные операции на отдельные категории и позволяет выдавать процессу только определенную их часть. Это является шагом вперед по сравнению с традиционным механизмом управления привилегиями, при котором процесс либо получает доступ ко всем операциям сразу (если его UID равен 0), либо полностью его лишается (если пользовательский идентификатор не равен 0). Начиная с версии 2.6.24, ядро Linux поддерживает назначение возможностей файлам, благодаря чему процесс, выполняющий программу, может получить определенный набор возможностей.

39.11. Упражнение

39.1. Измените программу из листинга 35.2 (sched_set.c) так, чтобы она могла задействовать возможности файла и выполняться от имени непривилегированного пользователя.

40 Учет входа в систему

Механизм учета входа в систему занимается записью сведений о том, какие пользователи сейчас находятся в системе, а также о предыдущих входах и выходах. В данной главе мы рассмотрим файлы, хранящие эти данные, и библиотечные функции, позволяющие обновлять и получать их содержимое. Также будет пошагово описана процедура обновления этих файлов, которую должно выполнять приложение, отвечающее за вход пользователей в систему (и выход из нее).

40.1. Краткий обзор файлов utmp и wtmp

В UNIX-системах имеются два файла, содержащих информацию о входе пользователей в систему и их выходе из нее.

- Файл `utmp` хранит записи о пользователях, находящихся в системе в текущий момент времени (а также некую дополнительную информацию, о которой мы поговорим чуть позже). При каждом входе в систему в этот файл добавляется новая запись. Одно из его полей, `ut_user`, содержит имя пользователя. Во время выхода из системы данное поле очищается. Такие программы, как `who(1)`, используют содержимое файла `utmp` для вывода списка пользователей, находящихся в системе.
- Файл `wtmp` представляет собой учетный журнал с информацией обо всех входах в систему и выходах из нее (плюс ряд дополнительных сведений, речь о которых пойдет чуть позже). При каждом входе в систему в этот файл добавляется та же запись, что и в `utmp`. При выходе добавляется еще одна запись с тем же содержимым, но с обнуленным полем `ut_user`. Данные, хранящиеся в файле `wtmp`, можно просматривать и фильтровать с помощью команды `last(1)`.

В Linux файлы `utmp` и `wtmp` имеют пути `/var/run/utmp` и, соответственно, `/var/log/wtmp`. В целом, программу не должно заботить ее местоположение, так как информация о нем заложена в библиотеку glibc. Если вам все же нужно получить эти пути, то лучше не указывать их напрямую в своем коде, а воспользоваться константами `_PATH_UTMP` и `_PATH_WTMP`, объявленными в заголовочном файле `<paths.h>` (и в `<utmpx.h>`).

> Стандарт SUSv3 не предусматривает никаких символьных обозначений для путей к файлам utmp и wtmp. В Linux и BSD используются имена _PATH_UTMP и _PATH_WTMP, однако во многих других реализациях UNIX вместо этого определены константы UTMP_FILE и WTMP_FILE. Кроме того, в Linux указанные значения хранятся только в файле <utmp.h>, но не в <utmpx.h> или <paths.h>.

40.2. Программный интерфейс utmpx

Файлы `utmp` и `wtmp` присутствуют в системе UNIX с давних пор. За это время они постепенно эволюционировали и претерпели различные изменения на разных платформах — особенно заметна разница между BSD и System V. В System V Release 4 значительно

расширился программный интерфейс, в результате чего появилась новая (параллельная) структура utmpx, а также связанные с ней файлы utmpx и wtmpx. Функции, предназначенные для работы с ними, тоже имеют в своих названиях букву x, равно как и связанные с ними заголовочные файлы. Подобные нестандартные расширения этого программного интерфейса появились и во многих других реализациях UNIX.

В данной главе будет описана Linux-версия интерфейса utmpx, которая представляет собой смесь реализаций, входящих в состав систем BSD и System V. В отличие от последней Linux не создает параллельных файлов utmpx и wtmpx; вместо этого вся необходимая информация записывается в файлы utmp и wtmp. Но, чтобы сохранить совместимость с другими реализациями, для доступа к этим файлам Linux предоставляет сразу два программных интерфейса — традиционный utmp и utmpx, пришедший из System V. Оба они возвращают одни и те же данные (одно из немногих различий между этими интерфейсами заключается в том, что utmp содержит несколько реентерабельных функций). Однако здесь будет описан только интерфейс utmpx, так как он входит в стандарт SUSv3 и благодаря своей портируемости на другие системы является более предпочтительным.

Спецификация SUSv3 не покрывает все аспекты программного интерфейса utmpx (например, в ней не указано местоположение файлов utmp и wtmp). Содержимое файлов учета входа в систему зависит от конкретной реализации, а в ряде систем utmpx может поддерживать дополнительные функции, не предусмотренные стандартом SUSv3.

В главе 17 книги [Frisch, 2002] перечислены некоторые отличия в местоположении и применении файлов wtmp и utmp в разных реализациях UNIX. Там же можно найти описание работы с командой ac(1), позволяющей выводить информацию о входе в систему, хранящуюся в файле wtmp.

40.3. Структура utmpx

Каждая запись в файлах utmp и wtmp представляет собой структуру utmpx, объявленную в заголовочном файле <utmpx.h> (листинг 40.1).

В спецификацию структуры utmpx из состава SUSv3 не входят поля ut_host, ut_exit, ut_session и ut_addr_v6. Первые два из них присутствуют в большинстве других реализаций; поле ut_session тоже доступно в некоторых системах; а поле ut_addr_v6 поддерживается только в Linux. Стандарт SUSv3 описывает поля ut_line и ut_user, но не уточняет их длину.

Тип данных int32_t, с помощью которого в структуре utmpx определено поле ut_addr_v6, представляет собой 32-разрядное целое число.

Листинг 40.1. Определение структуры utmpx

```
#define _GNU_SOURCE
/* Без макроса _GNU_SOURCE эти два поля начинаются с "__" */

struct exit_status {
    short e_termination;    /* Код принудительного завершения процесса (сигнал) */
    short e_exit;           /* Код завершения процесса */
};

#define __UT_LINESIZE    32
#define __UT_NAMESIZE    32
#define __UT_HOSTSIZE    256

struct utmpx {
    short ut_type;                /* Тип записи */
```

```
    pid_t ut_pid;                 /* Идентификатор процесса, отвечающего
                                     за вход в систему */
    char ut_line[__UT_LINESIZE];  /* Имя терминального устройства */
    char ut_id[4];                /* Суффикс, состоящий из имени терминала
                                     или поля ID в выводе команды inittab(5) */
    char ut_user[__UT_NAMESIZE];  /* Имя пользователя */
    char ut_host[__UT_HOSTSIZE];  /* Имя компьютера для удаленного входа или версия
                                     ядра для сообщений уровня выполнения */
    struct exit_status ut_exit;   /* Код завершения процесса, помеченного DEAD_PROCESS
                                     (В Linux init(8) оставляет его пустым) */
    long ut_session;              /* Идентификатор сессии */
    struct timeval ut_tv;         /* Время создания записи */
    int32_t ut_addr_v6[4];        /* IP-адрес удаленного компьютера (в случае с IPv4
                                     используется только ut_addr_v6[0], а остальные
                                     элементы заполняются нулями) */
    char __unused[20];        /* Зарезервировано для будущего применения */
};
```

Каждое из стоковых полей в структуре `utmpx` завершается значением `NULL` (если только оно не занимает весь отведенный ему массив).

Что касается процессов, выполняющих вход в систему, содержимое полей `ut_line` и `ut_id` основывается на имени терминального устройства; поле `ut_line` хранит полное имя файла, а `ut_id` — только суффикс (то, что следует за `tty`, `pts` или `pty`; последние два используются для псевдотерминалов в стиле соответственно System V и BSD). Например, в случае с терминалом `/dev/tty2` поле `ut_line` будет равно `tty2`, а поле `ut_id` будет содержать `2`.

В оконной среде некоторые эмуляторы терминалов используют поле `ut_session` для записи идентификатора сессии своего окна (подробности об идентификаторах сессии см. в разделе 34.3).

Поле `ut_type` представляет собой целое число, описывающее тип записи, добавляемой в файл. Значения, которые ему можно присваивать, представлены в виде следующих констант (их числовое представление указано в скобках):

- `EMPTY` (`0`) — запись не содержит корректной учетной информации;
- `RUN_LVL` (`1`) — запись указывает на то, что во время загрузки или выключения системы ее уровень выполнения изменился (информацию о данном уровне можно найти на странице `init(8)` руководства). Чтобы получить определение этой константы из заголовочного файла `<utmpx.h>`, следует определить макрос проверки возможностей `_GNU_SOURCE`;
- `OOT_TIME` (`2`) — поле `ut_tv` данной записи содержит время загрузки системы. Обычно автором записей типа `RUN_LVL` и `BOOT_TIME` является процесс `init`. Он добавляет их в оба файла — `utmp` и `wtmp`;
- `NEW_TIME` (`3`) — поле `ut_tv` данной записи содержит обновленное время после изменения системных часов;
- `OLD_TIME` (`4`) — поле `ut_tv` данной записи содержит старое время, которое было актуально до изменения системных часов. Записи типа `OLD_TIME` и `NEW_TIME` записываются в файлы `utmp` и `wtmp` демоном NTP (или аналогичным), когда тот меняет системное время;
- `INIT_PROCESS` (`5`) — запись относится к потомку процесса `init` — например, `getty`. Подробности можно найти на странице `inittab(5)` руководства;
- `LOGIN_PROCESS` (`6`) — запись относится к лидеру сессии входа в систему — например, к процессу `login(1)`;
- `USER_PROCESS` (`7`) — запись относится к пользовательскому процессу (обычно это сессия входа в систему); имя пользователя попадает в поле `ut_user`. Сессия может быть запущена командой `login(1)` или каким-нибудь другим приложением, предоставляющим удаленный вход в систему (например, `ftp` или `ssh`);
- `DEAD_PROCESS` (`8`) — запись относится к процессу, выполнившему выход из системы.

Указываются числовые значения этих констант, поскольку различные приложения написаны из расчета на тот порядок, в котором они указаны выше. Например, в исходном коде программы agetty можно найти проверки следующего вида:

```
utp->ut_type >= INIT_PROCESS && utp->ut_type <= DEAD_PROCESS
```

Записи типа INIT_PROCESS обычно являются результатом вызова getty(8) (или похожей программы, такой как agetty(8) или mingetty(8)). Во время загрузки системы процесс init создает по одному потомку для каждой строчки в виртуальной консоли, и все они запускают программу getty. Та, в свою очередь, открывает терминал, запрашивает у пользователя его имя и затем выполняет команду login(1). Аутентифицировав пользователя и выполнив ряд других действий, команда login создает дочерний процесс, запускающий командную оболочку. Жизненный цикл подобной сессии входа в систему представлен четырьмя записями, которые добавляются в файл wtmp в следующем порядке:

- запись INIT_PROCESS, добавленная процессом init;
- запись LOGIN_PROCESS, добавленная процессом getty;
- запись USER_PROCESS, добавленная процессом login;
- запись DEAD_PROCESS, добавленная процессом init, когда тот обнаружил завершение дочернего процесса команды login (при выходе пользователя из системы).

Больше подробностей о поведении программ getty и login во время входа в систему можно найти в главе 9 книги [Stevens & Rago, 2005].

Некоторые версии init создают процесс getty до обновления файла wtmp. Как следствие, оба процесса соперничают друг с другом за возможность обновить этот файл, и в итоге порядок добавления записей INIT_PROCESS и LOGIN_PROCESS может быть противоположным тому, что описан выше.

40.4. Извлечение информации из файлов utmp и wtmp

Функции, описанные в данном разделе, предназначены для извлечения записей формата utmpx. По умолчанию в качестве источника информации выступает файл utmp, но это можно изменить с помощью функции utmpxname() (рассмотренной ниже).

Во время извлечения записей данные функции оперируют понятием *текущая позиция*. Она обновляется при каждом вызове.

Функция setutxent() позволяет перейти в начало файла utmp.

```
#include <utmpx.h>

void setutxent(void);
```

Обычно вызов setutxent() следует выполнять перед использованием любой функции вида getutx*() (см. ниже). Это позволяет избежать недопонимания в случаях, когда функции были задействованы каким-нибудь сторонним кодом, который мы выполнили ранее. В зависимости от задачи функция setutxent() может понадобиться и на последующих этапах работы программы.

Функции setutxent() и getutx*() открывают файл utmp, только если он не открыт кем-то другим. Закончив с ним работать, можно его закрыть, воспользовавшись функцией endutxent().

```
#include <utmpx.h>

void endutxent(void);
```

Функции `getutxent()`, `getutxid()` и `getutxline()` считывают запись из файла `utmp` и возвращают указатель на структуру `utmpx` (выделенный статически).

```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *ut);
struct utmpx *getutxline(const struct utmpx *ut);
```

Все три возвращают либо указатель на статически выделенную структуру `utmpx`, либо `NULL`, если не нашлось подходящей записи или был достигнут конец файла

Функция `getutxent()` извлекает из файла `utmp` следующую по порядку запись. Функции `getutxid()` и `getutxline()` выполняют поиск записей, соответствующих заданным критериям, начиная с текущей позиции (критерии задаются в виде структуры `utmpx`, на которую указывает аргумент `ut`).

Функция `getutxid()` ищет запись, основываясь на полях `ut_type` и `ut_id` аргумента `ut`:

- если поле `ut_type` равно `RUN_LVL`, `BOOT_TIME`, `NEW_TIME` или `OLD_TIME`, то `getutxid()` возвращает следующую запись, тип которой совпадает с заданным значением (записи этих типов не относятся к входу в систему). Данное обстоятельство позволяет искать сведения об изменениях системного времени и уровня выполнения;
- если поле `ut_type` содержит любое другое допустимое значение (`INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` или `DEAD_PROCESS`), то `getutxent()` возвращает следующую запись, чей тип совпадает с любым из указанных значений и поле `ut_id` которой равно одноименному полю в аргументе `ut`. Это позволяет перебирать файл в поиске записей, относящихся к определенному терминалу.

Функция ищет следующую запись, у которой либо поле `ut_type` равно `LOGIN_PROCESS` или `USER_PROCESS`, либо поле `ut_line` совпадает с тем, что указано в аргументе `ut`. Это помогает находить записи, связанные со входом в систему.

Если поиск не дает результатов (то есть если достигается конец файла, а подходящих записей не найдено), то функции `getutxid()` и `getutxline()` возвращают `NULL`.

В ряде UNIX-систем функции `getutxline()` и `getutxid()` задействуют статическое пространство, в которое записывается итоговая структура `utmpx`, в качестве кэша. Если запись, помещенная в этот кэш предыдущим вызовом `getutx*()`, соответствует критериям, заданным в аргументе `ut`, то чтение файла не выполняется; вызов просто еще раз возвращает ту же запись (стандарт SUSv3 допускает такое поведение). Следовательно, чтобы не возвращать одну и ту же запись много раз подряд, когда функции `getutxline()` и `getutxid()` вызываются внутри цикла, нужно обнулять эту статическую структуру данных, используя код следующего вида:

```
struct utmpx *res = NULL;
/* Остальной код опущен */

if (res != NULL)      /* Если значение 'res' было установлено предыдущим вызовом */
    memset(res, 0, sizeof(struct utmpx));
res = getutxline(&ut);
```

Библиотека glibc не выполняет подобного кэширования, но для обеспечения портируемости все равно следует выполнять данное обнуление.

Функции getutx*() возвращают указатель на статически выделенную структуру, поэтому не могут быть реентерабельными. Библиотека GNU языка C предоставляет такие версии традиционных функций для работы со структурой utmp (getutent_r(), getutid_r() и getutline_r()), но подобных аналогов для utmpx не существует (реентерабельные функции не предусмотрены в стандарте SUSv3).

Функции `getutx*()` по умолчанию работают со стандартным файлом `utmp`. Чтобы использовать другой файл (например, `wtmp`), следует вызвать функцию `utmpxname()` и указать нужный путь.

```
#define _GNU_SOURCE
#include <utmpx.h>

int utmpxname(const char *file);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `utmpxname()` всего лишь записывает копию переданного ей пути. Она не занимается открытием файлов, но закрывает открытые предыдущими вызовами. Это значит, что `utmpxname()` не возвращает ошибки, если указать неверный путь. Ошибка произойдет позже, когда одна из функций вида `getutx*()` не сможет открыть заданный файл (то есть будет возвращено значение `NULL`, а `errno` будет присвоено значение `ENOENT`).

Функция utmpxname() не входит в стандарт SUSv3, но большинство реализаций UNIX предоставляют либо ее, либо один из аналогов utmpname().

Пример программы

Программа, представленная в листинге 40.2, выводит содержимое файла формата `utmpx` с помощью некоторых из рассмотренных выше функций.

Листинг 40.2. Вывод содержимого файла формата utmpx

loginacct/dump_utmpx.c

```
#define _GNU_SOURCE
#include <time.h>
#include <utmpx.h>
#include <paths.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct utmpx *ut;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [utmp-pathname]\n", argv[0]);
    if (argc > 1)        /* Если указан другой файл, используем его */
        if (utmpxname(argv[1]) == -1)
            errExit("utmpxname");

    setutxent();
```

```
    printf("user    type    PID    line    id        host    date/time\n");

    while ((ut = getutxent()) != NULL) {  /* Последовательный поиск до конца файла */
        printf("%-8s ", ut->ut_user);
        printf("%-9.9s ",
               (ut->ut_type == EMPTY) ?         "EMPTY" :
               (ut->ut_type == RUN_LVL) ?       "RUN_LVL" :
               (ut->ut_type == BOOT_TIME) ?     "BOOT_TIME" :
               (ut->ut_type == NEW_TIME) ?      "NEW_TIME" :
               (ut->ut_type == OLD_TIME) ?      "OLD_TIME" :
               (ut->ut_type == INIT_PROCESS) ?  "INIT_PR" :
               (ut->ut_type == LOGIN_PROCESS) ? "LOGIN_PR" :
               (ut->ut_type == USER_PROCESS) ?  "USER_PR" :
               (ut->ut_type == DEAD_PROCESS) ?  "DEAD_PR" : "???");
        printf("%5ld %-6.6s %-3.5s %-9.9s ", (long) ut->ut_pid,
              ut->ut_line, ut->ut_id, ut->ut_host);
        printf("%s", ctime((time_t *) &(ut->ut_tv.tv_sec)));
    }

    endutxent();
    exit(EXIT_SUCCESS);
}
```

loginacct/dump_utmpx.c

Листинг сессии, приведенный ниже, демонстрирует результаты работы этой программы на примере файла `/var/run/utmp` (используется по умолчанию, если не вызвать функцию `utmpxname()`):

```
$ ./dump_utmpx
user     type         PID    line    id  host    date/time
LOGIN    LOGIN_PR    1761    tty1    1           Sat Oct 23 09:29:37 2010
LOGIN    LOGIN_PR    1762    tty2    2           Sat Oct 23 09:29:37 2010
lynley   USER_PR    10482    tty3    3           Sat Oct 23 10:19:43 2010
david    USER_PR     9664    tty4    4           Sat Oct 23 10:07:50 2010
liz      USER_PR     1985    tty5    5           Sat Oct 23 10:50:12 2010
mtk      USER_PR    10111    pts/0   /0          Sat Oct 23 09:30:57 2010
```

Здесь не приводится большая часть вывода, чтобы сделать его более лаконичным. Строчки со значениями с `tty1` по `tty5` относятся ко входу в систему из виртуальных консолей (`/dev/tty[1-6]`). Последняя строка описывает сессию `xterm` в псевдотерминале.

Вывод содержимого файла `/var/log/wtmp`, приведенный ниже, показывает, что при входе в систему и выходе из нее в данный файл добавляется две записи (все остальные строки, выведенные программой, выброшены). Выполнив последовательный поиск по файлу `wtmp` (с помощью функции `getutxline()`), можно отфильтровать эти записи по полю `ut_line`.

```
$ ./dump_utmpx /var/log/wtmp
user     type        PID   line  id  host         date/time
lynley   USER_PR   10482   tty3  3               Sat Oct 23 10:19:43 2010
         DEAD_PR   10482   tty3  3   2.4.20-4G   Sat Oct 23 10:32:54 2010
```

40.5. Получение имени текущего пользователя: getlogin()

Функция `getlogin()` возвращает имя пользователя, который работает в контролирующем терминале вызывающего процесса. Она использует информацию, хранящуюся в файле `utmp`.

```
#include <unistd.h>

char *getlogin(void);
```

Возвращает указатель на строку с именем пользователя или NULL при ошибке

Чтобы получить имя терминала, связанного со стандартным вводом вызывающего процесса, функция `getlogin()` выполняет вызов `ttyname()` (см. раздел 58.10). Затем она ищет в файле `utmp` запись, значение `ut_line` которой совпадает с этим именем. Если такая запись нашлась, то `getlogin()` возвращает ее поле `ut_user`.

Если запись не найдена, то `getlogin()` возвращает `NULL` и сигнализирует об ошибке с помощью значения `errno`. Это может произойти в ситуации, когда стандартный ввод процесса не связан с терминалом (ошибка `ENOTTY`) — к примеру, если процесс является демоном. Еще одна потенциальная причина возникновения ошибки — сессия терминала не записывается в файл `utmp`; так, например, поступают некоторые программные эмуляторы терминала.

Но даже в случаях, когда одному и тому же идентификатору пользователя в файле `/etc/passwd` соответствует сразу несколько имен (что довольно необычно), функция `getlogin()` способна вернуть то самое имя, которое применялось при входе в систему в текущем терминале, так как она берет данные из файла `utmp`. Для сравнения, вызов `getpwuid(getuid())` всегда возвращает первую подходящую запись, найденную в файле `/etc/passwd`, вне зависимости от того, какое имя использовалось при входе в систему.

Реентерабельная версия функции getlogin() под названием getlogin_r() предусмотрена стандартом SUSv3 и входит в библиотеку glibc.

Для поиска имени, под которым пользователь вошел в систему, можно также применять переменную среды LOGNAME. Однако значение данной переменной может быть изменено самим пользователем; это значит, что оно не подходит для безопасной идентификации.

40.6. Запись в файлы utmp и wtmp данных о пребывании в системе

При написании приложения, позволяющего входить в систему (такого как `login` или `sshd`), файлы `utmp` и `wtmp` необходимо обновлять следующим образом.

- Когда пользователь входит систему, в файл `utmp` нужно добавить запись, сигнализирующую об этом факте. Приложение должно проверить, существует ли такая запись для текущего терминала. В случае положительного ответа ее необходимо перезаписать; в противном случае в файл добавляется новая запись. Часто для корректного выполнения данной процедуры достаточно сделать вызов `pututxline()` (пример его использования показан в листинге 40.3). Итоговая запись формата `utmpx` должна содержать как минимум поля `ut_type`, `ut_user`, `ut_tv`, `ut_pid`, `ut_id` и `ut_line`. Поле `ut_type` должно быть равно `USER_PROCESS`. Поле `ut_id` должно содержать суффикс имени устройства (то есть терминала или псевдотерминала), с помощью которого пользователь вошел в систему, а в поле `ut_line` должно находиться его полное имя, но без `/dev/` в начале (пример того, как могут выглядеть эти два поля, показан в выводе программы из листинга 40.2). В файл `wtmp` добавляется запись с точно таким же содержимым.

  Имя терминала (представленное полями ut_line и ut_id) играет роль уникального ключа для записей в файле utmp.

- Когда пользователь выходит из системы, запись, добавленная ранее в файл `utmp`, должна быть стерта. Для этого создается новая запись типа `DEAD_PROCESS` с теми же полями `ut_id` и `ut_line`, которые использовались при входе, но с обнуленным полем `ut_user`. Она заменяет собой предыдущую запись. Ее копия также добавляется в файл `wtmp`.

  Если не удастся очистить запись utmp при выходе из системы (возможно, из-за сбоя программы), то процесс init сделает это за вас во время следующей перезагрузки; полю ut_type будет присвоено значение DEAD_PROCESS, а другие поля будут обнулены.

Обычно файлы `utmp` и `wtmp` защищены таким образом, чтобы только привилегированные пользователи могли вносить в них изменения. Точность вызова `getlogin()` зависит от целостности файла `utmp`. Ввиду этой и других причин права доступа к файлам `utmp` и `wtmp` никогда не должны позволять запись обычным пользователям.

Что такое сессия входа в систему? Как можно было бы ожидать, в соответствующие учетные файлы заносится информация о входе в систему с помощью программ `login`, `telnet` и `ssh`. Большинство FTP-клиентов также создают подобные учетные данные. Но относятся ли к этой категории записи, которые создаются при запуске каждого окна терминала в системе или, например, вызове команды `su`? Ответ на этот вопрос может варьироваться в зависимости от конкретной реализации UNIX.

Отдельные эмуляторы терминалов (например, xterm) предоставляют параметры командной строки или другие механизмы для определения, изменяет ли программа учетные файлы, связанные со входом в систему.

Функция `pututxline()` записывает структуру `utmpx`, на которую указывает аргумент `ut`, в файл `/var/run/utmp` (или какой-то другой, если перед этим был выполнен вызов `utmpxname()`).

```
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *ut);
```

Возвращает указатель на копию успешно измененной записи или `NULL` при ошибке

Прежде чем добавлять новую структуру, функция `pututxline()` сначала делает вызов `getutxid()` для обнаружения следующей записи, которая может быть заменена. Если такая запись существует, то она заменяется; в противном случае в конец файла добавляется новая структура. Часто перед вызовом `pututxline()` приложения используют одну из функций вида `getutx*()`, чтобы сделать текущей подходящую запись — отвечающую критериям структуры `utmpx`, на которую указывает аргумент `ut` вызова `getutxid()`. Если функция `pututxline()` обнаруживает, что данный шаг уже выполнен, то `getutxid()` не вызывается.

Если функция pututxline() делает внутри себя вызов getutxid(), то это не затрагивает статический участок памяти, в который функции getutx*() возвращают структуру utmpx. Данный аспект является одним из требований стандарта SUSv3.

Чтобы обновить файл `wtmp`, нужно просто открыть его и добавить новую запись. Поскольку это стандартная операция, в библиотеке glibc она реализована в виде функции `updwtmpx()`.

Функция `updwtmpx()` добавляет запись типа `utmpx`, на которую указывает `ut`, в файл, заданный в аргументе `wtmpx_file`.

```
#define _GNU_SOURCE
#include <utmpx.h>

void updwtmpx(char *wtmpx_file, struct utmpx *ut);
```

Эта функция не входит в стандарт SUSv3 и поддерживается всего несколькими другими реализациями UNIX. В других системах доступны похожие функции — `ogin(3)`, `logout(3)` и `logwtmp(3)`; они являются частью библиотеки glibc и описаны на соответствующих справочных страницах. Если же они отсутствуют, вам придется самостоятельно написать их аналоги (они имеют несложную реализацию).

Пример программы

Листинг 40.3 обновляет файлы `utmp` и `wtmp`, используя функции, описанные в данном разделе. Эта программа вносит необходимые изменения, чтобы впустить в систему пользователя, указанного в командной строке, и затем, после нескольких секунд ожидания, выполняет выход из системы. Обычно такие действия связаны с созданием и завершением сессии входа в систему. Эта программа задействует вызов `ttyname()` (описанный в разделе 58.10) для получения имени терминального устройства, связанного с файловым дескриптором.

Работа программы из листинга 40.3 продемонстрирована на примере сессии командной оболочки, приведенной ниже. Мы повышаем привилегии, чтобы иметь возможность обновлять учетные файлы, связанные со входом в систему, после чего используем нашу программу для создания записи для пользователя `mtk`:

```
$ su
Password:
# ./utmpx_login mtk
Creating login entries in utmp and wtmp
        using pid 1471, line pts/7, id /7
Нажимаем Ctrl+Z, чтобы приостановить программу
[1]+ Stopped                              ./utmpx_login mtk
```

Пока программа `utmpx_login` находилась в состоянии ожидания, мы нажали Ctrl+Z, чтобы приостановить ее выполнение и переключить ее в фоновый режим. Теперь воспользуемся программой из листинга 40.2 для просмотра содержимого файла `utmp`:

```
# ./dump_utmpx /var/run/utmp
user       type      PID line   id  host    date/time
cecilia   USER_PR    249 tty1   1           Fri Feb 1 21:39:07 2008
mtk       USER_PR   1471 pts/7  /7          Fri Feb 1 22:08:06 2008
# who
cecilia   tty1     Feb 1 21:39
mtk       pts/7    Feb 1 22:08
```

Мы воспользовались командой `who(1)` для демонстрации того, что ее вывод берется из файла `utmp`. Теперь просмотрим с помощью нашей программы содержимое файла `wtmp`:

```
# ./dump_utmpx /var/log/wtmp
user     type        PID line   id  host    date/time
cecilia  USER_PR     249 tty1   1           Fri Feb  1 21:39:07 2008
mtk      USER_PR    1471 pts/7  /7          Fri Feb  1 22:08:06 2008
# last mtk
mtk      pts/7                      Fri Feb 1 22:08 still logged in
```

Команда `last(1)` применена для иллюстрации того, что ее вывод основывается на файле `wtmp` (чтобы вывод программ `dump_utmpx` и `last` был более лаконичным, здесь не приводятся строки, которые не относятся к теме обсуждения).

Теперь воспользуемся командой `fg` для возврата программы `utmpx_login` в активный режим. Продолжив работу, `utmpx_login` добавляет в файлы `utmp` и `wtmp` записи о выходе из системы.

```
# fg
./utmpx_login mtk
Creating logout entries in utmp and wtmp
```

Теперь еще раз просмотрим содержимое файла `utmp`. Как видите, внесенная нами запись была заменена:

```
# ./dump_utmpx /var/run/utmp
user     type        PID line   id  host    date/time
cecilia  USER_PR     249 tty1   1           Fri  Feb 1 21:39:07 2008
DEAD_PR  1471        pts/7      /7          Fri  Feb 1 22:09:09 2008
# who
Cecilia  tty1     Feb 1 21:39
```

Последняя строчка этого вывода свидетельствует о том, что команда `who` проигнорировала запись `DEAD_PROCESS`.

Если вывести файл `wtmp`, мы увидим, что и здесь произошла замена нашей записи:

```
# ./dump_utmpx /var/log/wtmp
user     type        PID line   id  host    date/time
cecilia  USER_PR     249 tty1    1          Fri Feb  1 21:39:07 2008
mtk      USER_PR    1471 pts/7   /7         Fri Feb  1 22:08:06 2008
         DEAD_PR    1471 pts/7   /7         Fri Feb  1 22:09:09 2008
# last mtk
mtk                     pts/7      Fri Feb  1 22:08 - 22:09 (00:01)
```

В последней строке вышеприведенного вывода наглядно показано, как команда `last` ищет в файле `wtmp` записи о входе в систему и выходе из нее, чтобы вернуть время начала и завершения сессии пользователя.

Листинг 40.3. Обновление файлов utmp и wtmp

loginacct/utmpx_login.c

```
#define _GNU_SOURCE
#include <time.h>
#include <utmpx.h>
#include <paths.h>        /* Определения _PATH_UTMP и _PATH_WTMP */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct utmpx ut;
    char *devName;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s username [sleep-time]\n", argv[0]);

    /* Инициализируем запись входа в систему для файлов utmp и wtmp */

    memset(&ut, 0, sizeof(struct utmpx));
    ut.ut_type = USER_PROCESS;          /* Это запись о входе в систему */
    strncpy(ut.ut_user, argv[1], sizeof(ut.ut_user));
    if (time((time_t *) &ut.ut_tv.tv_sec) == -1)
        errExit("time");                /* Указываем текущее время */
    ut.ut_pid = getpid();
```

```
    /* Устанавливаем поля ut_line и ut_id на основе того, какой терминал связан
       со стандартным вводом. Мы исходим из того, что имена терминалов имеют вид
       "/dev/[pt]t[sy]*". Имя каталога "/dev/" занимает пять символов; префикс
       "[pt]t[sy]" – три символа (в итоге получается восемь символов). */

    devName = ttyname(STDIN_FILENO);
    if (devName == NULL)
        errExit("ttyname");
    if (strlen(devName) <= 8)       /* Это условие не должно выполняться */
        fatal("Terminal name is too short: %s", devName);

    strncpy(ut.ut_line, devName + 5, sizeof(ut.ut_line));
    strncpy(ut.ut_id, devName + 8, sizeof(ut.ut_id));

    printf("Creating login entries in utmp and wtmp\n");
    printf("        using pid %ld, line %.*s, id %.*s\n",
           (long) ut.ut_pid, (int) sizeof(ut.ut_line), ut.ut_line,
           (int) sizeof(ut.ut_id), ut.ut_id);

    setutxent();                    /* Переходим в начало файла utmp */
    if (pututxline(&ut) == NULL)  /* Добавляем в файл utmp запись о входе в систему */
        errExit("pututxline");
    updwtmpx(_PATH_WTMP, &ut);    /* Добавляем запись о входе в систему в файл wtmp */

    /* Засыпаем, давая пользователю возможность просмотреть файлы utmp и wtmp */

    sleep((argc > 2) ? getInt(argv[2], GN_NONNEG, "sleep-time") : 15);

    /* Теперь "выходим из системы"; используем значения из ранее
       инициализированной структуры 'ut', меняя их следующим образом: */

    ut.ut_type = DEAD_PROCESS;    /* Нужно для записи данных о выходе из системы */
    time((time_t *) &ut.ut_tv.tv_sec); /* Указываем время выхода */
    memset(&ut.ut_user, 0, sizeof(ut.ut_user));
                        /* Имя пользователя в записи о выходе равно NULL */
    printf("Creating logout entries in utmp and wtmp\n");
    setutxent();                    /* Переходим в начало файла utmp */
    if (pututxline(&ut) == NULL)  /* Заменяем ранее созданную в utmp запись */
        errExit("pututxline");
    updwtmpx(_PATH_WTMP, &ut);    /* Добавляем запись о выходе в файл wtmp */

    endutxent();
    exit(EXIT_SUCCESS);
}
```

loginacct/utmpx_login.c

40.7. Файл lastlog

Этот файл хранит информацию о времени последнего входа в систему каждого пользователя (в отличие от файла `wtmp`, в котором находятся данные обо всех входах и выходах из системы). В числе прочего файл `lastlog` позволяет программе `login` информировать пользователей о том, когда они в последний раз входили в систему (в начале новой сессии). Данный файл должен обновляться приложениями, отвечающими за вход в систему (наряду с `utmp` и `wtmp`).

Как и в случае с файлами `utmp` и `wtmp`, местоположение и формат `lastlog` могут варьироваться (а в ряде UNIX-систем он и вовсе отсутствует). В Linux этот файл имеет путь `/var/`

log/lastlog, на который указывает константа _PATH_LASTLOG, объявленная в заголовочном файле <paths.h>. Файл lastlog (по аналогии с utmp и wtmp) обычно защищен от записи со стороны пользователей, не имеющих соответствующих привилегий.

Записи в файле lastlog имеют следующий формат (описанный в <lastlog.h>):

```
#define UT_NAMESIZE         32
#define UT_HOSTSIZE        256

struct lastlog {
    time_t ll_time;               /* Время последнего входа в систему */
    char   ll_line[UT_NAMESIZE];  /* Терминал для удаленного входа */
    char   ll_host[UT_HOSTSIZE];  /* Имя компьютера, с которого
                                     был выполнен удаленный вход */
};
```

Обратите внимание: приведенные записи не содержат имени или идентификатора пользователя. Вместо этого они индексируются по UID. То есть, чтобы найти в файле lastlog запись для пользователя с идентификатором 1000, нужно перейти к подходящему байту (1000 * sizeof(struct lastlog)). Это продемонстрировано в программе из листинга 40.4, которая позволяет просматривать записи для пользователей, указанных в командной строке. По своим функциям она похожа на команду lastlog(1). Ниже показан пример вывода, возвращаемого в результате ее выполнения:

```
$ ./view_lastlog annie paulh
annie    tty2                     Mon Jan 17 11:00:12 2011
paulh    pts/11                   Sat Aug 14 09:22:14 2010
```

Для обновления файла lastlog его точно так же нужно открыть, перейти в подходящую позицию и выполнить запись.

Поскольку lastlog проиндексирован по пользовательскому идентификатору, мы не можем отличить записи о входе в систему, относящиеся к разным именам пользователей с одним и тем же UID (в разделе 8.1 отмечалось, что вход с одного UID под разными именами хоть и является необычным, но вполне возможен).

Листинг 40.4. Вывод содержимого файла lastlog

loginacct/view_lastlog.c

```
#include <time.h>
#include <lastlog.h>
#include <paths.h>                  /* Определение _PATH_LASTLOG */
#include <fcntl.h>
#include "ugid_functions.h"         /* Объявление userIdFromName() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct lastlog llog;
    int fd, j;
    uid_t uid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [username...]\n", argv[0]);

    fd = open(_PATH_LASTLOG, O_RDONLY);
    if (fd == -1)
        errExit("open");
```

```
    for (j = 1; j < argc; j++) {
        uid = userIdFromName(argv[j]);
        if (uid == -1) {
            printf("No such user: %s\n", argv[j]);
            continue;
        }

        if (lseek(fd, uid * sizeof(struct lastlog), SEEK_SET) == -1)
            errExit("lseek");
        if (read(fd, &llog, sizeof(struct lastlog)) <= 0) {
            printf("read failed for %s\n", argv[j]);
            /* Конец файла или ошибка */
            continue;
        }

        printf("%-8.8s %-6.6s %-20.20s %s", argv[j], llog.ll_line,
                llog.ll_host, ctime((time_t *) &llog.ll_time));
    }
    close(fd);
    exit(EXIT_SUCCESS);
}
```

loginacct/view_lastlog.c

40.8. Резюме

Механизм учета входа в систему записывает данные о пользователях, которые находятся в системе, а также сведения обо всех их предыдущих входах. Эта информация хранится в трех файлах: `utmp` (записи обо всех пользователях, находящихся в системе), `wtmp` (записи обо всех входах и выходах из системы) и `lastlog` (записи о времени последнего входа в систему каждого пользователя). Эти данные используются разными командами, такими как `who` и `last`.

Библиотека языка C предоставляет функции для извлечения и изменения информации, содержащейся в файлах учета входа в систему. Приложения, позволяющие входить в систему, должны задействовать настоящие функции для обновления соответствующих учетных данных, чтобы команды, работа которых зависит от этих данных, вели себя корректно.

Дополнительная информация

Если не считать справочную страницу `utmp(5)`, наиболее полезным источником информации о функциях учета входа в систему является исходный код различных приложений, которые их используют. Просмотрите, к примеру, исходные тексты программ `mingetty` (или `agetty`), `login`, `init`, `telnet`, `ssh` и `ftp`.

40.9. Упражнения

40.1. Реализуйте функцию `getlogin()`. Как отмечалось в разделе 40.5, в процессах, выполняющихся в рамках некоторых эмуляторов терминала, она может вести себя некорректно, поэтому ее работу лучше проверять в виртуальной консоли.

40.2. Измените программу из листинга 40.3 (`utmpx_login.c`) таким образом, чтобы она обновляла не только файлы `utmp` и `wtmp`, но и `lastlog`.

40.3. Прочитайте справочные страницы `login(3)`, `logout(3)` и `logwtmp(3)`. Реализуйте соответствующие функции.

40.4. Реализуйте упрощенную версию команды `who(1)`.

41 Основы разделяемых библиотек

Разделяемые библиотеки — это механизм размещения библиотечных функций в едином файле, доступном сразу нескольким выполняющимся процессам. Такой подход может сэкономить как дисковое пространство, так и оперативную память. В данной главе мы обсудим основы разделяемых библиотек, а в следующей рассмотрим ряд продвинутых возможностей, которыми они обладают.

41.1. Библиотека объектов

Один из способов сборки приложения заключается в компиляции его исходных файлов в объектные с последующей их компоновкой в итоговую исполняемую программу. Например:

```
$ cc -g -c prog.c mod1.c mod2.c mod3.c
$ cc -g -o prog_nolib prog.o mod1.o mod2.o mod3.o
```

> На самом деле сборка выполняется отдельной программой-компоновщиком, ld. Когда мы компилируем программу с помощью команды cc (или gcc), утилита ld вызывается автоматически, незаметно для нас. В Linux компоновщик всегда следует вызывать через компилятор, так как это позволяет запустить его с корректными параметрами и скомпоновать программу с подходящими библиотечными файлами.

Однако часто бывает так, что некоторые из исходных файлов можно было бы использовать в нескольких программах. Первым делом, чтобы не заниматься лишней работой, эти файлы можно скомпилировать только один раз и затем уже по необходимости компоновать их с разными исполняемыми файлами. И хотя такой подход уменьшает время компиляции, он все равно не избавляет от необходимости каждый раз указывать все объектные файлы на этапе компоновки. Более того, с увеличением количества таких файлов можно создать неразбериху в каталоге проекта.

Чтобы обойти эти проблемы, можно сгруппировать набор объектных файлов в единую сущность — *библиотеку объектов* (или *объектную библиотеку*). Библиотеки объектов бывают двух видов: *статические* и *разделяемые*. Последние являются более современными и имеют несколько преимуществ по сравнению со статическими (которые будут перечислены в разделе 41.3).

Небольшое отступление: включение отладочной информации во время компиляции программы

При запуске команды `cc`, показанной выше, мы использовали параметр `-g` для добавления отладочной информации в скомпилированную программу. В целом создание программ и библиотек, позволяющих выполнять отладку, является хорошей идеей (когда-то отладочную информацию иногда отключали, чтобы итоговый исполняемый файл задействовал меньше дискового пространства и оперативной памяти, но в современных реалиях эти ресурсы достаточно дешевые).

Кроме того, в некоторых архитектурах, таких как x86-32, не следует применять параметр `-fomit-frame-pointer`, поскольку он исключает возможность отладки (на таких платформах, как x86-64 данный параметр не мешает отлаживать программы, поэтому там он включен по умолчанию). По тем же причинам к исполняемым файлам и библиотекам не стоит применять утилиту `strip(1)`, убирающую из них отладочную информацию.

41.2. Статические библиотеки

Чтобы лучше понимать особенности и преимущества разделяемых библиотек, вначале кратко рассмотрим их статические аналоги.

Статические библиотеки (также известные как *архивы*) были первым видом библиотечных файлов, доступных в UNIX-системах. Они имеют следующие положительные стороны:

- можно поместить набор часто используемых объектных файлов в единую библиотеку, которую потом можно будет применять для сборки разных программ; при этом не нужно будет перекомпилировать оригинальные исходные тексты при компоновке каждой программы;
- упрощаются команды для компоновки. Вместо перечисления длинного списка объектных файлов можно указать всего лишь имя статической библиотеки. Компоновщик знает, как выполнять поиск по ней и извлекать объекты, необходимые для создания исполняемого файла.

Создание и редактирование статической библиотеки

Статическая библиотека, по сути, является обычным файлом, содержащим копии всех помещенных в него объектных файлов. В архиве также хранятся различные атрибуты для каждого объектного файла, включая права доступа, числовые идентификаторы пользователя и группы и время последнего изменения. Статическим библиотекам принято давать имена вида `lib`*`name`*`.a`.

Для создания и редактирования статических библиотек используется команда `ar(1)`, которая имеет следующую общую форму:

```
$ ar options archive object-file...
```

Аргумент `options` состоит из набора букв, одна из которых является *кодом операции*, а остальные — *модификаторами*, влияющим на то, как эта операция будет выполняться. Ниже приведена часть распространенных кодов операции.

- `r` (от англ. `replace` — «заменить»). Вставляет объектный файл в архив, заменяя им любой существующий файл с тем же именем. Это стандартный способ создания и обновления архивов. Таким образом, архив можно собрать с помощью следующих команд:

  ```
  $ cc -g -c mod1.c mod2.c mod3.c
  $ ar r libdemo.a mod1.o mod2.o mod3.o
  $ rm mod1.o mod2.o mod3.o
  ```

  Как видите, после создания библиотеки можно удалить оригинальные объектные файлы, поскольку они больше не нужны.
- `t` (от англ. `table of contents` — оглавление). Выводит оглавление архива. По умолчанию выводятся только имена объектных файлов. Но если дополнительно указать параметр `v` (от англ. `verbose` — «подробно»), можно просмотреть все атрибуты каждого файла в архиве, как показано ниже:

```
$ ar tv libdemo.a
rw-r--r-- 1000/100 1001016 Nov 15 12:26 2009 mod1.o
rw-r--r-- 1000/100 406668 Nov 15 12:21 2009 mod2.o
rw-r--r-- 1000/100 46672 Nov 15 12:21 2009 mod3.o
```

Дополнительные атрибуты каждого объекта слева направо: права доступа на момент добавления в архив, пользовательский и групповой идентификаторы, размер, а также дата и время последнего изменения.

- `d` (от англ. `delete` — «удалить»). Удаляет из архива заданный модуль, как показано в следующем примере:

```
$ ar d libdemo.a mod3.o
```

Использование статической библиотеки

Скомпоновать программу со статической библиотекой можно двумя способами. Первый из них таков: название файла библиотеки можно указать на этапе компоновки, как показано ниже:

```
$ cc -g -c prog.c
$ cc -g -o prog prog.o libdemo.a
```

Можно также поместить библиотеку в один из стандартных каталогов, по которым компоновщик выполняет поиск (например, `/usr/lib`), и затем указать ее имя (то есть имя файла без префикса `lib` и суффикса `.a`) с помощью параметра `-l`:

```
$ cc -g -o prog prog.o -ldemo
```

Если библиотека находится в каталоге, о котором компоновщику обычно ничего не известно, можно воспользоваться параметром `-L`, чтобы указать этот каталог отдельно:

```
$ cc -g -o prog prog.o -Lmylibdir -ldemo
```

Статическая библиотека может состоять из множества объектных модулей, но компоновщик выберет из них только те, которые нужны программе.

Скомпоновав программу, можно ее запустить как обычно:

```
$ ./prog
Called mod1-x1
Called mod2-x2
```

41.3. Краткий обзор разделяемых библиотек

Когда программа компонуется со статической библиотекой (или вовсе без использования библиотек), итоговый исполняемый файл содержит копии всех объектных модулей, скомпонованных с программой. Таким образом, несколько разных программ могут содержать в себе копии одних и тех же объектных модулей. Подобная избыточность несет в себе несколько недостатков:

- дисковое пространство уходит на хранение нескольких копий одних и тех же объектных модулей. Такие потери могут быть значительными;
- если несколько программ, применяющих одни и те же модули, выполняются одновременно, каждая из них будет хранить в виртуальной памяти свою отдельную копию этих модулей, увеличивая тем самым потребление виртуальной памяти в системе;
- если объектный модуль статической библиотеки требует каких-либо изменений (возможно, нужно закрыть дыру в безопасности или исправить ошибку), придется заново компоновать все исполняемые файлы, в которых этот модуль используется. Данный

недостаток усугубляется тем фактом, что системному администратору необходимо знать, с какими приложениями скомпонована библиотека.

Для устранения представленных недочетов были придуманы разделяемые библиотеки. Их ключевая идея состоит в том, что одна копия объектного модуля разделяется между всеми программами, задействующими его. Объектные модули не копируются в компонуемый исполняемый файл; вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются ее объектные модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память. Благодаря применению разделяемых библиотек исполняемые файлы требуют меньше места на диске и в виртуальной памяти (при выполнении).

Код разделяемых библиотек является общим для нескольких процессов, однако глобальные и статические переменные, объявленные внутри библиотеки, предоставляются в виде копий.

Кроме того, разделяемые библиотеки обладают следующими преимуществами:

- общий размер программ уменьшается, в связи с чем в некоторых случаях они могут быстрее загружаться в память, что ускоряет их запуск. Это относится только к большим разделяемым библиотекам, которые уже используются другими программами. На самом деле программа, первой загружающая разделяемую библиотеку, запускается дольше, поскольку данную библиотеку сначала нужно найти и загрузить в память;
- объектные модули не копируются в исполняемые файлы, а хранятся в единой разделяемой библиотеке, поэтому можно изменять общий код без необходимости выполнять повторную компоновку (ограничения описаны в разделе 41.8). Изменения можно вносить, даже когда запущенные программы уже применяют существующую версию разделяемой библиотеки.

Однако за эти дополнительные возможности приходится платить:

- разделяемые библиотеки более сложные по сравнению со статическими — как с точки зрения самой концепции, так и на практике, при их создании и сборке программ, которые их используют;
- разделяемые библиотеки должны быть скомпилированы с поддержкой адресно-независимого кода (см. подраздел 41.4.2), который ввиду применения дополнительных регистров (см. [Hubicka, 2003]) имеет издержки в большинстве архитектур;
- перемещение символов должно выполняться во время работы программы. Эта процедура требует, чтобы каждый символ в разделяемой библиотеке (переменная или функция) был изменен с учетом своего реального местоположения в виртуальной памяти. По этой причине программе, использующей разделяемую библиотеку, может понадобиться немного больше времени для выполнения, чем ее статически скомпонованному аналогу.

Еще одно применение разделяемых библиотек заключается в построении на их основе интерфейсов JNI (Java Native Interface), которые позволяют коду, написанному на языке Java, напрямую задействовать возможности операционной системы; для этого достаточно вызывать функции из разделяемой библиотеки. Подробную информацию см. в [Liang, 1999] и [Rochkind, 2004].

41.4. Создание и использование разделяемых библиотек. Первые шаги

Для начала, чтобы понять, как устроены разделяемые библиотеки, рассмотрим минимальную цепочку операций, необходимых для их сборки и применения. Пока что намеренно проигнорируем общепринятую систему именования файлов разделяемых библиотек,

описанную в разделе 41.6; она позволяет программам автоматически загружать самые свежие версии нужных им библиотек, а также делает возможным бесконфликтное сосуществование разных (так называемых *мажорных*) версий одной и той же библиотеки.

В этой главе мы сосредоточим свое внимание только на формате ELF (Executable and Linking Format — формат исполняемых и компонуемых файлов), поскольку именно он используется для создания исполняемых файлов и разделяемых библиотек в современных версиях Linux (а также во многих других реализациях UNIX).

ELF заменил собой старые форматы a.out и COFF.

41.4.1. Создание разделяемой библиотеки

Для построения разделяемой версии статической библиотеки, созданной ранее, нужно выполнить следующие шаги:

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -o libfoo.so mod1.o mod2.o mod3.o
```

Первая команда создает три объектных модуля (мы объясним назначение параметра `cc -fPIC` в следующем разделе). Команда `cc -shared` создает на основе этих модулей разделяемую библиотеку.

Имена разделяемых библиотек принято обрамлять префиксом `lib` и суффиксом `.so` (от англ. shared object — «разделяемый объект»).

Для демонстрации того, что параметры командной строки, которые мы задействуем при создании разделяемых библиотек, не зависят от компилятора, в наших примерах вместо `cc` указана команда `gcc`. Другой компилятор в иной UNIX-системе, вероятно, потребует пересмотра этих параметров.

Стоит также отметить, что скомпилировать исходные файлы и создать разделяемую библиотеку можно с помощью всего одной команды:

```
$ gcc -g -fPIC -Wall mod1.c mod2.c mod3.c -shared -o libfoo.so
```

Но чтобы четко разделить этапы компиляции и сборки библиотеки, в примерах, приводимых в данной главе, для этого будут использоваться две отдельные команды.

В отличие от статических готовые разделяемые библиотеки не позволяют добавлять или удалять отдельные объектные модули. Как и в случае с обычными исполняемыми программами, объектные файлы внутри разделяемой библиотеки теряют свою идентичность.

41.4.2. Адресно-независимый код

Параметр `cc -fPIC` заставляет компилятор сгенерировать *адресно-независимый код*. Это влияет на такие операции, как доступ к глобальным, статическим и внешним переменным, строковым константам, а также на то, как определяются адреса функций. Данные изменения позволяют размещать исполняемый код на любом участке виртуальной памяти. Такой механизм является необходимым для разделяемых библиотек, поскольку на этапе компоновки невозможно определить, куда именно будет загружен их код (точное местоположение разделяемой библиотеки в адресном пространстве зависит от различных факторов, таких как количество памяти, которую уже использует программа, загружающая библиотеку, и какие разделяемые библиотеки она успела загрузить).

Платформа Linux/x86-32 позволяет создавать исполняемые библиотеки на основе модулей, скомпилированных без параметра `-fPIC`. Однако в этом случае теряются некоторые преимущества разделяемых библиотек, так как страницы программного кода,

содержащие адресно-независимые ссылки, не могут быть разделены между процессами. В ряде архитектур невозможно собрать разделяемую библиотеку без параметра `-fPIC`.

Чтобы определить, был ли имеющийся объектный файл скомпилирован с использованием данного параметра, можно проверить наличие имени `_GLOBAL_OFFSET_TABLE_` в его таблице символов. Для этого подойдет любая из следующих двух команд:

```
$ nm mod1.o | grep _GLOBAL_OFFSET_TABLE_
$ readelf -s mod1.o | grep _GLOBAL_OFFSET_TABLE_
```

С другой стороны, если одна из нижеприведенных команд выведет что-либо на экран, это будет значить следующее: заданная разделяемая библиотека содержит как минимум один объектный модуль, скомпилированный без параметра `-fPIC`:

```
$ objdump --all-headers libfoo.so | grep TEXTREL
$ readelf -d libfoo.so | grep TEXTREL
```

Строка `TEXTREL` указывает на наличие объектного модуля, чей текстовый сегмент содержит ссылку, требующую перемещения кода на этапе выполнения.

Более подробно команды `nm`, `readelf` и `objdump` будут рассмотрены в разделе 41.5.

41.4.3. Использование статической библиотеки

Перед применением разделяемой библиотеки необходимо выполнить два шага, которые не требуются для работы со статическими библиотеками.

- Поскольку исполняемый файл больше не содержит копии нужных ему объектных модулей, он должен иметь возможность определять, какая разделяемая библиотека требуется для его работы. Для этого на этапе компоновки в исполняемый файл внедряется имя библиотеки (говоря в терминологии формата ELF, библиотечная зависимость записывается в метку `DT_NEEDED` исполняемого файла). Набор всех разделяемых библиотек, которые нужны программе, называют *списком динамических зависимостей*.
- Должен существовать механизм для поиска файла библиотеки по ее имени во время выполнения программы и последующей его загрузки в память, если он не был загружен до этого.

Внедрение имени библиотеки в исполняемый файл происходит автоматически при компоновке разделяемой библиотеки с программой:

```
$ gcc -g -Wall -o prog prog.c libfoo.so
```

Теперь, если запустить данную программу, можно будет увидеть следующее сообщение об ошибке:

```
$ ./prog
./prog: error in loading shared libraries: libfoo.so: cannot
open shared object file: No such file or directory
```

Это подводит нас ко второму обязательному шагу — *динамической компоновке*. Данная процедура разрешения внедренного имени библиотеки на этапе выполнения, производимая *динамическим компоновщиком* (который еще называют *динамически компонуемым загрузчиком* или *компоновщиком времени выполнения*). Он сам по себе является разделяемой библиотекой, `/lib/ld-linux.so.2`, применяемая всеми исполняемыми файлами формата ELF, использующими динамические библиотеки.

> Путь /lib/ld-linux.so.2 представляет собой обычную символьную ссылку на исполняемый файл динамического компоновщика. Имя данного файла имеет вид ld-version.so, где version — это версия библиотеки glibc, установленной в системе (например, ld-2.11.so). Путь к динами-

ческому компоновщику может варьироваться в зависимости от архитектуры. Например, на платформе IA-64 символьная ссылка на него выглядит как /lib/ld-linux-ia64.so.2.

Динамический компоновщик анализирует список динамических зависимостей программы и находит соответствующие библиотечные файлы, используя набор заранее заданных правил. Часть этих правил основывается на списке стандартных каталогов, в которых обычно хранятся разделяемые библиотеки (к примеру, `/lib` и `/usr/lib`). Причина сообщения об ошибке, приведенного выше, заключается в том, что библиотека находится в текущем каталоге, которая не учитывается динамическим компоновщиком при выполнении поиска.

Некоторые архитектуры (такие как zSeries, PowerPC64 и x86-64) поддерживают выполнение как 32-, так и 64-разрядных программ. В таких системах 32-разрядные библиотеки находятся в подкаталогх */lib, а их 64-разрядные версии — в подкаталогх */lib64.

Переменная среды LD_LIBRARY_PATH

Для оповещения динамического компоновщика о том, что разделяемая библиотека находится в нестандартном месте, можно воспользоваться переменной среды `LD_LIBRARY_PATH`, указав соответствующий каталог в качестве одного из элементов списка, разделенного двоеточиями. Для разделения каталогов можно также использовать точку с запятой, но, чтобы избежать интерпретации данного символа командной оболочкой, список в этом случае следует обрамить кавычками. Если переменная `LD_LIBRARY_PATH` определена, компоновщик начинает поиск разделяемой библиотеки с тех каталогов, которые в ней перечислены, и только потом переходит к стандартным библиотечным путям (позже вы увидите, что реальное приложение никогда не должно полагаться на эту переменную, но на данном этапе она облегчает работу с разделяемыми библиотеками). Следовательно, можно запустить программу с помощью следующей команды:

```
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

Синтаксис командной оболочки (bash, ksh и sh), использованный выше, позволяет определить переменную среды в рамках выполнения программы `prog`. Это определение сообщает динамическому компоновщику о том, что поиск разделяемых библиотек нужно проводить в текущем каталоге (то есть в `.`).

Пустое значение в списке LD_LIBRARY_PATH (то, что указано посередине в dirx::diry) эквивалентно . — текущему каталогу (но имейте в виду: присваивание пустого значения всей переменной LD_LIBRARY_PATH имеет совсем другой результат). Мы не используем этот синтаксис (стандарт SUSv3 не рекомендует применять его в переменной среды PATH).

Сравнение статической и динамической компоновки

Обычно термин *«компоновка»* используется для описания работы компоновщика, `ld`, который объединяет один или несколько объектных модулей в единый исполняемый файл. Приставка *«статическая»* иногда указывается специально, чтобы обозначить отличие этой процедуры от *динамической* компоновки — загрузки исполняемым файлом разделяемых библиотек на этапе выполнения. (Статическую компоновку иногда называют *компоновочным редактированием*, а статический компоновщик, такой как `ld`, — *компоновочным редактором*.) Через этап статической компоновки проходят все программы, в том числе и те, что задействуют разделяемые библиотеки (последние к тому же подлежат динамической компоновке во время выполнения).

41.4.4. Разделяемые библиотеки и имя soname

В примерах, продемонстрированных ранее, имя, которое встраивалось в исполняемый файл и затем искалось динамическим компоновщиком на этапе выполнения, являлось также именем самой разделяемой библиотеки. Это имя называют *реальным* (или *настоящим*). Однако такую библиотеку можно (а обычно так и делается) создать с использованием некоего псевдонима, который называется soname (от англ. shared object name — «имя разделяемого объекта»; метка `DT_SONAME` в терминологии формата ELF).

Если у разделяемой библиотеки есть имя soname, то во время статической компоновки оно встраивается в исполняемый файл вместо реального имени и, как следствие, используется динамическим компоновщиком для поиска библиотеки на этапе выполнения. Имя soname — это уровень абстракции, позволяющий выполняющейся программе задействовать версию библиотеки, отличную от той, с которой она была скомпонована (но совместимую с ней).

В разделе 41.6 будет показано, какими правилами принято руководствоваться при создании имени разделяемой библиотеки (реального и soname). Теперь же рассмотрим упрощенный пример, чтобы усвоить основные принципы.

Изначально soname указывается при создании разделяемой библиотеки:

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -Wl,-soname,libbar.so -o libfoo.so mod1.o mod2.o mod3.o
```

Параметр `-Wl,-soname,libbar.so` заставляет компоновщик пометить разделяемую библиотеку `libfoo.so` именем soname `libbar.so`.

Чтобы определить soname существующей разделяемой библиотеки, можно воспользоваться любой из следующих команд:

```
$ objdump -p libfoo.so | grep SONAME
  SONAME      libbar.so
$ readelf -d libfoo.so | grep SONAME
  0x0000000e (SONAME)     Library soname: [libbar.so]
```

Теперь, когда у нас есть разделяемая библиотека с именем soname, можно создать исполняемый файл с помощью уже знакомой команды:

```
$ gcc -g -Wall -o prog prog.c libfoo.so
```

Но на этот раз компоновщик обнаруживает, что библиотека `libfoo.so` содержит имя soname `libbar.so` и встраивает его в исполняемый файл.

Теперь, попытавшись запустить программу, мы увидим следующее:

```
$ LD_LIBRARY_PATH=. ./prog
prog: error in loading shared libraries: libbar.so: cannot open shared object file:
No such file or directory
```

Проблема заключается вот в чем: динамический компоновщик не может найти ничего, что имело бы имя `libbar.so`. Необходимо выполнить еще один шаг: создать символьную ссылку, связывающую soname с реальным именем библиотеки. Эта ссылка должна находиться в одном из тех каталогов, по которым выполняет поиск динамический компоновщик. Следовательно, можно запустить программу следующим образом:

```
$ ln -s libfoo.so libbar.so    Создаем в текущем каталоге символьную ссылку на soname
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

На рис. 41.1 показаны этапы компиляции и компоновки, используемые в ходе создания разделяемой библиотеки с именем soname, а также процедура компоновки программы с этой библиотекой и создание символьной ссылки на имя soname, необходимой для запуска программы.

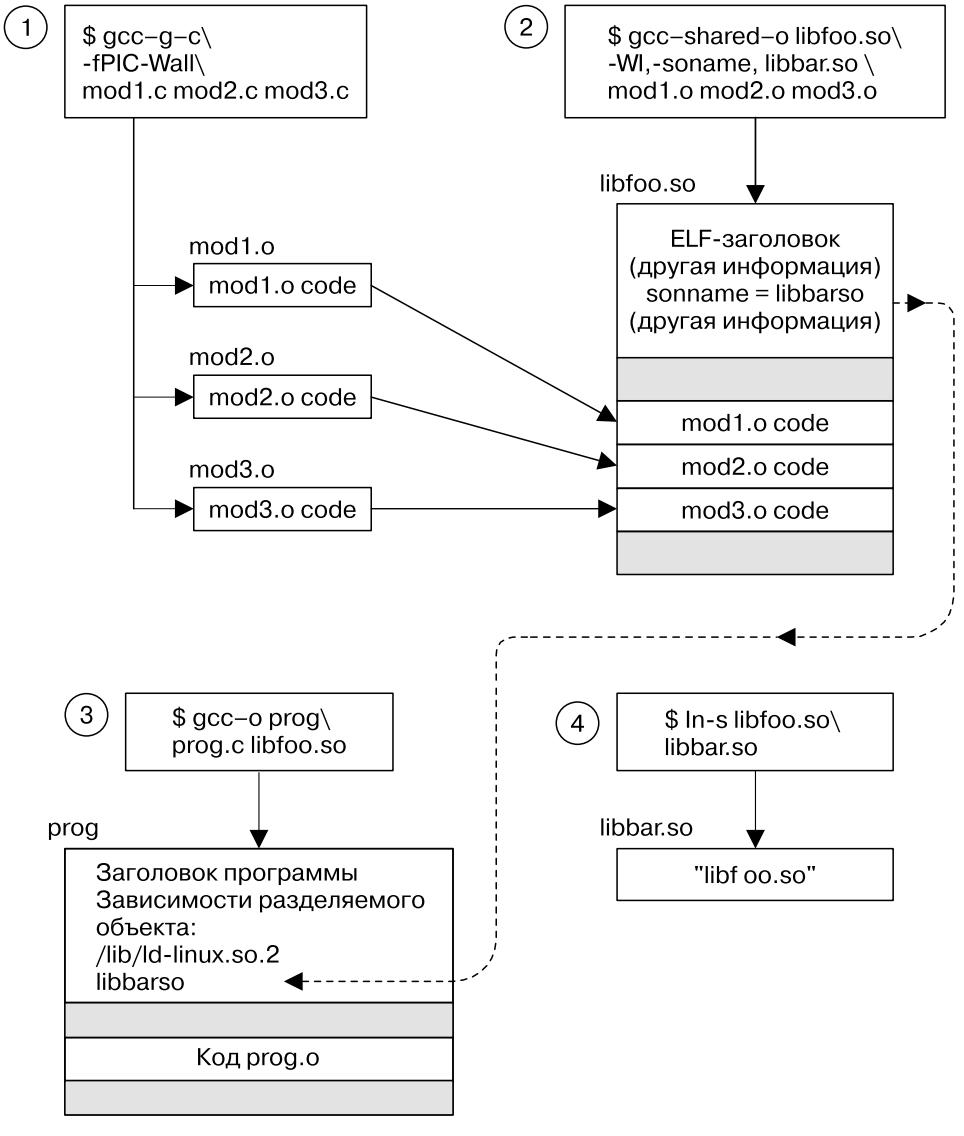


Рис. 41.1. Создание разделяемой библиотеки и компоновка с ней программы

На рис. 41.2 показано, через какие этапы проходит вышеприведенная программа при загрузке в память и подготовке к выполнению.

Чтобы выяснить, какую разделяемую библиотеку использует процесс, можно просмотреть содержимое /proc/PID/maps (поддерживается только в Linux; см. раздел 48.5).

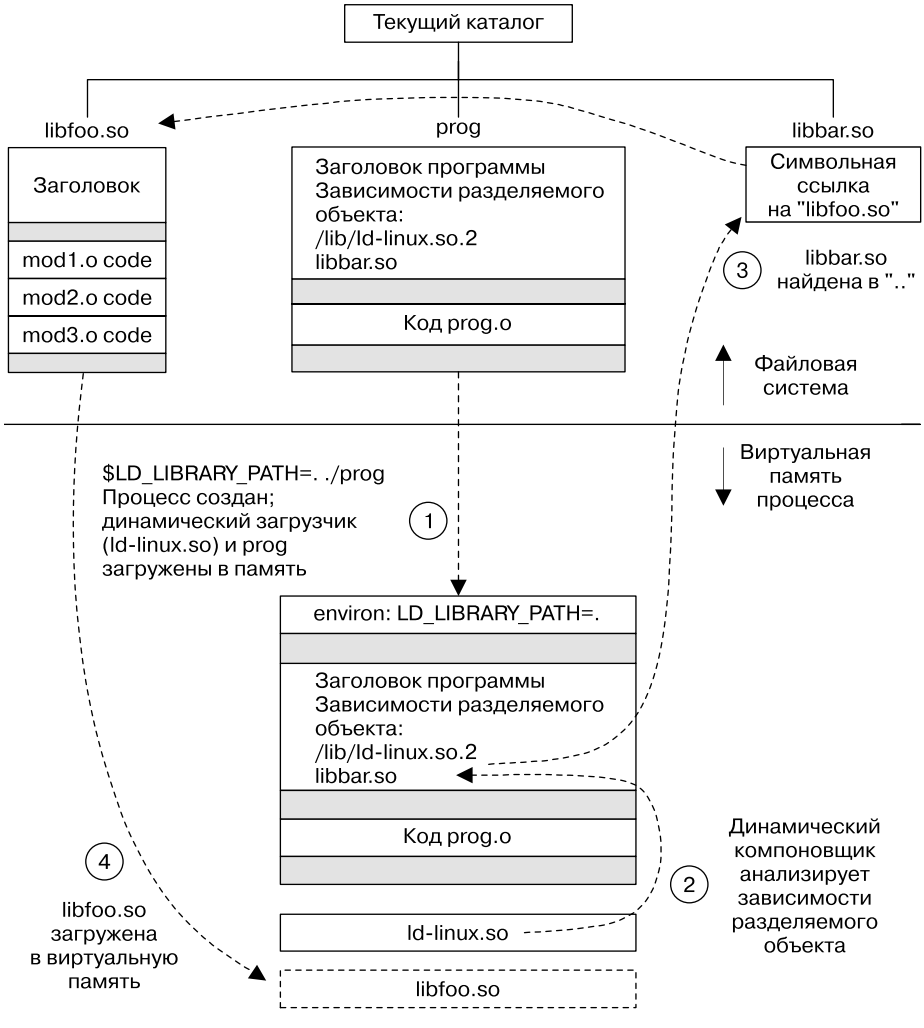


Рис. 41.2. Выполнение программы, загружающей разделяемую библиотеку

41.5. Полезные инструменты для работы с разделяемыми библиотеками

В этом разделе мы кратко рассмотрим несколько инструментов, которые могут пригодиться при анализе разделяемых библиотек, исполняемых файлов и скомпилированных объектов (файлов `.o`).

Команда ldd

Команда `ldd(1)` (от англ. list dynamic dependencies — «вывести динамические зависимости») выводит разделяемые библиотеки, необходимые для работы программы (или другой разделяемой библиотеки). Например:

```
$ ldd prog
        libdemo.so.1 => /usr/lib/libdemo.so.1 (0x40019000)
        libc.so.6 => /lib/tls/libc.so.6 (0x4017b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Команда `ldd` находит все модули, на которые ссылается библиотека (по тому же принципу, что и динамический компоновщик), и выводит результат в следующем виде:

имя_библиотеки => ссылается_на_путь

Для большинства исполняемых файлов в формате ELF команда `ldd` выведет как минимум `ld-linux.so.2` (динамический компоновщик) и `libc.so.6` (стандартную библиотеку языка C).

> Имя библиотеки языка C может отличаться в зависимости от архитектуры. Например, на платформах IA-64 и Alpha она называется libc.so.6.1.

Команды objdump и readelf

Команда `objdump` позволяет получить из исполняемого файла, скомпилированного объекта или разделяемой библиотеки различную информацию, включая дизассемблированный машинный код в двоичном формате. С ее помощью также можно вывести содержимое заголовков разных ELF-разделов упомянутых файлов; в этом контексте она напоминает команду `readelf`, которая выводит похожие данные, но в другом формате. Источники с подробным описанием команд `objdump` и `readelf` перечислены в конце настоящей главы.

Команда nm

Выводит список всех символов, определенных внутри объектной библиотеки или исполняемой программы. С ее помощью можно узнать, какой именно библиотеке принадлежит тот или иной символ. Например, чтобы понять, где определена функция `crypt()`, можно сделать следующее:

```
$ nm -A /usr/lib/lib*.so 2> /dev/null | grep ' crypt$'
/usr/lib/libcrypt.so:00007080 W crypt
```

Параметр `-A` говорит о том, что перед каждым символом должно быть указано имя библиотеки. Он нужен, поскольку команда `nm` по умолчанию сначала выводит имя библиотеки, а потом перечисляет все символы, которые ей принадлежат; это не подходит для выполнения фильтрации, как в нашем примере. Кроме того, мы отключили стандартный вывод ошибок, чтобы скрыть сообщения о файлах, формат которых данная команда не поддерживает. На основе результата, полученного выше, можно сказать, что функция `crypt()` определена в библиотеке libcrypt.

41.6. Версии и соглашение об именовании разделяемых библиотек

Теперь посмотрим, что собой представляет версионирование разделяемых библиотек. Обычно «соседние» версии совместимы друг с другом; то есть функции в каждом модуле предоставляют один и тот же интерфейс и являются семантически тожественными (то естьпроизводят идентичные результаты). Такие совместимые версии разделяемой библиотеки называют *минорными*. Однако время от времени приходится создавать новую *мажорную* версию библиотеки, несовместимую с предыдущей (из раздела 41.8 вы узнаете,

что именно может стать причиной несовместимости). При этом должна оставаться возможность запускать программы, которым требуется старая версия библиотеки.

Для решения данной задачи используется стандартное соглашение об именовании разделяемых библиотек, относящееся как к реальным именам, так и к soname.

Реальные имена, имена компоновки и soname

Каждая несовместимая версия разделяемой библиотеки имеет уникальный *идентификатор мажорной версии*, составляющий часть ее реального имени. Этот идентификатор принято изображать в виде числа, которое последовательно инкрементируется относительно последней несовместимой версии библиотеки. Реальное имя также содержит идентификатор, разделяющий совместимые минорные версии в рамках мажорной. Для реального имени принято использовать следующий формат: *имя_библиотеки*.so.*мажорный_идентификатор*.*минорный_идентификатор*.

В качестве идентификаторов минорных (как и мажорных) версий могут выступать любые строки, но, если следовать соглашению об именовании, это должны быть либо число, либо два числа, разделенных точкой; первое число обозначает минорную версию, а второе — номер патча или ревизии в рамках данной минорной версии. Ниже приведены некоторые примеры реальных имен разделяемых библиотек:

```
libdemo.so.1.0.1
libdemo.so.1.0.2        Минорная версия, совместимая с версией 1.0.1
libdemo.so.2.0.0        Новая мажорная версия, несовместимая с версиями 1.*
libreadline.so.5.0
```

Имя soname разделяемой библиотеки включает в себя тот же идентификатор мажорной версии, что и ее реальное имя, но не содержит минорной. Таким образом, имя soname имеет вид *имя_библиотеки*.so.*мажорный_идентификатор*.

Обычно soname создается в качестве относительной символьной ссылки в каталоге, содержащем реальное имя. Ниже показаны примеры отдельных имен soname и реальных имен, на которые они могут ссылаться.

```
libdemo.so.1        -> libdemo.so.1.0.2
libdemo.so.2        -> libdemo.so.2.0.0
libreadline.so.5    -> libreadline.so.5.0
```

Одна и та же мажорная версия разделяемой библиотеки может состоять из нескольких файлов, отличающихся номерами минорных версий. Обычно имя soname указывает на последнюю минорную версию в рамках мажорной (как показано выше, в случае с `libdemo.so`). Это обеспечивает корректную семантику версионирования во время использования разделяемой библиотеки выполняющимися программами. На этапе статической компоновки копия имени soname (не зависящая от минорной версии) встраивается в исполняемый файл. Ее символьная ссылка впоследствии может быть перенаправлена на более новую (минорную) версию разделяемой библиотеки; благодаря этому исполняемый файл может загружать ее самую свежую версию. Кроме того, разные мажорные версии библиотеки имеют разные имена soname, что позволяет им благополучно сосуществовать и быть доступными для программ, которые в них нуждаются.

Помимо реального имени и soname, для каждой разделяемой библиотеки обычно определяется еще и третье имя, *компоновочное*, используемое во время компоновки библиотеки с исполняемым файлом. Оно представляет собой символьную ссылку, содержащую только имя библиотеки, без номеров мажорных и минорных версий — *имя_библиотеки*.so. Это имя позволяет формировать команды компоновки, не зависящие от версий, которые автоматически применяются к корректной (то есть самой свежей) редакции библиотеки.

Обычно компоновочное имя создается в том же каталоге, что и файл, на который оно ссылается. Оно может быть привязано к реальному имени или soname последней мажорной версии библиотеки. Привязка к soname, как правило, предпочтительней, так как изменения в одном имени автоматически отражаются в другом (в разделе 41.7 вы увидите, что программа `ldconfig` автоматизирует процедуру обновления soname, и это неявно сказывается и на компоновочном имени, если выбрать вышеописанный подход).

Если нужно скомпоновать программу с более старой мажорной версией библиотеки, то нельзя использовать компоновочное имя. Вместо этого на этапе компоновки следует указать подходящее реальное имя или soname.

Ниже показаны примеры некоторых компоновочных имен:

```
libdemo.so            -> libdemo.so.2
libreadline.so        -> libreadline.so.5
```

В табл. 41.1 собрана краткая информация о soname, реальном и компоновочном именах разделяемой библиотеки, а на рис. 41.3 показаны отношения между ними.

Таблица 41.1. Краткая информация об именах разделяемой библиотеки

Имя	Формат	Описание
Реальное имя	имя_библиотеки.so.мажорный_идентификатор.минорный_идентификатор	Файл, хранящий библиотечный код; один экземпляр на каждую мажорную-плюс-минорную версию библиотеки
Soname	имя_библиотеки.so.мажорный_идентификатор	Один экземпляр на каждую мажорную версию библиотеки; встраивается в исполняемый файл во время компоновки; используется на этапе выполнения для поиска библиотеки с помощью символьной ссылки с тем же именем, которое указывает на подходящее (самое свежее) реальное имя
Компоновочное имя	имя_библиотеки.so	Символьная ссылка на последнее реальное имя или (что более вероятно) на последнее имя soname; в единственном экземпляре; позволяет создавать команды компоновки, не зависящие от версии

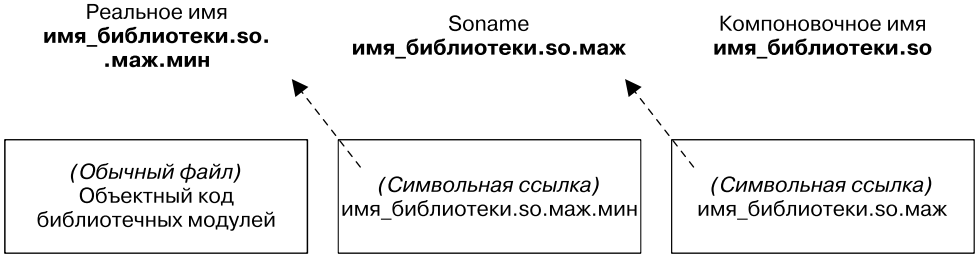


Рис. 41.3. Традиционная структура имен разделяемой библиотеки

Создание разделяемой библиотеки с применением общепринятых методик

Теперь, используя всю вышеприведенную информацию, мы покажем, как создать демонстрационную библиотеку, следуя общепринятым методикам. Для начала создадим объектные файлы:

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
```

Затем построим разделяемую библиотеку с реальным именем `libdemo.so.1.0.1` и soname `libdemo.so.1`.

```
$ gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.1 \
          mod1.o mod2.o mod3.o
```

Теперь можно создать символьные ссылки для soname и компоновочного имени:

```
$ ln -s libdemo.so.1.0.1 libdemo.so.1
$ ln -s libdemo.so.1 libdemo.so
```

Чтобы проверить результат, можно воспользоваться командой `ls` (с применением утилиты `awk`, которая отбирает только интересующие нас поля):

```
$ ls -l libdemo.so* | awk '{print $1, $9, $10, $11}'
lrwxrwxrwx libdemo.so -> libdemo.so.1
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
```

Построим наш исполняемый файл, применяя компоновочное имя (обратите внимание: в команде компоновки не упоминаются номера версий), и запустим программу уже привычным нам способом:

```
$ gcc -g -Wall -o prog prog.c -L. -ldemo
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

41.7. Установка разделяемых библиотек

До сих пор в наших примерах мы создавали разделяемые библиотеки в локальном пользовательском каталоге и затем задействовали переменную среды `LD_LIBRARY_PATH`, чтобы этот каталог была одним из тех, по которым динамический компоновщик выполняет поиск. Этот подход доступен как для обычных, так и для привилегированных пользователей, однако его не следует применять для реальных приложений. Обычно разделяемая библиотека вместе со всеми своими символьными ссылками устанавливается в один из стандартных каталогов — в частности, в один из следующих:

- `/usr/lib` — каталог, в который устанавливается большинство стандартных библиотек;
- `/lib` — в этот каталог следует устанавливать библиотеки, необходимые во время загрузки системы (так как на данном этапе каталог `/usr/lib` может быть еще недоступен);
- `/usr/local/lib` — в данный каталог следует устанавливать нестандартные и экспериментальные библиотеки (он также подходит в ситуациях, когда каталог `/usr/lib` подключается по сети и доступен сразу нескольким системам, а библиотеку при этом нужно установить локально);
- один из каталогов, перечисленных в файле `/etc/ld.so.conf` (который мы рассмотрим чуть ниже).

В большинстве случаев копирование файлов в любой из этих каталогов требует привилегий администратора.

После установки для soname и компоновочного имени должны быть созданы (относительные) символьные ссылки — обычно в том же каталоге, где находится сам файл библиотеки. Таким образом, чтобы установить демонстрационную библиотеку в `/usr/lib` (обновлять которую может только администратор), нужно выполнить следующие шаги:

```
$ su
Password:
# mv libdemo.so.1.0.1 /usr/lib
# cd /usr/lib
# ln -s libdemo.so.1.0.1 libdemo.so.1
# ln -s libdemo.so.1 libdemo.so
```

Последние две строчки в сессии командной строки создают символьные ссылки для soname и компоновочного имени.

ldconfig

Программа `ldconfig(8)` решает две потенциальные проблемы, присущие разделяемым библиотекам:

- разделяемая библиотека может находиться в разных каталогах. Ее загрузка может значительно замедлиться, если динамическому компоновщику придется искать ее во всех этих местах;
- при установке новых и удалении старых версий библиотеки некоторые символьные ссылки на имена soname могут оказаться неактуальными.

Для решения этих проблем программа `ldconfig` выполняет следующие действия.

1. Она проходит по стандартному набору каталогов и создает или обновляет файл кэша, `/etc/ld.so.cache`, содержащий мажорные версии всех найденных библиотек (точнее, последние минорные версии каждой из них). Динамический компоновщик, в свою очередь, использует данный кэш во время выполнения программы для превращения имени библиотеки в путь. Чтобы построить кэш, программа `ldconfig` сначала выполняет поиск по каталогм, указанным в файле `/etc/ld.so.conf`, а затем переходит к `/lib` и `/usr/lib`. Файл `/etc/ld.so.conf` содержит список каталогов (это должны быть полные пути), разделенных символами перевода строки, пробелами, табуляцией, запятыми или двоеточиями. В некоторых дистрибутивах в этот список входит каталог `/usr/local/lib` (в противном случае его, возможно, придется добавить вручную).

   Команда ldconfig -p выводит текущее содержимое файла /etc/ld.so.cache.

2. Она проверяет последнюю минорную версию (то есть версию с самым большим минорным номером) каждой мажорной версии библиотеки для поиска встроенного имени soname, и затем создает (или обновляет) относительные символьные ссылки для каждого именн soname в том же каталоге.

Для корректного выполнения этих действий программе `ldconfig` нужно, чтобы библиотеки соблюдали соглашения об именовании, описанные ранее (то есть реальное имя библиотеки должно включать мажорный и минорный идентификаторы, которые соответствующим образом инкрементируются от версии к версии).

По умолчанию программа `ldconfig` выполняет оба действия, описанных выше. Чтобы выборочно пропустить одно из них, можно воспользоваться параметрами командной строки: параметр `-N` предотвращает обновление кэша, а `-X` позволяет обойтись без создания

символьных ссылок для имен soname. Существует также параметр `-v` (от англ. verbose — «подробно»), который заставляет команду `ldconfig` выводить подробности о ее работе.

Программа `ldconfig` должна запускаться при установке каждой новой или обновлении/удалении имеющейся библиотеки, а также в случае изменения списка каталогов в файле `/etc/ld.so.conf`.

В качестве примера работы программы `ldconfig` рассмотрим ситуацию, когда нужно установить две разные мажорные версии библиотеки. Для этого можно сделать следующее:

```
$ su
Password:
# mv libdemo.so.1.0.1 libdemo.so.2.0.0 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.1 (changed)
        libdemo.so.2 -> libdemo.so.2.0.0 (changed)
```

Выше мы отфильтровали вывод `ldconfig` с целью получить только информацию, касающуюся библиотеки с именем `libdemo`.

Теперь, чтобы проверить наличие символьных ссылок для soname, выведем список файлов из каталога `/usr/lib`, в именах которых содержится название `libdemo`:

```
# cd /usr/lib
# ls -l libdemo* | awk '{print $1, $$9, $10, $11}'
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
lrwxrwxrwx libdemo.so.2 -> libdemo.so.2.0.0
-rwxr-xr-x libdemo.so.2.0.0
```

Нам все еще нужно создать символьную ссылку для компоновочного имени. Воспользуемся для этого следующей командой:

```
# ln -s libdemo.so.2 libdemo.so
```

Однако, если установить новую новую минорную версию нашей библиотеки, 2.x, то программа `ldconfig` обновит и компоновочное имя, поскольку оно указывает на последнюю версию soname. В этом можно убедиться на следующем примере:

```
# mv libdemo.so.2.0.1 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.1
        libdemo.so.2 -> libdemo.so.2.0.1 (changed)
```

При сборке и использовании приватной библиотеки (такой, которая не устанавливается в стандартные каталоги) программа `ldconfig` может помочь создать символьную ссылку на имя soname — достаточно указать параметр `-n`. Это заставит `ldconfig` обрабатывать только каталоги, заданные в командной строке, и предотвратит обновление файла с кэшем. В следующем примере мы воспользуемся программой `ldconfig` для обработки библиотек в текущем каталоге:

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.1 \
          mod1.o mod2.o mod3.o
$ /sbin/ldconfig -nv .
.:
        libdemo.so.1 -> libdemo.so.1.0.1
$ ls -l libdemo.so* | awk '{print $1, $9, $10, $11}'
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
```

При вызове программы `ldconfig` мы указали полный путь к ней, так как наша учетная запись не имеет повышенных привилегий и в ее переменной среды `PATH` не содержится каталог `/sbin`.

41.8. Совместимые и несовместимые библиотеки

Со временем в код разделяемой библиотеки приходится вносить изменения. Результатом этого будет новая версия библиотеки, которая либо *совместима* с предыдущими (то есть нужно поменять только минорный идентификатор в ее реальном имени), либо *нет* (в таком случае устанавливается новый мажорный номер).

Вносимые изменения совместимы с имеющейся версией библиотеки, если выполняются *все* условия, перечисленные ниже.

- Семантика каждой публичной функции и переменной в библиотеке остается неизменной. Иными словами, каждая функция сохраняет тот же список аргументов и возвращает то же значение, не меняя своего влияния на глобальные переменные и возвращаемые аргументы. Таким образом, изменения, относящиеся к улучшению производительности или устранению ошибок (что приводит к более строгому следованию заявленному поведению) могут считаться совместимыми.
- Из публичного программного интерфейса библиотеки не была удалена ни одна функция или переменная. Хотя добавление новых публичных функций и переменных не нарушает совместимость.
- Структуры, которые выделяются внутри всех функций и возвращаются ими, остаются неизменными. Аналогично не меняются структуры библиотеки, доступные для применения извне. Исключение из правила состоит в том, что в ряде обстоятельств в конец структуры могут быть добавлены новые элементы, хотя и это может оказаться источником непредвиденного поведения, если, например, вызывающая программа попытается выделить массив данного типа. Чтобы обойти описанные ограничения, разработчики библиотек иногда делают размер экспортируемых структур большим, чем того требует текущая версия библиотеки, добавляя несколько запасных полей «для будущего использования».

Если ни одно из этих условий не нарушается, новое имя библиотеки может быть сформировано путем инкрементации минорной версии. В противном случае необходимо создавать новый мажорный идентификатор.

41.9. Обновления разделяемых библиотек

Одно из преимуществ разделяемых библиотек заключается в том, что их новые мажорные или минорные версии можно устанавливать даже во время работы программ, которые используют текущую версию. Для этого достаточно создать новую ревизию библиотеки, установить ее в подходящий каталог и в случае необходимости обновить символьные ссылки на soname и компоновочное имя (или, как это принято, дать возможность программе `ldconfig` сделать всю работу за нас). Для выпуска новой минорной версии (то есть выполнения совместимого обновления) разделяемой библиотеки `/usr/lib/libdemo.1.0.1` потребовалось бы сделать следующее:

```
$ su
Password:
# gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
# gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.2 \
```

```
        mod1.o mod2.o mod3.o
# mv libdemo.so.1.0.2 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.2 (changed)
```

Если исходное компоновочное имя было установлено правильно (так, чтобы оно указывало на soname), не пришлось бы его менять.

Программа, запущенная ранее, продолжит использовать предыдущую минорную версию разделяемой библиотеки. Новая версия станет доступной только после перезапуска.

Если вслед за этим мы заходим создать новую мажорную версию (2.0.0), придется выполнить следующие шаги:

```
# gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
# gcc -g -shared -Wl,-soname,libdemo.so.2 -o libdemo.so.2.0.0 \
        mod1.o mod2.o mod3.o
# mv libdemo.so.2.0.0 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.2
        libdemo.so.2 -> libdemo.so.2.0.0 (changed)
# cd /usr/lib
# ln -sf libdemo.so.2 libdemo.so
```

Как можно видеть выше, `ldconfig` автоматически создает символьную ссылку на имя soname для новой мажорной версии. Но, как показывает последняя команда, обновлять символьную ссылку на компоновочное имя приходится вручную.

41.10. Задание каталогов для поиска библиотеки в объектном файле

Мы уже познакомились с двумя способами оповещения динамического компоновщика о местоположении разделяемой библиотеки: используя переменную среды `LD_LIBRARY_PATH` и путем установки библиотеки в один из стандартных библиотечных каталогов (`/lib`, `/usr/lib` или в один из перечисленных в файле `/etc/ld.so.conf`).

Но есть и третий путь: на этапе статического редактирования в исполняемый файл можно вставить список каталогов, в которых во время выполнения следует искать разделяемые библиотеки. Данный путь бывает полезен в ситуациях, когда библиотеки находятся в заранее определенных, но нестандартных местах, о которых не знает динамический компоновщик. Для этого при создании исполняемого файла можно воспользоваться параметром компоновщика `-rpath`:

```
$ gcc -g -Wall -Wl,-rpath,/home/mtk/pdir -o prog prog.c libdemo.so
```

Команда, приведенная выше, копирует строку `/home/mtk/pdir` в список путей времени выполнения (`rpath`) исполняемого файла `prog`, чтобы во время запуска этой программы динамический компоновщик выполнял поиск разделяемых библиотек и по заданному каталогу.

При необходимости параметр `-rpath` можно указать несколько раз; все каталоги будут объединены в упорядоченный список и помещены в исполняемый файл. То же самое можно сделать, задействуя один параметр `-rpath`, перечисляя пути через двоеточие. Во время выполнения динамический компоновщик будет искать по этим каталогам в порядке их указания с помощью параметра (-ов) `-rpath`.

Вместо параметра -rpath можно использовать переменную среды LD_RUN_PATH. Ей можно присвоить строку со списком путей, разделенных двоеточиями, который будет применяться в качестве rpath на этапе сборки исполняемого файла. Переменная LD_RUN_PATH учитывается только в том случае, если во время компоновки не было указано параметра -rpath.

Использование параметра -rpath во время сборки разделяемой библиотеки

Параметр компоновщика `-rpath` можно также применять при сборке разделяемой библиотеки. Допустим, у нас есть библиотека `libx1.so`, зависящая от другой, `libx2.so` (рис. 41.4). Представим также, что эти библиотеки находятся в нестандартных каталогах соответственно `d1` и `d2`. Теперь пройдемся по этапам, необходимым для их сборки и создания программы, которая их использует.

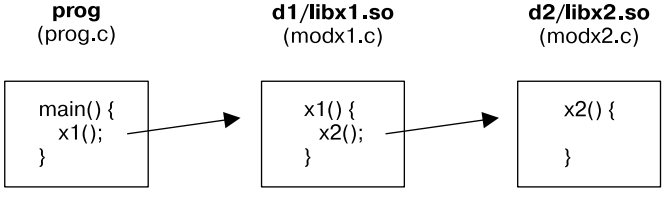


Рис. 41.4. Разделяемая библиотека, зависящая от другой разделяемой библиотеки

Для начала создадим файл `libx2.so` в каталоге `pdir/d2` (чтобы не усложнять пример, обойдемся без нумерации версий и задания имен soname вручную).

```
$ cd /home/mtk/pdir/d2
$ gcc -g -c -fPIC -Wall modx2.c
$ gcc -g -shared -o libx2.so modx2.o
```

Теперь создадим библиотеку `libx1.so` в каталоге `pdir/d1`. Поскольку `libx1.so` зависит от файла `libx2.so`, находящегося в нестандартном каталоге, путь к данному файлу будет указан с помощью параметра компоновщика `-rpath`. Это значение может отличаться от каталога, который применяется во время компоновки библиотеки (и задается с использованием параметра `-L`), но в данном случае оба пути совпадают.

```
$ cd /home/mtk/pdir/d1
$ gcc -g -c -Wall -fPIC modx1.c
$ gcc -g -shared -o libx1.so modx1.o -Wl,-rpath,/home/mtk/pdir/d2 \
            -L/home/mtk/pdir/d2 -lx2
```

Наконец, мы можем создать в каталоге `pdir` главную программу. Поскольку она использует библиотеку `libx1.so`, которая находится в нестандартном месте, опять задействуем параметр компоновщика `-rpath`:

```
$ cd /home/mtk/pdir
$ gcc -g -Wall -o prog prog.c -Wl,-rpath,/home/mtk/pdir/d1 \
            -L/home/mtk/pdir/d1 -lx1
```

Обратите внимание: при компоновке главной программы не потребовалось упоминать библиотеку `libx2.so`. Компоновщик способен сам проанализировать `rpath` в файле `libx1.so` и найти `libx2.so`; это позволяет удовлетворить требование, согласно которому все символы должны быть найдены на этапе статической компоновки.

Для просмотра списков `rpath` в файлах `prog` и `libx1.so` можно воспользоваться следующими командами:

```
$ objdump -p prog | grep PATH
  RPATH       /home/mtk/pdir/d1    Во время выполнения здесь будет
                                   произведен поиск libx1.so
$ objdump -p d1/libx1.so | grep PATH
  RPATH       /home/mtk/pdir/d2    Во время выполнения здесь будет
                                   произведен поиск libx2.so
```

Списки rpath также можно просматривать, анализируя вывод команды readelf --dynamic (или аналогичной ей readelf -d), используя утилиту grep.

Команда `ldd` позволяет просмотреть список всех динамических зависимостей программы `prog`:

```
$ ldd prog
        libx1.so => /home/mtk/pdir/d1/libx1.so (0x40017000)
        libc.so.6 => /lib/tls/libc.so.6 (0x40024000)
        libx2.so => /home/mtk/pdir/d2/libx2.so (0x4014c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Записи DT_RPATH и DT_RUNPATH в файлах формата ELF

Исходная спецификация формата ELF допускала встраивание в исполняемый файл только одного вида списка `rpath` или разделяемой библиотеки. Это относится к метке `DT_RPATH` в ELF-файле. Однако в последующих версиях данного формата эта метка считается устаревшей, а для представления списков `rpath` теперь используется запись `DT_RUNPATH`. Разница между двумя указанными списками заключается в их приоритете относительно переменной среды `LD_LIBRARY_PATH`, когда динамический компоновщик ищет разделяемые библиотеки во время выполнения программы: `DT_RPATH` имеет больший приоритет, а `DT_RUNPATH` — меньший (см. раздел 41.11).

По умолчанию компоновщик создает список `rpath` в виде метки `DT_RPATH`. Чтобы задействовать вместо этого запись `DT_RUNPATH`, следует дополнительно указать параметр `--enable–new–dtags` (`включить новые динамические метки`). Если пересобрать нашу программу с использованием этого параметра и проанализировать итоговый исполняемый файл с помощью утилиты `objdump`, можно увидеть следующее:

```
$ gcc -g -Wall -o prog prog.c -Wl,--enable-new-dtags \
        -Wl,-rpath,/home/mtk/pdir/d1 -L/home/mtk/pdir/d1 -lx1
$ objdump -p prog | grep PATH
  RPATH       /home/mtk/pdir/d1
  RUNPATH     /home/mtk/pdir/d1
```

Как видите, исполняемый файл содержит обе метки — `DT_RPATH` и `DT_RUNPATH`. Компоновщик дублирует список `rpath` для поддержки старых динамических компоновщиков, которые могут быть несовместимы с меткой `DT_RUNPATH` (поддержка ее была добавлена в glibc версии 2.2). Динамические компоновщики, совместимые с записью `DT_RUNPATH`, игнорируют `DT_RPATH` (см. раздел 41.11).

Использование переменной $ORIGIN в списке rpath

Представьте, что вам нужно распространять приложение, которое применяет собственные разделяемые библиотеки; при этом вы не хотите заставлять пользователя устанавливать их в один из стандартных каталогов. Вместо этого пользователь должен иметь возможность распаковать приложение в любом месте по своему усмотрению и сразу же его запустить. Проблема в том, что приложение не может определить местоположение своих разделяемых библиотек самостоятельно; мы должны попросить пользователя задать переменную `LD_LIBRARY_PATH` или предоставить небольшой установочный сценарий, ко-

торый будет определять соответствующие каталоги. Но ни один из указанных вариантов нам не подходит.

Для решения данной проблемы динамический компоновщик позволяет указать в параметре `-rpath` специальную строку, `$ORIGIN` (или `${ORIGIN}`), которую он умеет анализировать. Он интерпретирует ее как «каталог, содержащий приложение». Это значит, что мы, к примеру, можем собрать программу с помощью следующей команды:

```
$ gcc -Wl,-rpath,'$ORIGIN'/lib …
```

Здесь мы исходим из того, что во время выполнения разделяемые библиотеки будут доступны в подкаталоге `lib`, находящийся внутри каталога с нашим исполняемым файлом. Теперь можно предоставить пользователю простой установочный пакет с программой и нужными библиотеками, который можно установить в любое место и запускать оттуда программу (так называемое приложение под ключ).

41.11. Поиск разделяемых библиотек на этапе выполнения

При разрешении зависимостей библиотеки динамический компоновщик в первую очередь смотрит, содержит ли строка в списке `rpath` слеш (`/`); это бывает в тех случаях, когда при компоновке исполняемого файла был явно указан путь к библиотеке. Если слеш присутствует, то строка `rpath` интерпретируется в качестве пути (полного или относительного), по которому следует загружать библиотеку. В противном случае динамический компоновщик ищет библиотеку в соответствии со следующими правилами.

1. Если исполняемый файл содержит в своем списке `rpath` запись `DT_RPATH` с какими-либо каталогами и при этом не содержит списка `DT_RUNPATH`, то поиск будет выполнен по данным каталогам.
2. Если определена переменная среды `LD_LIBRARY_PATH`, то поиск будет выполнен последовательно по каждому каталогу, который в ней указан (и разделен двоеточиями). Если исполняемый файл устанавливает пользовательский или групповой идентификатор, то переменная `LD_LIBRARY_PATH` игнорируется. Это делается в целях безопасности, чтобы не дать пользователю обмануть динамический компоновщик, заставив его загрузить вместо требуемой библиотеки ее приватную версию с тем же именем.
3. Если в записи `DT_RUNPATH` списка `rpath` указаны какие-либо каталоги, то они используются во время поиска (в том порядке, в котором были перечислены во время компоновки программы).
4. Проверяется файл `/etc/ld.so.cache`, чтобы узнать, содержит ли он запись для соответствующей библиотеки.
5. Выполняется поиск по каталогам `/lib` и `/usr/lib` (именно в таком порядке).

41.12. Разрешение символов на этапе выполнения

Представьте, что глобальный символ (то есть функция или переменная) определен сразу в нескольких местах — например, в исполняемом файле и разделяемой библиотеке или в нескольких разных библиотеках. Как будет разрешена ссылка на этот символ?

Допустим, у нас есть главная программа и разделяемая библиотека, и в обеих определена глобальная функция `xyz()`, которая вызывается из другой библиотечной функции, как показано на рис. 41.5.

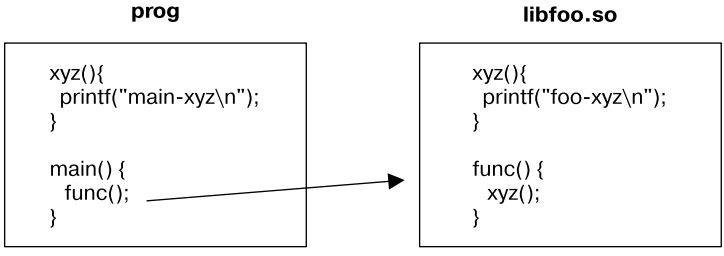


Рис. 41.5. Разрешение ссылки на глобальный символ

Собрав разделяемую библиотеку и исполняемый файл и затем запустив полученную программу, мы увидим следующее:

```
$ gcc -g -c -fPIC -Wall -c foo.c
$ gcc -g -shared -o libfoo.so foo.o
$ gcc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
main-xyz
```

В последней строчке мы видим, что определение `xyz()` из главной программы переопределяет (перекрывает) одноименную функцию в разделяемой библиотеке.

На первый взгляд это может показаться необычным, однако для такого поведения есть серьезные исторические причины. Первые реализации разделяемых библиотек были спроектированы так, что по умолчанию семантика процедуры разрешения символов в точности отражала ситуацию, когда приложение скомпоновано со статическими аналогами тех же библиотек. То есть в данном случае действует следующая логика:

- определение глобального символа в главной программе перекрывает определение в библиотеке;
- если глобальный символ определен сразу в нескольких библиотеках, то ссылка на него привязывается к первому определению, найденному в ходе сканирования библиотек, которое происходит в порядке их перечисления в командной строке во время статической компоновки (слева направо).

Приведенные правила обеспечивают относительно прямолинейный переход от статических к динамическим библиотекам, но могут вызвать ряд затруднений. Главная проблема такова: данная семантика несовместима с принципом, согласно которому разделяемая библиотека должна быть реализована в качестве самодостаточной подсистемы. Разделяемая библиотека по умолчанию не гарантирует, что ссылка на один из ее собственных глобальных символов будет привязана к ее же определению этого символа. Следовательно, свойства библиотеки могут измениться при включении ее в более крупный модуль. Это может привести к непредвиденным сбоям в приложении и усложнить раздельную отладку (например, когда вы пытаетесь воспроизвести проблему, используя другие разделяемые библиотеки или уменьшая их количество).

Для гарантии того, что в вышеописанном сценарии вызов `xyz()` в разделяемой библиотеке приведет к запуску именно той функции, которая в этой библиотеке определена, на этапе сборки компоновщику можно передать параметр `-Bsymbolic`:

```
$ gcc -g -c -fPIC -Wall -c foo.c
$ gcc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
$ gcc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
foo-xyz
```

Параметр компоновщика `-Bsymbolic` делает так, что ссылки на глобальный символ внутри разделяемой библиотеки в первую очередь должны привязываться к определению из этой библиотеки (если таковое существует). Стоит отметить: вне зависимости от данного параметра вызов `xyz()` из главной программы всегда приводит к запуску той версии `xyz()`, которая в ней определена.

41.13. Использование статической библиотеки вместо динамической

Почти во всех случаях разделяемые библиотеки являются наиболее предпочтительным выбором, однако иногда бывают ситуации, в которых более подходящей оказывается статическая библиотека. В частности, преимуществом статических библиотек может быть тот факт, что они позволяют скомпоновать в приложение весь код, необходимый ей для работы. Статическая компоновка, к примеру, подходит для пользователей, которые не могут или не хотят устанавливать в своей системе разделяемые библиотеки, или в случаях, когда программа должна выполняться в среде, где разделяемые библиотеки недоступны (например, в «тюрьме» `chroot`). Кроме того, даже обновление разделяемой библиотеки до совместимой версии может привести к ошибкам, вызывающим сбой приложения. Компонуя программу статически, мы защищаем ее от внешних изменений и гарантируем, что она содержит весь код, необходимый для ее выполнения (за счет увеличения ее размера и, как следствие, повышенного потребления дискового пространства и памяти).

Если в распоряжении компоновщика имеются оба типа библиотеки с тем же именем — статическая и разделяемая (например, когда во время компоновки указаны параметры `-Lsomedir -ldemo`, и при этом существуют файлы `libdemo.so` и `libdemo.a`), то он по умолчанию выбирает последнюю. Чтобы заставить его использовать статическую версию библиотеки, можно выполнить одно из следующих действий:

- указать путь к статической библиотеке (включая расширение `.a`) в командной строке `gcc`;
- указать для `gcc` параметр `-static`;
- задействовать параметры `gcc -Wl`, `-Bstatic` и `-Wl`, `-Bdynamic`, чтобы явно переключить компоновщик в режим использования статических библиотек. При этом можно применять параметры `-l` в командной строке `gcc`. Компоновщик обработает их в том порядке, в котором они были указаны.

41.14. Резюме

Объектная библиотека объединяет в себе скомпилированные объектные модули, которые могут быть использованы программами, скомпонованными с этой библиотекой. Как и другие реализации UNIX, Linux предоставляет два вида объектных библиотек: статические (не имевшие альтернативы в ранних UNIX-системах) и более современные разделяемые.

Разделяемые библиотеки имеют несколько преимуществ перед статическими, поэтому преобладают в современных реализациях UNIX. Данные преимущества по большей части следуют из того факта, что в итоговый исполняемый файл не записываются объектные модули библиотеки, с которой она скомпонована. Вместо этого (статический) компоновщик просто добавляет в файл программы информацию о разделяемых библиотеках, необходимые ей для работы. При запуске файла динамический компоновщик использует данную информацию для загрузки соответствующих библиотек. На этапе выполнения

в память загружается только одна копия разделяемой библиотеки, которая может применяться разными программами. Как следствие фактов, описанных выше, разделяемые библиотеки уменьшают объем дискового пространства и памяти, необходимые системе.

Имя soname предоставляет уровень абстракции для разрешения ссылок на разделяемые библиотеки во время выполнения. Если оно присутствует, то записывается статическим компоновщиком в итоговый исполняемый файл вместо реального имени библиотеки. Существует также система версионирования, когда реальное имя библиотеки имеет вид `имя_библиотеки.so.мажорный_идентификатор.минорный_идентификатор`. Это позволяет создавать программы, которые автоматически загружают последнюю минорную версию разделяемой библиотеки (без необходимости проводить повторную компоновку); благодаря этому также можно генерировать мажорные версии библиотеки, несовместимые с предыдущими.

Для нахождения разделяемой библиотеки на этапе выполнения динамический компоновщик следует стандартному набору правил, подразумевающему поиск по списку каталогов (таких как `/lib` и `/usr/lib`), в которых эта библиотека может быть установлена.

Дополнительная информация

Различную информацию, связанную со статическими и разделяемыми библиотеками, можно найти на справочных страницах `ar(1)`, `gcc(1)`, `ld(1)`, `ldconfig(8)`, `ld.so(8)`, `dlopen(3)` и `objdump(1)`, а также в документации `info` к командам `ld` и `readelf`. В книге [Drepper, 2004 (b)] освещается множество тонких нюансов написания разделяемых библиотек в Linux. Больше полезных сведений содержится в руководстве Program Library HOWTO Дэвида Уилера, доступном на сайте проекта LDP, www.tldp.org. Механизм разделяемых библиотек GNU во многом похож на аналогичную реализацию в операционной системе Solaris, поэтому вам стоит прочесть руководство по работе с компоновщиком и библиотеками (Linker and Libraries Guide) от компании Sun (доступном на docs.sun.com), в котором содержится дополнительная информация и примеры. [Levine, 2000] предоставляет введение в работу статических и динамических компоновщиков.

Информацию о GNU Libtool, инструменте, который скрывает от программиста детали реализации разделяемых библиотек, можно найти на www.gnu.org/software/libtool и в [Vaughan et al., 2000].

Документ Executable and Linking Format (формат исполняемых и компонуемых файлов), опубликованный комитетом Tools Interface Standards, содержит подробности о формате ELF; ознакомиться с ним можно на refspecs.freestandards.org/elf/elf.pdf. Множество полезных нюансов об этом формате можно также почерпнуть из [Lu, 1995].

41.15. Упражнение

41.1. Попробуйте скомпилировать программу с параметром `-static` и без него, чтобы увидеть разницу в размере исполняемых файлов, один из которых скомпонован с библиотекой языка C динамически, а другой — статически.

42 Продвинутые возможности разделяемых библиотек

Предыдущая глава была посвящена основным моментам, связанным с разделяемыми библиотеками. Теперь пришло время познакомиться с их продвинутыми возможностями, включая такие, как:

- динамическая загрузка разделяемых библиотек;
- управление видимостью символов, определенных в разделяемой библиотеке;
- использование компоновочных сценариев для создания версионных символов;
- применение функций инициализации и финализации для автоматического выполнения кода при загрузке и выгрузке библиотеки;
- предварительная загрузка разделяемой библиотеки;
- использование переменной `LD_DEBUG` для отслеживания работы динамического компоновщика.

42.1. Динамически загружаемые библиотеки

При запуске исполняемого файла динамический компоновщик загружает все разделяемые библиотеки, указанные в списке динамических зависимостей программы. Но иногда может понадобиться загрузить библиотеку чуть позже. Например, подключаемый модуль загружается, только когда нужен. Эта возможность предоставляется программным интерфейсом динамического компоновщика. Данный интерфейс обычно называют `dlopen`; изначально он был разработан для системы Solaris, но теперь большая его часть описана в стандарте SUSv3.

Интерфейс `dlopen` позволяет программе открыть разделяемую библиотеку во время выполнения, найти в ней функцию с подходящим именем и вызвать ее. Библиотеки, которые используются таким образом, обычно называют *динамически загружаемыми*; при этом они создаются точно так же, как и любые другие разделяемые библиотеки.

Основу `dlopen` составляют представленные ниже функции (все они входят в стандарт SUSv3):

- `dlopen()` — открывает разделяемую библиотеку и возвращает дескриптор, который можно использовать в последующих вызовах;
- `dlsym()` — ищет в библиотеке определенный символ (строку, содержащую имя функции или переменной) и возвращает его адрес;
- `dlclose()` — закрывает библиотеку, открытую ранее вызовом `dlopen()`;
- `dlerror()` — возвращает строку с сообщением об ошибке и применяется после неудачного завершения одной из вышеописанных функций.

Реализация glibc также содержит целый ряд дополнительных функций; некоторые из них будут описаны ниже.

Чтобы собрать в Linux программу, использующую программный интерфейс `dlopen`, нужно указать параметр `-ldl`; это позволит скомпоновать ее с библиотекой libdl.

42.1.1. Открытие разделяемой библиотеки: dlopen()

Функция `dlopen()` загружает в виртуальное адресное пространство вызывающего процесса разделяемую библиотеку с именем `libfilename` и инкрементирует счетчик открытых ссылок на нее.

```
#include <dlfcn.h>

void *dlopen(const char *libfilename, int flags);
```

Возвращает дескриптор библиотеки при успешном завершении или `NULL` при ошибке

Если имя `libfilename` содержит слеш (`/`), то `dlopen()` интерпретирует его как относительный или полный путь. В противном случае динамический компоновщик ищет разделяемую библиотеку по принципу, описанному в разделе 41.11.

В случае успеха `dlopen()` возвращает дескриптор, по которому можно ссылаться на библиотеку в последующих вызовах функций из программного интерфейса `dlopen`. Если произойдет ошибка (например, библиотеку не удалось найти), то `dlopen()` возвращает `NULL`.

Если разделяемая библиотека, указанная с помощью аргумента `libfilename`, зависит от других библиотек, то `dlopen()` загрузит их автоматически. При необходимости эта процедура выполняется рекурсивно. Мы будем называть набор загруженных таким образом библиотек *деревом зависимостей*.

Функцию `dlopen()` можно вызвать несколько раз для одной и той же библиотеки. При этом загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях станет возвращаться одно и то же значение `handle`. Однако программный интерфейс `dlopen` хранит счетчик ссылок для каждого дескриптора. С каждым вызовом `dlopen()` он инкрементируется, а декрементация происходит при вызове `dlclose()`; последний выгружает библиотеку из памяти только в том случае, если счетчик равен 0.

Аргумент `flags` представляет собой битовую маску, значение которой должно быть равно либо `RTLD_LAZY`, либо `RTLD_NOW`. Эти константы описаны ниже.

- `RTLD_LAZY` — неопределенные ссылки на функции библиотеки должны быть разрешены только при выполнении соответствующего кода. Если участок кода, которому требуется определенный символ, не выполняется, то данный символ не разрешается. Отложенное разрешение производится только для ссылок на функции; ссылки на переменные всегда разрешаются незамедлительно. Наличие флага `RTLD_LAZY` обеспечивает поведение, соответствующее обычной работе динамического компоновщика, когда тот загружает разделяемые библиотеки из списка динамических зависимостей исполняемого файла.
- `RTLD_NOW` — все неопределенные ссылки библиотеки должны быть немедленно разрешены вне зависимости от того, понадобятся ли они когда-нибудь; данные операции нужно произвести до завершения вызова `dlopen()`. Это замедляет загрузку библиотеки, но позволяет сразу же определить все ошибки, связанные со ссылками на функции. Такой подход может оказаться полезным при отладке приложения или в случае, когда при обнаружении неразрешенного символа программа должна завершиться немедленно, а не в ходе дальнейшего выполнения.

Присвоив переменной среды LD_BIND_NOW любое значение, кроме пустой строки, мы можем заставить динамический компоновщик немедленно выполнить поиск всех символов

(то есть как и в случае с RTLD_NOW) при загрузке разделяемой библиотеки, указанной в списке динамических зависимостей исполняемого файла. Эта переменная поддерживается в glibc 2.1.1 и выше. Ее установка нивелирует действие флага RTLD_LAZY при вызове dlopen().

В маску `flags` также можно включить дополнительные значения, описанные в стандарте SUSv3:

- `RTLD_GLOBAL` — символы в текущей библиотеке и ее дереве зависимостей можно задействовать для разрешения ссылок в других библиотеках, загруженных тем же процессом, а также для поиска с помощью вызова `dlsym()`;
- `RTLD_LOCAL` — это значение используется по умолчанию (если не указаны другие константы) и имеет эффект, противоположный `RTLD_GLOBAL`. Оно делает так, что символы в текущей библиотеке и ее дереве зависимостей становятся недоступными для разрешения ссылок в загружаемых библиотеках.

В стандарте SUSv3 не уточняется, какое значение используется по умолчанию, если ни одна из констант (`RTLD_GLOBAL` или `RTLD_LOCAL`) не указана явно. В большинстве реализаций UNIX, как и в Linux, по умолчанию устанавливается флаг `RTLD_LOCAL`, но есть и такие, в которых вместо этого задействован `RTLD_GLOBAL`.

Linux также поддерживает несколько нестандартных флагов.

- `RTLD_NODELETE` (начиная с glibc 2.2) — предотвращает выгрузку библиотеки при вызове `dlclose()`, даже если счетчик ссылок равен 0. Это значит, что статические переменные библиотеки не будут заново инициализированы при следующем вызове `dlopen()` (для библиотек, загружаемых автоматически динамическим компоновщиком, подобного эффекта можно достичь, используя параметр `gcc -Wl,-znodelete` на этапе их создания).
- `RTLD_NOLOAD` (начиная с glibc 2.2) — не дает загрузить библиотеку. Такое поведение имеет две цели. Во-первых, с помощью этого флага можно проверить, загружена ли заданная библиотека в адресное пространство процесса. В случае положительного ответа `dlopen()` возвращает ее дескриптор; в противном случае возвращается `NULL`. Во-вторых, данный флаг позволяет применить аргумент `flags` уже загруженной библиотеки. Например, можно указать в маске `flags` значение `RTLD_NOLOAD | RTLD_GLOBAL` при вызове `dlopen()` для библиотеки, открытой ранее с флагом `RTLD_LOCAL`.
- `RTLD_DEEPBIND` (начиная с glibc 2.3.4) — при разрешении символьных ссылок, созданных текущей библиотекой, поиск определений выполняется сначала в данной библиотеке и только потом в библиотеках, загруженных ранее. Это позволяет сделать библиотеку самодостаточной и отдавать приоритет собственным символам перед глобальными, если нужное имя встречается в уже загруженных библиотеках (похоже на то, как действует параметр компоновщика `-Bsymbolic`, описанный в разделе 41.12).

Флаги `RTLD_NODELETE` и `RTLD_NOLOAD` также реализованы в программном интерфейсе `dlopen` нескольких других операционных систем, включая Solaris. Флаг `RTLD_DEEPBIND` поддерживается только в Linux.

В особых случаях можно присвоить `libfilename` значение `NULL`. Данное действие заставит функцию `dlopen()` вернуть дескриптор главной программы (в стандарте SUSv3 это называется глобальным символьным объектом). Если указать этот дескриптор в последующем вызове `dlsym()`, поиск запрашиваемого символа в первую очередь будет проходить в главной программе, затем в разделяемых библиотеках, загруженных при запуске программы, и только потом в библиотеках, загруженных динамически с помощью флага `RTLD_GLOBAL`.

42.1.2. Анализ ошибок: dlerror()

При получении ошибки из `dlopen()` или другой функции, входящей в программный интерфейс `dlopen`, можно попытаться узнать ее причину, получив указатель на соответствующую строку, используя вызов `dlerror()`.

```
#include <dlfcn.h>

const char *dlerror(void);
```

Возвращает указатель на строку с описанием ошибки или `NULL`, если с момента последнего вызова `dlerror()` никаких ошибок не было

Функция `dlerror()` возвращает `NULL`, если с момента ее последнего вызова не возникало никаких ошибок. В следующем разделе вы увидите, как это может помочь на практике.

42.1.3. Получение адреса символа: dlsym()

Функция `dlsym()` ищет именованный символ (`symbol` — функцию или переменную) в библиотеке, на которую указывает дескриптор (`handle`), и в ее дереве зависимостей.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
```

Возвращает адрес символа или `NULL`, если символ не был найден

Если символ найден, `dlsym()` возвращает его адрес; в противном случае возвращается `NULL`. В качестве аргумента `handle` обычно выступает дескриптор библиотеки, возвращенный предыдущим вызовом `dlopen()`. Но это может быть и один из так называемых псевдодескрипторов, описанных ниже.

> У dlsym() есть родственная функция, dlvsym(handle, symbol, version), которая имеет похожее назначение, но может применяться для поиска в библиотеке версионных символов, чьи версии совпадают со значением аргумента version (версионирование символов будет описано в подразделе 42.3.2). Для получния объявления этой функции из заголовочного файла <dlfcn.h> нужно определить макрос проверки возможностей _GNU_SOURCE.

Значение символа, возвращенного функцией `dlsym()`, может быть равно `NULL`; тот же результат мы получим, если символ не был найден. Чтобы различать эти два случая, следует заранее вызвать `dlerror()` (для очистки любых ошибок, которые накопились до сего момента); затем, если повторный вызов `dlerror()`, сделанный после `dlsym()`, вернет ненулевое значение, то мы будем знать, что произошла ошибка.

При наличии в аргументе `symbol` имени переменной можно присвоить результат, возвращенный `dlsym()`, указателю соответствующего типа и, разыменовав его, извлечь значение этой переменной:

```
int *ip;
ip = (int *) dlsym(symbol, "myvar");
if (ip != NULL)
    printf("Value is %d\n", *ip);
```

Если аргумент `symbol` содержит имя функции, то эту функцию можно вызвать с помощью указателя, полученного в результате вызова `dlsym()`. Результат выполнения `dlsym()` можно поместить в указатель подходящего типа, как показано ниже:

```
int (*funcp)(int);    /* Указатель на функцию, принимающую целочисленный
                         аргумент и возвращающую целочисленный результат */
```

Однако мы не можем просто присвоить результат выполнения `dlsym()` такому указателю, как показано ниже:

```
funcp = dlsym(handle, symbol);
```

Причина в том, что стандарт C99 запрещает операцию присваивания между указателем на функцию и `void *`. В качестве решения можно воспользоваться (немного грубым) приведением типов:

```
*(void **) (&funcp) = dlsym(handle, symbol);
```

Получив указатель на функцию с помощью `dlsym()`, можно вызвать ее путем обычной для языка C операции разыменовывания:

```
res = (*funcp)(somearg);
```

Вместо синтаксиса вида `*(void **)`, приведенного выше, для присваивания значения, возвращенного функцией `dlsym()`, можно воспользоваться почти равнозначной альтернативой:

```
(void *) funcp = dlsym(handle, symbol);
```

Но если при компиляции задействовать параметр `gcc -pedantic`, то для этого кода будет выдано предупреждение `ANSI C forbids the use of cast expressions as lvalues` (Стандарт ANSI C запрещает использование приведения типов в качестве значений lvalue). Выражение `*(void **)` будет интерпретировано без замечаний, поскольку оно присваивает данные по адресу, *на который указывает* значение `lvalue`.

Во многих реализациях UNIX можно избавиться от предупреждений компилятора, применив следующее приведение типов:

```
funcp = (int (*) (int)) dlsym(handle, symbol);
```

Однако в первой технической поправке к стандарту SUSv3 (Technical Corrigendum Number 1, TC1), касающейся функции `dlsym()`, отмечается, что стандарт C99 требует, чтобы для подобных преобразований компиляторы выводили предупреждение, и предлагается использовать вместо этого синтаксис `*(void **)`, описанный выше.

> В поправке SUSv3 TC1 также отмечается, что из-за необходимости применения операции *(void **) будущая версия стандарта может содержать отдельный программный интерфейс, похожий на dlsym() и предназначенный для работы с указателями на данные и функции. Однако в стандарте SUSv4 никаких подобных изменений не предусмотрено.

Использование псевдодескрипторов в сочетании с функцией dlsym()

Вместо дескрипторов, возвращаемых вызовом `dlopen()`, аргументу `handle` функции `dlsym()` можно передать один из следующих *псевдодескрипторов*.

- `RTLD_DEFAULT` — поиск символа начинается с главной программы, после чего проходит по списку всех разделяемых библиотек, в том числе и загруженных динамически с помощью вызова `dlopen()` с флагом `RTLD_GLOBAL`. Это аналогично стандартному алгоритму поиска, который применяется динамическим компоновщиком.

- RTLD_NEXT — поиск символа выполняется по библиотекам, загруженным после вызова dlsym(). Это может пригодиться при создании функций-оберток, чьи имена совпадают с именами каких-то других функций, определенных где-либо. Например, можно определить в главной программе собственную версию функции malloc() (возможно, ведущую учет выделяемой памяти), которая будет вызывать одноименный оригинал; для этого она должна получить его адрес с помощью вызова func = dlsym(RTLD_NEXT, "malloc").

Значения псевдодескрипторов, описанные выше, не являются обязательными с точки зрения стандарта SUSv3 (хотя в нем они зарезервированы для будущего использования) и доступны не во всех UNIX-системах. Чтобы получить определение этих констант из заголовочного файла <dlfcn.h>, следует сперва определить макрос проверки возможностей _GNU_SOURCE.

Пример программы

Применение программного интерфейса dlopen продемонстрировано в листинге 42.1. Эта программа принимает два аргумента командной строки: имя разделяемой библиотеки, которую нужно загрузить, и имя функции, которую нужно вызвать из этой библиотеки. Ниже показаны примеры запуска данной программы:

```
$ ./dynload ./libdemo.so.1 x1
Called mod1-x1
$ LD_LIBRARY_PATH=. ./dynload libdemo.so.1 x1
Called mod1-x1
```

В первой команде функция dlopen() обнаруживает в названии библиотеки слеш и интерпретирует его как относительный путь (в данном случае это путь к библиотеке в текущем каталоге). Во второй команде мы указали путь поиска с помощью переменной LD_LIBRARY_PATH. Данный путь интерпретируется согласно стандартным правилам, которым следует динамический компоновщик (в нашем случае библиотека ищется в текущем каталоге).

Листинг 42.1. Использование программного интерфейса dlopen

shlibs/dynload.c

```
#include <dlfcn.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    void *libHandle;        /* Дескриптор для разделяемой библиотеки */
    void (*funcp)(void);    /* Указатель на функцию без аргументов */
    const char *err;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s lib-path func-name\n", argv[0]);

    /* Загружаем разделяемую библиотеку и получаем дескриптор
       для ее дальнейшего использования */

    libHandle = dlopen(argv[1], RTLD_LAZY);
    if (libHandle == NULL)
        fatal("dlopen: %s", dlerror());
    /* Ищем в библиотеке символ с именем, заданным в argv[2] */

     (void) dlerror();       /* Очищаем ошибки с помощью dlerror() */
```

```
    *(void **) (&funcp) = dlsym(libHandle, argv[2]);
    err = dlerror();
    if (err != NULL)
        fatal("dlsym: %s", err);

    /* Если адрес, возвращенный dlsym(), не равен NULL, пытаемся вызвать
       это значение как функцию, которая не принимает аргументов */

    if (funcp == NULL)
        printf("%s is NULL\n", argv[2]);
    else
        (*funcp)();

    dlclose(libHandle);                /* Закрываем библиотеку */

    exit(EXIT_SUCCESS);
}
```

shlibs/dynload.c

42.1.4. Закрытие разделяемой библиотеки: dlclose()

Функция `dlclose()` закрывает библиотеку.

```
include <dlfcn.h>

int dlclose(void *handle);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `dlclose()` декрементирует системный счетчик открытых ссылок на библиотеку, на которую указывает дескриптор `handle`. Когда счетчик доходит до нуля и ни один символ библиотеки больше не используется извне, библиотека выгружается. Такая же процедура выполняется (рекурсивно) для всех библиотек, входящих в ее дерево зависимостей. Во время завершения процесса вызов `dlclose()` автоматически выполняется для всех библиотек.

Начиная с glibc 2.2.3, разделяемая библиотека может совершить вызов atexit() (или on_exit()), чтобы установить функцию, которая будет автоматически запускаться при ее выгрузке.

42.1.5. Получение информации о загруженных символах: dladdr()

Возвращает структуру с информацией об адресе, заданном в аргументе `addr` (который обычно берется из ранее выполненного вызова `dlsym()`).

```
#define _GNU_SOURCE
#include <dlfcn.h>

int dladdr(const void *addr, Dl_info *info);
```

Возвращает ненулевое значение, если адрес `addr` был найден в разделяемой библиотеке; иначе возвращает `0`

Аргумент `info` представляет собой указатель на структуру, выделенную вызывающим кодом и имеющую такой вид:

```
typedef struct {
    const char *dli_fname;   /* Путь к разделяемой библиотеке, содержащей 'addr' */
    void       *dli_fbase;   /* Базовый адрес, по которому
                                загружена разделяемая библиотека */
    const char *dli_sname;   /* Имя ближайшего к 'addr' символа (во время выполнения) */
    void       *dli_saddr;   /* Значение символа, возвращенного в поле 'dli_sname' */
} Dl_info;
```

Первые два поля структуры `Dl_info` содержат путь к найденной разделяемой библиотеке и ее базовый адрес. Сведения об этом адресе находятся в последних двух полях. Если аргумент `addr` указывает на точный адрес библиотечного символа, `dli_saddr` будет иметь то же значение, которое было указано с помощью `addr`.

Функция `dladdr()` не входит в состав стандарта SUSv3 и поддерживается не всеми реализациями UNIX.

42.1.6. Получение доступа к символам главной программы

Представьте что мы динамически загрузили разделяемую библиотеку, используя `dlopen()`, и применяем `dlsym()` для получения адреса функции `x()` (которая принадлежит этой библиотеке) и последующего ее вызова. Если `x()`, в свою очередь, вызовет функцию `y()`, ее поиск будет происходить в одной из разделяемых библиотек, загруженных программой.

Но иногда бывает нужно, чтобы функция `x()` вызвала реализацию `y()` из главной программы (похоже на то, как работает механизм обратного вызова). Для этого мы должны сделать (глобальные) символы в главной программе доступными для динамического компоновщика, воспользовавшись при компоновке параметром `--export-dynamic`:

```
$ gcc -Wl,--export-dynamic main.c Плюс дополнительные параметры и аргументы
```

То же самое можно сделать с помощью следующей команды:

```
$ gcc -export-dynamic main.c
```

Любой из этих вариантов позволит динамически загруженной библиотеке получать доступ к глобальным символам в главной программе.

> Параметры gcc -rdynamic и gcc -Wl,-E являются дополнительными синонимами для -Wl,--export-dynamic.

42.2. Управление видимостью символов

Хорошо спроектированная библиотека должна оставлять видимыми только те символы (функции и переменные), которые составляют часть ее двоичного программного интерфейса (ABI). Этому есть несколько причин.

- Если разработчик библиотеки случайно экспортирует неустановленный интерфейс, авторы приложений, работающие с такой библиотекой, могут этим воспользоваться. Могут возникнуть проблемы с совместимостью в будущих обновлениях разделяемой библиотеки. Ее разработчик рассчитывает на наличие у него возможности изменять или удалять любые интерфейсы, не входящие в задокументированный ABI, а ее пользователи исходят из того, что они смогут продолжить использовать тот же интерфейс (с той же семантикой).

- Во время разрешения символов любая найденная функция или переменная, экспортируемая разделяемой библиотекой, может перекрыть собой определение, доступное в других библиотеках (см. раздел 41.12).
- Экспорт лишних интерфейсов увеличивает размер таблицы динамических символов, которая должна быть загружена при запуске программы.

Все эти проблемы можно минимизировать или вовсе устранить, если при разработке библиотеки экспортировать только символы, входящие в ее задокументированный двоичный программный интерфейс. Для управления данным процессом можно воспользоваться следующими методиками.

- В языке C можно использовать ключевое слово `static`, которое делает символ приватным относительно модуля с исходным кодом и недоступным для обращения со стороны других объектных файлов.

  Ключевое слово static имеет побочный эффект. Все ссылки на символ, помеченный им, будут привязаны к определению этого символа (при условии, что они находятся в том же исходном файле). Следовательно, такие ссылки не будут перекрываться определениями из других разделяемых библиотек (как это описано в разделе 41.12). Данное свойство ключевого слова static похоже на действие параметра компоновщика -Bsymbolic, описанного в разделе 41.12, с той лишь разницей, что ключевое слово static влияет только на один символ внутри одного исходного файла.

- Компилятор GNU языка C, `gcc`, предоставляет нестандартное определение атрибута, который выполняет то же действие, что и ключевое слово `static`:

  ```
  void
  __attribute__ ((visibility("hidden")))
  func(void) {
      /* Код */
  }
  ```

  Если ключевое слово `static` ограничивает видимость символа единственным исходным файлом, то атрибут `hidden` делает символ доступным во всех исходных файлах разделяемой библиотеки, но не за ее пределами.

  Как и ключевое слово static, атрибут hidden имеет побочный эффект, не давая перекрывать символ во время выполнения.

- Версионные сценарии (см. раздел 42.3) предоставляют полный контроль над видимостью символов и возможность выбирать те их версии, к которым будет привязана ссылка.
- Если при динамической загрузке разделяемой библиотеки (см. подраздел 42.1.1) указать в функции `dlopen()` флаг `RTLD_GLOBAL`, то символы, определенные в этой библиотеке, будут доступны для привязки только тем библиотекам, которые загружаются после нее. Параметр компоновщика `--export-dynamic` позволяет сделать глобальные символы главной программы доступными для динамически загружаемых библиотек. Больше подробностей о видимости символов см. в [Drepper, 2004 (b)].

42.3. Версионные сценарии компоновщика

Версионный сценарий — это текстовый файл, содержащий инструкции для компоновщика, `ld`. Чтобы использовать версионный сценарий, компоновщику следует указать параметр `--version-script`:

```
$ gcc -Wl,--version-script,myscriptfile.map ...
```

Версионные сценарии обычно (но не всегда) имеют расширение `.map`. Некоторые из способов их применения описаны в следующих подразделах.

42.3.1. Управление видимостью символов с помощью версионных сценариев

Одно из применений версионных сценариев заключается в управлении видимостью символов, которые могут случайно стать глобальными (то есть доступными для приложения, скомпонованного с этой библиотекой). В качестве простого примера рассмотрим такую ситуацию: мы собираем разделяемую библиотеку из трех исходных файлов — `vis_comm.c`, `vis_f1.c` и `vis_f2.c`, в которых определены функции `vis_comm()`, `vis_f1()` и `vis_f2()` соответственно. Функция `vis_comm()` не рассчитана на непосредственный доступ из приложений, скомпонованных с библиотекой, и должна вызываться из `vis_f1()` и `vis_f2()`. Представьте, что мы собираем эту библиотеку обычным способом:

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o
```

Если вывести с помощью следующей команды `readelf` динамические символы, экспортируемые библиотекой, то можно увидеть следующее:

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   30  12:  00000790   59    FUNC  GLOBAL  DEFAULT   10  vis_f1
   25  13:  000007d0   73    FUNC  GLOBAL  DEFAULT   10  vis_f2
   27  16:  00000770   20    FUNC  GLOBAL  DEFAULT   10  vis_comm
```

Эта разделяемая библиотека экспортирует три символа: `vis_comm()`, `vis_f1()` и `vis_f2()`. Однако нам нужно, чтобы экспортировались только символы `vis_f1()` и `vis_f2()`. Для этого можно воспользоваться следующим версионным сценарием:

```
$ cat vis.map
VER_1 {
    global:
        vis_f1;
        vis_f2;
    local:
        *;
};
```

Идентификатор `VER_1` является примером *версионной метки*. Как вы увидите в подразделе 42.3.2, версионный сценарий может содержать множество таких разделов, заключенных в фигурные скобки (`{}`) и обозначенных уникальной версионной меткой. Если нужно всего лишь управлять видимостью символов, то данную метку можно опустить; так позволено делать в современных версиях утилиты `ld`, хотя старые версии этого компоновщика требуют ее наличия. В данном случае мы используем анонимную версионную метку, при этом в сценарии больше не должно содержаться ни одного другого раздела.

Внутри раздела находится ключевое слово `global`, после которого через точку с запятой перечисляются символы, видимые за пределами библиотеки. Ключевое слово `local` описывает символы, не скрытые от внешнего мира. Для этого можно использовать шаблоны, такие как звездочка (`*`). Данные шаблоны аналогичны тем, что применяются при поиске по именам файлов — например, `*` и `?` (больше подробностей см. на странице `glob(7)` руководства). В нашем примере звездочка в списке `local` говорит о том, что все символы, не перечисленные в подразделе `global`, будут скрыты. Без этого функция `vis_comm()` осталась

бы видимой, поскольку глобальные символы в языке C по умолчанию доступны внешним разделяемым библиотекам.

Теперь можно собрать разделяемую библиотеку, используя версионный сценарий:

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o \
    -Wl,--version-script,vis.map
```

При повторном вызове команды `readelf` можно увидеть, что функция `vis_comm()` больше не доступна извне:

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   25   0: 00000730    73   FUNC GLOBAL  DEFAULT   11  vis_f2
   29  16: 000006f0    59   FUNC GLOBAL  DEFAULT   11  vis_f1
```

42.3.2. Версионирование символов

Версионирование символов позволяет разделяемой библиотеке предоставлять разные версии одной и той же функции. Каждая программа использует функцию той версии, которая была текущей на момент ее (статической) компоновки с библиотекой. Благодаря этому можно вносить в библиотеку несовместимые изменения, не увеличивая номер ее мажорной версии. В крайнем случае версионирование символов может заменить собой традиционный механизм назначения мажорных и минорных версий. Такое применение данной технологии практикуется в библиотеке glibc 2.3 и выше; это позволило сделать все ее версии, начиная с 2.0, совместимыми в рамках единого мажорного номера (`libc.so.6`).

Применение версионирования символов показано в следующем примере. Начнем с создания первой версии разделяемой библиотеки с помощью версионного сценария:

```
$ cat sv_lib_v1.c
#include <stdio.h>

void xyz(void) { printf("v1 xyz\n"); }
$ cat sv_v1.map
VER_1 {
        global: xyz;
        local: *;        # Скрываем любые другие символы
};
$ gcc -g -c -fPIC -Wall sv_lib_v1.c
$ gcc -g -shared -o libsv.so sv_lib_v1.o -Wl,--version-script,sv_v1.map
```

В версионных сценариях знак решетки (#) обозначает начало комментария.

Чтобы упростить этот пример, мы не станем вручную указывать имена soname и номера мажорных версий библиотеки.

На данном этапе версионный сценарий `sv_v1.map` используется только для управления видимостью символов разделяемой библиотеки; экспортируется лишь функция `xyz()`, тогда как все остальные символы (а таковых в нашем небольшом примере просто нет) скрываются. Теперь создадим программу `p1`, которая будет применять эту библиотеку:

```
$ cat sv_prog.c
#include <stdlib.h>

int
main(int argc, char *argv[])
```

```
{
    void xyz(void);

    xyz();

    exit(EXIT_SUCCESS);
}
$ gcc -g -o p1 sv_prog.c libsv.so
```

Запустив эту программу, мы увидим ожидаемый результат:

```
$ LD_LIBRARY_PATH=. ./p1
v1 xyz
```

Теперь предположим, что нам нужно изменить определение `xyz()` внутри библиотеки, но при этом программа `p1` должна иметь возможность использовать старую версию данной функции. Необходимо определить в библиотеке две версии `xyz()`:

```
$ cat sv_lib_v2.c
#include <stdio.h>

__asm__(".symver xyz_old,xyz@VER_1");
__asm__(".symver xyz_new,xyz@@VER_2");

void xyz_old(void) { printf("v1 xyz\n"); }

void xyz_new(void) { printf("v2 xyz\n"); }

void pqr(void) { printf("v2 pqr\n"); }
```

Две версии `xyz()` доступны с помощью функций `xyz_old()` и `xyz_new()`. Первая из них соответствует оригинальной версии `xyz()`, которая будет и дальше использоваться программой `p1`. Вторая предоставляет определение `xyz()`, доступное программам, скомпонованным с новой версией библиотеки.

Две директивы `.symver` связывают эти функции с разными версионными метками в модифицированном версионном сценарии, который мы задействуем для создания новой версии разделяемой библиотеки (показан чуть ниже). Первая директива говорит о том, что `xyz_old()` является реализацией функции `xyz()`, которая будет использоваться в программах, скомпонованных с версионной меткой `VER_1` (в нашем примере это программа `p1`), и что `xyz_new()` — реализация новой версии `xyz()`, доступной для программ, скомпонованных с меткой `VER_2`.

Во второй директиве `.symver` применяется обозначение `@@` вместо `@`; это говорит о том, что приложения, скомпонованные с библиотекой статически, должны быть привязаны по умолчанию именно к данному определению `xyz()`. Только одна из директив `.symver` должна быть помечена с помощью символов `@@`.

Версионный сценарий для нашей измененной библиотеки будет выглядеть так:

```
$ cat sv_v2.map
VER_1 {
        global: xyz;
        local: *;          # Скрываем любые другие символы
};

VER_2 {
        global: pqr;
} VER_1;
```

Данный сценарий содержит новую версионную метку, `VER_2`, которая зависит от метки `VER_1`. Эта зависимость обозначена следующей строчкой:

```
} VER_1;
```

Зависимости версионных меток отражают связи между соседними версиями библиотеки. С точки зрения семантики единственный эффект от зависимостей версионных меток в Linux состоит в том, что версионный раздел наследует ключевые слова `global` и `local` от раздела, от которого он зависит.

Зависимости можно объединять. К примеру, мы могли бы создать еще один версионный раздел `VER_3`, зависящий от `VER_2`, и т. д.

Имена версионных меток сами по себе ничего не значат. Их связи друг с другом определяются только их зависимостями; имена `VER_1` и `VER_2` были выбраны только для того, чтобы эти связи были более наглядными. Для упрощения администрирования в имена версионных меток рекомендуется включать имя пакета и номер версии. Например, версионные метки в glibc имеют следующий вид: `GLIBC_2.0`, `GLIBC_2.1` и т. д.

Версионная метка `VER_2` также указывает на то, что к ней будет привязана новая функция `pqr()`, которая экспортируется библиотекой. Если не объявить функцию `pqr()` таким способом, то ключевое слово `local`, наследованное меткой `VER_2` от `VER_1`, сделает ее невидимой за пределами библиотеки. Стоит также отметить: если бы мы просто опустили ключевое слово `local`, то символы `xyz_old()` и `xyz_new()` тоже были бы экспортированы (обычно это нежелательно).

Теперь соберем новую версию нашей библиотеки уже привычным способом:

```
$ gcc -g -c -fPIC -Wall sv_lib_v2.c
$ gcc -g -shared -o libsv.so sv_lib_v2.o -Wl,--version-script,sv_v2.map
```

Теперь можно создать новую программу, `p2`, которая использует новое определение функции `xyz()`; при этом программа `p1` будет применять ее старую версию.

```
$ gcc -g -o p2 sv_prog.c libsv.so
$ LD_LIBRARY_PATH=. ./p2
v2 xyz                                  Использует xyz@VER_2
$ LD_LIBRARY_PATH=. ./p1
v1 xyz                                  Использует xyz@VER_1
```

Зависимости версионной метки записываются в исполняемый файл на этапе статической компоновки. Если вывести с помощью команды `objdump -t` таблицы символов для обеих программ, то получим разные зависимости:

```
$ objdump -t p1 | grep xyz
08048380       F *UND* 0000002e            xyz@@VER_1
$ objdump -t p2 | grep xyz
080483a0       F *UND* 0000002e            xyz@@VER_2
```

Похожую информацию можно также получить, задействуя команду `readelf -s`.

Дополнительные сведения о версионировании символов можно найти на https://www.akkadia.org/drepper/symbol-versioning или воспользовавшись командой info ld scripts.

42.4. Инициализация и финализация функций

Можно определить одну или несколько функций, которые будут автоматически вызываться при загрузке и выгрузке разделяемой библиотеки. Это позволит выполнять

действия по инициализации и финализации во время работы с библиотеками. Функции инициализации и финализации вызываются вне зависимости от того, загружена библиотека автоматически или вручную, используя интерфейс `dlopen` (см. раздел 42.1).

Функции инициализации и финализации определяются с помощью атрибутов `constructor` и `destructor` компилятора `gcc`. Любую функцию, которую нужно выполнить при загрузке библиотеки, следует определить таким образом:

```
void __attribute__ ((constructor)) some_name_load(void)
{
    /* Код инициализации */
}
```

Функции выгрузки имеют похожее определение:

```
void __attribute__ ((destructor)) some_name_unload(void)
{
    /* Код финализации */
}
```

Вместо `some_name_load()` и `some_name_unload()` можно использовать любые другие имена на ваш выбор.

> Атрибуты constructor и destructor компилятора gcc можно также применять для создания функций инициализации и финализации в главной программе.

Функции _init() и _fini()

Существует и более старый способ инициализации и финализации разделяемых библиотек. Он заключается в создании внутри библиотеки двух функций, `_init()` и `_fini()`. Функция `void _init(void)` содержит код, который выполняется, когда библиотека впервые загружается процессом. Код функции `void _fini(void)` выполняется при выгрузке библиотеки.

Создав функции `_init()` и `_fini()`, нужно указать во время компиляции разделяемой библиотеки параметр `gcc -nostartfiles`, чтобы не дать компоновщику включить их стандартные версии (при желании можно выбрать для них другие имена, воспользовавшись параметрами `-Wl,-init` и `-Wl,-fini`).

Применение функций `_init()` и `_fini()` считается устаревшей практикой. Им на смену пришли атрибуты `constructor` и `destructor` компилятора `gcc`, которые среди прочих преимуществ позволяют определить несколько функций инициализации и финализации.

42.5. Предварительная загрузка разделяемых библиотек

Во время тестирования иногда может понадобиться переопределить функции (и другие символы), которые в обычных условиях были бы найдены динамическим компоновщиком на основе правил, описанных в разделе 41.11. Для этого переменной среды `LD_PRELOAD` можно присвоить строку с именами разделяемых библиотек, которые следует загрузить раньше других (имена разделяются двоеточиями). Поскольку данные библиотеки загружаются в первую очередь, их функции, запрашиваемые программой, будут использоваться автоматически, переопределяя любые одноименные символы, которые в противном случае пришлось бы искать динамическому компоновщику. Представьте, к примеру, что наша программа вызывает функции `x1()` и `x2()`, определенные в библиотеке `libdemo`. Запустив эту программу, мы увидим следующий вывод:

```
$ ./prog
Called mod1-x1 DEMO
Called mod2-x2 DEMO
```

В данном примере мы исходим из того, что разделяемая библиотека находится в одной из стандартных каталогов, поэтому не нужно использовать переменную среды `LD_LIBRARY_PATH`.

Мы могли бы выборочно переопределить функцию `x1()`, создав еще одну разделяемую библиотеку, `libalt.so`, с другой версией `x1()` внутри. Если предварительно загрузить эту библиотеку во время запуска программы, то получится следующий результат:

```
$ LD_PRELOAD=libalt.so ./prog
Called mod1-x1 ALT
Called mod2-x2 DEMO
```

Как видите, здесь вызывается версия `x1()`, определенная в библиотеке `libalt.so`, однако вызов `x2()`, определение которого не содержится в `libalt.so`, приводит к запуску функции `x2()` из библиотеки `libdemo.so`.

Переменная среды `LD_PRELOAD` управляет процедурой предварительной загрузки для каждого отдельного процесса. Чтобы сделать то же самое на уровне системы, можно воспользоваться файлом `/etc/ld.so.preload`, который содержит список библиотек, разделенных пробельными символами (библиотеки, указанные в переменной `LD_PRELOAD`, загружаются раньше перечисленных в `/etc/ld.so.preload`).

Из соображений безопасности программы, устанавливающие пользовательские и групповые идентификаторы, игнорируют переменную `LD_PRELOAD`.

42.6. Мониторинг работы динамического компоновщика: LD_DEBUG

Иногда бывает полезно проследить за работой динамического компоновщика, чтобы, например, понять, где именно он ищет библиотеки. Для этого можно использовать переменную среды `LD_DEBUG`. Присвоив ей одно (или несколько) из стандартных ключевых слов, мы получим различные сведения, касающиеся того, как динамический компоновщик проводит поиск.

Если присвоить переменной `LD_DEBUG` значение `help`, то динамический компоновщик выведет справочную информацию, а сама команда *не* будет выполнена:

```
$ LD_DEBUG=help date
Valid options for the LD_DEBUG environment variable are:

  Libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  all         all previous options combined
  statistics  display relocation statistics
  unused      determine unused DSOs
  help        display this help message and exit
```

В следующем примере представлен несколько сокращенный вариант вывода, который получается при запросе сведений о процессе поиска библиотек:

```
$ LD_DEBUG=libs date
     10687:     find library=librt.so.1 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/librt.so.1
     10687:     find library=libc.so.6 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libc.so.6
     10687:     find library=libpthread.so.0 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libpthread.so.0
     10687:     calling init: /lib/libpthread.so.0
     10687:     calling init: /lib/libc.so.6
     10687:     calling init: /lib/librt.so.1
     10687:     initialize program: date
     10687:     transferring control: date
Tue Dec 28 17:26:56 CEST 2010
     10687:     calling fini: date [0]
     10687:     calling fini: /lib/librt.so.1 [0]
     10687:     calling fini: /lib/libpthread.so.0 [0]
     10687:     calling fini: /lib/libc.so.6 [0]
```

Значение `10687`, выводимое в начале каждой строчки, — идентификатор процесса, который мы отслеживаем. Оно помогает при мониторинге нескольких процессов (например, родительского и дочернего).

Результат по умолчанию записывается в стандартный вывод ошибок, но можно перенаправить его любое другое место, назначив переменной среды `LD_DEBUG_OUTPUT` подходящий путь.

При желании переменной `LD_DEBUG` можно присвоить сразу несколько параметров, разделяя их запятыми (но без пробелов). Особенно подробным является вывод параметра `symbols` (который отслеживает поиск символов динамическим компоновщиком).

Переменная `LD_DEBUG` подходит как для библиотек, загружаемых автоматически, так и для тех, что загружаются вручную с помощью `dlopen()`.

Из соображений безопасности переменная `LD_DEBUG` игнорируется программами, устанавливающими пользовательские и групповые идентификаторы (начиная с glibc 2.2.5).

42.7. Резюме

Динамический компоновщик предоставляет программный интерфейс `dlopen`, позволяющий программам самостоятельно загружать дополнительные разделяемые библиотеки во время выполнения. Это позволяет реализовать подключаемые модули.

Важным аспектом проектирования разделяемых библиотек является управление видимостью символов, поскольку это делает возможным экспорт только тех функций и переменных, которые должны быть доступны программам, скомпонованным с нашей библиотекой. Мы рассмотрели различные методики, помогающие управлять видимостью символов, включая использование версионных сценариев, позволяющих контролировать все нюансы данного процесса.

Мы также показали, как версионные сценарии можно применять для экспорта из библиотеки нескольких вариантов одного и того же символа, которые будут использоваться разными программами, скомпонованными с этой библиотекой. (Каждое приложение задействует определение, являющееся текущим на момент его статической компоновки.) Такой подход — альтернатива традиционному версионированию библиотек с помощью номеров минорных и мажорных версий в реальном имени библиотеки.

Определение функций инициализации и финализации внутри разделяемой библиотеки позволяет автоматически выполнять код при ее загрузке и выгрузке.

Переменная среды `LD_PRELOAD` дает возможность загружать разделяемые библиотеки заранее. Используя этот механизм, можно выборочно переопределить функции и другие символы, которые в противном случае были бы найдены в других библиотеках.

Для мониторинга работы динамического компоновщика можно присваивать различные значения переменной среды `LD_DEBUG`.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 41.14.

42.8. Упражнения

42.1. Напишите программу, которая демонстрирует, что библиотека, закрытая с помощью функции `dlclose()`, не выгружается из памяти, если ее символы используются другой библиотекой.

42.2. Добавьте в программу из листинга 42.1 (`dynload.c`) вызов `dladdr()` для получения сведений об адресе, возвращенном функцией `dlsym()`. Выведите значения полей полученной структуры `Dl_info` и проверьте, соответствуют ли они вашим ожиданиям.

43 Краткий обзор межпроцессного взаимодействия

В этой главе представлен краткий обзор механизмов, с помощью которых процессы и потоки выполнения могут общаться друг с другом и синхронизировать свои действия. В следующих главах эта тема будет рассмотрена более подробно.

43.1. Классификация IPC-механизмов

На рис. 43.1 показано богатое разнообразие механизмов взаимодействия и синхронизации в UNIX. Все они делятся на три категории.

- *Взаимодействие*. Эти механизмы отвечают за обмен данными между процессами.
- *Синхронизация*. Эти механизмы отвечают за синхронизацию работы процессов или потоков.
- *Сигналы*. Изначально они были предназначены для других целей, однако в некоторых ситуациях их можно использовать в качестве средства синхронизации. В редких случаях с их помощью можно организовать взаимодействие: номер сигнала как таковой является своеобразным видом информации, а сигналы реального времени могут нести в себе дополнительные данные (целое число или указатель). Подробное описание сигналов см. в главах 20–22.

Некоторые из этих механизмов занимаются синхронизацией, но в целом ко всем им применим термин *«межпроцессное взаимодействие» (IPC)*.

Как показано на рис. 43.1, разные механизмы часто предоставляют похожие возможности, связанные с IPC. Тому есть несколько причин:

- похожие механизмы эволюционировали параллельно в разных вариантах UNIX и позже были перенесены на другие системы. Например, очереди FIFO были разработаны для System V, а (потоковые) сокеты впервые появились в BSD;
- некоторые новые механизмы были разработаны для устранения недостатков более ранних аналогов. Например, механизмы POSIX IPC (очереди сообщений, семафоры и разделяемая память) появились в качестве улучшенной версии более старых средств работы с IPC из System V.

Ряд механизмов, объединенных на рис. 43.1, на самом деле обладают довольно разными возможностями. Например, потоковые сокеты могут использоваться для взаимодействия по сети, тогда как очереди FIFO позволяют обмениваться данными только между процессами на одном компьютере.

43.2. Средства взаимодействия

Различные механизмы взаимодействия, представленные на рис. 43.1, позволяют процессам обмениваться данными друг с другом (эти же механизмы могут быть использованы для обмена информацией между потоками одного и того же процесса, но такая

необходимость возникает не часто — потоки могут взаимодействовать с помощью общих глобальных переменных).

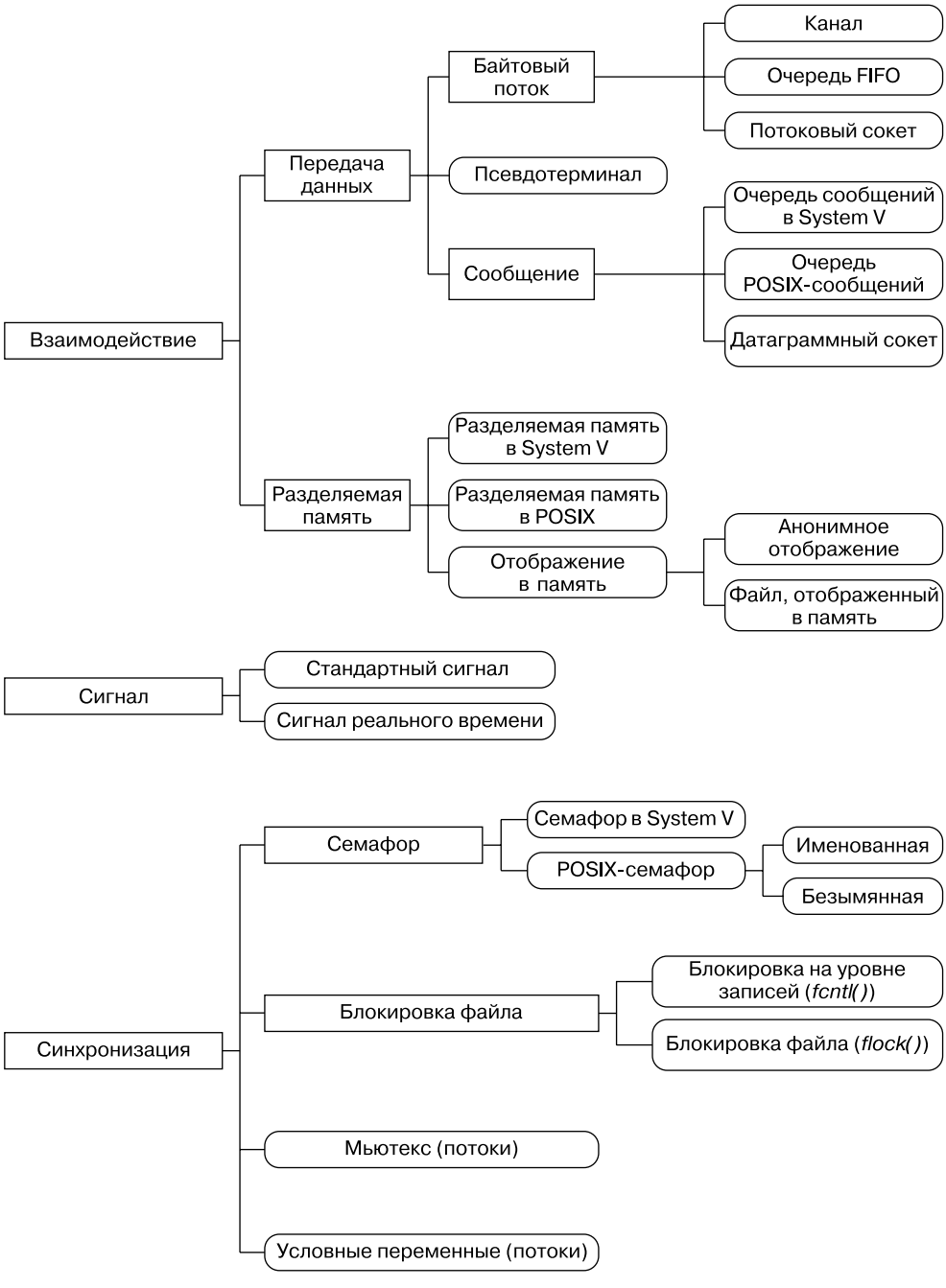


Рис. 43.1. Классификация IPC-механизмов в UNIX

Средства взаимодействия можно разделить на две категории.

- *Средства передачи данных*. Ключевой отличительной особенностью этих механизмов является понятие чтения и записи. Для общения друг с другом один процесс записывает данные в IPC-механизм, а другой считывает их оттуда. Такой подход подразумевает двухэтапную передачу данных между пользовательской и системной памятью: во время записи информация переходит из пространства пользователя в пространство ядра, а во время чтения — в обратном направлении (на рис. 43.2 данный случай проиллюстрирован на примере канала).
- *Разделяемая память*. Позволяет процессам обмениваться информацией путем размещения ее в участках памяти, которые являются общими для обоих процессов (в ядре это достигается за счет привязки записей в таблице страниц памяти каждого из процессов к одним и тем же страницам физической памяти — см. рис. 45.2). Чтобы данные были доступны другому процессу, их можно поместить в сегмент разделяемой памяти. Поскольку данное действие не требует выполнения системных вызовов или передачи информации между пользовательской и системной памятью, этот механизм способен обеспечить очень быстрое взаимодействие.

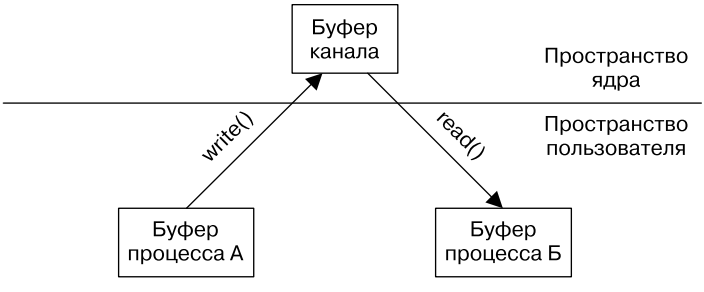


Рис. 43.2. Обмен данными между двумя процессами с помощью канала

Передача данных

Средства передачи данных можно разделить на еще более мелкие подкатегории.

- *Байтовый поток*. Данные, которые передаются с помощью каналов, очередей FIFO и датаграммных сокетов, представляют собой сплошной поток байтов. Каждая операция чтения может получить из IPC-механизма произвольное количество байтов, вне зависимости от размера блока, записанного передающим процессом. Это традиционная для UNIX модель «файл как последовательность байтов».
- *Сообщение*. Данные, передающиеся с помощью очередей сообщений (System V и POSIX) и датаграммных соектов, принимают вид отдельных сообщений. Каждая операция чтения возвращает отдельное сообщение, полученное в том виде, в котором оно было записано отправителем. Мы не можем забрать из IPC-механизма только часть сообщения, как и получить сразу несколько сообщений за одну операцию чтения.
- *Псевдотерминалы*. Это средство взаимодействия, предназначенное для использования в особых ситуациях. Подробности описаны в главе 60.

Средства передачи данных имеют несколько основных отличий от разделяемой памяти.

- У механизма передачи данных может быть несколько получателей, однако операции чтения являются «разрушающими» — то есть однажды прочитанные данные уже недоступны для другого процесса.

Чтобы сделать операции чтения из сокета неразрушающими, можно воспользоваться флагом MSG_PEEK (см. раздел 57.3). UDP-сокеты (датаграммы интернет-домена) позволяют адресовать одно и то же сообщение разным получателям (см. раздел 57.12).

- Синхронизация процессов получателя и отправителя выполняется автоматически. Если получатель попытается извлечь данные, которые отсутствуют в соответствующем механизме, операция чтения по умолчанию будет заблокирована до тех пор, пока данные не будут записаны туда каким-нибудь процессом.

Разделяемая память

Большинство современных UNIX-систем предоставляет три вида: разделяемая память System V, разделяемая память POSIX и отображение в память. Различия между ними будут рассмотрены в следующих главах, когда будет описан каждый из этих механизмов (в частности, см. раздел 50.5).

Ниже отмечены основные особенности разделяемой памяти.

- Разделяемая память обеспечивает быстрое взаимодействие, но эта скорость нивелируется необходимостью выполнения синхронизации. Например, процесс не должен обращаться к структуре данных в разделяемой памяти, когда та обновляется другим процессом. Традиционным средством синхронизации данных в разделяемой памяти являются семафоры.
- Данные, размещенные в разделяемой памяти, видны всем процессам, которые используют эту память (в отличие от механизма разрушающего чтения, описанного выше, в разделе о передаче данных).

43.3. Средства синхронизации

Средства синхронизации, представленные на рис. 43.1, позволяют процессам координировать свои действия. С помощью синхронизации процессы могут избежать таких проблем, как одновременное обновление сегмента разделяемой памяти или части одного и того же файла. Без этого одновременное обновление могло бы привести к получению некорректных результатов.

UNIX-системы предоставляют следующие механизмы синхронизации.

- *Семафоры.* Семафор — целое число, хранящееся в пространстве ядра, которое ни при каких обстоятельствах не может быть меньше нуля. Процесс может увеличивать и уменьшать его значение. Ядро блокирует попытку сделать семафор отрицательным до тех пор, пока тот не вырастет до нужного уровня. (Процесс также может запросить неблокирующую операцию; в этом случае вместо блокировки ядро сразу же вернет сообщение об ошибке, указывающее на то, что операция не может быть выполнена в данный момент.) Назначение семафора определяется самим приложением. Процесс декрементирует семафор (скажем, с 1 до 0), чтобы зарезервировать эксклюзивный доступ к какому-нибудь общему ресурсу, и затем, завершив работу, инкрементирует его обратно, делая ресурс доступным для других процессов. Часто применяется двоичный семафор, который может быть равен только 0 или 1. Но если приложение работает с несколькими экземплярами разделяемого ресурса, то ему следует использовать семафор, чье максимальное значение равно количеству этих экземпляров. Linux поддерживает семафоры из System V и стандарта POSIX, которые, в сущности, имеют похожие функции.
- *Блокировки файлов.* Этот метод синхронизации специально создан для координации действий разных процессов, которые работают с одним и тем же файлом. С его помощью также можно координировать доступ к другим общим ресурсам. Блокировки файлов бывают

двух видов: блокировки чтения (разделяемые) и блокировки записи (эксклюзивные). Блокировкой чтения одного и того же файла (или его участка) может владеть неограниченное количество процессов. Но если файл (или его участок) заблокирован на запись каким-либо процессом, остальные процессы не могут ни считывать его содержимое, ни записывать в него. В Linux механизм блокировки файлов предоставляется с помощью системных вызовов `flock()` и `fcntl()`. Первый позволяет процессам устанавливать разделяемые или эксклюзивные блокировки на весь файл целиком. Ввиду своих ограниченных возможностей данный вызов редко используется в наши дни. Второй обеспечивает блокировку на уровне записей, позволяя процессам устанавливать неограниченное количество блокировок для чтения и записи разных участков одного и того же файла.

- *Мьютексы и условные переменные*. Эти средства синхронизации обычно применяются в сочетании с потоками, как описано в главе 30.

Ряд реализаций UNIX, включая дистрибутивы Linux, в которых библиотека glibc поддерживает NPTL-потоки, позволяют разделять между процессами мьютексы и условные переменные. Эта возможность предусмотрена стандартом SUSv3, но не является обязательной. Она поддерживается не во всех UNIX-системах и обычно не используется для синхронизации процессов.

При выборе средства для межпроцессной синхронизации обычно учитываются функциональные требования. Для координации доступа к файлу лучшим выбором, как правило, является блокировка записей. Для координации доступа к другим видам общих ресурсов часто выбирают семафоры.

Средства взаимодействия тоже могут использоваться для синхронизации. Например, в разделе 44.3 будет показано, как синхронизировать действия родительского и дочернего процессов с помощью канала. В целом синхронизация может быть выполнена за счет любого механизма передачи данных; в таких случаях это выглядит как обмен сообщениями, проходящими через выбранный механизм.

43.4. Сравнение IPC-механизмов

Когда речь заходит об IPC, мы сразу же сталкиваемся с разнообразием вариантов, которое поначалу может вызвать недоумение. В следующих главах, описывающих каждый из IPC-механизмов, есть разделы, в которых сравниваются несколько похожих средств. Ниже мы рассмотрим несколько общих моментов, способных повлиять на выбор того или иного механизма.

Идентификация IPC-объектов и их дескрипторов

Чтобы получить доступ к IPC-объекту, у процесса должна быть возможность его идентифицировать; после «открытия» объекта процесс должен использовать некий дескриптор, который на него ссылается. Эти свойства в контексте разных видов IPC-механизмов перечислены в табл. 43.1.

Таблица 43.1. Идентификаторы и дескрипторы в различных видах IPC-механизмов

Механизм	**Имя для идентификации объекта**	**Дескриптор, ссылающийся на объект в программе**
Канал	Без имени	Дескриптор файла
Очередь FIFO	Pathname	Дескриптор файла
Сокет домена UNIX	Pathname	Дескриптор файла

Механизм	Имя для идентификации объекта	Дескриптор, ссылающийся на объект в программе
Сокет интернет-домена	IP-адрес + номер порта	Дескриптор файла
Очередь сообщений в System V	IPC-ключ в System V	IPC-идентификатор в System V
Семафор в System V	IPC-ключ в System V	IPC-идентификатор в System V
Разделяемая память в System V	IPC-ключ в System V	IPC-идентификатор в System V
Очередь POSIX-сообщений	IPC-путь в POSIX	mqd_t (дескриптор очереди сообщений)
Именованный POSIX-семафор	IPC-путь в POSIX	sem_t * (указатель на семафор)
Безымянный POSIX-семафор	Без имени	sem_t * (указатель на семафор)
Разделяемая память в POSIX	IPC-путь в POSIX	Дескриптор файла
Анонимное отображение	Без имени	Отсутствует
Файл, отображенный в память	Pathname	Дескриптор файла
Блокировка flock()	Pathname	Дескриптор файла
Блокировка fcntl()	Pathname	Дескриптор файла

Возможности

Существуют функциональные различия, которые могут повлиять на выбор того или иного IPC-механизма. Для начала кратко перечислим различия между средствами передачи данных и разделяемой памятью.

- Средства передачи данных используют операции чтения и записи, а переданная информация потребляется только одним процессом-адресатом. Управление потоком данных между отправителем и получателем, а также их синхронизация (чтобы блокировать получателя, который пытается прочитать данные из пустого источника) автоматически осуществляется ядром. Эта модель хорошо согласуется с архитектурой многих приложений.
- Другим видам приложений лучше подходит модель с разделяемой памятью. Она позволяет одному процессу открывать доступ к своим данным для множества других процессом, разделяя с ними один и тот же участок памяти. «Операции» взаимодействия предельно просты: процесс работает с участком памяти точно так же, как он это делает с любой частью своего адресного пространства. С другой стороны, описанная модель может усложниться ввиду необходимости выполнения синхронизации (и, возможно, управления потоком данных). Этот механизм хорошо подходит для приложений, которым нужно управлять разделяемым состоянием (например, общей структурой данных).

  Стоит отметить несколько моментов, касающихся механизмов передачи данных.

- Некоторые механизмы передают данные в виде байтов (каналы, очереди FIFO и потоковые сокеты), а другие ориентированы на обмен сообщениями (очереди сообщений и датаграммные сокеты). Предпочтительность того или иного подхода зависит от конкретного приложения (можно использовать модель обмена сообщениями поверх механизма, оперирующего байтовыми потоками; для этого подойдут

символы-разделители, сообщения фиксированной длины или заголовки сообщений, в которых содержится их общая длина; см. раздел 44.8).

- Отличительной чертой очередей сообщений System V и POSIX (по сравнению с другими механизмами передачи данных) является то, что сообщения могут быть доставлены не в том порядке, в котором их отправляли. Это достигается за счет назначения различных числовых приоритетов.
- Каналы, очереди FIFO и сокеты реализованы в виде файловых дескрипторов. Все эти IPC-механизмы поддерживают целый ряд альтернативных моделей ввода/вывода, которые будут описаны в главе 59: мультиплексированный ввод/вывод (системные вызовы `select()` и `poll()`), ввод/вывод, основанный на сигналах, и программный интерфейс `epoll` (только в Linux). Основное преимущество этих методик таково: они позволяют приложению отслеживать одновременно несколько файловых дескрипторов, чтобы понять, возможен ли ввод/вывод на каком-либо из них. Для сравнения, в System V очереди сообщений не используют файловые дескрипторы и не поддерживают указанные методики.

  В Linux очереди POSIX-сообщений тоже реализованы в виде файловых дескрипторов и поддерживают альтернативные методики ввода/вывода, описанные выше. Однако такое поведение не описано в стандарте SUSv3 и не поддерживается в большинстве реализаций.

- Очереди POSIX-сообщений предоставляют механизм оповещений, который может послать процессу сигнал или создать новый поток выполнения в ответ на появление сообщения в пустой очереди.
- Сокеты доменов UNIX позволяют передавать файловый дескриптор между процессами. Благодаря этому один процесс может открыть файл и сделать его доступным для другого процесса, который в противном случае не мог бы обратиться к данному файлу. Эта возможность кратко описывается в подразделе 57.13.3.
- UDP-сокеты (датаграммы интернет-домена) позволяют отправителю передавать сообщения определенной (мультивещание) или произвольной (широковещание) группе получателей. Эта возможность будет кратко описана в разделе 57.12.

Что касается механизмов синхронизации процессов, необходимо отметить следующее.

- Блокировки записей, установленные с помощью вызова `fcntl()`, принадлежат процессу, их установившему. Эта принадлежность используется ядром для обнаружения взаимных блокировок (ситуации, в которой два или больше процесса удерживают блокировки и не дают друг другу получить доступ к нужным ресурсам). Если взаимные блокировки все же происходят, ядро сигнализирует об этом, отклоняя запрос на блокировку одного из процессов и возвращая ошибку из вызова `fcntl()`. В System V и POSIX семафоры не поддерживают принцип владения, поэтому для них не действует механизм обнаружения взаимных блокировок.
- Блокировки записей, установленные с использованием вызова `fcntl()`, автоматически снимаются при завершении процесса, владеющего блокировкой. POSIX-семафоры не предоставляют аналогичную возможность.

Сетевое взаимодействие

Из всех IPC-механизмов, представленных на рис. 43.1, только сокеты позволяют процессам взаимодействовать по сети. Они обычно применяются в доменах: либо в UNIX, обеспечивающем общение между процессами в рамках одной системы, либо в интернет-домене, который дает возможность процессам, находящимся на разных компьютерах, взаимодействовать по сети TCP/IP. Для перехода от UNIX- к интернет-доменам обычно требуются минимальные изменения, поэтому в приложения, использующие домены UNIX, можно относительно легко добавить поддержку сети.

Портируемость

Современные реализации UNIX поддерживают большинство IPC-механизмов, представленных на рис. 43.1. Однако механизмы, относящиеся к стандарту POSIX (очереди сообщений, семафоры и разделяемая память) распространены не так широко, как их аналоги из мира System V; особенно это касается старых систем (реализация очередей POSIX-сообщений и поддержка POSIX-семафоров появились в Linux только в ветке ядра 2.6.*x*). Поэтому с точки зрения портируемости IPC-механизмы из System V могут быть более предпочтительными, чем их POSIX-альтернативы.

Доступность

Во втором столбце табл. 43.2 указано важное свойство IPC-объектов каждой разновидности, — система прав доступа, определяющая, какие процессы могут работать с объектом. Ниже перечислены ряд подробностей, касающихся различных систем.

- В некоторых IPC-механизмах (например, в очередях FIFO и сокетах) имена объектов хранятся в файловой системе, а доступ к ним определяется соответствующей маской прав доступа к файлу, описывающей привилегии для владельца, группы и всех остальных (см. раздел 15.4). В System V объекты не хранятся в файловой системе, но у каждого из них есть маска, семантика которой похожа на используемую для файлов.
- Несколько IPC-механизмов (каналы, анонимное отображение в память) доступны только связанным процессам (как отмечено ниже). Эта *связь* присутствует на уровне вызова `fork()`. Чтобы два процесса получили доступ к объекту, один из них должен создать данный объект и затем вызвать `fork()`. После этого дочерний процесс наследует дескриптор, ссылающийся на объект, что позволяет процессу разделить его со своим родителем.
- Доступность безымянного POSIX-семафора определяется доступностью разделяемого участка памяти, в котором он находится.
- Чтобы установить блокировку на файл, процесс должен иметь дескриптор, ссылающийся на данный файл (на практике это значит, что он должен иметь права для его открытие).
- Доступ к сокету интернет-домена (то есть соединение с ним и передача ему датаграммы) не ограничен. При необходимости управление доступом следует реализовывать на уровне приложения.

Таблица 43.2. Доступность и сохраняемость различных типов IPC-объектов

Механизм	**Доступность**	**Сохранямость**
Канал	Только для родственного процесса	Процесс
Очередь FIFO	Через маску прав доступа	Процесс
Сокет домена UNIX	Через маску прав доступа	Процесс
Сокет интернет-домена	Для любого процесса	Процесс
Очередь сообщений в System V	Через маску прав доступа	Ядро
Семафор в System V	Через маску прав доступа	Ядро
Разделяемая память в System V	Через маску прав доступа	Ядро
Очередь POSIX-сообщений	Через маску прав доступа	Ядро
Именованный POSIX-семафор	Через маску прав доступа	Ядро

Продолжение ⇗

Таблица 43.2 *(продолжение)*

Механизм	Доступность	Сохранямость
Безымянный POSIX-семафор	Через права доступа к памяти	По-разному
Разделяемая память в POSIX	Через маску прав доступа	Ядро
Анонимное отображение	Только для родственного процесса	Процесс
Файл, отображенный в память	Через маску прав доступа	Файловая система
Блокировка flock()	Через открытие файла	Процесс
Блокировка fcntl()	Через открытие файла	Процесс

Сохраняемость

Термин *«сохранямость»* (устойчивость) относится к продолжительности существования IPC-объекта (см. третий столбец табл. 43.2). С этой точки зрения объекты можно разделить на три категории.

- *Сохраняемость на уровне процесса.* Объект существует, пока открыт хотя бы одним процессом. После закрытия его всеми процессами ядро освобождает его ресурсы, а любые непрочитанные данные уничтожаются. Примером таких объектов являются каналы, очереди FIFO и сокеты.

  Сохраняемость данных очереди FIFO отличается от сохраняемости ее имени. Имя хранится в файловой системе и продолжает существовать даже после закрытия всех файловых дескрипторов, ссылавшихся на очередь.

- *Сохраняемость на уровне ядра.* Объект существует до тех пор, пока его явно не удалят или пока система не будет выключена. Жизненный цикл объекта не зависит от того, открыт ли он каким-либо процессом. Это значит, что, к примеру, один процесс может создать объект, записать в него данные и закрыть (или просто завершиться). Затем другой процесс может открыть данный объект и прочитать хранящуюся в нем информацию. Примером объектов с сохраняемостью на уровне ядра являются IPC-механизмы в System V и POSIX. Это их свойство будет использовано в примерах программ, представленных в следующих главах при описании данных механизмов: для каждого механизма будет рассмотрена отдельную программу, которая создает и удаляет объект, а также осуществляет взаимодействие или синхронизацию.
- *Сохраняемость на уровне файловой системы.* Объект сохраняет свою информацию даже после перезагрузки системы. Он прекращает свое существование только после удаления вручную. Единственным типом объектов, который обладает сохраняемостью на уровне файловой системы, является файл, отраженный на разделяемую память.

Производительность

В некоторых ситуациях разные IPC-механизмы могут демонстрировать существенные различия в производительности. Однако в последующих главах мы воздержимся от сравнений в этом контексте по нескольким причинам:

- производительность IPC-механизма может не оказывать заметного влияния на общую производительность приложения и не быть единственным фактором при выборе средства межпроцессного взаимодействия;
- относительная производительность различных IPC-механизмов может варьироваться в зависимости от реализации UNIX или версии ядра Linux;

- и самое важное: производительность IPC-механизма зависит от того, как именно и в какой среде его используют. Это относится к объему данных, передаваемых в каждой IPC-операции, необходимости переключения контекста процесса при передаче каждой порции данных и общей загруженности системы.

Если производительность IPC играет решающую роль, единственный способ выбрать подходящий механизм — провести измерения в среде, которая соответствует целевой системе. Для этого можно даже написать абстрактный программный слой, инкапсулирующий подробности реализации IPC-механизма, и затем провести тестирование производительности с помощью разных средств межпроцессного взаимодействия.

43.5. Резюме

В этой главе мы кратко рассмотрели различные механизмы, которые могут применяться для синхронизации процессов (и потоков) и их взаимодействия.

Среди средств взаимодействия, доступных в Linux, можно выделить каналы, очереди FIFO, сокеты, очереди сообщений и разделяемую память. Синхронизация в Linux выполняется за счет семафоров и файловых блокировок.

Во многих случаях для выполнения одной и той же задачи можно выбирать из нескольких механизмов взаимодействия и синхронизации. В этой главе мы сравнивали разные методики в различных контекстах, пытаясь выделить те особенности, которые могут повлиять на выбор того или иного механизма.

В следующих главах мы более подробно остановимся на каждом механизме взаимодействия и синхронизации.

43.6. Упражнения

43.1. Напишите программу, измеряющую пропускную способность каналов. В качестве аргументов командной строки она должна принимать количество блоков данных, которые нужно послать, и размер каждого такого блока. После создания канала программа делится на два процесса: дочерний, записывающий блоки данных в канал с максимально возможной скоростью, и родительский, считывающий данные блоки. После прочтения всей информации родительский процесс должен вывести время, которое ему для этого понадобилось, а также скорость передачи данных (в байтах в секунду). При измерении пропускной способности используйте разные размеры блоков.

43.2. Повторите предыдущее упражнение для очередей сообщений в System V и POSIX, а также для сокетов UNIX- и интернет-доменов. Примените все эти программы для сравнения относительной производительности различных IPC-механизмов в Linux. Если у вас есть доступ к другим UNIX-системам, выполните аналогичное сравнение и в них.

44 Каналы и очереди FIFO

Эта глава посвящена каналам и очередям FIFO. Каналы являются наиболее старым средством межпроцессного взаимодействия в UNIX-системах; они появились в третьем издании UNIX в начале 1970-х. Этот механизм предоставляет элегантное решение распространенной задачи: имея два процесса, выполняющие две разные программы (команды), нужно сделать так, чтобы командная оболочка направила вывод одного из них в ввод другого. Каналы могут использоваться для передачи данных между родственными процессами (мы проясним значение слова «родственные» чуть позже). Очереди FIFO — одна из вариаций концепции каналов. Их важной особенностью является то, что их можно применять для организации взаимодействия *любых* процессов.

44.1. Краткий обзор

Любому пользователю командной оболочки встречались команды с применением каналов. Например, команда, представленная ниже, считает количество файлов в каталоге:

```
$ ls | wc –l
```

Для выполнения данной команды командная оболочка создает два процесса и выполняет в них соответственно утилиты `ls` и `ws` (это делается с помощью вызовов `fork()` и `exec()`, описанных в главах 24 и 27). На рис. 44.1 показано, как оба процесса используют канал.

Помимо прочего, рис. 44.1 иллюстрирует то, почему каналы (англ. pipe — «труба») получили такое название. Мы можем рассматривать их как фрагмент трубы, которая позволяет пропускать через себя потоки данных, направляя их от одного процесса к другому.

Рис. 44.1. Использование канала для соединения двух процессов

При взгляде на рис. 44.1 можно обратить внимание на способ подключения процессов к каналу: тот, что записывает данные (`ls`), соединяет свой стандартный вывод (файловый дескриптор 1 — fd 1) с записывающим концом канала, а считывающий процесс (`wc`) подключает свой стандартный ввод (файловый дескриптор 0 — fd 0) к его другому концу, предназначенному для чтения. Как следствие, ни один из процессов не знает о существовании канала; они просто используют стандартные файловые дескрипторы для чтения и записи. Командная оболочка должна проделать некоторую работу, чтобы наладить такое взаимодействие; в разделе 44.4 объясняется, как это происходит.

В следующих подразделах будет рассмотрен целый ряд важных особенностей, присущих каналам.

Канал — это поток байтов

Когда мы говорим, что канал представляет собой поток байтов, мы имеем в виду следующее: при его использовании не существует понятия сообщений или их границ. Процесс, считывающий данные из канала, может прочитать блок любого размера, независимо от того, насколько большой блок был туда записан другим процессом. Кроме того, данные проходят через канал последовательно — байты считываются из него в том же порядке, в котором они были записаны. Произвольный доступ к содержимому канала с помощью вызова `lseek()` невозможен.

Реализацию принципа отдельных сообщений в канале следует делать на уровне приложения. Это вполне выполнимая задача (см. раздел 44.8), но для подобных целей лучше применять другие IPC-механизмы, такие как очереди сообщений и датаграммные сокеты (мы рассмотрим их в следующих главах).

Чтение из канала

Любая попытка прочитать данные из пустого канала блокируется до тех пор, пока кто-нибудь не запишет в этот канал хотя бы один байт. Если закрыть другой конец канала, процесс, выполняющий чтение, получит конец файла (то есть операция `read()` вернет `0`), как только дочитает оставшиеся данные.

Каналы являются однонаправленными

Данные внутри канала могут перемещаться только в одном направлении. Один конец канала используется для записи, а другой — для чтения.

В ряде других реализаций UNIX (в частности, тех, что происходят от четвертой редакции System V) каналы являются двунаправленными (так называемые *потоковые каналы*). Они не описаны ни в одном стандарте, связанном с UNIX, поэтому даже в системах, в которых они поддерживаются, на их семантику лучше не полагаться. В качестве альтернативы можно применять парные потоковые сокеты доменов UNIX (создаются с помощью вызова `socketpair()`, описанного в разделе 53.5), предоставляющие стандартизированный двунаправленный механизм взаимодействия, семантически эквивалентный потоковым каналам.

Запись данных, чей объем не превышает PIPE_BUF байт, является гарантированно атомарной

Если несколько процессов выполняют запись в один и тот же канал, их данные никогда не перемешаются, если ни один из них не станет записывать более чем `PIPE_BUF` байт за раз.

Стандарт SUSv3 требует, чтобы значение `PIPE_BUF` было не менее `_POSIX_PIPE_BUF` (`512`). Система должна определить `PIPE_BUF` (внутри `<limits.h>`) и/или позволить вызову `fpathconf(fd, _PC_PIPE_BUF)` возвращать действующее ограничение на максимальный размер атомарной записи. Значение `PIPE_BUF` варьируется в зависимости от системы — например, в FreeBSD 6.0 оно равно 512 байт, в Tru64 5.1 — 4096 байт, а в Solaris 8 — 5120 байт. В Linux `PIPE_BUF` имеет значение `4096`.

Если записываемый в канал блок данных превышает `PIPE_BUF` байт, ядро может разбить его на более мелкие части и передавать их по мере того, как считывающий процесс удаляет их из канала (вызов `write()` блокируется, пока в канал не будут записаны все данные). Если только один процесс выполняет запись в канал (что обычно и происходит), то все это не имеет значения. Но при наличии нескольких таких процессов большие блоки могут быть разбиты на сегменты произвольного размера (который может быть меньше `PIPE_BUF` байт) и переданы, чередуясь с блоками других процессов.

Ограничение PIPE_BUF определяет то, когда именно данные будут переданы в канал. При записи не более PIPE_BUF байт вызов write() при необходимости может заблокироваться, пока в канале не освободится достаточно места для автоматического выполнения этой операции. Если записываемый блок данных превышает PIPE_BUF байт, то вызов write() передает ту их часть, которая полностью заполнит канал, и затем блокируется до тех пор, пока данные не будут удалены из канала каким-нибудь считывающим процессом. При прерывании такой операции обработчиком сигнала вызов write() возобновляет работу и возвращает количество успешно переданных байтов (которое будет меньше запрашиваемого объема — так называемая *частичная запись*).

Каналы имеют ограниченную вместимость

Канал — это всего лишь буфер, находящийся в памяти ядра. И его максимальная вместимость ограничена. Если канал заполнен, то все дальнейшие операции записи в него блокируются, пока считывающий процесс не удалит из него часть данных.

В стандарте SUSv3 нет никаких требований относительно вместимости каналов. В версиях ядра Linux, предшествовавших 2.6.11, вместимость была равна размеру страницы памяти в системе (например, 4096 байт на платформе x86-32); но, начиная с версии 2.6.11, канал может хранить в себе до 55 536 байт. Это ограничение отличается в других реализациях UNIX.

В общем случае приложению не нужно знать точную вместимость канала. Если нужно предотвратить блокировку записывающих процессов, то процессы, считывающие данные, должны изымать содержимое канала по мере его поступления.

> Теоретически нет никаких причин, по которым канал не мог бы работать с меньшей вместимостью, даже если это однобайтный буфер. Увеличение размера продиктовано требованиями к производительности: каждый раз, когда записывающий процесс заполняет канал, ядро вынуждено переключать контекст, чтобы считывающий процесс мог извлечь данные из канала. Чем больше буфер, тем реже приходится переключать контекст.
>
> Начиная с Linux 2.6.35, вместимость канала можно менять. Вызов fcntl(fd, F_SETPIPE_SZ, size) (доступен только в Linux) назначает каналу fd размер size байт. Непривилегированный процесс может установить для канала любую вместимость в диапазоне от размера страницы памяти и до значения, хранящегося в файле /proc/sys/fs/pipe-max-size. По умолчанию значение pipe-max-size равно 1 048 576 байт. Привилегированный процесс (CAP_SYS_RESOURCE) может переопределить это ограничение. Выделяя место для канала, Linux может округлить size до значения, которое система находит более подходящим. Вызов fcntl(fd, F_GETPIPE_SZ) возвращает фактический размер выделенного канала.

44.2. Создание и использование каналов

Системный вызов pipe() создает новый канал.

```
include <unistd.h>

int pipe(int filedes[2]);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Успешный вызов pipe() возвращает массив filedes с двумя открытыми файловыми дескрипторами: один для чтения (filedes[0]), второй — для записи (filedes[1]).

Как и в случае с любыми другими дескрипторами, для выполнения ввода/вывода в контексте канала можно использовать системные вызовы `read()` и `write()`. Данные, записанные в канал, немедленно становятся доступными для чтения на другом его конце. Операция чтения извлекает из канала количество байтов, не больше запрошенного, но не меньше того, которое сейчас доступно (если канал пустой, операция блокируется).

Для работы с каналом также можно применять функции стандартного ввода/вывода (`printf()`, `scanf()` и т. д.), предварительно получив файловый поток из одного из дескрипторов в массиве `filedes` (см. раздел 13.7), задействуя вызов `fdopen()`. Но при этом следует учитывать потенциальные проблемы с буферизацией ввода/вывода, описанные в разделе 44.6.

Вызов ioctl(fd, FIONREAD, &cnt) возвращает количество непрочитанных байтов в канале или очереди FIFO, на которые ссылается дескриптор fd. Он доступен и в ряде других реализаций, но не описан в стандарте SUSv3.

На рис. 44.2 изображен канал, созданный с помощью вызова `pipe()`. Каждый конец данного канала доступен вызывающему процессу в виде файловых дескрипторов.

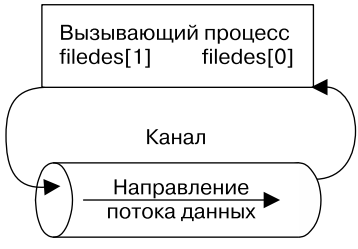


Рис. 44.2. Файловые дескрипторы процесса после создания канала

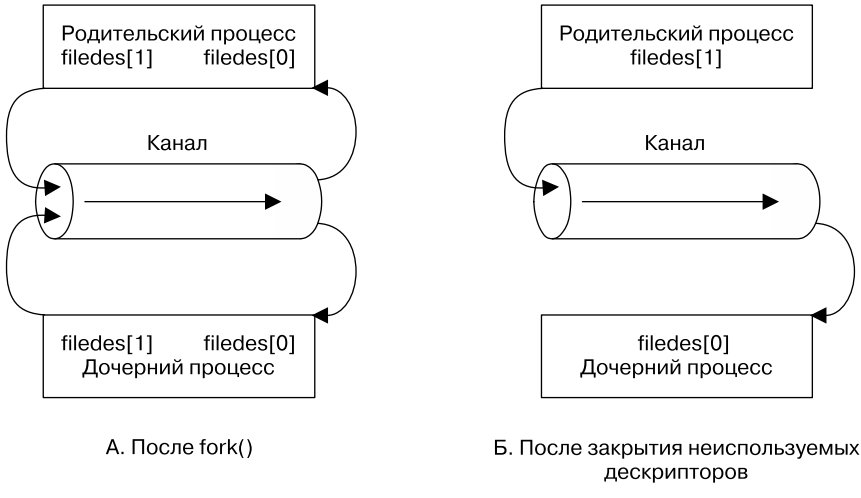
В рамках одного процесса канал можно применять несколькими способами (один из них будет рассмотрен в подразделе 59.5.2). Обычно канал используется для обеспечения взаимодействия двух процессов. Чтобы их соединить с помощью канала, вслед за `pipe()` делается вызов `fork()`. Во время его выполнения потомок наследует копии файловых дескрипторов родителя (см. раздел 24.2.1), что приводит к ситуации, представленной на рис. 44.3, *слева*.

Теоретически оба процесса, родительский и дочерний, могут как считывать данные из канала, так и записывать их в него, однако на практике такая ситуация не является типичной. Как правило, сразу же после вызова `fork()` один из процессов закрывает дескриптор, связанный с записывающим концом канала, а другой делает то же самое для считывающего конца. Например, если родитель собирается передавать данные потомку, он должен закрыть свой дескриптор, предназначенный для чтения (`filedes[0]`); в то же время потомок должен закрыть дескриптор для записи (`filedes[1]`). В результате получается ситуация, изображенная в правой части рис. 44.3. Код, реализующий данный пример, показан в листинге 44.1.

Листинг 44.1. Этапы создания канала, подходящего для передачи данных от родителя к потомку

```
int filedes[2];

if (pipe(filedes) == -1)           /* Создаем канал */
    errExit("pipe");

switch (fork()) {                  /* Создаем дочерний процесс */
case -1:
    errExit("fork");
```

```
case 0: /* Потомок */
    if (close(filedes[1]) == -1)    /* Закрываем неиспользуемый записывающий конец */
        errExit("close");

    /* Теперь потомок читает из канала */
    break;

default: /* Родитель */
    if (close(filedes[0]) == -1)    /* Закрываем неиспользуемый считывающий конец */
        errExit("close");

    /* Теперь родитель записывает в канал */
    break;
}
```



А. После fork()

Б. После закрытия неиспользуемых дескрипторов

Рис. 44.3. Подготовка канала к передаче данных от родителя к потомку

Родитель и потомок редко сохраняют возможность считывать данные из одного канала, поскольку в этом случае нельзя точно определить, какой из процессов первым выполнит чтение; то есть процессы должны конкурировать за получение информации. Чтобы этого избежать, пришлось бы применять какой-нибудь механизм синхронизации. Но для двунаправленного взаимодействия существует более простой вариант: создать два канала — по одному для передачи данных в каждом направлении (при использовании такой методики следует учитывать возможность взаимных блокировок, когда оба процесса блокируются, пытаясь прочитать из пустых каналов или записать в каналы, которые уже заполнены).

Теоретически в канал можно записывать неограниченное количество процессов, но обычно применяется только один записывающий процесс (пример того, когда может пригодиться несколько таких процессов, будет показан в разделе 44.3). Для сравнения, очередь FIFO допускает ситуации, когда запись из нескольких процессов может быть оправданной (этот случай будет рассмотрен в разделе 44.8).

Каналы обеспечивают взаимодействие родственных процессов

До сих пор мы обсуждали каналы, предназначенные для взаимодействия родительского и дочернего процессов. Однако они дают возможность взаимодействовать любым двум

(или более) родственным процессам, общий предок которых создал канал до выполнения цепочки вызовов `fork()` (вот что мы имели в виду, когда упоминали *родственные процессы* в начале этой главы). Например, канал позволяет организовать взаимодействие процесса и его внука: сначала процесс открывает канал, затем создает дочерний процесс, который, в свою очередь, создает внука. Обычно каналы используются для взаимодействия родственных процессов одного уровня: родитель открывает канал и создает двух потомков. Именно это и делает командная оболочка, когда формирует конвейер.

> Правило, согласно которому каналы могут применяться только для взаимодействия родственных процессов, имеет одно исключение. Сокет домена UNIX позволяет передать файловый дескриптор канала любому процессу (эта методика кратко описана в разделе 57.13.3).

Закрытие неиспользуемых файловых дескрипторов канала

Данное действие требуется не только для того, чтобы не дать процессу превысить ограничение на открытые дескрипторы; это необходимое условие корректной работы каналов. Теперь подумаем, почему неиспользуемые файловые дескрипторы для чтения из канала и записи в него должны быть закрыты.

Процесс, читающий из канала, закрывает его записывающий дескриптор, чтобы иметь возможность обнаружить конец файла (прочитав оставшиеся в канале данные), когда другой процесс закончит вывод и закроет свой дескриптор.

Если читающий процесс не закроет записывающий конец канала, то не сможет узнать, когда другой процесс завершит свой ввод, даже после прочтения всех доступных данных. Вместо этого операция `read()` будет заблокирована в ожидании новой информации, поскольку ядро знает: у данного канала остался по крайней мере один записывающий дескриптор. Тот факт, что дескриптор открыт самим считывающим процессом, не имеет никакого значения; теоретически этот процесс может записывать в канал, несмотря на то что он заблокирован при попытке чтения. Например, операция `read()` может быть прервана обработчиком сигнала, который записывает данные в канал (в подразделе 59.5.2 вы увидите реалистичность данного сценария).

Записывающий процесс закрывает считывающий дескриптор канала по другой причине. Если попытаться записать в канал, на другом конце которого нет открытого дескриптора, ядро пошлет сигнал `SIGPIPE`. По умолчанию это приводит к завершению процесса, хотя он может его перехватить или проигнорировать — в таком случае запись в канал завершится ошибкой `EPIPE` (канал поврежден). Получение сигнала `SIGPIPE` или ошибки `EPIPE` позволяет узнать состояние канала, поэтому неиспользуемый считывающий дескриптор следует закрывать.

> Стоит отметить, что прерывание операции write() сигналом SIGPIPE обрабатывается особым образом. Обычно, когда обработчик сигнала прерывает запись (или другой «медленный» вызов), операция либо автоматически перезапускается, либо завершается ошибкой EINTR — это зависит от того, был ли обработчик установлен с помощью флага SA_RESTART для вызова sigaction() (см. раздел 21.5). Сигнал SIGPIPE ведет себя иначе, поскольку автоматический перезапуск записи или оповещение о ее прерывании обработчиком (предполагается, что тогда операция write() может быть перезапущена вручную) не имеет никакого смысла. Ни в том ни в другом случае последующий вызов write() не сможет завершиться успешно, так как канал останется поврежденным.

Если записывающий процесс не закроет считывающий конец канала, то сможет продолжать запись даже после того, как другой процесс закроет свой считывающий дескриптор. В какой-то момент записывающий процесс заполнит канал, и следующая попытка записи будет навсегда заблокирована.

Еще одна причина для закрытия неиспользуемых дескрипторов состоит в том, что канал можно уничтожить и освободить его ресурсы только после того, как будут закрыты все дескрипторы во всех процессах, ссылающихся на данный канал. В этот момент любые данные, которые в нем еще оставались, теряются.

Пример программы

Программа, представленная в листинге 44.2, демонстрирует применение канала для взаимодействия родительского и дочернего процессов. Данный пример иллюстрирует ранее озвученный нами факт: каналы являются потоками байтов. Для этого родитель выполняет запись за одну операцию, а потомок считывает данные небольшими блоками.

Главная программа открывает канал с помощью вызова `pipe()` ❶ и вызывает `fork()`, чтобы создать дочерний процесс ❷. После этого родитель закрывает свой файловый дескриптор для чтения из канала ❽ и записывает в другой конец канала строку ❾, переданную программе в виде аргумента командной строки. Затем родитель закрывает считывающий конец канала ❿ и делает вызов `wait()`, чтобы дождаться завершения дочернего процесса ⓫. После закрытия своего дескриптора для записывающего конца канала ❸ потомок входит в цикл, в котором считывает ❹ из канала блоки данных (размером до `BUF_SIZE` байт) и записывает ❻ их в стандартный вывод. Обнаружив символ конца файла ❺, потомок выходит из цикла ❼, выводит в конце символ новой строки, закрывает свой дескриптор для чтения из канала и завершается.

Вот какой результат можно получить, если запустить программу из листинга 44.2:

```
$ ./simple_pipe 'It was a bright cold day in April, '\
'and the clocks were striking thirteen.'
It was a bright cold day in April, and the clocks were striking thirteen.
```

Листинг 44.2. Использование канала для взаимодействия родительского и дочернего процессов

pipes/simple_pipe.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10

int
main(int argc, char *argv[])
{
    int pfd[2];                 /* Файловые дескрипторы канала */
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "–help") == 0)
        usageErr("%s string\n", argv[0]);

❶   if (pipe(pfd) == -1)    /* Создаем канал */
        errExit("pipe");

❷   switch (fork()) {
    case -1:
        errExit("fork");

    case 0:                      /* Потомок читает из канала */
❸       if (close(pfd[1]) == -1)  /* Записывающий конец не используется */
            errExit("close – child");
```

```
        for (;;) {  /* Считываем данные из канала и направляем их в стандартный вывод */
❹           numRead = read(pfd[0], buf, BUF_SIZE);
            if (numRead == -1)
                errExit("read");
❺           if (numRead == 0)
                break;    /* Конец файла */
❻           if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("child – partial/failed write");
        }

❼       write(STDOUT_FILENO, "\n", 1);
        if (close(pfd[0]) == -1)
            errExit("close");
        _exit(EXIT_SUCCESS);

    default:                        /* Родитель записывает в канал */
❽       if (close(pfd[0]) == -1)  /* Считывающий конец не используется */
            errExit("close – parent");

❾       if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
            fatal("parent – partial/failed write");

❿       if (close(pfd[1]) == -1)  /* Потомок увидит символ завершения файла */
            errExit("close");
⓫       wait(NULL);               /* Ждем завершения потомка */
        exit(EXIT_SUCCESS);
    }
}
```

pipes/simple_pipe.c

44.3. Каналы как средство синхронизации процессов

В разделе 24.5 мы объяснили, как, используя сигналы, можно синхронизировать действия родительского и дочернего процессов, чтобы избежать состояния гонки. Как показано в прототипе программы из листинга 44.3, каналы можно применять для достижения тех же результатов. Данная программа создает несколько дочерних процессов (по одному для каждого аргумента командной строки), каждый из которых должен завершить какое-то действие (для его симуляции мы временно приостанавливаем выполнение с помощью вызова `sleep()`). Родитель ждет, когда все его потомки завершат свои действия.

Для выполнения синхронизации родитель, прежде чем создавать потомков ❷, открывает канал ❶. Каждый потомок наследует файловый дескриптор для записывающего конца канала и закрывает его, как только завершит действие ❸. Когда все потомки закроют дескрипторы, вызов `read()`, сделанный родителем ❺, завершится, возвратив символ конца файла (`0`). После этого родитель может начать выполнять другую работу (обратите внимание: для корректной работы данной методики родительский процесс обязательно должен закрыть записывающий конец канала ❹, который он не использует; в противном случае при попытке чтения из канала он будет заблокирован навсегда).

Пример вывода, показанный ниже, можно увидеть, если запустить программу из листинга 44.3 с тремя дочерними процессами, которые приостанавливаются на 4, 2 и 6 секунд соответственно:

```
$ ./pipe_sync 4 2 6
08:22:16  Parent started
```

```
08:22:18  Child 2 (PID=2445) closing pipe
08:22:20  Child 1 (PID=2444) closing pipe
08:22:22  Child 3 (PID=2446) closing pipe
08:22:22  Parent ready to go
```

Листинг 44.3. Использование канала для синхронизации нескольких процессов

pipes/pipe_sync.c

```
#include "curr_time.h"      /* Определение currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];              /* Канал для синхронизации процессов */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1], "–help") == 0)
        usageErr("%s sleep-time…\n", argv[0]);

    setbuf(stdout, NULL); /* Отключаем буферизацию стандартного вывода,
                             так как мы завершаем потомка с помощью _exit() */
    printf("%s    Parent started\n", currTime("%T"));

(1) if (pipe(pfd) == -1)
        errExit("pipe");

    for (j = 1; j < argc; j++) {
(2)     switch (fork()) {
        case -1:
        errExit("fork %d", j);

        case 0: /* Потомок */
            if (close(pfd[0]) == -1) /* Считывающий конец не используется */
                errExit("close");

    /* Потомок выполняет какую-то работу, после чего позволяет родителю его завершить */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                        /* Симулируем работу */
            printf("%s    Child %d (PID=%ld) closing pipe\n",
                    currTime("%T"), j, (long) getpid());
(3)         if (close(pfd[1]) == -1)
                errExit("close");

            /* Теперь потомок может заняться другими задачами… */

            _exit(EXIT_SUCCESS);

        default: /* Родитель входит в цикл, чтобы создать следующего потомка */
            break;
        }
    }

    /* Дойдя сюда, родитель закрывает записывающий конец канала,
       чтобы мы могли обнаружить символ EOF */

(4) if (close(pfd[1]) == -1)   /* Записывающий конец не используется */
        errExit("close");
```

```
    /* Родитель может заняться другой работой, после чего синхронизируется с потомками */

❺   if (read(pfd[0], &dummy, 1) != 0)
        fatal("parent didn't get EOF");
    printf("%s    Parent ready to go\n", currTime("%T"));

    /* Родитель может приступить к выполнению других задач… */

    exit(EXIT_SUCCESS);
}
```

pipes/pipe_sync.c

Синхронизация с помощью каналов имеет преимущество по сравнению с ранее представленным примером, в котором для этого используются сигналы: она позволяет координировать действия одного процесса с множеством других (родственных) процессов. Стандартные сигналы в данном случае не подходят, поскольку их нельзя поместить в очередь (с другой стороны, их можно транслировать всем членам группы процессов).

Существуют и другие способы синхронизации (например, задействуя несколько каналов). К тому же эту методику можно дополнить таким образом, чтобы вместо закрытия канала каждый потомок записывал в него сообщение с идентификатором процесса и какой-то информацией о своем состоянии. Как вариант, потомок может записывать в канал единственный байт. Родитель впоследствии мог бы сосчитать и проанализировать эти сообщения. Такой подход предохраняет от случайного завершения потомка, игнорируя закрытие канала вручную.

44.4. Использование каналов для соединения фильтров

При создании канала файловым дескрипторам, которые применяются на обоих его концах, назначаются наименьшие доступные номера. Поскольку дескрипторы 0, 1 и 2 обычно заняты, задействуют какие-то более высокие числа. Есть ли способ достичь ситуации, показанной на рис. 44.1, когда два фильтра (то есть программы, которые читают из стандартного ввода и записывают в стандартный вывод) соединены каналом так, чтобы стандартный вывод одной программы был направлен *в* канал, а стандартный ввод другой брался *из* канала? К тому же как этого добиться без изменения исходного кода самих фильтров?

Ответ следующий: дублировать файловые дескрипторы, используя методики, описанные в разделе 5.5. Традиционно для достижения желаемого результата выполняется следующая цепочка вызовов:

```
int pfd[2];

pipe(pfd);   /* Выделим для канала (к примеру) файловые дескрипторы 3 и 4 */

/* Здесь выполняем другие шаги, например fork() */

close(STDOUT_FILENO);    /* Освобождаем файловый дескриптор 1 */
dup(pfd[1]);             /* При дублировании используется наименьший
                            свободный номер дескриптора, то есть fd 1 */
```

Конечным результатом вышеприведенных шагов является то, что стандартный вывод процесса привязывается к записывающему концу канала. Аналогичная цепочка вызовов применяется для привязки считывающего конца канала к стандартному вводу процесса.

Стоит отметить: данные шаги основаны на предположении о том, что файловые дескрипторы процесса с номерами 0, 1 и 2 уже открыты (командная оболочка обычно делает это автоматически для каждой программы, которая в ней выполняется). Если бы дескриптор 0 был закрыт до вышеприведенных вызовов, то мы бы ошибочно привязали к записывающему концу канала стандартной *ввод* процесса. Чтобы исключить такую возможность, `close()` и `dup()` можно заменить вызовом `dup2()`, показанным ниже; это позволит нам явно задать дескриптор, который будет привязан к концу канала:

```
dup2(pfd[1], STDOUT_FILENO);  /* Закрываем дескриптор 1 и повторно устанавливаем
                                 связь с записывающим концом канала */
```

Продублировав `pfd[1]`, мы получили два дескриптора, ссылающихся на записывающий конец канала: дескриптор 1 и `pfd[1]`. Поскольку неиспользуемые файловые дескрипторы канала следует закрывать, мы сделаем это после вызова `dup2()`:

```
close(pfd[1]);
```

Код, показанный выше, требует предварительного открытия стандартного вывода. Предположим, что стандартный ввод/вывод был закрыт перед вызовом `pipe()`. В этом случае вызов `pipe()` выделил бы для канала два дескриптора — например, `pfd[0]` со значением `0` и `pfd[1]` со значением `1`. Следовательно, эквивалентом предыдущих вызовов `dup2()` и `close()` был бы такой код:

```
dup2(1, 1);   /* Ничего не делает */
close(1);     /* Закрывает единственный дескриптор для записывающего конца канала */
```

В целях безопасности эти вызовы рекомендуется заключить внутрь инструкции `if` следующего вида:

```
if (pfd[1] != STDOUT_FILENO) {
    dup2(pfd[1], STDOUT_FILENO);
    close(pfd[1]);
}
```

Пример программы

Программа, представленная в листинге 44.4, использует методики, описанные в данном разделе для получения того же результата, что и у кода из листинга 44.1. Открыв канал, мы создаем два дочерних процесса. Первый привязывает свой стандартный вывод к записывающему концу канала, после чего выполняет команду `ls`. Второй привязывает свой стандартный ввод к считывающему концу канала и выполняет команду `wc`.

Листинг 44.4. Использование канала для соединения команд ls и wc

pipes/pipe_ls_wc.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                 /* Файловые дескрипторы канала */

    if (pipe(pfd) == -1)        /* Создаем канал */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");
```

```
    case 0:             /* Первый потомок выполняет 'ls', записывая результат в канал */
        if (close(pfd[0]) == -1)    /* Считывающий конец не используется */
            errExit("close 1 ");

        /* Дублируем стандартный вывод на записывающем конце канала;
           закрываем лишний дескриптор */

        if (pfd[1] != STDOUT_FILENO) {  /* Проверка на всякий случай */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
            errExit("dup2 1");
            if (close(pfd[1]) == -1)
            errExit("close 2");
        }

        execlp("ls", "ls", (char *) NULL);   /* Записывает в канал */
        errExit("execlp ls");

    default:   /* Родитель выходит из этого блока, чтобы создать следующего потомка */
        break;
    }

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:  /* Второй потомок выполняет 'wc', считывая ввод из канала */
        if (close(pfd[1]) == -1)  /* Записывающий конец не используется */
            errExit("close 3");

        /* Дублируем стандартный ввод на считывающем конце канала;
           закрываем лишний дескриптор */

        if (pfd[0] != STDIN_FILENO) {    /* Проверка на всякий случай */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                errExit("dup2 2");
            if (close(pfd[0]) == -1)
                errExit("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);  /* Читает из канала */
        errExit("execlp wc ");

    default:            /* Родитель выходит из этого блока */
        break;
    }

  /* Родитель закрывает лишние дескрипторы канала и ждет завершения дочерних процессов */

    if (close(pfd[0]) == -1)
        errExit("close 5");
    if (close(pfd[1]) == -1)
        errExit("close 6");
    if (wait(NULL) == -1)
        errExit("wait 1");
    if (wait(NULL) == -1)
        errExit("wait 2");

    exit(EXIT_SUCCESS);
}
```

pipes/pipe_ls_wc.c

Запустив программу из листинга 44.4, мы увидим следующее:

```
$ ./pipe_ls_wc
     24
$ ls | wc -l        Проверяем результаты с помощью консольных команд
     24
```

44.5. Взаимодействие с консольными командами с помощью канала: popen()

Каналы часто используют для выполнения консольных команд — в частности, для считывания их вывода или передачи им какого-нибудь ввода. Для упрощения этой задачи предусмотрены функции `popen()` и `pclose()`:

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

Возвращает файловый поток или `NULL`, если произошла ошибка

```
int pclose(FILE *stream);
```

Возвращает код завершения дочернего процесса или `-1`, если произошла ошибка

Функция `popen()` открывает канал и создает дочерний процесс, запускающий командную оболочку, которая, в свою очередь, создает еще один дочерний процесс для выполнения строки, переданной в аргументе `command`. Аргумент `mode` представляет собой строку, определяющую, будет ли процесс читать из канала (`mode` равен `r`) или записывать в него (`mode` равен `w`). Это взаимоисключающие режимы, поскольку каналы являются однонаправленными. В зависимости от значения аргумента `mode` происходит одно из двух: либо стандартный вывод выполняемой команды соединяется с записывающим концом канала, либо ее стандартный ввод соединяется со считывающим концом канала (рис. 44.4).

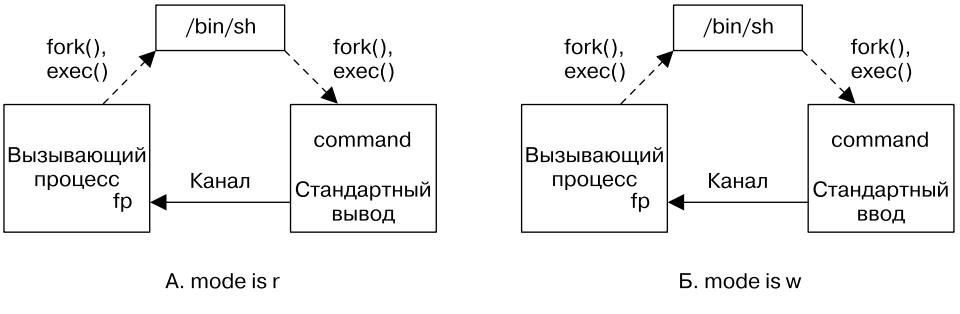


Рис. 44.4. Общая схема взаимодействия процессов и применения канала с помощью вызова popen()

При успешном выполнении `popen()` возвращает указатель на файловый поток, который можно применять в библиотечных функциях стандартного ввода/вывода. При возникновении ошибки (например, когда значение `mode` не равно `r` или `w`, не удалось открыть канал или вызов `fork()`, создающий потомков, завершился неудачей) `popen()` возвращает `NULL` и устанавливает значение `errno`, чтобы сигнализировать о причине ошибки.

После завершения `popen()` вызывающий процесс использует канал для чтения вывода команды `command` или для передачи ей какого-нибудь ввода. Когда команда закрывает записывающий конец канала, вызывающий процесс получает символ конца файла (аналогично ведут себя каналы, созданные с помощью вызова `pipe()`). Если команда закрывает считывающий конец канала, то процесс получает сигнал `SIGPIPE` и ошибку `EPIPE`.

По окончании ввода/вывода вызывается функция `pclose()`, закрывающая канал и ждущая завершения дочерней командной оболочки (функция `fclose()` в этом случае не подходит, так как не ждет завершения потомка). В случае успеха `pclose()` возвращает код завершения (см. раздел 26.1.3) дочерней командной оболочки (который равен коду завершения последней выполненной в нем команды — если только оболочка не была завершена по сигналу). Как и в случае с вызовом `system()` (см. раздел 27.6), если командная оболочка не может быть запущена, то функция `pclose()` возвращает такое же значение, как если бы оболочка была закрыта с помощью вызова `_exit(127)`. В случае возникновения какой-то другой ошибки `pclose()` возвращает `-1`. Возможна также ситуация, в которой не удается получить код завершения. Чуть ниже мы объясним, как это может произойти.

Стандарт SUSv3 требует, чтобы во время ожидания кода завершения дочерней командной оболочки функция `pclose()`, как и `system()`, автоматически перезапускала вызов `waitpid()`, который она выполняет внутри, если тот был прерван обработчиком сигнала.

В целом все написанное нами в разделе 27.6 относительно вызова `system()` применимо и к функции `popen()`. Она всего лишь предоставляет больше удобств: открывает канал, дублирует дескрипторы, закрывает те из них, что не используются, а также берет на себя все нюансы запуска вызовов `fork()` и `exec()`. Кроме того, команда управляется оболочкой. Но за такое удобство приходится платить производительностью, ведь для этого создается два дополнительных процесса: один для оболочки, а другой — для одной или нескольких команд, которые она выполняет. По аналогии с `system()` функция `popen()` никогда не должна вызываться из привилегированных программ.

Вызов `system()` и функции `popen()` и `pclose()` имеют определенное сходство, однако между ними существуют и различия. Оно вытекает из того факта, что `system()` инкапсулирует выполнение консольных команд внутри единого вызова, тогда как в случае с `popen()` вызывающий процесс работает параллельно консольной команде, после чего вызывает `pclose()`. Разница заключается в аспектах, представленных ниже.

- Поскольку вызывающий процесс и выполняемая команда работают параллельно, стандарт SUSv3 требует, чтобы функция `popen()` *не* игнорировала сигналы `SIGINT` и `SIGQUIT`. Если они сгенерированы с помощью клавиатуры, их получает как вызывающий процесс, так и выполняемая команда. Это вызвано тем, что оба процесса находятся в одной и той же группе, а, как мы знаем из раздела 34.5, сигналы, сгенерированные терминалом, получают все участники (активной) группы процессов.
- Между выполнением функций `popen()` и `pclose()` вызывающий процесс может создать других потомков, поэтому стандарт SUSv3 требует, чтобы вызов `popen()` *не* блокировал сигнал `SIGCHLD`. То есть процесс, который вызывает `wait()` перед функцией `pclose()`, может получить статус потомка, созданного с помощью `popen()`. В таком случае последующий вызов `pclose()` возвратит `-1` и присвоит `errno` ошибку `ECHILD`, сигнализируя о невозможности извлечения статуса потомка.

Пример программы

В листинге 44.5 показан пример использования функций `popen()` и `pclose()`. Программа многократно считывает шаблон имени файла **2** и задействует `popen()`, чтобы получить результат передачи этого шаблона команде `ls` **5** (до появления библиотечной функции `glob()` похожий подход использовался в старых версиях UNIX для генерирования имен файлов).

Листинг 44.5. Поиск файлов по шаблону с помощью функции popen()

pipes/popen_glob.c

```
#include <ctype.h>
#include <limits.h>
#include "print_wait_status.h"       /* Для printWaitStatus() */
#include "tlpi_hdr.h"

❶ #define POPEN_FMT "/bin/ls –d %s 2> /dev/null"
#define PAT_SIZE 50
#define PCMD_BUF_SIZE (sizeof(POPEN_FMT) + PAT_SIZE)

int
main(int argc, char *argv[])
{
    char pat[PAT_SIZE];              /* Поиск по шаблону */
    char popenCmd[PCMD_BUF_SIZE];
    FILE *fp;             /* Файловый поток, возвращенный вызовом popen() */
    Boolean badPattern;              /* Некорректные символы в 'pat'? */
    int len, status, fileCnt, j;
    char pathname[PATH_MAX];

    for (;;) {            /* Считываем шаблон, выводим результаты поиска */
        printf("pattern: ");
        fflush(stdout);
❷       if (fgets(pat, PAT_SIZE, stdin) == NULL)
            break;                  /* Конец файла */
        len = strlen(pat);
        if (len <= 1)               /* Пустая строка */
            continue;

        if (pat[len – 1] == '\n')  /* Убираем завершающий символ новой строки */
            pat[len – 1] = '\0';

        /* Следим за тем, чтобы шаблон содержал только допустимые символы, то есть
           буквы, цифры, подчеркивания, точки и символы поиска по шаблону,
           поддерживаемые командной оболочкой (мы используем более строгое
           определение допустимого, чем командная оболочка, которая позволяет
           добавлять любые символы, если они находятся внутри кавычек). */

❸       for (j = 0, badPattern = FALSE; j < len && !badPattern; j++)
            if (!isalnum((unsigned char) pat[j]) &&
                    strchr("_*?[^-].", pat[j]) == NULL)
                badPattern = TRUE;

        if (badPattern) {
            printf("Bad pattern character: %c\n", pat[j – 1]);
            continue;
        }
        /* Создаем и выполняем команду glob 'pat' */

❹       snprintf(popenCmd, PCMD_BUF_SIZE, POPEN_FMT, pat);
        popenCmd[PCMD_BUF_SIZE – 1] = '\0';  /* Добавляем в конце строки
                                                нулевой символ */
❺       fp = popen(popenCmd, "r");
        if (fp == NULL) {
            printf("popen() failed\n");
            continue;
        }
```

```
    /* Перебираем итоговый список путей, пока не доходим до конца файла */

        fileCnt = 0;
        while (fgets(pathname, PATH_MAX, fp) != NULL) {
            printf("%s", pathname);
            fileCnt++;
        }

        /* Закрываем канал, извлекаем и выводим код завершения */

        status = pclose(fp);
        printf("    %d matching file%s\n", fileCnt,
               (fileCnt != 1) ? "s" : "");
        printf("    pclose() status = %#x\n", (unsigned int) status);
        if (status != -1)
            printWaitStatus("\t", status);
    }

    exit(EXIT_SUCCESS);
}
```

pipes/popen_glob.c

Использование программы из листинга 44.5 показано на примере следующей сессии командной строки. Здесь мы указываем два шаблона: первому соответствуют два файла, а для второго не находится ни одного совпадения:

```
$ ./popen_glob
pattern: popen_glob*                Соответствует двум файлам
popen_glob
popen_glob.c
    2 matching files
    pclose() status = 0
        child exited, status=0
pattern: x*                         Не соответствует ни одному файлу
    0 matching files
    pclose() status = 0x100         ls(1) завершается со статусом 1
        child exited, status=1
pattern: ^D$                        Нажмите Ctrl+D, чтобы выйти
```

Построение команды поиска по шаблону ❶❹ в листинге 44.5 требует некоторого объяснения. Сам поиск выполняется командной оболочкой. Команда `ls` используется всего лишь для построчного вывода совпавших файлов. Вместо этого можно было бы воспользоваться командой `echo`, но это повлекло бы нежелательные результаты: если не обнаружено ни одного совпадения, то командная оболочка оставляет шаблон без изменений и просто выводит его с помощью команды `echo`. Команда `ls` в данной ситуации, прежде чем завершиться со статусом 1, возвращает в вывод `stderr` (который мы перенаправили в `/dev/null`) сообщение об ошибке, а стандартный вывод оставляет пустым.

Стоит также обратить внимание на проверку ввода, производимую в листинге 44.5 ❸. Она делается для того, чтобы не дать `popen()` выполнить произвольную консольную команду в случае некорректного ввода. Представьте, что эта проверка не проводится и пользователь ввел такую строку:

```
pattern: ; rm *
```

В этом случае программа передала бы в функцию `popen()` следующую команду, что привело бы к катастрофическим результатам:

```
/bin/ls –d ; rm * 2> /dev/null
```

Подобную проверку всегда следует проводить в программах, запускающих произвольные команды на основе пользовательского ввода, используя функции `popen()` (или `system()`). Как вариант символы, которые не нужно проверять, можно было бы заключить в кавычки — таким образом командная оболочка не стала бы их интерпретировать.

44.6. Каналы и буферизация стандартного ввода/вывода

Поскольку указатель на файловый поток, возвращаемый вызовом `popen()`, не ссылается на терминал, библиотека `stdio` применяет к данному потоку блочную буферизацию (см. раздел 13.2). Это значит, что по умолчанию при выполнении функции `popen()` в режиме `w` вывод передается дочернему процессу на другом конце канала только после заполнения буфера или закрытия самого канала с помощью вызова `pclose()`. Во многих случаях это не вызывает никаких проблем. Однако если нужно, чтобы потомок получал данные из канала немедленно, то есть два варианта: можно периодически вызывать `fflush()` или отключить буферизацию, используя вызов `setbuf(fp, NULL)`. То же самое можно сделать, если мы создаем канал, задействуя системный вызов `pipe()`, а затем применяем `fdopen()` для получения потока стандартного вывода, связанного с записывающим концом канала.

Но все может оказаться сложнее, если процесс, вызывающий `popen()`, читает из канала (то есть при указании режима `r`). В таком случае, если потомок использует библиотеку `stdio` (и при этом не делает вызовов `fflush()` или `setbuf()`), то его вывод дойдет до вызывающего процесса либо после заполнения буфера, либо в результате вызова `fclose()`. (То же самое относится к ситуации, когда мы читаем из канала, созданного с помощью вызова `pipe()`, а процесс на другом конце записывает данные, применяя библиотеку `stdio`.) Если такое поведение вас не устраивает, исправить его почти невозможно — разве что можно отредактировать исходный код программы, выполняющейся в дочернем процессе, и добавить туда вызов `setbuf()` или `fflush()`.

Если у вас нет доступа к исходному коду, канал можно заменить псевдотерминалом — IPC-каналом, один конец которого подключается к процессу и ведет себя как терминал. В итоге библиотека `stdio` будет буферизировать вывод построчно. Псевдотерминалы будут описаны в главе 60.

44.7. Очереди FIFO

С точки зрения семантики очередь FIFO похожа на канал. Ее принципиальное отличие заключается в том, что у нее есть имя в рамках файловой системы и ее можно открывать так же, как обычный файл. Это позволяет использовать очереди FIFO для взаимодействия процессов, не имеющих отношения друг к другу (например, между клиентом и сервером).

Для работы с открытой очередью FIFO служат те же системные вызовы, что и в случае с каналами и другими файлами (то есть `read()`, `write()` и `close()`). По аналогии с каналами очередь FIFO имеет записывающий и считывающий конец, а данные, которые из них читаются, поступают в том же порядке, в каком они были записаны. На этом и основывается название FIFO — *«первым пришел — первым ушел»* (англ. first in, first out). Такие очереди иногда называют *именованными каналами*.

Как и в случае с каналами, данные внутри очереди FIFO теряются при закрытии последнего дескриптора, который на нее ссылается.

Очередь FIFO можно создать, задействуя консольную команду `mkfifo`:

```
$ mkfifo [ -m mode ] pathname
```

где pathname — имя создаваемой очереди, а параметр -m используется, чтобы задать режим доступа (так же как это делается в команде chmod).

Функции fstat() и stat(), если их применить к очереди FIFO (или каналу), возвращают в поле st_mode структуры stat (см. раздел 15.1) тип файла S_IFIFO. При выводе с помощью команды ls –l очередь FIFO содержит в первом столбце тип p, а команда ls –F добавляет к ее имени символ канала (|).

Функция mkfifo() позволяет создать очередь FIFO с заданным именем.

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Аргумент mode задает права доступа к новой очереди FIFO. Они состоят из констант, к которым применяется побитовое ИЛИ (см. табл. 15.4). Как и к обычным правам доступа, к ним применяется атрибут umask (см. подраздел 15.4.6).

Изначально очереди FIFO создавались с помощью системного вызова mknod(pathname, S_IFIFO, 0), который позволяет создавать различные виды файлов, включая файлы устройств (в стандарте SUSv3 применение mknod() ограничивается созданием очередей FIFO). В стандарте POSIX.1-1990 появился более простой, хотя и менее универсальный программный интерфейс, mkfifo(). В большинстве UNIX-систем он реализован в виде обертки вокруг mknod().

Новую очередь FIFO может открыть любой процесс, имеющий соответствующие права доступа (см. подраздел 15.4.3).

Операция открытия очереди FIFO имеет немного непривычную семантику. Обычно единственное разумное применение FIFO подразумевает наличие считывающего и записывающего процессов на каждом конце очереди. Следовательно, ее открытие для чтения (флаг O_RDONLY для вызова open()) блокируется до тех пор, пока другой процесс не откроет ее для записи (флаг O_WRONLY для вызова open()). И наоборот: открытие очереди для записи блокируется, пока другой процесс не откроет ее для чтения. Иными словами, открытие очереди FIFO синхронизирует считывающий и записывающий процессы. Если другой конец очереди уже открыт (возможно, каждый из концов уже открыт другими процессами), то вызов open() сразу же успешно завершается.

В большинстве реализаций UNIX (в том числе и в Linux) блокировку открытия очереди FIFO можно обойти, если указать флаг O_RDWR. В данном случае операция open() немедленно вернет файловый дескриптор для работы с очередью. Однако это противоречит модели ввода/вывода, предусмотренной для очередей FIFO, и в стандарте SUSv3 отдельно подчеркивается, что их открытие с использованием флага O_RDWR приводит к неопределенным результатам; таким образом, при написании портируемых программ применение данного флага следует избегать. Если нужно предотвратить блокировку при открытии очереди FIFO, то функции open() можно передать стандартизированный флаг O_NONBLOCK, предназначенный специально для этого (см. раздел 44.9).

Нежелательность применения флага O_RDWR при открытии очереди FIFO может быть вызвана еще одной причиной. Если передать его функции open(), то вызывающий процесс не сможет увидеть символ конца файла в полученном файловом дескрипторе, поскольку к записывающему концу очереди всегда будет подключен еще один дескриптор — тот, из которого процесс считывает данные.

Использование очередей FIFO и команды tee(1) для создания двойного конвейера

Одним из свойств конвейеров командной строки является их линейность; каждый процесс в конвейере считывает данные, сгенерированные его предшественником, и передает их следующему процессу. Очередь FIFO позволяет создать внутри конвейера ответвление, благодаря которому копия вывода процесса может быть передана еще одному адресату. Для этого необходимо применить команду `tee`, записывающую две копии того, что она прочитала в своем стандартном вводе: одну в стандартный вывод, а другую — в файл, заданный в ее аргументе командной строки.

Если вместо файла передать команде `tee` очередь FIFO, то ее дублированный вывод смогут одновременно читать два процесса. Это показано на примере следующей сессии командной строки. Создается очередь FIFO с именем `myfifo`, запускается фоновая команда `wc`, открывающая данную очередь для чтения (она будет заблокирована до открытия записывающего конца). Затем эта команда выполняет конвейер, который передает вывод `ls` команде `tee` (а та уже отправляет его дальше по конвейеру команде `sort` и в очередь FIFO). Параметр `–k5n` команды `sort` позволяет отсортировать вывод `ls` в порядке возрастания пятого столбца (столбцы разделены пробелами).

```
$ mkfifo myfifo
$ wc –l < myfifo &
$ ls –l | tee myfifo | sort –k5n
(Итоговый вывод не показан)
```

Схематически приведенные команды можно изобразить так, как они показаны на рис. 44.5.

> Программа tee носит такое название благодаря своей «форме» (tee переводится как «тройник»). По принципу работы она похожа на канал, но с дополнительным ответвлением, через которое проходит копия вывода. Схематично это можно представить в виде буквы T (см. рис. 44.5). Помимо решения вышеописанной задачи, команда tee также может пригодиться при отладке конвейеров и для сохранения промежуточных результатов каких-нибудь сложных конвейерных вычислений.

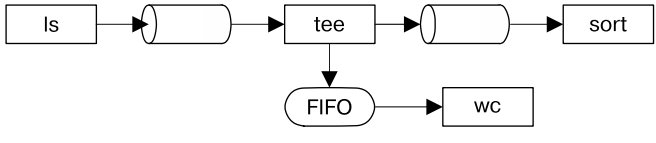


Рис. 44.5. Использование очереди FIFO и команды tee(1) для дублирования конвейера

44.8. Клиент-серверные приложения на основе очередей FIFO

В данном разделе будет представлено простое клиент-серверное приложение, использующее очереди FIFO для межпроцессного взаимодействия. Сервер занимается назначением уникальных последовательных чисел каждому клиенту, который их запрашивает. В ходе описания этого приложения вы познакомитесь с несколькими концепциями и методиками проектирования серверов.

Краткий обзор приложения

В нашем примере все клиенты отправляют серверу свои запросы с помощью единой серверной очереди FIFO. Общеизвестное имя очереди (`/tmp/seqnum_sv`) определяется в заголовочном файле в листинге 44.6. Данное имя — постоянное, поэтому все клиенты знают, как соединиться с сервером. (В примере очередь FIFO создается в каталоге `/tmp`, поскольку это позволяет запускать программы без изменений в большинстве систем. Но, как отмечается в разделе 38.7, создание файлов в публичных каталогах, доступных для записи, таких как `/tmp`, чревато различными проблемами с безопасностью и в реальных приложениях является нежелательным.)

> При работе с клиент-серверными приложениями мы будем постоянно сталкиваться с понятием «общеизвестный адрес» (или имя), с помощью которого сервер становится доступным для клиентов. Это один из механизмов, позволяющий клиентам определять местонахождение сервера. Еще один вариант заключается в предоставлении некоего DNS-сервера, в котором можно регистрировать различные сервисы, предоставляемые другими серверами. Когда клиенту нужно узнать местоположение сервиса, он связывается с DNS-сервером. Благодаря такому решению процедура размещения серверов становится более гибкой, хотя для этого необходимо проделать некую дополнительную работу. Естественно, в таком случае клиенты и серверы должны знать, где находится сам DNS-сервер; обычно он привязан к общеизвестному адресу.

Однако для отправки ответов всем клиентам недостаточно единственной очереди FIFO, поскольку клиенты стали бы за нее соперничать и, возможно, читать ответные сообщения друг друга вместо своих собственных. Таким образом, каждый клиент создает уникальную очередь FIFO, с помощью которой сервер доставляет ему свой ответ. При этом сервер должен знать, как найти данную очередь. Одним из решений проблемы является генерирование клиентом пути к своей очереди и передача этого пути вместе с запросом. Как вариант, клиент и сервер могут задействовать соглашение относительно построения пути к очереди FIFO; в таком случае вместе с запросом клиент будет отправлять информацию, необходимую для определения данного пути. Именно это решение используется в нашем примере. Имя каждой клиентской очереди формируется по шаблону (`CLIENT_FIFO_TEMPLATE`), который состоит из пути с идентификатором клиентского процесса. Применение идентификатора процесса упрощает генерирование имени, уникального для каждого конкретного клиента.

На рис. 44.6 показано, как наше приложение использует очереди FIFO для взаимодействия серверного и клиентского процессов.

Заголовочный файл (см. листинг 44.6) описывает форматы запросов, передающиеся от клиентов к серверу, и ответов, которые сервер возвращает клиентам.

Как вы помните, данные внутри каналов и очередей FIFO представляют собой потоки байтов; границ между отдельными сообщениями не существует. То есть при передаче нескольких сообщений одному процессу (в данном случае серверу) отправитель и получатель должны иметь одинаковое представление о том, как эти сообщения разделяются. Здесь возможны различные подходы.

- Каждое сообщение заканчивается *символом-разделителем*, таким как символ новой строки (примером подобного подхода является функция `readLine()` из листинга 55.1). В нашем случае это значит одно из двух: либо данный символ не должен встречаться в сообщениях, либо нужно его экранировать. Например, слеш плюс символ новой строки могут обозначать настоящий перевод строки в сообщениях, а сам слеш можно представить как `\\`. Одним из недостатков такого подхода является то, что сервер должен перебирать данные в очереди FIFO байт за байтом, пока не найдет символ-разделитель.

- Каждое сообщение содержит *заголовок фиксированной длины*, в котором описывается *количество байтов* в остальной части сообщения. В этом случае считывающий процесс сначала читает заголовок, пришедший из очереди FIFO, и затем на основе полученной информации определяет количество байтов, которое ему нужно прочитать, чтобы извлечь все сообщение целиком. Преимуществом такого подхода является возможность использовать сообщения произвольной длины, а недостатком — потенциальные проблемы в случае, если в канал было записано некорректное сообщение (содержащее заголовок неправильной длины).
- *Сообщения имеют фиксированную длину*, которая известна серверу. Преимущество данного подхода заключается в простоте реализации. Однако таким образом ограничивается максимальный размер передаваемых данных и пропускная способность канала тратится впустую (так как короткие сообщения должны быть чем-нибудь заполнены, чтобы соответствовать установленной длине). Кроме того, если один из клиентов случайно или преднамеренно отправит сообщение нестандартного размера, то это нарушит процесс чтения всех последующих сообщений; в такой ситуации серверу будет непросто восстановить данные.

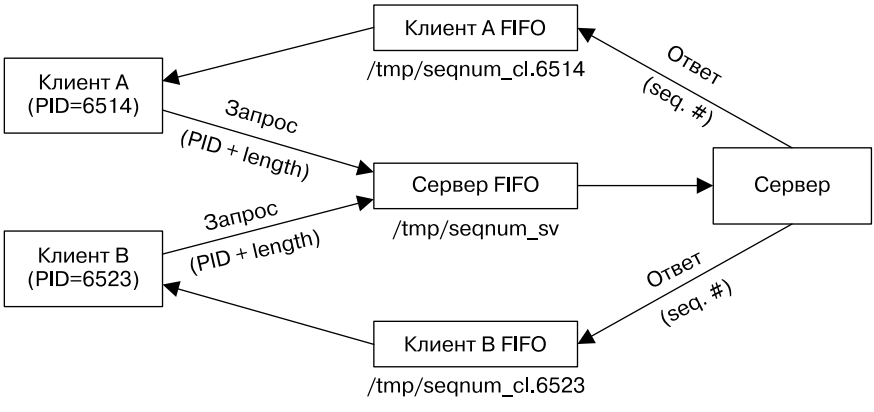


Рис. 44.6. Применение очередей FIFO в приложении с одним сервером и несколькими клиентами

Эти три методики проиллюстрированы на рис. 44.7. Имейте в виду, что в каждой из них общая длина сообщения не должна превышать `PIPE_BUF` байт, иначе оно может быть разбито ядром на несколько частей и перемешано с сообщениями от других отправителей.

> Во всех трех методиках, описанных выше, для приема сообщений от всех клиентов используется единый канал (FIFO). Вместо этого каждое сообщение можно передавать по отдельному соединению. Отправитель открывает канал взаимодействия, отсылает свое сообщение и закрывает канал. Когда процесс-получатель обнаруживает символ конца файла, он знает, что сообщение завершено. Данный подход был бы невозможен, если бы несколько отправителей удерживали открытой одну и ту же очередь FIFO, поскольку при закрытии одним из них этой очереди получатель не смог бы увидеть символ конца файла. Однако такая методика подходит для применения в сочетании с потоковыми сокетами, когда серверный процесс создает уникальный канал взаимодействия для каждого входящего соединения, инициированного клиентом.

Из методик, описанных выше, в нашем примере мы используем третью: каждый клиент отправляет серверу сообщения фиксированной длины. Сообщение определяется структурой `request`, описанной в листинге 44.6. Каждый запрос включает в себя иден-

тификатор клиентского процесса, что позволяет серверу получить имя очереди FIFO, которую клиент применяет для получения ответа. Запрос также содержит поле `seqLen`, определяющее, сколько последовательных чисел нужно выделить для соответствующего клиента. Ответное сообщение, отправляемое сервером, состоит из единственного поля `seqNum`, которое представляет собой начальное число в диапазоне, выделенном для клиента.

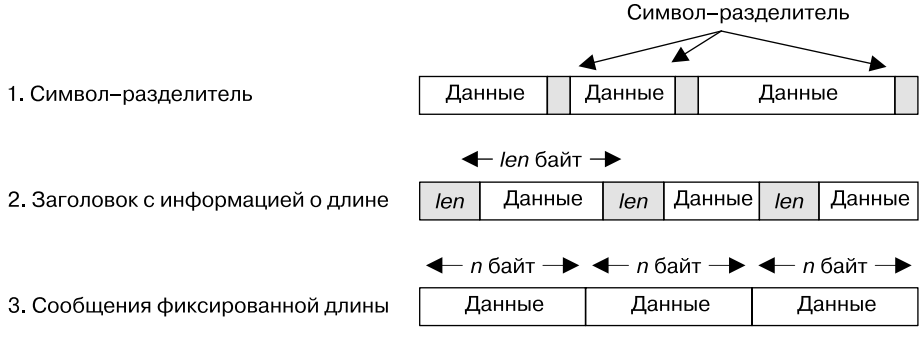


Рис. 44.7. Разделение сообщений в байтовом потоке

Листинг 44.6. Заголовочный файл для fifo_seqnum_server.c и fifo_seqnum_client.c

pipes/fifo_seqnum.h

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define SERVER_FIFO "/tmp/seqnum_sv"
     /* Общеизвестное имя серверной очереди FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
     /* Шаблон для построения имени клиентской очереди FIFO */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
     /* Место, необходимое для пути клиентской очереди FIFO */

struct request {            /* Запрос (клиент --> сервер)
*/
    pid_t pid;              /* PID клиента */
    int seqLen;             /* Длина запрашиваемой последовательности */
};

struct response {           /* Ответ (сервер --> клиент) */
    int seqNum;             /* Начало последовательности */
};
```

pipes/fifo_seqnum.h

Серверная программа

В листинге 44.7 представлен код сервера, выполняющего следующие действия.

- Создает серверную очередь FIFO с общеизвестным именем ❶ и открывает ее для чтения ❷. Он должен быть запущен раньше любого из клиентов, чтобы на момент отправки клиентского запроса очередь FIFO уже существовала. Операция `open()`, выполняемая сервером, блокируется до тех пор, пока первый клиент не откроет другой конец очереди для записи.

- Снова открывает серверную очередь FIFO ❸, на этот раз для записи. Данная операция никогда не будет блокироваться, поскольку очередь FIFO уже была открыта для чтения. Так делается для того, чтобы сервер не увидел символ завершения файла в случае, если все клиенты закроют записывающий конец очереди.
- Игнорирует сигнал `SIGPIPE` ❹. Если сервер попытается записать в клиентскую очередь FIFO, у которой нет считывающего процесса, то он всего лишь получит ошибку `EPIPE` из системного вызова `write()`. В противном случае ему был бы послан сигнал `SIGPIPE` (по умолчанию завершающий процесс).
- Входит в цикл, который считывает входящие клиентские запросы и отвечает на каждый из них ❺. Чтобы отправить ответ, сервер формирует имя очереди FIFO клиента ❻ и затем открывает ее ❼.
- Столкнувшись с ошибкой при открытии клиентской очереди FIFO, сервер отклоняет запрос клиента ❽.

Это пример *итерационного сервера*, который последовательно считывает и обрабатывает запросы каждого из клиентов. Модель данного сервера подходит в ситуации, когда клиентские запросы можно быстро обработать и вернуть ответ, чтобы не задерживать других клиентов. Альтернативой ему является *параллельный сервер*, обрабатывающий каждый клиентский запрос в отдельном дочернем процессе (или потоке). Мы вернемся к проектированию серверных приложений в главе 56.

Листинг 44.7. Итерационный сервер на основе очередей FIFO

pipes/fifo_seqnum_server.c

```
#include <signal.h>
#include "fifo_seqnum.h"
int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0;                /* Это наш «сервис» */

    /* Создаем общеизвестную очередь FIFO и открываем ее для чтения */
    umask(0);                      /* Получаем нужные нам права доступа */
❶   if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
            && errno != EEXIST)
        errExit("mkfifo %s", SERVER_FIFO);
❷   serverFd = open(SERVER_FIFO, O_RDONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

    /* Открываем дополнительный записывающий дескриптор,
       чтобы никогда не получить символ конца файла */
❸   dummyFd = open(SERVER_FIFO, O_WRONLY);
    if (dummyFd == -1)
        errExit("open %s", SERVER_FIFO);

❹   if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");

❺   for (;;) {              /* Считываем запросы и отправляем ответы */
        if (read(serverFd, &req, sizeof(struct request))
```

```
                != sizeof(struct request)) {
            fprintf(stderr, "Error reading request; discarding\n");
            continue;     /* Либо частичное прочтение, либо ошибка */
        }

        /* Открываем клиентскую очередь FIFO (предварительно созданную клиентом) */

❻       snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                 (long) req.pid);
❼       clientFd = open(clientFifo, O_WRONLY);
        if (clientFd == -1) {  /* Открыть не удалось, отклоняем запрос */
            errMsg("open %s", clientFifo);
❽           continue;
        }

        /* Отправляем ответ и закрываем очередь FIFO */

        resp.seqNum = seqNum;
        if (write(clientFd, &resp, sizeof(struct response))
                != sizeof(struct response))
            fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
        if (close(clientFd) == -1)
            errMsg("close");

        seqNum += req.seqLen; /* Обновляем номер нашей последовательности */
    }
}
```

pipes/fifo_seqnum_server.c

Клиентская программа

В листинге 44.8 приведен код клиента, выполняющего следующие действия.

- Создает очередь FIFO, которая будет использоваться для получения ответа от сервера ❷. Это делается до отправки запроса, чтобы на момент, когда сервер попытается ее открыть и послать сообщение, очередь уже существовала.
- Создает сообщение для сервера, состоящее из идентификатора клиентского процесса и числа (взятого из опционального аргумента командной строки), обозначающего длину последовательности, которую клиент хочет получить от сервера ❹ (если аргумент командной строки не указан, по умолчанию берется длина 1).
- Открывает серверную очередь FIFO ❺ и отправляет серверу сообщение ❻.
- Открывает клиентскую очередь FIFO ❼, считывает и выводит ответ сервера ❽.

Еще один момент, на который стоит обратить внимание, — обработчик выхода ❶. Он устанавливается с помощью вызова `atexit()` ❸ и следит за тем, чтобы при завершении клиентского процесса его очередь FIFO была удалена. Как вариант можно было бы просто добавить вызов `unlink()` сразу после открытия клиентской очереди. Такой подход сработал бы, ведь на данном этапе клиент и сервер уже выполнили бы блокирующие вызовы `open()` и обладали бы открытыми файловыми дескрипторами для очереди; кроме того, удаление имени очереди FIFO из файловой системы не повлияло бы на эти дескрипторы (или на открытые файловые дескрипторы, на которые они указывают).

Вот пример того, что можно увидеть при запуске клиентской и серверной программ:

```
$ ./fifo_seqnum_server &
[1] 5066
$ ./fifo_seqnum_client 3    Запрашиваем последовательность из трех чисел
0                           Назначенная последовательность начинается с 0
```

```
$ ./fifo_seqnum_client 2    Запрашиваем последовательность из двух чисел
3                           Назначенная последовательность начинается с 3
$ ./fifo_seqnum_client      Запрашиваем одно число
5
```

Листинг 44.8. Клиент для сервера, генерирующего числовые последовательности
pipes/fifo_seqnum_client.c

```
#include "fifo_seqnum.h"

static char clientFifo[CLIENT_FIFO_NAME_LEN];

static void     /* Вызывается при выходе для удаления клиентской очереди FIFO */
(1) removeFifo(void)
{
    unlink(clientFifo);
}

int
main(int argc, char *argv[])
{
    int serverFd, clientFd;
    struct request req;
    struct response resp;

    if (argc > 1 && strcmp(argv[1], "–help") == 0)
        usageErr("%s [seq-len…]\n", argv[0]);

  /* Создаем нашу очередь FIFO (до отправки запроса, чтобы избежать состояния гонки) */
    umask(0);              /* Получаем нужные нам права доступа */
(2) snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
            (long) getpid());
    if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
                && errno != EEXIST)
        errExit("mkfifo %s", clientFifo);

(3) if (atexit(removeFifo) != 0)
        errExit("atexit");

    /* Создаем запрос, открываем серверную очередь FIFO и отправляем запрос */
(4) req.pid = getpid();
    req.seqLen = (argc > 1) ? getInt(argv[1], GN_GT_0, "seq-len") : 1;

(5) serverFd = open(SERVER_FIFO, O_WRONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

(6) if (write(serverFd, &req, sizeof(struct request)) !=
            sizeof(struct request))
        fatal("Can't write to server");

    /* Открываем нашу очередь FIFO, считываем и выводим ответ */
(7) clientFd = open(clientFifo, O_RDONLY);
    if (clientFd == -1)
        errExit("open %s", clientFifo);

(8) if (read(clientFd, &resp, sizeof(struct response))
            != sizeof(struct response))
        fatal("Can't read response from server");
```

```
    printf("%d\n", resp.seqNum);
    exit(EXIT_SUCCESS);
}
```

pipes/fifo_seqnum_client.c

44.9. Неблокирующий ввод/вывод

Как отмечалось ранее, при открытии очереди FIFO процесс блокируется, если другой ее конец еще не был открыт. Иногда такая блокировка является нежелательной, и для этих случаев при вызове `open()` можно указать флаг `O_NONBLOCK`:

```
fd = open("fifopath", O_RDONLY | O_NONBLOCK);
if (fd == -1)
    errExit("open");
```

Если другой конец очереди уже открыт, то флаг `O_NONBLOCK` никак не влияет на вызов `open()` — очередь сразу же успешно открывается, как обычно. Действие этого флага проявляется только в случае, если другой конец очереди FIFO еще не открыт, и зависит оно от того, какой именно конец открывается — считывающий или записывающий:

- если очередь FIFO открывается для чтения и ни один процесс не подключен к ее записывающему концу, то вызов `open()` сразу же успешно завершается (как будто другой конец уже был открыт);
- если очередь FIFO открывается для записи и ни один процесс не подключен к ее считывающему концу, то вызов `open()` завершается ошибкой `ENXIO`.

Асимметричность флага `O_NONBLOCK`, зависящая от того, какой конец очереди FIFO открывается, можно объяснить следующим образом. При открытии очереди для чтения отсутствие записывающего процесса не является проблемой, поскольку любая попытка чтения из очереди просто не возвращает никаких данных. Но если попытаться записать в очередь, к которой не подключен считывающий процесс, то вызов `write()` приведет к получению сигнала `SIGPIPE` и ошибки `EPIPE`.

В табл. 44.1 описывается семантика открытия очереди FIFO, в том числе и с учетом флага `O_NONBLOCK`, описанного выше.

Таблица 44.1. Семантика вызова open() для очереди FIFO

Тип open()		**Результат вызова open()**	
Открытие для	**Дополнительные флаги**	**Другой конец FIFO открыт**	**Другой конец FIFO закрыт**
Чтение	Нет (блокировка)	Сразу же успешно завершается	Блокируется
	O_NONBLOCK	Сразу же успешно завершается	Сразу же успешно завершается
Запись	Нет (блокировка)	Сразу же успешно завершается	Блокируется
	O_NONBLOCK	Сразу же успешно завершается	Выдается ошибка (ENXIO)

Использование флага `O_NONBLOCK` при открытии очередей FIFO служит двум основным целям:

- позволяет одному процессу открыть оба конца очереди. Сначала процесс открывает считывающий конец, указывая флаг `O_NONBLOCK`, и затем открывает конец для записи;

- предотвращает взаимную блокировку процессов, которые открывают две очереди FIFO.

Взаимная блокировка — ситуация, при которой два или более процесса блокируются, потому что каждый из них ждет, когда другой завершит выполнение каких-то действий. Два процесса, изображенные на рис. 44.8, блокируют друг друга. Каждый из них ждет возможности открыть очередь FIFO для чтения. Такого положения можно было бы избежать, если бы каждый из процессов мог выполнить свой следующий шаг (открытие очереди для записи). Эту конкретную проблему можно было бы решить, изменив порядок выполнения шагов 1 и 2 в процессе Y, не трогая процесс X (или наоборот). Однако в некоторых приложениях достичь такого порядка выполнения может быть непросто. Вместо этого при открытии очереди FIFO для чтения один из процессов (или оба) может указать флаг `O_NONBLOCK`.

Процесс X	Процесс Y
1. Открытие FIFO для чтения блокируется.	1. Открытие FIFO для чтения блокируется.
2. Открытие FIFO для записи	2. Открытие FIFO для записи

Рис. 44.8. Взаимная блокировка процессов, открывающих две очереди FIFO

Неблокирующие чтение и запись

Флаг `O_NONBLOCK` способен влиять на семантику не только вызова `open()`, но и последующих операций `read()` и `write()`, так как продолжает действовать для открытого файлового дескриптора. Мы рассмотрим это влияние в следующем разделе.

Но иногда нужно изменить состояние флага `O_NONBLOCK` для очереди FIFO (или файла другого типа). Такая необходимость может возникнуть в следующих случаях.

- Очередь FIFO открыта с флагом `O_NONBLOCK`, но мы хотим, чтобы последующие вызовы `read()` и `write()` выполнялись в блокирующем режиме.
- Мы хотим включить неблокирующий режим для файлового дескриптора, возвращенного вызовом `pipe()`. Если взять более универсальный случай, может возникнуть необходимость изменить неблокирующее состояние любого файлового дескриптора, полученного любым способом, кроме как из вызова `open()`. Например, это может быть дескриптор, возвращенный вызовом `socket()`, или один из трех стандартных дескрипторов, которые автоматически открываются для каждой программы, запускаемой из командной оболочки.
- Логика нашего приложения может потребовать установки и сброса флага `O_NONBLOCK` для какого-нибудь файлового дескриптора.

Для этих целей послужит вызов `fcntl()`, который включает или выключает флаг состояния открытого файла `O_NONBLOCK`. Чтобы включить данный флаг, можно воспользоваться следующим кодом (опуская проверку ошибок):

```
int flags;

flags = fcntl(fd, F_GETFL);  /* Получаем флаги состояния открытых файлов */
flags |= O_NONBLOCK;         /* Включаем бит O_NONBLOCK */
fcntl(fd, F_SETFL, flags);   /* Обновляем флаги состояния открытых файлов */
```

А для его выключения достаточно следующих строк:

```
flags = fcntl(fd, F_GETFL);
flags &= ~O_NONBLOCK;          /* Выключаем бит O_NONBLOCK */
fcntl(fd, F_SETFL, flags);
```

44.10. Семантика вызовов read() и write() в контексте каналов и очередей FIFO

В табл. 44.2 описана семантика операции `read()` для каналов и очередей FIFO, в том числе и с учетом действия флага `O_NONBLOCK`.

Единственное различие между блокирующим и неблокирующим чтением проявляется, когда при отсутствии данных открывается записывающий конец. В этом случае обычный вызов `read()` блокируется, а его неблокирующая версия завершается ошибкой `EAGAIN`.

Таблица 44.2. Семантика чтения n байт из канала или очереди FIFO, содержащих p байт

O_NONBLOCK включен?	**Данные, доступные в канале или очереди FIFO (p)**			
	p = 0, записывающий конец открыт	**p = 0, записывающий конец закрыт**	**p < n**	**p >= n**
Нет	Блокируется	Возвращает 0 (EOF)	Считывает p байт	Считывает n байт
Да	Возникает ошибка (EAGAIN)	Возвращает 0 (EOF)	Считывает p байт	Считывает n байт

Влияние флага `O_NONBLOCK` на запись в канал или очередь FIFO оказывается более сложным из-за наличия ограничения `PIPE_BUF`. Поведение операции `write()` описано в табл. 44.3.

Таблица 44.3. Семантика записи n байт в канал или очередь FIFO

O_NONBLOCK включен?	**Считывающий конец открыт**		**Считывающий конец закрыт**
	n <= PIPE_BUF	**n > PIPE_BUF**	
Нет	Автоматически записывает n байт; может блокироваться, пока не будет прочитано достаточно данных, чтобы можно было выполнить write()	Записывает n байт; может блокироваться, пока не будет прочитано достаточно данных для завершения write(); данные разных записывающих процессов могут перемешаться	SIGPIPE + EPIPE
Да	Если для немедленной записи n байт достаточно места, write() завершается успешно автоматически; в противном случае возвращается ошибка EAGAIN	Если у нас хватает места для немедленной записи некоего количества данных, записывается от 1 до n байт (которые могут перемешаться с данными, записываемыми другими процессами); в противном случае write() завершается ошибкой EAGAIN	

Флаг `O_NONBLOCK` приводит к сбою записи в канал или очередь FIFO (с ошибкой `EAGAIN`) в любой ситуации, в которой немедленная передача данных невозможна. Это значит следующее: при записи `PIPE_BUF` байт вызов `write()` завершится неудачей, если в канале или очереди FIFO окажется недостаточно места; такая ситуация связана с тем, что ядро не может немедленно завершить операцию и в то же время не может выполнить частичную запись, поскольку это бы нарушило требование к атомарности записи сообщений, не превышающих `PIPE_BUF` байт.

Если за один раз записывается больше PIPE_BUF байт, то процедура записи не обязательно должна быть атомарной. В связи с этим вызов write() передает как можно больше данных (частичная запись), чтобы заполнить канал или очередь FIFO. В таком случае write() возвращает количество переданных байтов, а вызывающий процесс должен повторить попытку записи позже — для передачи оставшихся данных. Но если канал или очередь FIFO оказываются заполненными и не могут принять ни одного байта, то вызов write() завершается ошибкой EAGAIN.

44.11. Резюме

Каналы стали первым средством межпроцессного взаимодействия в UNIX-системах. Они часто используются в командной оболочке и других приложениях. Канал является однонаправленным потоком байтов с ограниченной пропускной способностью, который можно применять для организации взаимодействия родственных процессов. В него можно записать блок данных любого размера, но только блоки, не превышающие PIPE_BUF байт, являются гарантировано атомарными. Каналы применяются не только для IPC, но и как метод синхронизации процессов.

Задействуя каналы, следует проявлять осмотрительность при закрытии неиспользуемых дескрипторов, чтобы считывающий процесс смог обнаружить символ конца файла, а записывающий процесс — получить сигнал SIGPIPE или ошибку EPIPE (обычно проще всего сделать так, чтобы записывающий процесс игнорировал сигнал SIGPIPE и обнаруживал «сбой» канала с помощью ошибки EPIPE).

Функции popen() и pclose() позволяют программе передавать или получать данные от стандартных консольных команд, не вникая в подробности создания канала, запуска командной оболочки и закрытия неиспользуемых файловых дескрипторов.

Очереди FIFO работают по тому же принципу, что и каналы, но со следующими особенностями: для их создания применяется вызов mkfifo(), они имеют имя в рамках файловой системы и могут быть открыты любым процессом с подходящими правами доступа. Открытие очереди FIFO для чтения по умолчанию блокируется до тех пор, пока другой процесс не откроет ее для записи (и наоборот).

В этой главе мы рассмотрели целый ряд сопутствующих тем. Вначале мы узнали, как дублировать файловые дескрипторы, чтобы привязать стандартный вывод или ввод фильтра к каналу. В ходе рассмотрения примера клиент-серверного приложения, основанного на очередях FIFO, мы затронули несколько тем, касающихся клиент-серверной архитектуры, включая назначение серверу общеизвестного адреса и разницу между итерационным и параллельным подходами. При разработке примера программы, работающей с очередью FIFO, мы отметили, что данные, передающиеся по каналу в виде байтовых потоков, иногда имеет смысл разбивать на отдельные сообщения; мы познакомились с различными способами, как можно этого добиться.

В конце мы обратили внимание на то, как флаг O_NONBLOCK (неблокирующий ввод/вывод) влияет на открытие очереди FIFO и на передачу данных в/из нее. Этот флаг может пригодиться, если мы не хотим блокировать работу, открывая очередь. Он также может оказаться полезным в ситуации, когда нужно предотвратить блокировку чтения, если данные отсутствуют, или записи, если в канале или очереди FIFO не хватает места.

Дополнительная информация

Реализация каналов рассматривается в [Bach, 1986] и [Bovet & Cesati, 2005]. Полезные подробности о каналах и очередях FIFO можно найти в [Vahalia, 1996].

44.12. Упражнения

44.1. Напишите программу, которая обеспечивает двунаправленное взаимодействие родительского и дочернего процесса с помощью двух каналов. Родитель должен входить в цикл, считывая блоки текста из стандартного ввода и использовать один из каналов для передачи этого текста своему потомку. Тот, в свою очередь, должен перевести текст в верхний регистр и вернуть его обратно родителю по другому каналу. Родитель считывает полученный от дочернего процесса текст, направляет его в стандартный вывод и возвращается к выполнению цикла.

44.2. Реализуйте функции `popen()` и `pclose()`. Они не требуют обработки сигнала, которая происходит в вызове `system()` (см. раздел 27.7), но вам все равно нужно быть осторожными, чтобы правильно подключить концы канала к файловым потокам каждого из процессов и убедиться в том, что все неиспользуемые дескрипторы, ссылающиеся на канал, закрыты. Поскольку потомки, созданные несколькими вызовами `popen()`, могут выполняться одновременно, вам придется создать структуру данных, связывающую указатели на файловый поток с идентификаторами соответствующих процессов. (Если воспользоваться для этого массивом, значение, возвращаемое функцией `fileno()` и содержащее дескриптор соответствующего файлового потока, можно применять в качестве индекса.) Получение из данной структуры корректного идентификатора позволит функции `pclose()` выбрать потомка, завершения работы которого нужно ждать. Кроме того, применение такой структуры будет соответствовать требованию стандарта SUSv3, согласно которому новый дочерний процесс должен закрыть все открытые файловые потоки, созданные ранее с помощью вызовов `popen()`.

44.3. Сервер в листинге 44.7 (`fifo_seqnum_server.c`) при каждом запуске начинает последовательность с 0. Измените программу так, чтобы она использовала запасной файл, обновляемый при назначении каждого элемента последовательности (вам может пригодиться флаг `O_SYNC` для операции `open()`, описанный в подразделе 4.3.1). При запуске программа должна проверить наличие файла и при его наличии применить содержащееся в нем значение для инициализации текущего элемента последовательности. Если запасной файл не найден, то программа должна его создать и начать назначение чисел последовательности с 0 (в качестве альтернативы можно было бы воспользоваться отображением файла в память, описанным в главе 45).

44.4. Добавьте в сервер из листинга 44.7 (`fifo_seqnum_server.c`) код, который при получении сигналов `SIGINT` или `SIGTERM` удаляет серверную очередь FIFO и завершает программу.

44.5. Сервер в листинге 44.7 (`fifo_seqnum_server.c`) выполняет повторную операцию открытия очереди FIFO с флагом `O_WRONLY`, чтобы никогда не встретить символ конца файла при использовании считывающего дескриптора (`serverFd`). Вместо этого можно применить альтернативный подход: при обнаружении символа конца файла сервер закрывает считывающий дескриптор и снова открывает очередь FIFO для чтения (эта операция будет заблокирована, пока следующий клиент не откроет очередь для записи). Какой недостаток у данного подхода?

44.6. Сервер в листинге 44.7 (`fifo_seqnum_server.c`) полагается на предсказуемое поведение клиентских процессов. Но если клиент создаст клиентскую очередь FIFO и отправит серверу запрос, но сама очередь при этом не будет открыта, то попытка сервера ее открыть заблокируется, а запросы других клиентов останутся без ответа (такие действия, если они выполняются умышленно, можно квалифицировать как DoS-атаку — от англ. denial-of-service attack). Разработайте план для борьбы с данной проблемой. Дополните код сервера (а возможно, и клиента из листинга 44.8) соответствующим образом.

44.7. Напишите программу для проверки работы неблокирующих операций открытия и ввода/вывода в контексте очередей FIFO (см. раздел 44.9).

45 Отображение в память

Эта глава посвящена системному вызову `mmap()`, который предназначен для создания отображений в память. Они могут использоваться для межпроцессного взаимодействия, а также ряда других задач. Сначала мы кратко рассмотрим основные понятия, связанные с данным вызовом, а затем изучим его более глубоко.

45.1. Краткий обзор

Системный вызов `mmap()` создает в виртуальном адресном пространстве процесса новое *отображение в память*. Отображения бывают двух видов.

- *Отображение файла*. Файл отражается непосредственно на виртуальную память вызывающего процесса. По завершении этой процедуры его содержимое становится доступным для байтовых операций на соответствующем участке памяти. Страницы отображения по мере необходимости (автоматически) загружаются из соответствующего файла. Данный вид отображения также известен как *отображение, основанное на файле,* или *файл, отображенный в память*.
- *Анонимное отображение*. С ним не связан никакой файл. Вместо этого его страницы заполняются нулями.

  Данную процедуру также можно представить в виде отображения виртуального файла, который всегда содержит нули (что на самом деле ближе к действительности).

Участок памяти в отображении одного процесса можно разделять с отображением другого процесса (то есть записи таблицы со страницами каждого из процессов будут указывать на одни и те же страницы физической памяти). Этого можно достичь двумя способами:

- когда два процесса отображают один и тот же участок файла, страницы памяти, к которым они получают доступ, становятся для них общими;
- дочерний процесс, созданный с помощью вызова `fork()`, наследует копии родительских отображений, указывающих на те же страницы физической памяти, что и отображения родителя.

Когда два или более процесса разделяют одни и те же страницы, каждый из них может видеть изменения, вносимые в эти страницы другими процессами. Однако данное положение зависит от того, является отображение *приватным* или *разделяемым*.

- *Приватное отображение (MAP_PRIVATE)*. Изменения, вносимые в содержимое отображения, остаются невидимыми для других процессов; в случае с файловым отображением они не передаются в исходный файл. В вышеописанной ситуации страницы приватного отображения изначально являются разделяемыми, однако изменения каждого отображения распространяются только на процесс, которому они принадлежат. Чтобы достичь этого, ядро использует методику копирования при записи (см. подраздел 24.2.2). То есть когда процесс пытается изменить содержимое страницы, ядро

создает для него ее копию (и корректирует таблицу страниц процесса). В связи с этим отображение `MAP_PRIVATE` иногда называют *приватным, копируемым при записи*.

- *Разделяемое отображение (MAP_SHARED)*. Изменения, вносимые в содержимое отображения, доступны другим процессам, которые разделяют то же отображение; в случае с файловым отображением изменения передаются в исходный файл.

Два атрибута отображений, описанных выше (файловые/анонимные, приватные/разделяемые), можно комбинировать четырьмя разными способами, как показано в табл. 45.1.

Таблица 45.1. Назначения разных видов отображения в память

Видимость изменений	**Вид отображения**	
	Файловое	**Анонимное**
Приватное	Инициализация памяти из содержимого файла	Выделение памяти
Разделяемое	Ввод/вывод, отображенный в память; разделение памяти между процессами (IPC)	Разделение памяти между процессами (IPC)

Ниже описано, как создается и используется каждый из четырех типов отображения.

- *Приватное файловое отображение*. Содержимое отображения инициализируется с помощью участка файла. Несколько процессов, отображающих один файл, изначально разделяют одни и те же страницы памяти; но, поскольку задействовано копирование при записи, изменения, вносимые процессом в отображение, не видны остальным процессам. Основное применение этого вида отображения заключается в инициализации участка памяти, используя содержимое файла. Например, можно инициализировать сегменты процесса, хранящие его код и данные, заполнив их соответствующими участками двоичного исполняемого файла или разделяемой библиотеки.
- *Приватное анонимное отображение*. Каждое такое отображение, создаваемое с помощью вызова `mmap()`, отличается от других отображений данного вида, созданных тем же (или другим) процессом (это значит, что они не применяют одни и те же страницы физической памяти). И хотя дочерний процесс наследует отображения родителя, процедура копирования при записи гарантирует, что после вызова `fork()` родитель и потомок не будут видеть изменения друг друга, вносимые в это отображение. Основное применение приватного анонимного отображения заключается в выделении (и заполнении нулями) памяти для процесса (например, при выделении больших блоков памяти `malloc()` использует вызов `mmap()`).
- *Разделяемое файловое отображение*. Все процессы, отображающие один участок того же файла, разделяют одни и те же страницы физической памяти, которые изначально инициализированы с помощью этого участка. Изменения, вносимые в отображение, передаются обратно в файл. Отображение данного типа служит двум целям. Во-первых, оно позволяет *отобразить в память ввод/вывод*. Это значит, что файл загружается на какой-то участок виртуальной памяти процесса и все изменения, вносимые в данную память, автоматически записываются в сам файл. Таким образом, ввод/вывод, отображенный в память, является альтернативой вызовам `read()` и `write()`, которые используются для чтения и записи файла. Во-вторых, этот вид отображения позволяет неродственным процессам разделять один и тот же участок памяти для обеспечения (быстрого) межпроцессного взаимодействия.

- *Разделяемое анонимное отображение*. Как и в случае с приватным анонимным отображением, каждый вызов `mmap()` создает новое отображение со своими отдельными страницами памяти. Разница такова: к страницам отображения не применяется процедура копирования при записи. Это значит, что, когда потомок после вызова `fork()` наследует отображение родителя, родительский и дочерний процессы разделяют одни и те же страницы физической памяти, и изменения, вносимые одним процессом, видны другому. Разделяемые анонимные отображения позволяют организовать межпроцессное взаимодействие по тому же принципу, что и в случае с сегментами разделяемой памяти в System V, но только между родственными процессами.

Мы подробно рассмотрим каждый из этих типов отображений в оставшейся части данной главы.

Отображения теряются, когда процесс выполняет вызов `exec()`, но наследуются потомками при вызове `fork()`. Тип отображения (`MAP_PRIVATE` или `MAP_SHARED`) тоже наследуется.

Информация обо всех отображениях процесса доступна в специальном файле вида `/proc/PID/maps` (только в Linux).

Вызов mmap() можно также применять в сочетании с POSIX-объектами разделяемой памяти, что позволяет разделять участок памяти между неродственными процессами, не создавая при этом на диске связанный с ним файл (как в случае с разделяемым отображением файла). POSIX-объекты разделяемой памяти будут описаны в главе 50).

45.2. Создание отображения: mmap()

Системный вызов `mmap()` создает новое отображение в виртуальном адресном пространстве вызывающего процесса.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd,
           off_t offset);
```

Возвращает начальный адрес отображения при успешном завершении или `MAP_FAILED`, если произошла ошибка

Аргумент `addr` обозначает виртуальный адрес, по которому будет находиться отображение. Если присвоить ему значение `NULL`, то ядро само выберет подходящий адрес. Это предпочтительный способ создания отображения. Но также можно указать ненулевой адрес, который будет учитываться ядром при размещении отображения в памяти. На практике данный адрес будет как минимум округлен, чтобы совпасть с началом ближайшей страницы. В любом случае ядро выберет адрес, не конфликтующий ни с одним другим отображением (если аргумент `flags` содержит значение `MAP_FIXED`, то аргумент `addr` должен быть привязан к странице; этот флаг описывается в разделе 45.10).

В случае успеха `mmap()` возвращает начальный адрес нового отображения. В случае ошибки возвращается значение `MAP_FAILED`.

В Linux (как и в большинстве других разновидностей UNIX) константа MAP_FAILED равна ((void *) – 1). Но в стандарте SUSv3 она указана по той причине, что спецификация языка C не может гарантировать отличие ((void *) – 1) от значения, которое возвращается при успешном выполнении mmap().

Аргумент length обозначает размер отображения в байтах. И хотя он не должен быть кратным размеру страниц памяти в системе (возвращаемому вызовом sysconf(_SC_PAGESIZE)), при создании отображения ядро использует единицы измерения, благодаря которым length автоматически округляется до ближайшего следующего числа, кратного размеру страницы.

Аргумент prot представляет собой битовую маску, позволяющую задать защиту отображения. Он может быть равен либо значению PROT_NONE, либо сочетанию любых других трех флагов (к которым применяется побитовое ИЛИ), перечисленных в табл. 45.2.

Таблица 45.2. Значения защиты памяти

Значение	Описание
PROT_NONE	Этот участок недоступен
PROT_READ	Содержимое данного участка можно прочитать
PROT_WRITE	Содержимое этого участка можно изменить
PROT_EXEC	Содержимое данного участка можно выполнить

Битовая маска flags состоит из параметров, управляющих различными аспектами работы отображения. В их число обязательно должно входить *одно* из следующих значений.

- MAP_PRIVATE — создает приватное отображение. Изменения, вносимые в содержимое участка, не видны другим процессам, использующим то же отображение; в случае с файловым отображением изменения не передаются обратно в исходный файл.
- MAP_SHARED — создает разделяемое отображение. Изменения, вносимые в содержимое участка, видны другим процессам, отображающим тот же участок с помощью атрибута MAP_SHARED; в случае с файловым отображением изменения передаются обратно в исходный файл. Обновление файла может происходить не сразу; см. описание системного вызова msync() в разделе 45.5.

Помимо MAP_PRIVATE и MAP_SHARED, аргумент flags может принимать и другие флаги, разделенные побитовым ИЛИ. Мы обсудим их в разделах 45.6 и 45.10.

Оставшиеся аргументы, fd и offset, используются в сочетании с файловыми отображениями (и игнорируются анонимными). Аргумент fd — это файловый дескриптор файла, который нужно отобразить. Аргумент offset задает начальную позицию отображения в файле и должен быть кратным размеру страницы памяти в системе. Чтобы отобразить весь файл целиком, в качестве offset можно указать 0, а length передать размер файла. Мы вернемся к файловым отображениям в разделе 45.5.

Подробнее о защите памяти

Как уже отмечалось выше, аргумент prot вызова mmap() обозначает тип защиты нового отображения в память. Он может содержать значение PROT_NONE или маску, состоящую из одного или нескольких флагов: PROT_READ, PROT_WRITE и PROT_EXEC. Если процесс, пытаясь получить доступ к участку памяти, нарушит его защиту, то ядро пошлет этому процессу сигнал SIGSEGV.

> В стандарте SUSv3 говорится о том, что для оповещения о нарушениях защиты памяти должен использоваться сигнал SIGSEGV, но в некоторых системах вместо него применяется сигнал SIGBUS.

Страницы, защищенные флагом PROT_NONE, среди прочего используются в начале и конце участка памяти, выделяемого процессом. Если процесс случайно затронет

страницу, помеченную как PROT_NONE, то ядро уведомит его об этом факте, сгенерировав сигнал SIGSEGV.

Сведения о защите хранятся в таблицах виртуальной памяти, выделяемых для каждого отдельного процесса. Таким образом, разные процессы могут отобразить данные на один и тот же участок памяти, но с разной защитой.

Тип защиты можно изменить с помощью системного вызова mprotect() (см. раздел 46.1).

В ряде реализаций UNIX защита, которая на самом деле применяется к страницам отображения, может отличаться от той, что указана в аргументе prot. В частности, это может быть связано с аппаратными ограничениями (например, в старой версии архитектуры x86-32), из-за которых во многих UNIX-системах флаг PROT_READ может подразумевать PROT_EXEC (и наоборот), а в отдельных реализациях флаг PROT_READ автоматически применяется вместе с PROT_WRITE. Однако приложения не должны полагаться на такое поведение; аргумент prot всегда должен указывать необходимую защиту памяти.

> Современные образцы архитектуры x86-32 предоставляют аппаратную поддержку маркировки таблиц со страницами флагом NX (от англ. no execute — «не выполнять»), а Linux, начиная с версии 2.6.8, использует эту возможность для корректного разделения прав доступа PROT_READ и PROT_EXEC на платформе Linux/x86-32.

Стандартные ограничения выравнивания для offset и addr

Согласно стандарту SUSv3, аргумент offset вызова mmap() должен выравниваться по странице; то же самое касается аргумента addr, если указан флаг MAP_FIXED. Linux соответствует этой спецификации. Но впоследствии было отмечено, что данные требования расходятся с более ранними стандартами, которые не так жестко регулировали поведение указанных аргументов. В результате такой формулировки ряд стандартных реализаций формально (и без особой на то причины) перестали считаться таковыми. Стандарт SUSv4 возвращается к менее строгим требованиям:

- реализация может требовать, чтобы аргумент offset был кратным размеру страницы в системе;
- при указании флага MAP_FIXED реализация может требовать выравнивания аргумента addr по странице;
- если указан флаг MAP_FIXED, а addr не равен нулю, то аргументы addr и offset будут иметь остаток после деления, равный размеру страницы в системе.

> Похожая ситуация сложилась с аргументом addr в вызовах mprotect(), msync() и munmap(). Стандарт SUSv3 гласит, что он должен быть выровнен по странице, однако в стандарте SUSv4 это требование смягчено и оставлено на усмотрение реализации.

Пример программы

В листинге 45.1 демонстрируется применение вызова mmap() для создания приватного файлового отображения. Данная программа является упрощенной версией команды cat(1). Она отображает файл (целиком), имя которого задано в виде аргумента командной строки, и затем записывает содержимое отображения в стандартный вывод.

Листинг 45.1. Использование mmap() для создания приватного файлового отображения

mmap/mmcat.c

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    char *addr;
    int fd;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        errExit("open");

    /* Получаем размер файла и указываем на его основе размеры отображения
       и буфера, которые будут записаны */

    if (fstat(fd, &sb) == -1)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (write(STDOUT_FILENO, addr, sb.st_size) != sb.st_size)
        fatal("partial/failed write");
    exit(EXIT_SUCCESS);
}
```

mmap/mmcat.c

45.3. Удаление отображения с участка памяти: munmap()

Системный вызов `munmap()` выполняет действие, обратное `mmap()`, — удаляет отображение из виртуального адресного пространства вызывающего процесса.

```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `addr` обозначает начальный адрес участка памяти, с которого нужно удалить отображение. Он должен совпадать с началом страницы (в стандарте SUSv3 это обязательное требование, но стандарт SUSv4 всего лишь предусматривает такую *возможность*).

Аргумент `length` представляет собой неотрицательное целое число, обозначающее размер участка, с которого будет удаляться отображение (в байтах). Диапазон адресов будет округлен до следующего значения, кратного размеру страницы в системе.

Обычно отображение удаляется целиком. Следовательно, нужно указать адрес, возвращенный предыдущим вызовом `mmap()`, и использовать то же значение `length`, что и при создании отображения. Например:

```
addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (addr == MAP_FAILED)
    errExit("mmap");

/* Код для работы с отображенным участком */

if (munmap(addr, length) == -1)
    errExit("munmap");
```

Но можно также выполнить частичное удаление. В этом случае отображение либо урезается, либо делится на две части — в зависимости от места выполнения удаления. Кроме того, существует возможность указать диапазон адресов, охватывающий несколько отображений, в результате чего все они будут удалены.

Если в диапазоне адресов, заданных с помощью аргументов `addr` и `length`, не обнаруживается отображение, то вызов `munmap()` не выполняет никаких действий и просто возвращает `0` (сигнализируя об успехе).

Во время удаления отображения ядро уничтожает все блокировки памяти, удерживаемые процессом в заданном диапазоне адресов (эти блокировки устанавливаются с использованием вызовов `mlock()` или `mlockall()`; см. раздел 46.2).

Когда процесс завершается или выполняет вызов `exec()`, все его отображения удаляются автоматически.

Чтобы записать содержимое отображения обратно в исходный файл, перед удалением следует сделать вызов `msync()` (см. раздел 45.5).

45.4. Отображение файлов

Для отображения файла нужно выполнить следующие шаги.

1. Получить дескриптор файла (обычно с помощью вызова `open()`).
2. Передать этот файловый дескриптор вызову `mmap()` в виде аргумента `fd`.

В результате `mmap()` отобразит содержимое открытого файла на адресное пространство вызывающего процесса. После завершения вызова `mmap()` можно закрыть файловый дескриптор, не затрагивая отображения. Но в некоторых случаях его лучше оставить открытым (примеры приводятся в листинге 45.1 и в главе 50).

> Вызов mmap() позволяет отображать не только обычные файлы, хранящиеся на диске, но и содержимое различных физических и виртуальных устройств, таких как жесткие и оптические диски или /dev/mem.

Файл, на который ссылается дескриптор `fd`, должен быть открыт с правами доступа, соответствующими значениям аргументов `prot` и `flags`. В частности, файл всегда должен быть доступен для чтения, а если в `flags` указаны флаги `PROT_WRITE` и `MAP_SHARED`, то также должен позволять запись. Аргумент `offset` обозначает начальный байт участка отображаемого файла и должен быть кратным размеру страницы в системе. Если он равен 0, то файл будет отображаться с самого начала. Аргумент `length` обозначает количество байтов, которое нужно отобразить. Сочетание `ffset` и `length` позволяет определить тот участок файла, что будет отображен в память (рис. 45.1).

> В Linux страницы файла отображаются при первом доступе. Это значит следующее: изменения, внесенные в участок файла после вызова mmap(), но перед доступом к соответствующей части (то есть странице) отображения, могут быть видны процессу (если страница не

была загружена в память до сего момента каким-то другим способом). Данное поведение зависит от конкретной реализации; портируемым приложениям не следует полагаться на то, что ядро поведет себя именно так.

45.4.1. Приватные файловые отображения

Наиболее часто приватные файловые отображения используются в следующих целях.

- Позволить нескольким процессам выполнять одну и ту же программу или открывать доступ (на чтение) к общему текстовому сегменту с помощью разделяемой библиотеки (при этом сегмент отображен из соответствующего участка исходного исполняемого или библиотечного файла).

  Исполняемый текстовый сегмент обычно защищен маской PROT_READ | PROT_EXEC, которая позволяет только чтение и выполнение, но при его отображении вместо MAP_SHARED применяется флаг MAP_PRIVATE. Это делается для того, чтобы отладчики или самомодифицирующиеся программы, вносящие изменения в программный код (предварительно откорректировав защиту памяти), не могли повлиять на исходный исполняемый файл или другие процессы.

- Отображать инициализированный сегмент данных исполняемого файла или разделяемой библиотеки. Такие отображения делаются приватными, чтобы изменения, вносимые в содержимое отображенного сегмента данных, не распространялись на исходный файл.

В обоих этих случаях вызов `mmap()` обычно используется незаметно для программы, так как такие отображения создаются программным загрузчиком и динамическим компоновщиком. Еще один, менее распространенный способ применения приватного файлового отображения заключается в организации обычного ввода для программы. Это похоже на то, как разделяемые файловые отображения используются для ввода/вывода в памяти (о чем пойдет речь в следующем разделе), но в данном случае задействуется только ввод.

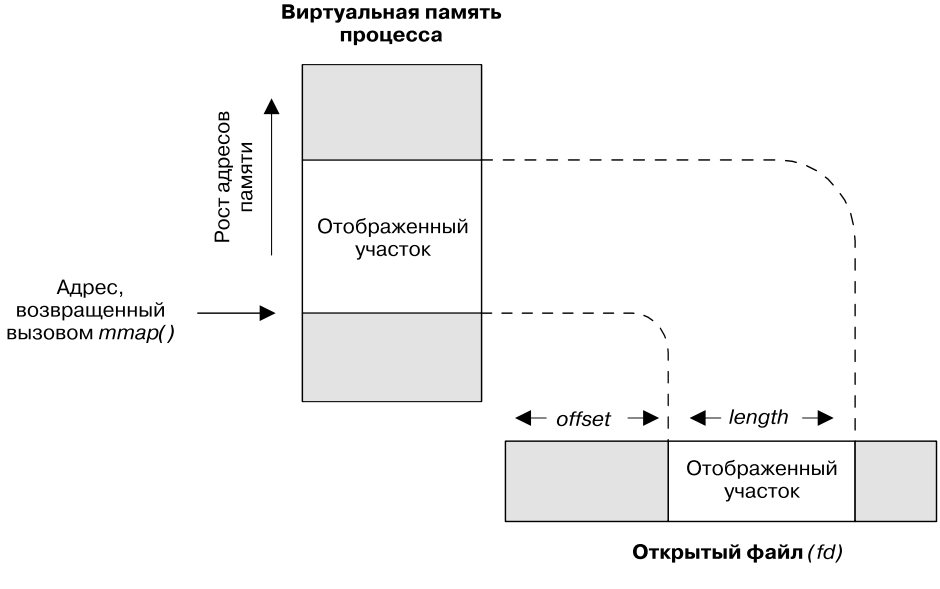


Рис. 45.1. Структура файла, отображенного в память

45.4.2. Разделяемые файловые отображения

Когда несколько процессов создают разделяемые отображения одного и того же участка файла, отображенные страницы физической памяти становятся для них общими. Кроме того, изменения содержимого отображений передаются обратно в исходный файл. Фактически файл становится хранилищем страниц этого участка памяти, как показано на рис. 45.2 (данная диаграмма упрощена: опущен тот факт, что отображенные страницы физической памяти обычно не являются смежными).

Разделяемые файловые отображения используются для двух задач: отображения ввода/вывода в память и межпроцессного взаимодействия. Каждый из этих случаев будет рассмотрен ниже.

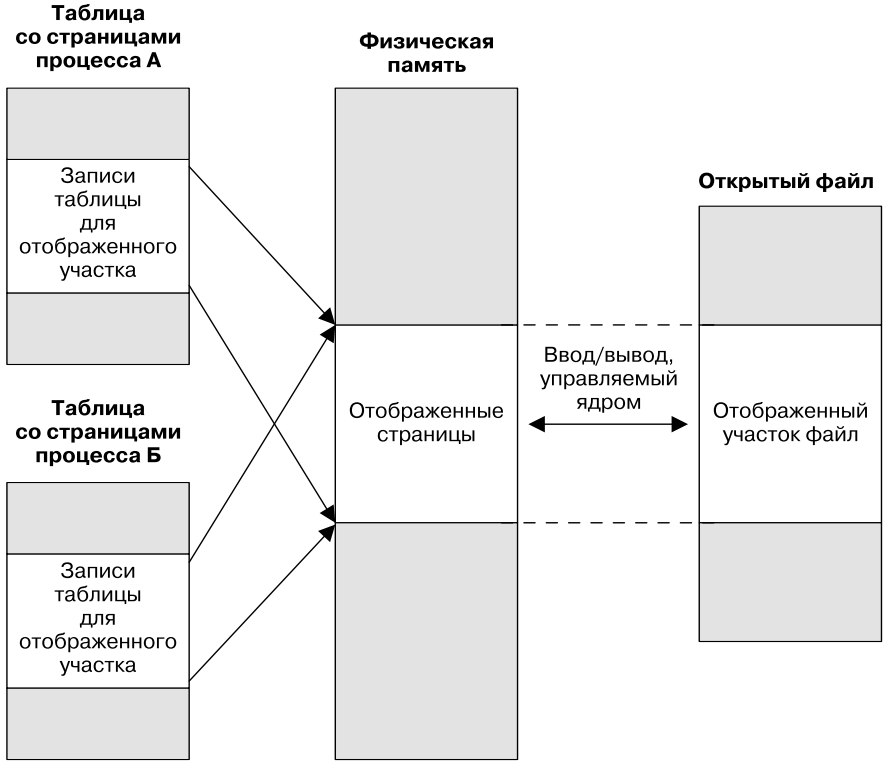


Рис. 45.2. Два процесса, совместно отображающие один и тот же участок файла

Отображение ввода/вывода в память

Поскольку содержимое разделяемого файлового отображения берется из файла, в который возвращаются любые изменения, внесенные в само отображение, можно выполнять файловый ввод/вывод, просто работая с байтами памяти, — все изменения будут автоматически применены ядром к исходному файлу (обычно для преобразования содержимого отображения создается структура данных, соответствующая содержимому файла). Такой подход называется *отображением ввода/вывода в память* и составляет альтернативу использованию вызовов `read()` и `write()` для чтения и записи файлов.

Отображение ввода/вывода в память имеет два потенциальных преимущества:

- непосредственная работа с памятью вместо применения системных вызовов `read()` и `write()` позволяет упростить логику некоторых приложений;
- в ряде случаев это может улучшить производительность ввода/вывода по сравнению с использованием традиционных системных вызовов.

Причины, по которым отображение ввода/вывода в память может привести к улучшению производительности, заключаются в следующем.

- Обычные операции `read()` и `write()` подразумевают двойную передачу данных: сначала между файлом и кэшем буфера ядра, а затем — между этим кэшем и буфером в пользовательском пространстве. Второго этапа можно избежать, если задействовать `mmap()`. При вводе данные доступны пользовательскому процессу сразу, как только ядро отобразит соответствующие блоки файла в память. При выводе этому процессу всего лишь нужно изменить содержимое памяти, а диспетчер памяти ядра сам автоматически обновит исходный файл.
- Помимо экономии на передаче данных между пространством ядра и пользователя, `mmap()` может также снизить потребление памяти. При использовании операций `read()` и `write()` данные хранятся в двух буферах, один из которых находится в пользовательском пространстве, а другой принадлежит ядру. Вызову `mmap()` нужен только один общий буфер, разделяемый между пространствами ядра и пользователя. Кроме того, этот единый буфер ядра может применяться несколькими процессами, которые выполняют ввод/вывод с одним и тем же файлом, что еще больше снижает потребление памяти.

Выгода от улучшенной производительности отображения ввода/вывода в большинстве случаев проявляется при выполнении множества операций произвольного доступа к большому файлу. При последовательном доступе вызов `mmap()` почти (или совсем) не имеет существенных преимуществ по сравнению с операциями `read()` и `write()`, при условии, что используется буфер достаточных размеров, который позволяет избежать большого количества системных вызовов для ввода/вывода. Причина заключается вот в чем: независимо от применяемой методики все содержимое файла будет считано с диска в память всего один раз и по сравнению с этим выгода от пониженного потребления памяти и отказа от передачи данных между пространством пользователя и ядра является несущественной.

> Отображение ввода/вывода в память может иметь некоторые недостатки. В случае с чтением/записью небольшого объема данных издержки от применения этой методики (то есть отображение, отказы страницы, удаление отображения и обновление буфера ассоциативной трансляции в физической памяти) могут превысить затраты на использование операций read() и write(). Кроме того, иногда ядру не удается эффективно выполнить обратную запись в изменяемое отображение (в данном случае производительность можно повысить за счет вызовов msync() или sync_file_range()).

Межпроцессное взаимодействие на основе разделяемого файлового отображения

Поскольку все процессы, отображающие один и тот же участок файла, получают доступ к общим страницам физической памяти, разделяемое файловое отображение можно также использовать для организации (быстрого) межпроцессного взаимодействия. Данный подход отличается тем, что изменения, вносимые в содержимое выбранного участка, автоматически применяются к исходному файлу. Это может пригодиться в случаях, когда содержимое разделяемой памяти должно сохраняться при перезапуске приложения или всей системы.

Пример программы

В листинге 45.2 приводится простой пример использования вызова `mmap()` для создания разделяемого файлового отображения. Сначала данная программа отображает файл, заданный в виде первого аргумента командной строки. Затем она выводит значение строки, находящейся в начале отображенного участка. В завершение второй аргумент командной строки, если он был указан, копируется на соответствующий участок разделяемой памяти.

Применение данной программы показано на примере следующей сессии командной оболочки. Для начала создадим 1024-байтный файл, заполненный нулями:

```
$ dd if=/dev/zero of=s.txt bs=1 count=1024
1024+0 records in
1024+0 records out
```

Теперь воспользуемся нашей программой, чтобы отобразить этот файл и скопировать строку в отображенный участок:

```
$ ./t_mmap s.txt hello
Current string=
Copied "hello" to shared memory
```

Программа не вывела ничего, что касалось бы текущей строки, поскольку исходное значение отображенных файлов начинается с нулевого байта (то есть со строки нулевой длины).

Теперь опять попытаемся отобразить файл и скопировать на отображенный участок новую строку:

```
$ ./t_mmap s.txt goodbye
Current string=hello
Copied "goodbye" to shared memory
```

В завершение выведем файл (по восемь символов в строке), чтобы проанализировать его содержимое:

```
$ od -c -w8 s.txt
0000000   g   o   o   d   b   y   e nul
0000010 nul nul nul nul nul nul nul nul
*
0002000
```

Наша программа достаточно тривиальна, так что не использует никакого механизма для синхронизации доступа к отображенному файлу со стороны нескольких процессов. Однако реальные приложения обычно без этого не обходятся. Существует целый ряд методик, включая семафоры (см. главу 49) и блокировку файлов (см. главу 51).

Системный вызов `msync()`, применяемый в листинге 45.2, будет рассмотрен в разделе 45.5.

Листинг 45.2. Использование mmap() для создания разделяемого файлового отображения

mmap/t_mmap.c

```
#include <sys/mman.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MEM_SIZE 10

int
main(int argc, char *argv[])
{
```

```
    char *addr;
    int fd;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file [new-value]\n", argv[0]);

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open");

    addr = mmap(NULL, MEM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)            /* Дескриптор 'fd' больше не нужен */
        errExit("close");

    printf("Current string=%.*s\n", MEM_SIZE, addr);
         /* Мера безопасности: ограничиваем вывод MEM_SIZE байтами */

    if (argc > 2) {                 /* Обновляем содержимое участка */
        if (strlen(argv[2]) >= MEM_SIZE)
            cmdLineErr("'new-value' too large\n");

        memset(addr, 0, MEM_SIZE);     /* Заполняем участок нулями */
        strncpy(addr, argv[2], MEM_SIZE - 1);
        if (msync(addr, MEM_SIZE, MS_SYNC) == -1)
            errExit("msync");

        printf("Copied \"%s\" to shared memory\n", argv[2]);
    }

    exit(EXIT_SUCCESS);
}
```

mmap/t_mmap.c

45.4.3. Крайние случаи

Часто размер отображения кратен размеру страницы памяти и точно вписывается в границы отображенного файла. Но так бывает не всегда, поэтому ниже вы увидите, что происходит, когда данные условия не выполняются.

На рис. 45.3 показан следующий случай: отображение точно вписывается в границы отображенного файла, но размер участка при этом не кратен размеру страницы памяти в системе (мы будем исходить из того, что он равен 4096 байтам).

Поскольку размер отображения не кратен размеру страницы памяти, он округляется до следующего кратного. Итоговое значение оказывается меньше, чем размер самого файла, поэтому оставшиеся байты отображаются так, как показано на рис. 45.3.

Попытки доступа за пределы отображения приводят к генерированию сигнала `SIGSEGV` (предполагается, что в данном месте нет какого-то другого отображения). По умолчанию это приводит к завершению процесса и сбрасыванию дампа памяти.

Все становится сложнее, когда отображение выходит за пределы отображаемого файла (рис. 45.4). Как и прежде, длина отображения, которое не кратно размеру страницы памяти в системе, округляется. Но в данном случае байты в округленном участке (на диаграмме это байты с 2200 по 4095) хоть и остаются доступными, но не привязываются к исходному файлу (поскольку в самом файле нет соответствующих

байтов). Вместо этого они заполняются нулями (таково требование стандарта SUSv3). Тем не менее эти байты являются доступными для других процессов, отображающих файл (если указали аргумент `length` достаточного размера). Изменения этих байтов не записываются в файл.

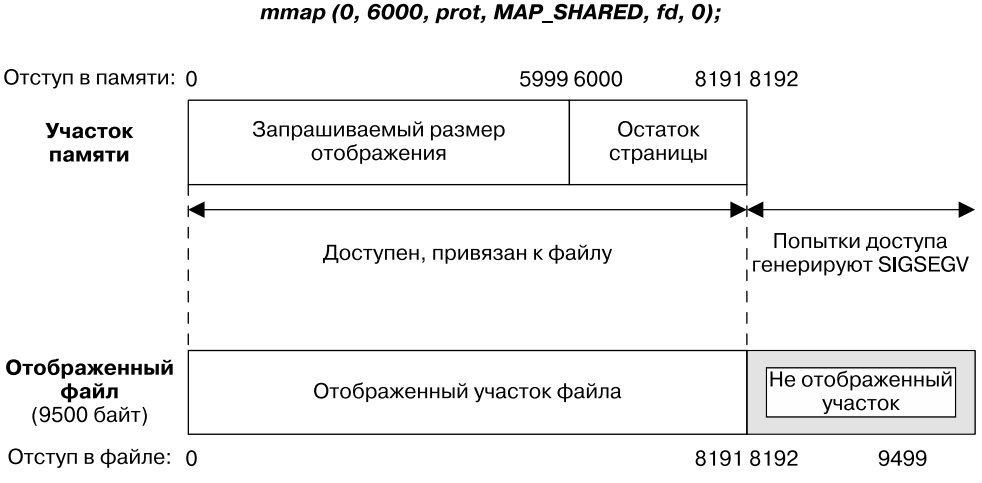


Рис. 45.3. Отображение, чья длина не кратна размеру страницы памяти в системе

Если отражение содержит страницы, выходящие за пределы отображенного участка (на рис. 45.4 это байты, начиная с 4096), попытки получения доступа к адресам внутри указанных страниц приводят к генерированию сигнала `SIGBUS`, который предупреждает процесс о том, что данный адрес не связан с участком исходного файла. Как и прежде, попытки выйти за пределы отображения приводят к получению сигнала `SIGSEGV`.

Из описания, приведенного выше, может сложиться впечатление, что в создании отображения, превышающего размер исходного файла, нет никакого смысла. Однако эти дополнительные области становятся доступными по мере увеличения самого файла (например, когда используются вызовы `ftruncate()` или `write()`).

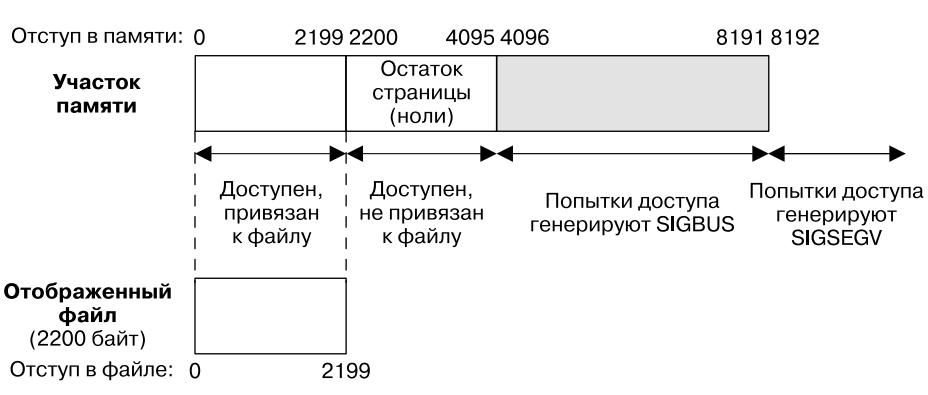


Рис. 45.4. Отображение в память выходит за пределы отображаемого файла

45.4.4. Взаимодействие защиты памяти и режима доступа к памяти

Один из моментов, освещенный недостаточно хорошо, — это взаимодействие защиты памяти, заданной с помощью аргумента `prot` в вызове `mmap()`, и режима, в котором открыт отображенный файл. В целом можно утверждать: флаги `PROT_READ` и `PROT_EXEC` требуют, чтобы отображаемый файл был открыт в режиме `O_RDONLY` или `O_RDWR`, а для защиты `PROT_WRITE` нужны режимы открытия `O_WRONLY` или `O_RDWR`.

Но все усложняется из-за ограниченной гибкости защиты памяти, предоставляемой некоторыми аппаратными платформами (см. раздел 45.2). Для таких случаев необходимо сделать следующие замечания.

- С флагом открытия файла `O_RDWR` совместимы любые комбинации защиты памяти.
- С файлом, открытым в режиме `O_WRONLY`, не совместима ни одна комбинация защиты памяти, даже `PROT_WRITE` (генерируется ошибка `EACCES`). Это вытекает из того факта, что ряд аппаратных платформ не предусматривает доступа к странице только на запись. Как отмечалось в разделе 45.2, в таких архитектурах защита `PROT_WRITE` подразумевает использование флага `PROT_READ`; то есть страница, в которую можно записывать, доступна также для чтения. Операции чтения несовместимы с режимом `O_WRONLY`, запрещающим вывод исходного содержимого файла.
- Результаты открытия файла в режиме `O_RDONLY` зависят от того, какой флаг был указан при вызове `mmap()` — `MAP_PRIVATE` или `MAP_SHARED`. В первом случае `mmap()` поддерживает любые комбинации защиты памяти, поскольку изменения приватной страницы все равно никогда не записываются в исходный файл. Во втором случае с режимом `O_RDONLY` совместима только защита `PROT_READ` (`PROT_READ | PROT_EXEC`). Это логично, ведь защита `PROT_WRITE` позволяет обновлять отображенный файл.

45.5. Синхронизация отображенного участка памяти: msync()

Ядро автоматически возвращает изменения содержимого отображения типа `MAP_SHARED` обратно в исходный файл, но по умолчанию нет никаких гарантий относительно того, когда именно произойдет такая синхронизация (стандарт SUSv3 этого не требует).

Системный вызов `msync()` предоставляет приложению возможность самостоятельно синхронизировать разделяемое отображение с отображенным файлом. Это может пригодиться в разнообразных ситуациях. Например, чтобы гарантировать целостность информации, база данных может сбрасывать изменения на диск с помощью `msync()`. Выполняя данный вызов, мы делаем обновления изменяемого отображения видимыми для других процессов, которые считывают исходный файл.

```
#include <sys/mman.h>

int msync(void *addr, size_t length, int flags);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргументы `addr` и `length` вызова `msync()` обозначают начальный адрес и размер участка памяти, который нужно синхронизировать. Адрес должен быть выровнен по странице, а длина округлена до следующего кратного размеру страницы в системе (в стандарте

SUSv3 требование к выравниванию является *обязательным*; в SUSv4 всего лишь предусмотрена такая *возможность*).

Аргумент `flags` может принимать одно из следующих значений:

- `MS_SYNC` — выполняет синхронную запись в файл. Вызов блокируется, пока все измененные страницы участка памяти не будут записаны на диск;
- `MS_ASYNC` — выполняет асинхронную запись в файл. Измененные страницы участка памяти записываются на диск не сразу, но немедленно становятся видимыми для других процессов, выполняющих операцию `read()` для соответствующего участка файла.

  Различия между этими двумя значениями можно сформулировать иначе: после операции `MS_SYNC` участок памяти синхронизирован с диском, тогда как после `MS_ASYNC` — только с кэшем буфера ядра.

  Если после операции MS_ASYNC не предпринять никаких дополнительных действий, то измененные страницы на участке памяти в какой-то момент будут автоматически сброшены на диск потоком выполнения ядра pdflush (в Linux 2.4 и более ранних версиях он назывался kupdated). В Linux существует два (нестандартных) метода ускорить вывод. Вслед за msync() можно сделать вызов fsync() (или fdatasync()), указав дескриптор соответствующего отображения. Данный вызов блокируется до тех пор, пока кэш буфера не будет синхронизирован с диском. Как вариант можно инициировать асинхронную запись страниц с помощью операции posix_fadvise() POSIX_FADV_DONTNEED (в Linux эти два случая имеют некоторые особенности, не описанные в стандарте SUSv3).

  В аргументе `flags` можно указать еще одно дополнительное значение. `MS_INVALIDATE` — удаляет кэшированные копии отображенных данных. После сброса на диск любых измененных страниц участка памяти те из них, что не соответствуют исходному файлу, помечаются как недействительные. При следующем обращении в них будут скопированы соответствующие участки файла. Как следствие, любые изменения, внесенные в файл другим процессом, становятся доступными на участке памяти.

Как и многие другие современные реализации UNIX, Linux предоставляет так называемую систему *унифицированной виртуальной памяти*. Это значит, что по возможности отображения и блоки буферного кэша разделяют одни и те же страницы физической памяти. Таким образом обеспечивается связность данных, получаемых через отображение и системные вызовы ввода/вывода (`read()`, `write()` и т. д.), а `msync()` используется только для принудительного сброса содержимого отображенного участка на диск.

Однако система унифицированной виртуальной памяти не предусмотрена стандартом SUSv3 и присутствует не во всех UNIX-системах. В таких реализациях, чтобы сделать видимыми изменения содержимого отображения для других процессов, читающих файл, необходимо выполнить вызов `msync()`. И наоборот, чтобы запись в файл, выполненная другим процессом, стала доступной на отображенном участке памяти, необходимо использовать флаг `MS_INVALIDATE`. Многопроцессные приложения, в которых для работы с одним и тем же файлом применяются как `mmap()`, так и системные вызовы ввода/вывода, должны корректно выполнять операцию `msync()`, иначе их нельзя будет перенести в системы, не поддерживающие систему унифицированной виртуальной памяти.

45.6. Дополнительные флаги вызова mmap()

Помимо `MAP_PRIVATE` и `MAP_SHARED`, аргумент `flags` вызова `mmap()` в Linux поддерживает ряд других значений (которые можно перечислять через побитовое ИЛИ). Все они собраны в табл. 45.3. Из них только `MAP_FIXED` входит в стандарт SUSv3 (вместе с `MAP_PRIVATE` и `MAP_SHARED`).

Таблица 45.3. Значения битовой маски для аргумента flags вызова mmap()

Значение	Описание	SUSv3
MAP_ANONYMOUS	Создает анонимное отображение	
MAP_FIXED	Не округляет аргумент addr (см. раздел 45.10)	*
MAP_LOCKED	Блокирует отображенные страницы в памяти (начиная с Linux 2.6)	
MAP_HUGETLB	Создает отображение, которое использует большие страницы (начиная с Linux 2.6.32)	
MAP_NORESERVE	Управляет резервированием пространства файла подкачки (см. раздел 45.9)	
MAP_PRIVATE	Изменения, вносимые в отображенные данные, являются приватными	*
MAP_POPULATE	Заполняет страницы отображения (начиная с Linux 2.6)	
MAP_SHARED	Изменения отображенных данных доступны другим процессам и возвращаются обратно в исходный файл (противоположность флагу MAP_PRIVATE)	*
MAP_UNINITIALIZED	Не очищает анонимные отображения (начиная с Linux 2.6.33)	

В следующем списке перечислены подробности о значениях аргумента `flags` из табл. 45.3 (за исключением `MAP_PRIVATE` и `MAP_SHARED`, которые уже были рассмотрены):

- `MAP_ANONYMOUS` — создает анонимное отображение — то есть не связанное с файлом. Этот флаг будет подробно описан в разделе 45.7;
- `MAP_FIXED` — об этом флаге рассказывается в разделе 45.10;
- `MAP_HUGETLB` (начиная с Linux 2.6.32) — имеет такое же назначение для вызова `mmap()`, как и флаг `SHM_HUGETLB` для сегментов разделяемой памяти в System V;
- `MAP_LOCKED` (начиная с Linux 2.6) — выполняет предварительную загрузку и блокирует отображенные страницы в памяти; похоже на действие вызова `mlock()`. Привилегии, необходимые для использования этого флага, а также ограничения, налагаемые на его работу, будут описаны в разделе 46.2;
- `MAP_NORESERVE` — определяет, нужно ли выполнять предварительное резервирование пространства файла подкачки для отображения. Подробности см. в разделе 45.9;
- `MAP_POPULATE` (начиная с Linux 2.6) — заполняет страницы отображения. В случае с файловым отображением выполняется упреждающее чтение файла. Это значит, что последующие обращения к содержимому отображения не будут блокироваться сбоями страницы (если только в результате давления страницы памяти не успели сброситься на диск);
- `MAP_UNINITIALIZED` (начиная с Linux 2.6.33) — предотвращает заполнение нулями анонимного отображения. Это улучшает производительность, но несет в себе потенциальную угрозу безопасности, поскольку выделенные страницы могут содержать конфиденциальную информацию, оставленную предыдущим процессом. Таким образом, данный флаг предназначен для использования во встраиваемых платформах, где производительность имеет решающее значение, а вся система находится под управлением одного или нескольких приложений. Действует только в ядрах, сконфигурированных с параметром `CONFIG_MMAP_ALLOW_UNINITIALIZED`.

45.7. Анонимные отображения

Анонимным называют отображение, которое не связано с исходным файлом. В этом разделе вы узнаете, как создаются такие отображения и для чего используются их приватные и разделяемые разновидности.

MAP_ANONYMOUS и /dev/zero

В Linux существует два разных, но равноценных способа создания анонимного отображения с помощью вызова `mmap()`.

- Указать значение `MAP_ANONYMOUS` для аргумента `flags` и `-1` для аргумента `fd` (в Linux при использовании данного флага значение аргумента `fd` игнорируется; но некоторые разновидности UNIX требуют, чтобы в этом случае аргумент `fd` был равен `-1` — портируемые приложения должны поступать именно так).

  Для получения определения MAP_ANONYMOUS из заголовочного файла <sys/mman.h> необходимо определить один из двух макросов проверки возможностей: _BSD_SOURCE или _SVID_SOURCE. В Linux для совместимости с рядом других UNIX-систем, в которых используются другие названия, в качестве синонима флага MAP_ANONYMOUS доступна константа MAP_ANON.

- Открыть файл устройства `/dev/zero` и передать полученный дескриптор вызову `mmap()`.

  /dev/zero представляет собой виртуальное устройство, при чтении из которого всегда возвращаются нули. Запись в него всегда отклоняется. Обычно /dev/zero применяется для заполнения файла нулями (например, с помощью команды dd(1)).

При использовании любой из этих методик содержимое итогового отображения заполняются нулями, а аргумент `offset` игнорируется (так как нет исходного файла, для которого можно было бы указать отступ). Примеры применения обоих подходов будут показаны чуть ниже.

Использование флага MAP_ANONYMOUS и файла /dev/zero не предусмотрено стандартом SUSv3, хотя большинство реализаций UNIX поддерживает обе методики. Причина существования разных подходов с одним и тем же результатом заключается в том, что один из них пришел из мира BSD (MAP_ANONYMOUS), а другой — из System V (/dev/zero).

Анонимные отображения типа MAP_PRIVATE

Анонимные отображения типа `MAP_PRIVATE` служат для выделения блоков памяти, принадлежащих только одному процессу и заполненных нулями. Чтобы создать такое отображение, можно воспользоваться файлом устройства `/dev/zero`:

```
fd = open("/dev/zero", O_RDWR);
if (fd == -1)
    errExit("open");
addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (addr == MAP_FAILED)
    errExit("mmap");
```

Реализация вызова malloc(), входящая в библиотеку glibc, использует анонимные отображения типа MAP_PRIVATE для выделения блоков памяти, превышающих MMAP_THRESHOLD байт. Это позволяет эффективно уничтожать такие блоки (с помощью munmap()), если передать их вызову free() (таким образом снижается и вероятность фрагментации памяти при многократном выделении и уничтожении больших блоков). Константа MMAP_THRESHOLD

по умолчанию равна 128 байтам, но данное значение можно откорректировать, задействуя библиотечную функцию mallopt().

Анонимные отображения типа MAP_SHARED

Анонимные отображения типа MAP_SHARED позволяют родственным процессам (например, родителю и потомку) работать с одним и тем же участком памяти, не используя связанный с ним отображенный файл.

Анонимные отображения типа MAP_SHARED доступны в Linux, начиная с версии 2.4.

Методика, которую мы применяли для отображений типа MAP_ANONYMOUS, подходит и для MAP_SHARED:

```
addr = mmap(NULL, length, PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (addr == MAP_FAILED)
            errExit("mmap");
```

Если вслед за кодом, приведенным выше, выполнить вызов fork(), то новый дочерний процесс унаследует отображение и получит доступ к тому же участку памяти, что и родитель.

Пример программы

Программа, представленная в листинге 45.3, демонстрирует использование MAP_ANONYMOUS или файла /dev/zero (на выбор) для разделения отображенного участка между родительским и дочерним процессами. Выбор методики зависит от того, был ли определен макрос USE_MAP_ANON на этапе компиляции программы. Перед вызовом fork() родитель инициализирует разделяемый участок с помощью значения 1. Затем потомок инкрементирует это общее целое число и завершается; дождавшись завершения потомка, родитель выводит получившееся число. При выполнении данной программы мы увидим следующее:

```
$ ./anon_mmap
Child started, value = 1
In parent, value = 2
```

Листинг 45.3. Разделение анонимного отображения между родительским и дочерним процессами

mmap/anon_mmap.c

```
#ifdef USE_MAP_ANON
#define _BSD_SOURCE             /* Получаем определение MAP_ANONYMOUS */
#endif
#include <sys/wait.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int *addr;                  /* Указатель на общий участок памяти */

#ifdef USE_MAP_ANON             /* Используем MAP_ANONYMOUS */
    addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");
```

```
#else                           /* Отображаем /dev/zero */
    int fd;

    fd = open("/dev/zero", O_RDWR);
    if (fd == -1)
        errExit("open");

    addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)        /* Больше не нужен */
        errExit("close");
#endif

    *addr = 1;                  /* Инициализируем целое число на отображенном участке */

    switch (fork()) {           /* Родитель и потомок разделяет отображение */
    case -1:
        errExit("fork");

    case 0:                     /* Потомок инкрементирует общее число и завершается */
        printf("Child started, value = %d\n", *addr);
         (*addr)++;

        if (munmap(addr, sizeof(int)) == -1)
            errExit("munmap");
        exit(EXIT_SUCCESS);

    default:                    /* Родитель ждет завершения потомка */
        if (wait(NULL) == -1)
            errExit("wait");
        printf("In parent, value = %d\n", *addr);

        if (munmap(addr, sizeof(int)) == -1)
            errExit("munmap");
        exit(EXIT_SUCCESS);
    }
}
```

mmap/anon_mmap.c

45.8. Изменение отображенного участка памяти: mremap()

В большинстве UNIX-систем нельзя изменить местоположение и размер существующего отображения отображения. Однако Linux предоставляет (недоступный в других реализациях) вызов `mremap()`, который делает возможными такие изменения.

Аргументы `old_address` и `old_size` обозначают местоположение и размер существующего отображения, которое мы хотим расширить или уменьшить. Адрес, указанный в `old_address`, должен быть выровнен по странице; обычно это значение, возвращенное предыдущим вызовом `mmap()`. Новый запрашиваемый размер указывается с помощью аргумента `new_size`. Значения `old_size` и `new_size` округляются до следующего кратного размеру страницы памяти в системе.

```
#define _GNU_SOURCE
#include <sys/mman.h>

void *mremap(void *old_address, size_t old_size, size_t new_size,
             int flags, ...);
```

Возвращает начальный адрес измененного отображения при успешном завершении или MAP_FAILED, если произошла ошибка

Изменяя отображение, ядро может переместить его в рамках виртуального адресного пространства процесса. Возможность такого перемещения определяется аргументом `flags`; это битовая маска, которая может быть равна либо нулю, либо следующим значениям.

- `MREMAP_MAYMOVE` — если указать данный флаг, ядро, исходя из требований к свободной памяти, может переместить отображение внутри виртуального адресного пространства процесса. В противном случае, если на текущем участке не хватает свободного места для расширения отображения, генерируется ошибка `ENOMEM`.
- `MREMAP_FIXED` (начиная с Linux 2.4) — этот флаг можно использовать только в сочетании с `MREMAP_MAYMOVE`. Его действие аналогично применению флага `MAP_FIXED` в вызове `mmap()` (см. раздел 45.10). Если его указать, то вызов `mremap()` сможет принять дополнительный аргумент `void *new_address` — адрес, выровненный по странице, куда следует переместить отображение. Любые другие отображения, находящиеся в диапазоне, заданном с помощью `new_address` и `new_size`, уничтожаются.

В случае успеха `mremap()` возвращает начальный адрес отображения. Поскольку данное значение может отличаться от предыдущего начального адреса (если был указан флаг `MREMAP_MAYMOVE`), указатели, ссылающиеся на этот участок, могут утратить свою актуальность. Следовательно, приложения, выполняющие вызов `mremap()`, должны ссылаться на адреса в отображенном участке с помощью отступов, избегая абсолютных значений.

В Linux существует функция realloc(), которая использует вызов mremap() для эффективного перемещения больших блоков памяти, выделенных ранее путем операции mmap() MAP_ANONYMOUS (я уже упоминал о данной возможности вызова malloc() из библиотеки glibc в разделе 45.7). Применение для этих целей вызова mremap() позволяет избежать копирования данных во время перемещения.

45.9. Флаг MAP_NORESERVE и перерасход пространства подкачки

Некоторые приложения создают большие (обычно приватные и анонимные) отображения, но используют только небольшую часть выделенной памяти. Например, ряд научных программ выделяют огромные массивы, но помещают в разные его участки всего несколько элементов (такие массивы называют *разреженными*).

Если бы ядро всегда выделяло (или резервировало) для таких отображений достаточно места в файле подкачки, то большая его часть тратилась бы впустую. Вместо этого оно способно резервировать страницы отображения по мере необходимости (то есть когда приложение пытается получить к ним доступ). Такой подход называется *отложенным резервированием пространства подкачки* и позволяет приложению использовать виртуальную память, объем которой превышает совокупность физической памяти и пространства подкачки.

Иными словами, отложенное резервирование позволяет перерасходовать пространство подкачки. Это работает до тех пор, пока все процессы не попытаются получить доступ ко всему диапазону адресов своих отображений. В таком случае физическая память и пространство подкачки будут исчерпаны и ядро будет вынуждено снизить давление на память, принудительно завершив один или несколько процессов. В идеале оно попытается выбрать процесс, вызывающий проблемы с памятью (см. описание *OOM killer* чуть ниже), но так происходит не всегда. Поэтому в некоторых случаях от отложенного резервирования лучше отказаться и заставить систему выделять все пространство подкачки, необходимое для размещения отображения.

Способ резервирования пространства подкачки ядром зависит от использования флага `MAP_NORESERVE` при вызове `mmap()`, а также от интерфейсов `/proc`, которые влияют на перерасход пространства подкачки на уровне системы. Эти факторы сгруппированы в табл. 45.4.

Таблица 45.4. Управление резервированием пространства подкачки при вызове mmap()

Значение overcommit_memory	**Указан ли флаг MAP_NORESERVE при вызове mmap()?**	
	Нет	**Да**
0	Очевидный перерасход запрещается	Перерасход разрешен
1	Перерасход разрешен	Перерасход разрешен
2 (начиная с Linux 2.6)	Разрешается фиксированный перерасход	

Файл `/proc/sys/vm/overcommit_memory` (доступный только в Linux) содержит целое число, определяющее поведение ядра касательно перерасхода пространства подкачки. Версии ядра ниже 2.6 поддерживали только два значения: `0` и больше нуля. Первое означает, что очевидный перерасход запрещается (при условии использования флага `MAP_NORESERVE`), а второе разрешает перерасход при любых условиях.

Запрет очевидного перерасхода означает, что можно создавать отображения, размер которых не превышает объем свободной памяти, доступной в текущий момент времени. Это ограничение не будет распространяться на уже существующие отображения (поскольку они могут не задействовать все отображенные ими страницы).

Начиная с версии 2.6, вместо положительных значений применяется `1`, а значение `2` (или больше) приводит к использованию политики *фиксированного перерасхода*. В этом случае ядро ведет строгий учет памяти, выделяемой вызовами `mmap()`, и ограничивает доступное пространство по следующей формуле:

```
[размер файла подкачки] + [размер физической памяти] * overcommit_ratio / 100
```

Значение `overcommit_ratio` представляет собой целое процентное значение и хранится в файле `/proc/sys/vm/overcommit_ratio` (доступном только в Linux). По умолчанию оно равно `50`; это значит, что при выделении адресного пространства ядро может успешно превысить размер доступной в системе памяти на 50 % (если только все процессы не попытаются одновременно использовать всю выделенную им память).

Стоит отметить, что мониторинг перерасхода памяти касается только следующих типов отображения:

- приватные отображения, доступные для записи (как файловые, так и анонимные), выделяемые отдельно для каждого процесса, который их задействует;
- разделяемые анонимные отображения, существующие в единственном экземпляре (поскольку они разделяются всеми процессами).

В резервировании пространства подкачки для приватного отображения, доступного только для чтения, нет необходимости: поскольку содержимое отображения не может быть изменено, не нужно использовать это пространство. Оно также не требуется для разделяемых файловых отображений, так как отображенный файл сам играет роль файла подкачки.

При вызове `fork()` дочерний процесс наследует не только отображение, но и его параметр `MAP_NORESERVE`. Этот флаг не предусмотрен стандартом SUSv3, но поддерживается в нескольких реализациях UNIX, включая Linux.

В данном разделе мы рассмотрели ситуации, в которых вызов mmap() может не суметь увеличить адресное пространство процесса ввиду системных ограничений, касающихся физической памяти и пространства подкачки. Причиной также может оказаться ограничение на ресурсы RLIMIT_AS (описанное в разделе 36.3), ограничивающее максимальный размер адресного пространства, выделяемый для вызывающего процесса.

OOM killer

Выше упоминалось, что при использовании отложенного резервирования память может быть исчерпана, если приложения попытаются получить доступ ко всему диапазону своих отображений. В таком случае для освобождения памяти ядро прибегает к принудительному завершению процессов.

Подсистема ядра, предназначенная для выбора процессов, которые следует завершить при нехватке памяти, известна под названием OOM killer (OOM от англ. out-of-memory — «нехватка памяти»). Данный механизм пытается выбрать наиболее подходящие для завершения процессы; критерии, учитываемые им при этом, зависят от целого ряда факторов. Например, чем больше памяти потребляет процесс, тем выше вероятность того, что OOM killer выберет именно его. Среди других факторов, принимаемых во внимание, можно отметить низкое значение `nice` (то есть больше 0) и попытки создания множества дочерних процессов. Ядро предпочитает не трогать следующие процессы:

- привилегированные процессы, поскольку они, вероятно, выполняют важные задачи;
- процессы, напрямую работающие с устройствами, так как их принудительное завершение может оставить устройство в нерабочем состоянии;
- процессы, которые работают продолжительное время или потребили значительный объем ресурсов процессора, так как их принудительное завершение может означать, что вся их работа была проделана впустую.

Для принудительного завершения процесса OOM killer отправляет ему сигнал `SIGKILL`.

В Linux с версии 2.6.11 существует файл `/proc/PID/oom_score`; в нем хранится «вес», который ядро назначает процессу, когда возникает необходимость в вызове OOM killer. Чем больше данное значение, тем выше вероятность того, что процесс при необходимости будет выбран для принудительного завершения. В версии ядра 2.6.11 также появился файл `/proc/PID/oom_adj`, с помощью которого можно повлиять на значение `oom_score` процесса. Этому файлу можно назначить любое число в диапазоне от –16 до +15; отрицательные значения снижают `oom_score`, а положительные — повышают. Если указать специальное значение `-17`, процесс перестанет рассматриваться системой OOM killer как кандидат на завершение. Дальнейшие подробности можно найти на странице `proc(5)` руководства.

45.10. Флаг MAP_FIXED

Если указать в аргументе `flags` вызова `mmap()` флаг `MAP_FIXED`, ядро будет интерпретировать адрес, заданный в аргументе `addr`, буквально, а не как точку отсчета. В таком случае адрес должен быть изначально выровнен по странице.

В целом флаг MAP_FIXED не следует применять в портируемых приложениях; вместо этого лучше передать аргументу addr значение NULL, позволяя системе автоматически выбрать адрес для размещения отображения.

Но существует ситуация, в которой портируемое приложение могло бы воспользоваться флагом MAP_FIXED. Речь идет о случае, когда при вызове mmap() участок памяти длиной length, начинающийся с адреса addr, перекрывает собой страницы предыдущего отображения и заменяет их. Можно отобразить несколько участков файла (или файлов) на один непрерывный участок памяти. Для этого нужно сделать следующее.

1. Использовать mmap() для создания анонимного отображения (см. раздел 45.7). В данном вызове аргументу addr передается значение NULL, а флаг MAP_FIXED опускается. Это позволяет ядру выбрать адрес для отображения.
2. Задействовать несколько вызовов mmap() с флагом MAP_FIXED, чтобы отобразить (то есть наложить) участки файла на разные участки отображения, созданного в предыдущем шаге.

Мы могли бы пропустить первый шаг и сразу применить вызов mmap() с флагом MAP_FIXED для создания набора смежных отображений в диапазоне адресов, выбранном приложением, но такой подход был бы менее универсальным. Как отмечалось выше, портируемое приложение должно воздерживаться от попыток создания нового отображения по фиксированному адресу. Первый шаг позволяет избежать проблем с портируемостью, так как выбор цельного диапазона адресов для нашего отображения перекладывается на ядро.

Начиная с Linux 2.6 того же эффекта можно достичь, используя системный вызов remap_file_pages(), который будет описан в следующем разделе. Однако применение флага MAP_FIXED является более портируемым, так как вызов remap_file_pages() доступен только в Linux.

45.11. Нелинейные отображения: remap_file_pages()

Файловые отображения, созданные с помощью вызова mmap(), являются линейными: между страницами отображенного файла и страницами участка памяти существует точное, последовательное соответствие. Это подходит для большинства приложений. Но в ряде ситуаций приходится создавать большое количество нелинейных отображений — таких, в которых порядок размещения страниц файла в памяти меняется. Пример данного отображения показан на рис. 45.5.

Один из способов создания нелинейного отображения описан в предыдущем разделе и заключается в использовании нескольких вызовов mmap() с флагом MAP_FIXED. Однако такой подход не очень хорошо масштабируется. Проблема в том, что каждый вызов mmap() создает в области виртуальной памяти ядра (англ. virtual memory area, VMA) структуру данных. И каждый раз область VMA тратит время на подготовку и потребляет некий объем памяти ядра, не предназначенной для сбрасывания на диск. Кроме того, наличие большого количества таких областей может ухудшить производительность диспетчера виртуальной памяти; в частности, есть вероятность существенного увеличения времени обработки отказа страницы (с этой проблемой сталкивались некоторые большие системы управления базами данных, хранящие в одном файле несколько разных отображений).

Каждая строка в файле /proc/PID/maps представляет одну область VMA.

Начиная с версии 2.6, ядро Linux поддерживает системный вызов `remap_file_pages()` для создания нелинейных отображений, который не приводит к появлению множества областей VMA. Он используется следующим образом.

1. Делается вызов `mmap()` для создания отображения.
2. Выполняются несколько вызовов `remap_file_pages()`, меняющие соответствие между страницами памяти и страницами файла (вызов `remap_file_pages()` всего лишь манипулирует таблицами с записями о страницах, принадлежащими процессу).
3. Вызов `remap_file_pages()` дает возможность отобразить одну и ту же страницу файла на разные сегменты отображенного участка памяти.

```
#define _GNU_SOURCE
#include <sys/mman.h>

int remap_file_pages(void *addr, size_t size, int prot, size_t pgoff,
                     int flags);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргументы `pgoff` и `size` обозначают участок файла, который нужно переместить в памяти. Первый из них указывает на начальную позицию участка файла в единицах, равных размеру страницы памяти (возвращаемый вызовом `sysconf(_SC_PAGESIZE)`). Аргумент `size` определяет длину участка в байтах. Значение `addr` служит двум целям:

- определяет существующее отображение, чьи страницы мы хотим поменять местами. Иными словами, это адрес, который должен находиться в пределах участка памяти, отображенного ранее с помощью вызова `mmap()`;
- указывает на адрес в памяти, по которому находятся страницы файла, заданные с использованием аргументов `pgoff` и `size`.

Аргументы `addr` и `size` должны быть кратными размеру страницы памяти в системе. В противном случае они округляются до ближайшего кратного.

Представьте, что мы применяем следующий вызов `mmap()` для отображения трех страниц открытого файла, на который ссылается дескриптор `fd`, и в итоге аргументу `addr` присваивается адрес `0x4001a000`:

```
ps = sysconf(_SC_PAGESIZE);    /* Получаем размер страницы в системе */
addr = mmap(0, 3 * ps, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Тогда следующие вызовы создадут нелинейные отображения, показанные на рис. 45.5:

```
remap_file_pages(addr, ps, 0, 2, 0);
            /* Отображает страницу файла 0 на страницу памяти 2 */
remap_file_pages(addr + 2 * ps, ps, 0, 0, 0);
            /* Отображает страницу файла 2 на страницу памяти 0 */
```

Есть еще два аргумента для вызова `remap_file_pages()`, которые не упоминались ранее.

- Аргумент `prot` игнорируется и должен быть равен `0`. В будущем с его помощью можно будет менять способ защиты участка памяти, с которым работает вызов `remap_file_pages()`. В текущей реализации действуют те же параметры защиты, что и во всей области VMA.

  Области VMA также применяются в виртуальных машинах и сборщиках мусора, у которых должна быть возможность защищать от записи отдельные страницы. Изначально вызов remap_file_pages() должен был уметь это делать, но пока данный механизм не реализован.

- Аргумент `flags`, который на текущий момент не используется.

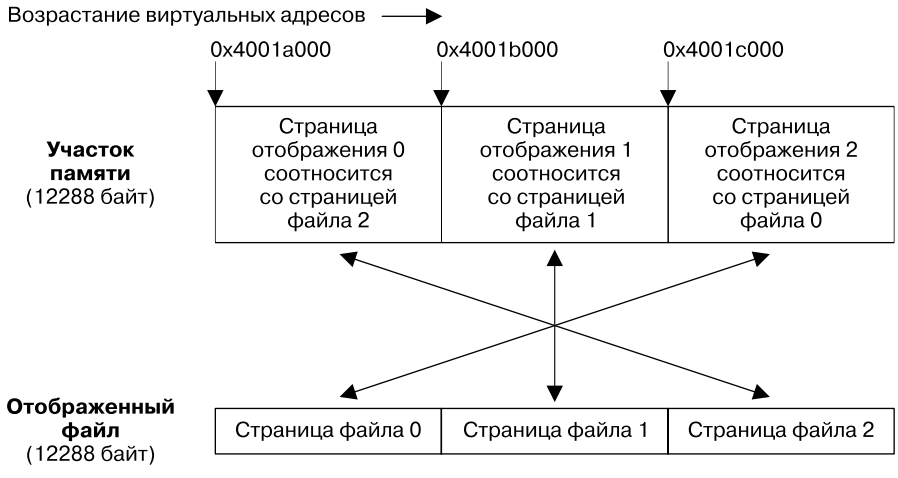


Рис. 45.5. Нелинейное отображение файла

В своей текущей реализации вызов `remap_file_pages()` способен работать только с разделяемыми отображениями (`MAP_SHARED`).

Системный вызов `remap_file_pages()` поддерживается только в Linux; он не предусмотрен стандартом SUSv3 и отсутствует в других UNIX-системах.

45.12. Резюме

Системный вызов `mmap()` создает отображение в виртуальном адресном пространстве вызывающего процесса. Системный вызов `munmap()` выполняет обратную операцию, удаляя отображение из этого пространства.

Отображения бывают двух видов: файловые и анонимные. В первом случае в память отображается содержимое определенного участка файла. Анонимное отображение (созданное с помощью флага `MAP_ANONYMOUS` или путем отображения устройства `/dev/zero`) не связано ни с каким файлом; его содержимое заполнено нулями.

Отображения могут быть либо приватными (`MAP_PRIVATE`), либо разделяемыми (`MAP_SHARED`). Разница между ними определяет, будут ли видны изменения разделяемой памяти другим процессам; в случае с файловым отображением это влияет на то, будет ли ядро записывать изменения участка памяти обратно в исходный файл. Когда процесс отображает файл с помощью флага `MAP_PRIVATE`, изменения, которые он вносит в данное отображение, не видны другим процессам и не возвращаются в отображенный файл. Флаг `MAP_SHARED` обеспечивает противоположное поведение: изменения участка памяти попадают в исходный файл и видны другим процессам.

Ядро автоматически применяет изменения, внесенные к отображению типа `MAP_SHARED`, к исходному файлу, однако не дает никаких гарантий относительно времени выполнения таких операций. Приложение может задействовать системный вызов `msync()`, чтобы инициировать немедленную синхронизацию участка памяти с отображенным файлом.

Отображения в память используются для многих задач, среди которых:

- выделение памяти, доступной только одному процессу (приватные анонимные отображения);
- инициализация содержимого сегментов с кодом и данными процесса (приватные файловые отображения);

- разделение памяти между процессами, связанными вызовом `fork()` (разделяемые анонимные отображения);
- отображение ввода/вывода в память, возможно, в сочетании с разделением памяти между неродственными процессами (разделяемые файловые отображения).

При доступе к содержимому отображений можно столкнуться с двумя сигналами: `SIGSEGV` и `SIGBUS`. Первый генерируется при попытке доступа, нарушающей защиту отображения (или если мы обращаемся к адресу, который не был отображен). Второй возникает для файловых отображений при обращении к участку, не связанному с соответствующим участком файла (то есть когда отображение больше, чем его исходный файл).

Перерасход пространства подкачки позволяет системе выделять для процессов память, объем которой превышает размер доступного физического и виртуального пространства. Такой подход возможен по той причине, что процессы обычно не задействуют всю память, выделенную им. Перерасходом можно управлять как на уровне вызова `mmap()`, используя флаг `MAP_NORESERVE`, так и на общесистемном уровне с помощью файлов `/proc`.

Системный вызов `mremap()` позволяет изменять размер существующих отображений. Для создания нелинейных файловых отображений можно применять системный вызов `remap_file_pages()`.

Дополнительная информация

Сведения о реализации вызова `mmap()` в Linux можно найти в [Bovet & Cesati, 2005]. Подробности реализации `mmap()` в других UNIX-системах доступны в таких источниках: [McKusick et al., 1996] (BSD), [Goodheart & Cox, 1994] и [Vahalia, 1996] (System V Release 4).

45.13. Упражнения

45.1. Напишите программу, аналогичную команде `cp(1)`, применяющей вызовы `mmap()` и `memcpy()` (вместо `read()` или `write()`) для копирования файла из одного места в другое. (Используйте вызов `fstat()`, чтобы получить размер заданного файла, который затем можно будет указать при создании отображения в память; для задания размера итогового файла подойдет вызов `ftruncate()`.)

45.2. Напишите программы, подтверждающие доставку сигналов `SIGBUS` и `SIGSEGV` в обстоятельствах, описанных в подразделе 45.4.3.

45.3. Напишите программу, которая использует методику `MAP_FIXED`, описанную в разделе 45.10, для создания нелинейного отображения, аналогичного показанному на рис. 45.5.

46 Операции с виртуальной памятью

Эта глава посвящена системным вызовам, предназначенным для выполнения различных операций с виртуальным адресным пространством процесса:

- системный вызов `mprotect()` изменяет защиту участка виртуальной памяти;
- системные вызовы `mlock()` и `mlockall()` «запирают» участок виртуального пространства в рамках физической памяти и не дают сбросить его на диск;
- системный вызов `mincore()` позволяет процессу определять, находятся ли страницы виртуального пространства в физической памяти;
- системный вызов `madvise()` дает возможность сообщить ядру о потенциальной модели поведения процесса в контексте использования участка виртуальной памяти.

Некоторые из этих системных вызовов могут быть особенно полезны в сочетании с участками разделяемой памяти (см. главы 45 и 50), но вы можете применять их для работы с любыми сегментами виртуального адресного пространства.

> Методики, описанные в данной главе, на самом деле не имеют никакого отношения к межпроцессному взаимодействию; я включил их в эту часть книги, поскольку они иногда используются вместе с разделяемой памятью.

46.1. Изменение защиты памяти: mprotect()

Системный вызов `mprotect()` изменяет защиту страниц виртуальной памяти в диапазоне длиной `length` байт, который начинается с адреса `addr`.

```
#include <sys/mman.h>

int mprotect(void *addr, size_t length, int prot);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Значение, переданное в аргументе `addr`, должно быть кратным размеру страницы памяти в системе (который возвращается вызовом `sysconf(_SC_PAGESIZE)`). Стандарт SUSv3 гласит, что аргумент `addr` *должен* быть выровнен по странице; в SUSv4 всего лишь предусмотрена такая *возможность*. Поскольку защита распространяется на целые страницы, значение *`length`* на практике округляется до следующего кратного размеру страницы в системе.

Аргумент `prot` представляет собой битовую маску, задающую новую защиту для данного участка памяти. Он должен быть равен либо `PROT_NONE`, либо сочетанию значений `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`, к которым применено побитовое ИЛИ. Все эти флаги имеют то же значение, что и в вызове `mmap()` (табл. 45.2).

Если процесс попытается обратиться к участку памяти, нарушив заданную защиту, то ядро пошлет ему сигнал `SIGSEGV`.

Вызов `mprotect()` позволяет изменить защиту участка отображенной памяти, заданную с помощью `mmap()`, как показано в листинге 46.1. Эта программа создает анонимное

отображение, любой доступ к которому изначально закрыт (PROT_NONE). Затем она открывает доступ для чтения и записи. Прежде чем выполнить данное изменение, программа использует вызов system(), чтобы запустить консольную команду, выводящую строчку из файла /proc/PID/maps, относящуюся к отображенному участку. Это позволит увидеть, как изменилась защита памяти (ту же информацию можно получить, вручную разобрав файл /proc/self/maps, но вызов system() позволяет сократить код программы). Запустив данное приложение, мы увидим следующее:

```
$ ./t_mprotect
Before mprotect()
b7cde000-b7dde000 ---s 00000000 00:04 18258    /dev/zero (deleted)
After mprotect()
b7cde000-b7dde000 rw-s 00000000 00:04 18258    /dev/zero (deleted)
```

Последняя строка вывода говорит о том, что вызов mprotect() изменил права доступа к участку памяти на PROT_READ | PROT_WRITE.

Листинг 46.1. Изменение защиты памяти с помощью вызова mprotect()

vmem/t_mprotect.c

```
#define _BSD_SOURCE        /* Получаем определение MAP_ANONYMOUS из <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

#define LEN (1024 * 1024)

#define SHELL_FMT "cat /proc/%ld/maps | grep zero"
#define CMD_SIZE (sizeof(SHELL_FMT) + 20)
     /* Выделяем дополнительное место для строки с целым числом */

int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    char *addr;

    /* Создаем анонимное отображение с полным отсутствием доступа */

    addr = mmap(NULL, LEN, PROT_NONE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    /* Выводим строку из /proc/self/maps, относящуюся к отображению */

    printf("Before mprotect()\n");
    snprintf(cmd, CMD_SIZE, SHELL_FMT, (long) getpid());
    system(cmd);

    /* Изменяем защиту памяти, разрешая чтение и запись */

    if (mprotect(addr, LEN, PROT_READ | PROT_WRITE) == -1)
        errExit("mprotect");

    printf("After mprotect()\n");
    system(cmd);     /* Проверяем защиту, считывая файл /proc/self/maps */

        exit(EXIT_SUCCESS);
}
```

vmem/t_mprotect.c

46.2. Блокирование памяти: mlock() и mlockall()

Иногда возникает необходимость «запереть» часть виртуальной памяти процесса (или всю целиком), чтобы она гарантированно оставалась в рамках физического адресного пространства. Одной из причин этого может быть повышение производительности. Доступ к заблокированным участкам памяти никогда не будет задержан из-за сбоя в работе страницы. Это полезно в случаях, когда требуется обеспечить низкое время отзыва.

Еще одна причина для запирания памяти связана с безопасностью. Если страница виртуальной памяти, содержащая конфиденциальную информацию, не попадает в пространство подкачки, ее копия никогда не будет записана на диск. Сброс данных на диск теоретически чреват тем, что позже их могут оттуда прочитать (злоумышленник может намеренно спровоцировать такую ситуацию, запустив программу, которая потребляет большой объем памяти, и тем самым заставив сбросить пространство подкачки страницы других процессов). Чтение информации из этого пространства можно выполнить даже после завершения процесса, поскольку ядро не гарантирует обнуления данных, там хранящихся (обычно чтение из устройства подкачки разрешено только привилегированным процессам).

> Режим сна в настольных и переносных компьютерах сохраняет копию физической памяти на диск вне зависимости от блокировки памяти.

В данном разделе мы познакомимся с системными вызовами, предназначенными для частичной или полной блокировки и разблокировки виртуальной памяти процесса. Но сначала рассмотрим ограничение на ресурсы, которое регулирует эти операции.

Ограничение на ресурсы RLIMIT_MEMLOCK

Ограничение `RLIMIT_MEMLOCK`, касающееся количества байтов, доступных процессу для запирания в памяти, было кратко рассмотрено в разделе 36.3. Теперь мы познакомимся с ним более подробно.

В ядрах Linux до версии 2.6.9 только привилегированные процессы (`CAP_IPC_LOCK`) могли блокировать память, а мягкое ограничение `RLIMIT_MEMLOCK` регулировало максимальное количество байтов, доступных для блокировки.

В Linux 2.6.9 модель блокировки памяти претерпела изменения. Теперь небольшие участки адресного пространства могут запираться и непривилегированными процессами. Это может пригодиться программам, которым нужно поместить в заблокированную память конфиденциальную информацию небольшого объема, чтобы исключить возможность ее записи на диск; например, утилита `pgp` использует такую возможность для фразовых паролей. Результаты данных изменений следующие:

- привилегированные процессы могут запирать любые объемы памяти, без ограничений (то есть флаг `RLIMIT_MEMLOCK` игнорируется);
- непривилегированные процессы теперь могут запирать память, объем которой ограничен мягким ограничением `RLIMIT_MEMLOCK`.

По умолчанию значения мягкого и жесткого ограничений `RLIMIT_MEMLOCK` равны восьми страницам (на платформе x86-32, например, это 32 768 байт).

Ограничение `RLIMIT_MEMLOCK` затрагивает:

- вызовы `mlock()` и `mlockall()`;
- вызов `mmap()` с флагом `MAP_LOCKED`, который используется для запирания отображения при его создании (см. раздел 45.6);
- вызов `shmctl()` с флагом `SHM_LOCK`, применяемым для блокирования сегментов разделяемой памяти в System V.

Блокировка применяется постранично, поскольку единицей управления виртуальной памятью является страница. Когда проверяется ограничение, значение RLIMIT_MEMLOCK округляется до *наименьшего* кратного размеру страницы памяти в системе.

Ограничение RLIMIT_MEMLOCK имеет всего лишь одно (мягкое) значение, но, по сути, он определяет сразу два отдельных ограничения.

- В случае с операциями mlock(), mlockall() и вызовом mmap() с флагом MAP_LOCKED данное ограничение определяет максимальное количество байтов виртуального адресного пространства, которое может заблокировать каждый отдельный процесс.
- Для вызова shmctl() с флагом SHM_LOCK ограничение RLIMIT_MEMLOCK определяет максимальное количество байтов разделяемого сегмента адресного пространства, которые можно заблокировать с помощью реального пользовательского идентификатора для текущего процесса. Когда процесс выполняет операцию shmctl() SHM_LOCK, ядро проверяет общий объем байтов разделяемой памяти типа System V, уже заблокированный с применением реального UID вызывающего процесса. Если размер сегмента, который нужно заблокировать, не выходит за пределы ограничения RLIMIT_MEMLOCK для текущего процесса, операция завершается успешно.

В случае с разделяемой памятью типа System V ограничение RLIMIT_MEMLOCK имеет иную семантику, поскольку память в таком сегменте может продолжать существовать даже без привязки к какому-либо процессу (чтобы ее очистить, нужно вручную вызвать shmctl() с флагом IPC_RMID и дождаться, когда все процессы отсоединят от нее свое адресное пространство).

Блокировка и разблокировка участков памяти

Для выполнения этих операций процесс может использовать вызовы mlock() и munlock().

```
#include <sys/mman.h>

int mlock(void *addr, size_t length);
int munlock(void *addr, size_t length);
```

Возвращают 0 при успешном завершении или -1 при ошибке

Системный вызов mlock() блокирует все страницы вызывающего процесса в диапазоне виртуальных адресов длиной length, начинающегося с addr. В отличие от аналогичного аргумента в ряде других вызовов, связанных с памятью, addr не обязательно выравнивать по странице; ядро автоматически выбирает ближайшую страницу, адрес начала которой не превышает addr. Однако стандарт SUSv3 разрешает реализации требовать, чтобы аргумент addr был кратным размеру страницы в системе, и портируемые приложения, использующие вызовы mlock() и munlock(), должны следовать этому требованию.

Поскольку единицей блокирования является целая страница, конец блокируемого участка совпадает с концом следующей страницы, адрес которой больше length плюс addr. Например, в системе, где размер страницы равен 4096 байтам, вызов mlock(2000, 4000) заблокирует диапазон байтов с 0 по 8191.

> Чтобы узнать, сколько всего памяти заблокировал текущий процесс, можно прочитать поле VmLck в файле /proc/PID/status.

После успешного вызова mlock() все страницы в заданном диапазоне гарантированно запираются в физической памяти. Вызов mlock() дает сбой, если для блокирования всех запрашиваемых страниц не хватает физической памяти или запрос нарушает ограничение на ресурсы RLIMIT_MEMLOCK.

Пример использования вызова `mlock()` показан в разделе 46.2.

Системный вызов `munlock()` выполняет обратную операцию, удаляя из памяти блокировку, установленную ранее вызывающим процессом. Аргументы `addr` и `length` интерпретируются так же, как и в `mlock()`. Разблокировка набора страниц не гарантирует, что они перестанут храниться в физическом памяти, — они будут удалены оттуда только в том случае, если память потребуется другим процессам.

Блокировки могут удаляться не только вручную, благодаря вызову `munlock()`, но и автоматически. Это происходит в следующих ситуациях:

- при завершении процесса;
- при удалении заблокированных страниц из отображения с помощью вызова `munmap()`;
- если заблокированные страницы перекрываются в результате вызова `mmap()` с флагом `MAP_FIXED`.

Подробности о семантике блокирования памяти

Ниже мы рассмотрим некоторые особенности семантики блокирования памяти.

Блокировки памяти не наследуются потомком, созданным путем вызова `fork()`, и не сохраняются на протяжении работы `exec()`.

Если несколько процессов разделяют какой-то набор страниц (например, отображение типа `MAP_SHARED`), то эти страницы остаются запертыми в физической памяти, пока хотя бы один процесс удерживает соответствующую блокировку.

Блокировки памяти не накапливаются в рамках одного процесса. Если процесс многократно вызовет `mlock()` для определенного диапазона виртуальных адресов, то это приведет к установке всего лишь одной блокировки, которую можно будет убрать с помощью единственного вызова `munlock()`. С другой стороны, при отображении одних и тех же страниц (то есть того же файла) на разные участки памяти того же процесса и блокировке каждого из этих отображений страницы будут оставаться запертыми в физической памяти до тех пор, пока все отображения не будут разблокированы.

Тот факт, что единицей измерения блокировок памяти являются страницы, а также то, что блокировки не могут накапливаться, означает следующее: независимое применение вызовов `mlock()` и `munlock()` к разным структурам данных на одной виртуальной странице логически неверно. Представьте, к примеру, что у нас есть две структуры данных внутри одной страницы виртуальной памяти и на них ссылаются указатели `p1` и `p2`. Теперь выполним следующие вызовы:

```
mlock(*p1, len1);
mlock(*p2, len2);        /* На самом деле ни на что не влияет */
munlock(*p1, len1);
```

Все вышеприведенные вызовы завершатся успешно, но в результате будет разблокирована вся страница; таким образом, структура данных, на которую указывает `p2`, не запирается в физической памяти.

Обратите внимание на то, что семантика операции `shmctl() SHM_LOCK` отличается от семантики вызовов `mlock()` и `mlockall()`:

- после операции `SHM_LOCK` страницы блокируются, только если последующие обращения к ним приводят к отказу. Вызовы `mlock()` и `mlockall()` возвращаются только после того, как проверят запертые в физической памяти страницы на отказы;
- операция `SHM_LOCK` работает скорее с разделяемым сегментом памяти, чем с процессом (по данной причине значение поля `VmLck` в файле `/proc/PID/status` не учитывает размер любых подключенных сегментов памяти типа System V, которые были заблокированы операцией `SHM_LOCK`). Это значит, что после отказа страница остается в физической памяти, даже если все процессы отключатся от соответствующего сегмента памяти. Участок,

запертый в физической памяти с помощью вызова `mlock()` (или `mlockall()`), остается там только до тех пор, пока хотя бы один процесс удерживает блокировку данного участка.

Блокировка и разблокировка всей памяти процесса

Для выполнения этих операций процесс может использовать вызовы `mlockall()` и `munlockall()`.

```
#include <sys/mman.h>

int mlockall(int flags);
int munlockall(void);
```

Возвращают `0` при успешном завершении или `-1` при ошибке

Системный вызов `mlockall()` блокирует все страницы в виртуальном адресном пространстве процесса, уже отображенные или являющиеся таковыми в будущем — это зависит от битовой маски `flags`, в которой побитовое ИЛИ применяется к одной или нескольким константам, представленным ниже.

- `MCL_CURRENT` — блокирует все страницы, отображенные на виртуальное адресное пространство процесса. Это относится к страницам, выделенным для отображений, стека и сегментов с текстом и данными. После успешного вызова с указанием данного флага все страницы вызывающего процесса гарантированно запираются в физической памяти. Флаг `MCL_CURRENT` не влияет на страницы, которые впоследствии будут выделены в виртуальном адресном пространстве процесса; для них следует использовать флаг `MCL_FUTURE`.
- `MCL_FUTURE` — блокирует все страницы, которые впоследствии будут отображены на виртуальное адресное пространство процесса. Это, например, могут быть страницы разделяемого участка памяти, отображенные с помощью вызовов `mmap()` или `shmat()`, часть кучи, растущей снизу вверх, или стека, который растет сверху вниз. В результате применения флага `MCL_FUTURE` последующие операции выделения памяти (например, `mmap()`, `sbrk()` или `malloc()`) могут завершиться неудачей, или же расширение стека может привести к получению сигнала `SIGSEGV`. Это происходит, когда в системе не хватает памяти, которую можно было бы выделить процессу, или когда исчерпывается мягкое ограничение на ресурсы `RLIMIT_MEMLOCK`.

Все правила относительно ограничений, жизненного цикла и наследования блокировок памяти, действующих для вызова `mlock()`, относятся и к `mlockall()`.

Системный вызов `munlockall()` разблокирует все страницы вызывающего процесса и отменяет последствия от любых ранее выполненных вызовов `mlockall(MCL_FUTURE)`. Как и в случае с `munlock()`, удаление разблокированных страниц из физической памяти не гарантируется.

46.3. Определение местонахождения памяти: mincore()

Системный вызов `mincore()` дополняет операции блокировки памяти. Он сообщает, какие страницы в диапазоне виртуальных адресов находятся в физической памяти и в случае доступа к ним не приведут к отказу.

Вызов `mincore()` не входит в стандарт SUSv3. Он доступен во многих реализациях UNIX, но не во всех. В Linux он поддерживается, начиная с ядра версии 2.4.

```
#define _BSD_SOURCE                /* Или: #define _SVID_SOURCE */
#include <sys/mman.h>

int mincore(void *addr, size_t length, unsigned char *vec);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Системный вызов mincore() возвращает сведения о местоположении страниц виртуального адресного пространства длиной length байт, которое начинается с адреса addr. Адрес, указанный в аргументе addr, должен быть выровнен по странице, а значение length на практике округляется до следующего кратного размеру страницы памяти в системе, поскольку информация относится к целым страницам.

Сведения о местонахождении памяти возвращается в аргументе vec, который должен представлять собой массив размером (length + PAGE_SIZE - 1) / PAGE_SIZE байт (в Linux *vec* имеет тип unsigned char *; в ряде других реализаций UNIX вместо этого используется тип char *). Если соответствующая страница находится в физической памяти, то устанавливается младший бит в каждом байте. Установка остальных битов не предусмотрена в некоторых UNIX-системах, в связи с чем портируемые приложения должны проверять только этот единственный бит.

Информация, возвращаемая mincore(), может измениться после выполнения вызова и проверки элементов vec. Только страницы, заблокированные вызовами mlock() или mlockall(), гарантированно остаются в физической памяти.

Применение вызовов mlock() и mincore() демонстрируется в листинге 46.2. Выделив и отобразив участок памяти с помощью mmap(), эта программа вызывает mlock() для блокирки либо всего участка, либо групп страниц с постоянными интервалами (каждый аргумент командной строки программы выражается в виде страниц; программа переводит их в байты, как того требуют вызовы mmap(), mlock() и mincore()). Перед и после mlock() она вызывает mincore(), чтобы получить сведения о местоположении страниц заданного участка памяти, и отображает эту информацию в графическом виде.

Листинг 46.2. Использование вызовов mlock() и mincore()

vmem/memlock.c

```
#define _BSD_SOURCE      /* Получаем объявление mincore() и определение
                            MAP_ANONYMOUS из <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

/* Выводим местоположение страниц в диапазоне [addr .. (addr + length - 1)] */

static void
displayMincore(char *addr, size_t length)
{
    unsigned char *vec;
    long pageSize, numPages, j;

    pageSize = sysconf(_SC_PAGESIZE);
    numPages = (length + pageSize - 1) / pageSize;
    vec = malloc(numPages);
    if (vec == NULL)
        errExit("malloc");

    if (mincore(addr, length, vec) == -1)
        errExit("mincore");
```

```
    for (j = 0; j < numPages; j++) {
        if (j % 64 == 0)
            printf("%s%10p: ", (j == 0) ? "" : "\n", addr + (j * pageSize));
        printf("%c", (vec[j] & 1) ? '*' : '.');
    }
    printf("\n");
    free(vec);
}

int
main(int argc, char *argv[])
{
    char *addr;
    size_t len, lockLen;
    long pageSize, stepSize, j;

    if (argc != 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-pages lock-page-step lock-page-len\n", argv[0]);

    pageSize = sysconf(_SC_PAGESIZE);
    if (pageSize == -1)
        errExit("sysconf(_SC_PAGESIZE)");

    len =       getInt(argv[1], GN_GT_0, "num-pages") * pageSize;
    stepSize =  getInt(argv[2], GN_GT_0, "lock-page-step") * pageSize;
    lockLen =   getInt(argv[3], GN_GT_0, "lock-page-len") * pageSize;
    addr = mmap(NULL, len, PROT_READ, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    printf("Allocated %ld (%#lx) bytes starting at %p\n",
             (long) len, (unsigned long) len, addr);

    printf("Before mlock:\n");
    displayMincore(addr, len);

    /* Запираем в памяти страницы, указанные с помощью аргументов командной строки */
    for (j = 0; j + lockLen <= len; j += stepSize)
        if (mlock(addr + j, lockLen) == -1)
            errExit("mlock");

    printf("After mlock:\n");
    displayMincore(addr, len);
    exit(EXIT_SUCCESS);
}
```

vmem/memlock.c

В следующей сессии командной строки показан пример выполнения программы из листинга 46.2. Мы выделяем семь групп по три смежные страницы в каждой (всего 32 страницы):

```
$ su                                        Повышаем привилегии
Password:
# ./memlock 32 8 3
Allocated 131072 (0x20000) bytes starting at 0x4014a000
Before mlock:
0x4014a000: ................................
After mlock:
0x4014a000: ***.....***.....***.....***.....
```

В этом программном выводе для обозначения страниц, которые находятся в физической памяти и пространстве подкачки, используются соответственно звездочки и точки. Как можно видеть в последней строке вывода, в каждой группе из восьми страниц три находятся в физическом адресном пространстве.

В данном примере мы получили повышенные привилегии, чтобы программа могла задействовать вызов `mlock()`. Начиная с Linux 2.6.9, данный шаг стал необязательным (при условии, что объем блокируемой памяти не превышает мягкое ограничение `LIMIT_MEMLOCK`).

46.4. Предсказание модели использования памяти в будущем: madvise()

Системный вызов `madvise()` служит для улучшения производительности программы путем информирования ядра о том, как именно вызывающий процесс (скорее всего) будет применять страницы памяти в диапазоне длиной `length` байт, начиная с адреса `addr`. С помощью данной информации ядро может повысить эффективность операций ввода/вывода, выполняемых с файлом, который отображается на эти страницы (обсуждение файловых отображений см. в разделе 45.4). В ядре Linux вызов `madvise()` доступен с версии 2.4.

```
#define _BSD_SOURCE
#include <sys/mman.h>

int madvise(void *addr, size_t length, int advice);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Значение `addr` должно быть выровнено по странице, а аргумент `length` в итоге округляется к следующему значению, кратному размеру страницы памяти в системе. Аргумент `advice` может быть равен одной из следующих констант.

- `MADV_NORMAL` — поведение по умолчанию. Страницы передаются в виде небольших групп. Это приводит к упреждающему и отложенному чтению.
- `MADV_RANDOM` — обращение к страницам на данном участке будет произвольным, поэтому упреждающее чтение не повлечет никакой выгоды. Таким образом, в каждой операции чтения ядро должно извлекать как можно меньший объем данных.
- `MADV_SEQUENTIAL` — обращение к страницам на этом участке будет последовательным. Таким образом, ядро может прибегнуть к агрессивному упреждающему чтению; после доступа к страницам их можно быстро освобождать.
- `MADV_WILLNEED` — страницы на этом участке следует считывать наперед, готовясь к будущему доступу. Операция `MADV_WILLNEED` по своим результатам похожа на вызов `readahead()` (доступный только в Linux) и `posix_fadvise()` с флагом `POSIX_FADV_WILLNEED`.
- `MADV_DONTNEED` — вызывающий процесс больше не требует, чтобы страницы на данном участке находились в физической памяти. Результаты применения этого флага варьируются в зависимости от реализации UNIX. Для начала посмотрим, как он ведет себя в Linux. Отображенные страницы на участке `MAP_PRIVATE` всегда отклоняются; это значит, что будут потеряны все внесенные в них изменения. Диапазон адресов виртуальной памяти остается доступным, но при следующем доступе к каждой странице произойдет отказ, который приведет к ее повторной инициализации (с помощью либо содержимого отображенного на нее файла, либо (если это анонимное отображение) нулей). Данную особенность можно использовать как средство повторной инициализации содержимого

участка MAP_PRIVATE. В случае с участком MAP_SHARED ядро *может* отклонить измененные страницы в зависимости от архитектуры (на платформе x86 этого не происходит). Такое же поведение встречается и в ряде других систем. Но существуют реализации UNIX, в которых MADV_DONTNEED просто информирует ядро о том, что заданные страницы при необходимости можно сбросить на диск. Портируемые приложения не должны полагаться на деструктивную семантику флага MADV_DONTNEED, действующую в Linux.

В Linux 2.6.16 было добавлено три новых нестандартных значения advice: MADV_DONTFORK, MADV_DOFORK и MADV_REMOVE. В Linux 2.6.32 и 2.6.33 появилось еще четыре нестандартных значения: MADV_HWPOISON, MADV_SOFT_OFFLINE, MADV_MERGEABLE и MADV_UNMERGEABLE. Эти константы предназначены для особых случаев и описаны в справочной странице madvise(2).

В большинстве реализаций UNIX вызов madvise() поддерживает для аргумента advice как минимум те константы, которые описаны выше. Но в SUSv3 данный программный интерфейс носит другое имя, posix_madvise(), а ко всем соответствующим значениям advice добавляется префикс POSIX_. Таким образом, стандартные константы выглядят как POSIX_MADV_NORMAL, POSIX_MADV_RANDOM, POSIX_MADV_SEQUENTIAL, POSIX_MADV_WILLNEED и POSIX_MADV_DONTNEED. Этот альтернативный интерфейс реализован в библиотеке glibc (начиная с версии 2.2) и использует вызовы madvise(), однако в некоторых UNIX-системах не поддерживается.

Согласно стандарту SUSv3 функция posix_madvise() не должна влиять на семантику программы. Но в библиотеке glibc до версии 2.7 операция POSIX_MADV_DONTNEED была реализована с помощью вызова madvise() с флагом MADV_DONTNEED, который идет вразрез с этим требованием (как было описано выше). Начиная с glibc 2.7, при использовании константы POSIX_MADV_DONTNEED функция-обертка posix_madvise() не выполняет никаких действий и, следовательно, не влияет на семантику программы.

46.5. Резюме

В данной главе были рассмотрены различные операции, которые можно применять к виртуальной памяти процесса:

- системный вызов mprotect() изменяет защиту участка виртуальной памяти;
- системные вызовы mlock() и mlockall() «запирают» участок виртуального пространства в рамках физической памяти и не дают сбросить его на диск;
- системный вызов mincore() сообщает, какие страницы виртуального пространства находятся в физической памяти;
- системный вызов madvise() и функция posix_madvise() дают возможность сообщить ядру о потенциальной модели поведения процесса в контексте использования участка виртуальной памяти.

46.6. Упражнения

46.1. Проверьте влияние ограничения RLIMIT_MEMLOCK, написав программу, которая устанавливает соответствующее ограничение и затем пытается заблокировать больше памяти, чем позволено.

46.2. Напишите программу, проверяющую действие вызова madvise() с флагом MADV_DONTNEED на отображение типа MAP_PRIVATE, доступное для записи.

47 Введение в межпроцессное взаимодействие стандарта POSIX

Стандарт POSIX.1b содержит ряд расширений реального времени, которые формируют набор IPC-механизмов (одной из целей создателей данного стандарта было разработать средства межпроцессного взаимодействия, лишенные недостатков аналогичных механизмов из состава System V). В совокупности эти механизмы называются POSIX IPC. Их можно разделить на три категории.

- *Очереди сообщений* могут использоваться для передачи информации между процессами. Считывающий и записывающий процессы обмениваются блоками (сообщениями) с четкими границами (в отличие от каналов, которые предоставляют сплошной байтовый поток). Стандарт POSIX позволяет назначать каждому сообщению отдельный приоритет; сообщения с более высоким приоритетом передаются раньше остальных.
- *Семафоры* позволяют синхронизировать действия нескольких процессов. POSIX-семафоры представляют собой целые числа, которые управляются ядром и не могут быть меньше 0. Они отличаются простотой использования: каждый из них выделяется и управляется отдельно с помощью всего лишь двух операций — увеличения и уменьшения значения семафора на 1.
- *Разделяемая память* позволяет нескольким процессам работать с одним и тем же участком памяти. В стандарте POSIX этот механизм применяется для быстрого межпроцессного взаимодействия. Изменения, внесенные в разделяемую память одним процессом, сразу же становятся доступными для других процессов, разделяющих тот же участок.

В данной главе мы познакомимся со средствами POSIX IPC, уделяя внимание характерным для них свойствам.

47.1. Краткий обзор программных интерфейсов

Механизмы POSIX IPC имеют целый ряд общих свойств, которые будут подробно рассмотрены на следующих нескольких страницах. В табл. 47.1 представлены их программные интерфейсы.

> Если не считать упоминаний в табл. 47.1, в этой главе будет проигнорирован тот факт, что POSIX-семафоры бывают двух видов: именованные и анонимные. Первые похожи на другие механизмы POSIX IPC, описанные в данной главе: они имеют имена и доступны для любого процесса с подходящими правами доступа к заданному объекту. У анонимных семафоров нет соответствующего идентификатора; они помещаются на участок памяти, который разделяется несколькими процессами (или потоками одного процесса). Семафоры обоих этих типов будут подробно описаны в главе 49.

Таблица 47.1. Перечень программных интерфейсов для работы с объектами POSIX IPC

Интерфейс	Очереди сообщений	Семафоры	Разделяемая память
Заголовочный файл	<mqueue.h>	<semaphore.h>	<sys/mman.h>
Тип объекта	mqd_t	sem_t *	int (дескриптор файла)
Создание/открытие	mq_open()	sem_open()	shm_open() + + mmap()
Закрытие	mq_close()	sem_close()	munmap()
Удаление	mq_unlink()	sem_unlink()	shm_unlink()
Выполнение IPC	mq_send(), mq_receive()	sem_post(), sem_wait(), sem_getvalue()	Работа с адресами на разделяемом участке памяти
Прочие операторы	mq_setattr() — устанавливает атрибуты, mq_getattr() — получает атрибуты, mq_notify() — запрашивает уведомление	sem_init() — инициализирует анонимный семафор, sem_destroy() — уничтожает анонимный семафор	(нет)

Имена IPC-объектов

Чтобы получить доступ к объекту POSIX IPC, нужно его как-то распознать. Единственным портируемым средством такой идентификации, предусмотренным стандартом SUSv3, является обращение по имени, которое начинается со слеша, например `/myobject`. Такой портируемый способ именования IPC-объектов поддерживается в Linux и некоторых других системах (например, в Solaris).

В Linux имена для разделяемой памяти и очередей сообщений POSIX ограничены `NAME_MAX` (`255`) символами. Для семафоров это ограничение меньше на четыре символа, поскольку к их именам автоматически добавляется префикс `sem`.

Стандарт SUSv3 не запрещает использовать имена, не соответствующие формату `/myobject`, но отмечает, что семантика таких имен определяется конкретной реализацией. Правила именования IPC-объектов могут разниться в некоторых системах. Например, в Tru64 5.1 эти объекты создаются в файловой системе и интерпретируются как полный или относительный путь. Если вызывающий процесс не имеет права создавать файлы в заданном каталоге, то операция `open` завершится неудачей. Это значит, что в системе Tru64 непривилегированные процессы не могут создавать имена вида `/myobject`, поскольку им обычно не разрешается изменять содержимое корневого каталога (`/`). Похожие правила формирования имен, которые можно передавать IPC-вызову `open`, действуют и в других реализациях. Таким образом, в портируемых приложениях процедуру генерирования имен IPC-объектов следует выносить в отдельные функции или заголовочные файлы, которые можно адаптировать для целевой системы.

Создание или открытие IPC-объекта

У каждого IPC-механизма есть своя операция открытия (`mq_open()`, `sem_open()` или `shm_open()`) — аналог традиционного для UNIX системного вызова `open()`, предназначенного для работы с файлами. В зависимости от указанного имени IPC-вызов `open` выполнит одно из этих двух действий:

- создаст новый объект с заданным именем, откроет его и вернет соответствующий дескриптор;
- откроет существующий объект и вернет его дескриптор.

Дескриптор, возвращаемый IPC-вызовом `open`, аналогичен файловому дескриптору, который возвращается традиционным системным вызовом `open()`, — он используется в последующих операциях с заданным объектом.

Тип полученного дескриптора зависит от типа объекта. Например, дескриптор очереди сообщений представляет собой структуру данных `mqd_t`; дескриптором семафора является указатель типа `sem_t *`; в случае с разделяемой памятью это обычный файловый дескриптор.

Все IPC-вызовы `open` предусматривают как минимум три аргумента: `name`, `oflag` и `mode`. Рассмотрим их на примере вызова `shm_open()`:

```
fd = shm_open("/mymem", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
```

Данные аргументы аналогичны тем, что используются в традиционном для UNIX системном вызове `open()`. Аргумент `name` идентифицирует объект, который нужно создать или открыть. Аргумент `oflag` представляет собой битовую маску, которая может включать в себя следующие флаги:

- `O_CREAT` — создает объект, если его еще не существует. Если опустить этот флаг в ситуации, когда заданный объект не был создан заранее, то получим ошибку `ENOENT`;
- `O_EXCL` — если объект уже существует и при этом указан флаг `O_CREAT`, то получим ошибку `EEXIST`. Проверка существования объекта и его создание — две атомарные операции (см. раздел 5.1). Без `O_CREAT` данный флаг ни на что не влияет.

В зависимости от типа объекта аргумент `oflag` может также содержать значения `O_RDONLY`, `O_WRONLY` и `O_RDWR`, похожие на аналогичные константы в вызове `open()`. Некоторые IPC-механизмы поддерживают дополнительные флаги.

Оставшийся аргумент, `mode`, содержит битовую маску, определяющую права доступа, необходимые для работы с новым объектом, если таковой создается в результате вызова (то есть если был указан флаг `O_CREAT`, а самого объекта еще не существовало). Значения, которые можно передать этому аргументу, такие же, как и в случае с файлами (см. табл. 15.4). По аналогии с системным вызовом `open()` к маске с правами доступа применяется атрибут `umask` процесса (см. подраздел 15.4.6). Пользовательские и групповые права владения новым объектом берутся из действующих идентификаторов пользователя и группы процесса, выполняющего IPC-вызов `open`. (Если быть предельно точным, в Linux владение новым объектом POSIX IPC определяется ID исполняемого файла, которые обычно совпадают с действующими идентификаторами; см. раздел 9.5.)

> В реализациях, в которых IPC-объекты хранятся в обычной файловой системе, стандарт SUSv3 разрешает использовать GID родительского каталога в качестве идентификатора группы IPC-объекта.

Закрытие IPC-объекта

Очереди сообщений и семафоры POSIX поддерживают IPC-вызов `close`, сигнализирующий о том, что вызывающий процесс завершил использование объекта и система может освободить любые связанные с ним ресурсы. Для закрытия объекта разделяемой памяти POSIX нужно удалить соответствующее отображение с помощью вызова `munmap()`.

IPC-объекты автоматически закрываются, когда процесс завершает свою работу или выполняет вызов `exec()`.

Права доступа к IPC-объектам

IPC-объекты имеют такую же маску с правами доступа, как и файлы, поэтому обращение к ним и к файлам ограничивается по одному и тому же принципу (см. подраздел 15.4.3). Различие только в том, что в случае с объектами POSIX IPC права на выполнение не имеют никакого смысла.

Начиная с версии 2.6.19, ядро Linux позволяет использовать списки контроля доступа для объектов разделяемой памяти и именованных семафоров POSIX. В настоящее время ACL-списки не поддерживаются для очередей сообщений POSIX.

Удаление и жизненный цикл IPC-объектов

Как и в случае с открытыми файлами, ядро ведет учет ссылок на объекты POSIX IPC. Это упрощает определение того, может ли объект быть удален безопасным образом.

Для каждого IPC-объекта предусмотрена операция `unlink`, аналогичная традиционному системному вызову `unlink()`, который применяется для файлов. Она немедленно удаляет имя объекта, а потом и сам объект, когда он перестает применяться другими процессами (то есть когда количество ссылок на него становится равным 0). В случае с очередями сообщений и семафорами это значит, что объект уничтожается после того, как его закроют все процессы; удаление объекта разделяемой памяти происходит, когда последний использовавший его процесс удалил отображение с помощью вызова `munmap()`.

После операции удаления IPC-вызов `open` с тем же именем вернет дескриптор на новый объект (или ошибку, если не был указан флаг `O_CREAT`).

Как и в System V, объекты POSIX IPC хранятся на уровне ядра. После создания они продолжают существовать, пока не будут удалены или система не будет выключена. Таким образом, если процесс создаст объект, изменит его состояние и завершится, этот объект продолжит существовать и будет доступен для других процессов, запущенных позже.

Вывод и удаление объектов POSIX IPC с помощью командной строки

Стандарт POSIX не предусматривает консольных команд для вывода и удаления IPC-объектов. Однако во многих системах, в том числе и Linux, IPC-объекты реализованы в рамках реальной или виртуальной файловой системы, подключенной где-то внутри корневого каталога (`/`), поэтому для их вывода и удаления можно использовать стандартные утилиты `ls` и `rm` (хотя стандарт SUSv3 не предусматривает такого их применения). Основной проблемой здесь является нестандартный формат имен IPC-объектов и их местоположение в рамках файловой системы.

В Linux объекты POSIX IPC находятся в виртуальных файловых системах, подключенных к каталогам, для которых установлен закрепляющий бит. Он запрещает удаление (см. подраздел 15.4.5); его наличие означает, что непривилегированный процесс может удалять только IPC-объекты, созданные им самим.

Компиляция программ, использующих POSIX IPC

В Linux программы, применяющие механизмы POSIX IPC, должны быть скомпонованы с *библиотекой реального времени* под названием `librt`. Для этого команде `cc` следует указать параметр `-lrt`.

47.2. Резюме

POSIX IPC — это собирательное название трех IPC-механизмов: очередей сообщений, семафоров и разделяемой памяти. Все они разработаны в стандарте POSIX.1b в качестве альтернативы аналогичным средствам из System V.

Интерфейс POSIX IPC больше соответствует традиционной для UNIX файловой модели. IPC-объекты идентифицируются с помощью имен и управляются вызовами `open`, `close` и `unlink`, которые похожи на аналогичные системные вызовы для работы с файлами.

Интерфейс, предоставляемый POSIX IPC, во многих отношениях превосходит IPC-интерфейс System V, хотя и уступает ему в плане портируемости на другие системы.

48 Очереди сообщений стандарта POSIX

Эта глава посвящена очередям сообщений стандарта POSIX, позволяющим процессам обмениваться данными в виде отдельных блоков (сообщений). Очереди POSIX-сообщений похожи на аналогичный механизм в System V, но имеют несколько заметных отличий:

- для очередей сообщений POSIX ведется учет ссылок. Очередь, помеченная для удаления, уничтожается только после того, как будет закрыта всеми процессами, которые ее используют;
- POSIX-сообщения имеют приоритет, обеспечивающий строгий порядок передачи (и, следовательно, получения);
- уведомление о доступности POSIX-сообщения в очереди может быть передано асинхронно.

Очереди POSIX-сообщений появились в Linux относительно недавно. Необходимая поддержка была реализована только в ядре версии 2.6.6 (кроме того, требуется библиотека `gibc` 2.3.4 или новее).

> Поддержка очередей сообщений POSIX в ядре является опциональной и настраивается с помощью параметра CONFIG_POSIX_MQUEUE.

48.1. Краткий обзор

Ниже перечислены основные функции программного интерфейса для управления очередями POSIX-сообщений:

- `mq_open()` — создает новую очередь сообщений или открывает уже существующую, возвращая соответствующий дескриптор, который можно использовать в дальнейшем;
- `mq_send()` — записывает сообщение в очередь;
- `mq_receive()` — считывает сообщение из очереди;
- `mq_close()` — закрывает очередь сообщений, открытую ранее текущим процессом;
- `mq_unlink()` — удаляет имя очереди сообщений; сама очередь удаляется после того, как будет закрыта всеми процессами.

Назначение функций, перечисленных выше, должно быть понятно из их названий. Интерфейс для работы с очередями POSIX-сообщений обладает двумя специфическими свойствами.

- Каждая очередь сообщений имеет определенный набор атрибутов. Часть из них могут быть установлены во время создания или открытия очереди с помощью вызова `mq_open()`. Для просмотра и изменения этих атрибутов предусмотрены две функции: `mq_getattr()` и `mq_setattr()`.
- Функция `mq_notify()` позволяет процессу зарегистрировать оповещение, которое будет сигнализировать о наличии того или иного сообщения в очереди. Это делается путем доставки сигнала или вызова функции в отдельном потоке.

48.2. Открытие, закрытие и удаление очереди сообщений

В данной главе мы рассмотрим функции, применяемые для открытия, закрытия и удаления очередей сообщений.

Открытие очереди сообщений

Функция `mq_open()` создает новую или открывает существующую очередь сообщений.

```
#include <fcntl.h>          /* Определяет константы вида O_* */
#include <sys/stat.h>       /* Определяет константы для аргумента mode */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

Возвращает дескриптор очереди сообщений при успешном завершении или `(mqd_t) -1`, если произошла ошибка

Аргумент `name` идентифицирует очередь сообщений и соответствует правилам, приведенным в разделе 47.1.

Аргумент `oflag` представляет собой битовую маску, управляющую различными аспектами работы функции `mq_open()`. Значения, которые можно включать в эту маску, собраны в табл. 48.1.

Таблица 48.1. Битовые значения для аргумента oflag функции mq_open()

Флаг	Описание
O_CREAT	Создает очередь, если она еще не существует
O_EXCL	В сочетании с O_CREAT создает очередь с уникальным именем
O_RDONLY	Открывает только для чтения
O_WRONLY	Открывает только для записи
O_RDWR	Открывается для чтения и записи
O_NONBLOCK	Открывает в неблокирующем режиме

Одним из назначений флага `oflag` является определение того, какая именно операция выполняется — открывается существующая очередь или создается и затем открывается новая. При отсутствии в `oflag` флага `O_CREAT` будет создана новая пустая очередь с именем `name` (при условии, что это имя еще не занято). Если в маске `oflag` одновременно указаны флаги `O_CREAT` и `O_EXCL`, а очередь с заданным именем уже существует, то функция `mq_open()` завершится неудачей.

Аргумент `oflag` также определяет тип доступа к очереди сообщений, который получит вызывающий процесс; для этого необходимо указать один из трех флагов: `O_RDONLY`, `O_WRONLY` или `O_RDWR`.

Оставшийся флаг, `O_NONBLOCK`, делает так, чтобы очередь открывалась в неблокирующем режиме. Если последующий вызов `mq_receive()` или `mq_send()` нельзя выполнить без блокировки, то он немедленно завершается ошибкой `EAGAIN`.

Если функция mq_open() используется для открытия уже существующей очереди сообщений, то ей нужно передать всего два аргумента. Но при указании флага O_CREAT еще два аргумента станут обязательными: mode и attr (если имя name уже занято, то эти два аргумента игнорируются). Они применяются следующим образом.

- Аргумент mode представляет собой битовую маску, которая определяет права доступа для новой очереди сообщений. Можно указывать те же битовые значения, что и для файлов (см. табл. 15.4; кроме того, как и в случае с вызовом open(), к маске mode применяется атрибут umask (см. подраздел 15.4.6). Чтобы читать из очереди (mq_receive()), соответствующий класс пользователей должен получить права для чтения; для передачи данных в очередь (mq_send()) требуются права на запись.
- Аргумент attr представляет собой структуру mq_attr, определяющую атрибуты новой очереди сообщений. Если он равен NULL, то очередь создается с атрибутами по умолчанию, которые действуют в текущей системе. Мы опишем структуру mq_attr в разделе 48.4.

В случае успешного выполнения функция mq_open() возвращает *дескриптор очереди сообщений* — значение типа mqd_t, которое будет применяться в последующих вызовах для обращения к этой открытой очереди. В стандарте SUSv3 есть только одно замечание относительно данного значения: оно не должно быть массивом; то есть это обязательно должен быть тип, пригодный для использования в операции присваивания или передачи по значению в качестве аргумента функции (в Linux тип mqd_t тождественен int, но в Solaris, к примеру, определен как void *).

Образец применения функции mq_open() показан в листинге 48.2.

Влияние fork(), exec() и завершения процесса на дескрипторы очередей сообщений

Во время вызова fork() дочерний процесс получает копии дескрипторов очередей сообщений, открытых его родителем, и эти дескрипторы ссылаются на те же очереди (подробней о дескрипторах очередей сообщений речь пойдет в разделе 48.3). Потомок не наследует от родителя подписки на уведомления о сообщениях.

Когда процесс выполняет вызов exec() или завершается, все дескрипторы открытых им очередей сообщений закрываются. В результате этого все подписки процесса на оповещения о поступлении сообщений в соответствующие очереди отменяются.

Закрытие очереди сообщений

Функция mq_close() закрывает дескриптор очереди сообщений mqdes.

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Если вызывающий процесс подписался на оповещения о поступлении сообщений в очередь с помощью дескриптора mqdes (см. раздел 48.6), то подписка автоматически удаляется; позже другой процесс сможет подписаться на оповещения из той же очереди.

Дескриптор очереди сообщений автоматически закрывается при завершении процесса или вызове exec(). По аналогии с файловыми, дескрипторы очередей сообщений, которые больше не нужны, следует закрывать вручную, чтобы не столкнуться с проблемой превышения ограничения на количество открытых дескрипторов.

Закрывая очередь сообщений, мы ее не удаляем (точно так же, как, вызывая `close()`, мы не удаляем сам файл). Для удаления очередей предусмотрена функция `mq_unlink()`, аналогичная вызову `unlink()`.

Удаление очереди сообщений

Функция `mq_unlink()` удаляет очередь сообщений с именем `name` и делает ее кандидатом на уничтожение, которое произойдет, когда ее перестанут использовать все процессы (это может случиться немедленно, если все процессы, открывавшие данную очередь, уже успели ее закрыть).

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Пример использования функции `mq_unlink()` показан в листинге 48.1.

Листинг 48.1. Применение функции mq_unlink() для удаления очереди POSIX-сообщений

pmsg/pmsg_unlink.c

```
#include <mqueue.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

    if (mq_unlink(argv[1]) == -1)
        errExit("mq_unlink");
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_unlink.c

48.3. Связь между дескрипторами и очередями сообщений

Данная связь аналогична отношениям между файловым дескриптором и самим открытым файлом (см. рис. 5.2). Дескриптор очереди сообщений принадлежит конкретному процессу и ссылается на запись в общесистемной таблице открытых дескрипторов этого типа; данная запись в свою очередь указывает на объект очереди. Эта связь проиллюстрирована на рис. 48.1.

В Linux очереди POSIX-сообщений реализованы в виде индексных дескрипторов (i-node) в виртуальной файловой системе; дескрипторы таких очередей представляют собой файловые дескрипторы (обычные и открытые). Однако с точки зрения стандарта SUSv3 эти аспекты реализации не являются обязательными и неактуальны в некоторых других UNIX-системах. Мы еще вернемся к данному вопросу в разделе 48.7, поскольку Linux предоставляет ряд нестандартных возможностей, основанных на особенностях именно этой реализации.

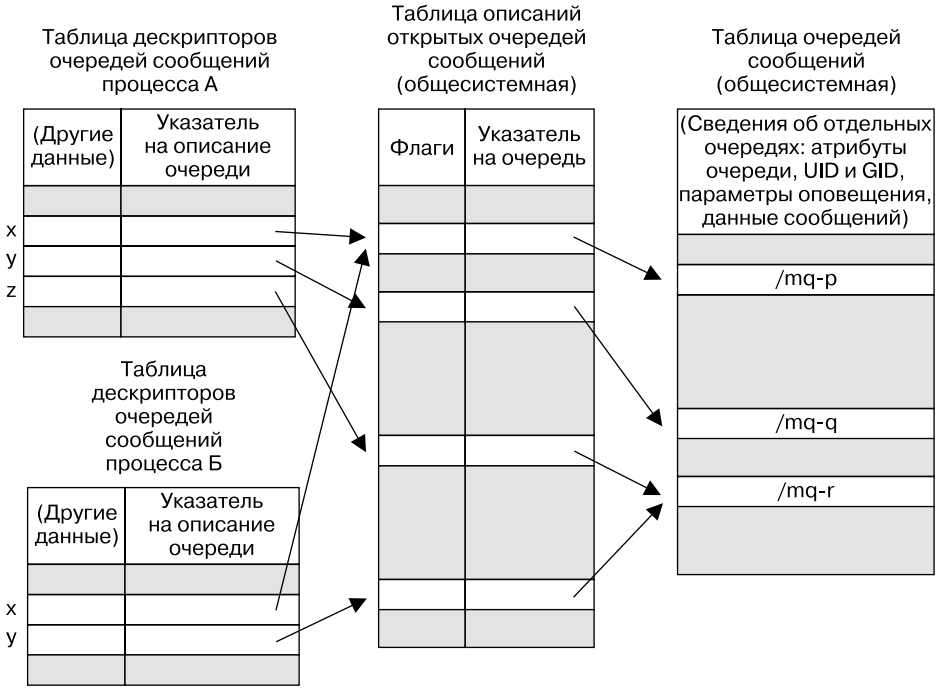


Рис. 48.1. Связь между структурами данных ядра для очередей POSIX-сообщений

Рисунок 48.1 помогает прояснить пару моментов относительно использования дескрипторов очередей сообщений (каждый из которых является аналогом файлового дескриптора).

- Описание открытой очереди сообщений содержит набор флагов. Стандарт SUSv3 предусматривает только один такой флаг, `O_NONBLOCK`, определяющий, должен ли ввод/вывод быть неблокирующим.
- Два разных процесса могут обладать дескрипторами (на диаграмме обозначены как `x`), которые ссылаются на одно и то же описание открытой очереди сообщений. Это может произойти, если процесс вызовет `fork()` после открытия очереди. Такие дескрипторы разделяют состояние флага `O_NONBLOCK`.
- Два разных процесса могут обладать открытыми дескрипторами, ссылающимися на разные описания, связанные с одной и той же очередью сообщений (например, дескриптор `z` в процессе А и дескриптор `y` в процессе Б ссылаются на `/mq-r`). Это происходит в ситуации, когда оба процесса задействуют функцию `mq_open()` для открытия одной и той же очереди.

48.4. Атрибуты очередей сообщений

Каждая из трех функций — `mq_open()`, `mq_getattr()` и `mq_setattr()` — принимает аргумент, который является указателем на структуру `mq_attr`. Она определена в заголовочном файле `<mqueue.h>` и имеет следующий вид:

```
struct mq_attr {
    long mq_flags;       /* Флаги описания очереди сообщений: 0 или
                            O_NONBLOCK [mq_getattr(), mq_setattr()] */
```

```
    long mq_maxmsg;      /* Максимальное количество сообщений в
                            очереди [mq_open(), mq_getattr()] */
    long mq_msgsize;     /* Максимальный размер сообщения (в байтах)
                            [mq_open(), mq_getattr()] */
    long mq_curmsgs;     /* Количество сообщений в очереди на текущий
                            момент [mq_getattr()] */
};
```

Прежде чем приступить к подробному рассмотрению структуры `mq_attr`, необходимо отметить следующее.

- Только часть этих полей используется всеми тремя функциями. Функции, в которых применяется каждое из полей, обозначены в комментариях к определению структуры (см. выше).
- Структура содержит сведения об описании открытой очереди сообщений (`mq_flags`), связанной с дескриптором сообщения и информацией об очереди, на которую тот ссылается (`mq_maxmsg`, `mq_msgsize`, `mq_curmsgs`).
- Часть полей содержит информацию, записанную на момент создания очереди с помощью функции `mq_open()` (`mq_maxmsg` и `mq_msgsize`); в других хранятся сведения о текущем состоянии описания открытой очереди сообщений (`mq_flags`) или о самой очереди (`mq_curmsgs`).

Установка атрибутов очереди сообщений во время ее создания

При создании очереди сообщений с помощью вызова `mq_open()` ее атрибуты определяются следующими полями структуры `mq_attr`:

- поле `mq_maxmsg` определяет максимальное количество сообщений, которое можно поместить в очередь, используя вызов `mq_send()`. Это значение должно быть больше 0;
- поле `mq_msgsize` определяет максимальный размер каждого сообщения, которое может оказаться в очереди. Это значение тоже должно быть больше 0.

Вместе оба поля позволяют ядру определить максимальный объем памяти, который может потребоваться для данной очереди сообщений.

Атрибуты `mq_maxmsg` и `mq_msgsize` инициализируются во время создания очереди и больше не могут быть изменены. В разделе 48.8 мы опишем два файла в каталоге `/proc`, устанавливающие общесистемное ограничение на значения, которые можно присвоить атрибутам `mq_maxmsg` и `mq_msgsize`.

Программа из листинга 48.2 предоставляет интерфейс командной строки к функции `mq_open()` и демонстрирует применение структуры `mq_attr`.

Чтобы задать атрибуты очереди сообщений, предусмотрено два параметра командной строки: `-m` для `mq_maxmsg` и `-s` для `mq_msgsize`. При указании любого из них функции `mq_open()` будет передан ненулевой аргумент `attrp`. Полям структуры `mq_attr`, на которую указывает этот аргумент, по умолчанию присваиваются некие значения (при условии, что был указан только один из параметров командной строки). Если ни один из параметров не указан, то функции `mq_open()` в качестве аргумента `attrp` передается значение `NULL`. В таком случае очередь сообщений создается с атрибутами, применяемыми по умолчанию в текущей системе.

Листинг 48.2. Создание очереди сообщений POSIX

pmsg/pmsg_create.c

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
```

```
static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] [-m maxmsg] [-s msgsize] mq-name "
            "[octal-perms]\n", progName);
    fprintf(stderr, "    -c          Create queue (O_CREAT)\n");
    fprintf(stderr, "    -m maxmsg   Set maximum # of messages\n");
    fprintf(stderr, "    -s msgsize  Set maximum message size\n");
    fprintf(stderr, "    -x          Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    mqd_t mqd;
    struct mq_attr attr, *attrp;

    attrp = NULL;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 2048;
    flags = O_RDWR;

    /* Разбираем параметры командной строки */

    while ((opt = getopt(argc, argv, "cm:s:x")) != -1) {
        switch (opt) {
    case 'c':
            flags |= O_CREAT;
            break;

        case 'm':
            attr.mq_maxmsg = atoi(optarg);
            attrp = &attr;
            break;

        case 's':
            attr.mq_msgsize = atoi(optarg);
            attrp = &attr;
            break;

        case 'x':
            flags |= O_EXCL;
            break;

        default:
            usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");
```

```
    mqd = mq_open(argv[optind], flags, perms, attrp);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_create.c

Извлечение атрибутов очереди сообщений

Функция `mq_getattr()` возвращает структуру `mq_attr`, содержащую сведения об атрибутах очереди сообщений и саму очередь, связанную с дескриптором `mqdes`.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Помимо полей `mq_maxmsg` и `mq_msgsize`, уже описанных выше, возвращенная структура, на которую указывает аргумент `attr`, содержит следующие поля.

- `mq_flags` — это флаги для описания открытой очереди сообщений, связанной с дескриптором `mqdes`. Допускается только один флаг: `O_NONBLOCK`. Он инициализируется путем аргумента `oflag` в вызове `mq_open()` и может быть изменен с помощью функции `mq_setattr()`.
- `mq_curmsgs` — это количество сообщений, находящихся в очереди на текущий момент. За время с начала вызова `mq_getattr()` и до его завершения какой-то другой процесс может изменить данный атрибут, прочитав сообщение из очереди или записав его туда.

Программа, представленная в листинге 48.3, использует функцию `mq_getattr()`, чтобы извлечь атрибуты очереди сообщений, заданные в качестве аргументов командной строки, и затем направляет их в стандартный вывод.

Листинг 48.3. Извлечение атрибутов очереди сообщений POSIX

pmsg/pmsg_getattr.c

```
#include <mqueue.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    mqd_t mqd;
    struct mq_attr attr;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

    mqd = mq_open(argv[1], O_RDONLY);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    printf("Maximum # of messages on queue:   %ld\n", attr.mq_maxmsg);
    printf("Maximum message size:             %ld\n", attr.mq_msgsize);
```

```
    printf("# of messages currently on queue: %ld\n", attr.mq_curmsgs);
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_getattr.c

В следующей сессии командной строки мы воспользуемся программой из листинга 48.2 для создания очереди сообщений с атрибутами, которые действуют по умолчанию в текущей системе (то есть передадим `NULL` в качестве последнего аргумента функции `mq_open()`), а затем выведем эти атрибуты с помощью программы из листинга 48.3. Таким образом мы покажем, какие параметры применяются по умолчанию в Linux.

```
$ ./pmsg_create -cx /mq
$ ./pmsg_getattr /mq
Maximum # of messages on queue: 10
Maximum message size: 8192
# of messages currently on queue: 0
$ ./pmsg_unlink /mq                          Удаляем очередь сообщений
```

По выводу, представленному выше, можно понять, что в Linux стандартными значениями для атрибутов `mq_maxmsg` и `mq_msgsize` являются числа 10 и 8192 соответственно.

Эти значения могут сильно варьироваться в зависимости от реализации. Портируемые приложения должны явно указывать данные атрибуты, не полагаясь на стандартные значения.

Изменение атрибутов очереди сообщений

Функция `mq_setattr()` устанавливает атрибуты очереди сообщений, связанной с дескриптором `mqdes`, и в случае необходимости возвращает сведения о самой очереди.

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `mq_setattr()` выполняет следующие действия:

- использует поле `mq_flags` в структуре `mq_attr`, на которую указывает аргумент `newattr`, чтобы изменить флаги описания очереди сообщений, связанной с дескриптором `mqdes`;
- если аргумент `oldattr` не равен `NULL`, то функция возвращает структуру `mq_attr`, содержащую предыдущие флаги описания очереди сообщений и ее атрибуты (то есть дублирует работу функции `mq_getattr()`).

Согласно стандарту SUSv3 функция `mq_setattr()` может изменить только состояние флага `O_NONBLOCK`.

Разные системы могут предоставлять другие флаги, доступные для изменения, и, возможно, в будущем в стандарте SUSv3 будут добавлены новые флаги. Учитывая все это, портируемые приложения должны изменять состояние флага `O_NONBLOCK` в три этапа: сначала извлечь значение `mq_flags` с помощью функции `mq_getattr()`, модифицировать бит `O_NONBLOCK` и затем вызвать `mq_setattr()` для установки новых параметров `mq_flags`. Например, чтобы включить `O_NONBLOCK`, нужно сделать следующее:

```
if (mq_getattr(mqd, &attr) == -1)
    errExit("mq_getattr");
```

```
attr.mq_flags |= O_NONBLOCK;
if (mq_setattr(mqd, &attr, NULL) == -1)
    errExit("mq_getattr");
```

48.5. Обмен сообщениями

В этом разделе мы уделим внимание функциям, применяемым для отправки сообщений в очередь и получения их оттуда.

48.5.1. Отправка сообщений

Функция `mq_send()` добавляет сообщение из буфера `msg_ptr` в очередь сообщений, на которую ссылается дескриптор `mqdes`.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `msg_len` обозначает длину сообщения, на которое указывает `msg_ptr`. Это значение не должно превышать атрибут `mq_msgsize` очереди; в противном случае функция `mq_send()` завершится ошибкой `EMSGSIZE`. Сообщения нулевой длины являются допустимыми.

Каждое сообщение имеет приоритет (целое положительное число), указанный в аргументе `msg_prio`. Сообщения в очереди размещаются в порядке убывания приоритета (наименьшим является 0). Когда в очередь добавляется новое сообщение, оно занимает место сразу за всеми сообщениями одного с ним приоритета. Если приложение не нуждается в использовании приоритетов, аргументу `msg_prio` можно всегда передавать значение 0.

Стандарт SUSv3 позволяет реализации определить максимально возможный приоритет; это делается либо с помощью константы `MQ_PRIO_MAX`, либо через значение, возвращаемое вызовом `sysconf(_SC_MQ_PRIO_MAX)`. Стандарт SUSv3 требует, чтобы данное ограничение не было меньше 32 (`_POSIX_MQ_PRIO_MAX`); то есть доступны приоритеты в диапазоне как минимум от 0 до 31. Но реальный диапазон может заметно варьироваться. Например, в Linux эта константа равна 32 768, в Solaris — 32, а в Tru64 — 256.

При заполненной очереди сообщений (то есть достигнутом ограничении `mq_maxmsg`) дальнейшие вызовы `mq_send()` будут либо блокироваться, пока в очереди не освободится место, либо немедленно завершаться ошибкой `EAGAIN`, если был установлен флаг `O_NONBLOCK`.

Программа, представленная в листинге 48.4, является интерфейсом командной строки к функции `mq_send()`. Ее использование будет продемонстрировано в следующем разделе.

Листинг 48.4. Запись сообщения в очередь сообщений POSIX

pmsg/pmsg_send.c

```
#include <mqueue.h>
#include <fcntl.h>              /* Для определения O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
```

```
{
    fprintf(stderr, "Usage: %s [-n] mq-name msg [prio]\n", progName);
    fprintf(stderr, "    -n              Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;

    flags = O_WRONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind + 1 >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    prio = (argc > optind + 2) ? atoi(argv[optind + 2]) : 0;

    if (mq_send(mqd, argv[optind + 1],
        strlen(argv[optind + 1]), prio) == -1)
        errExit("mq_send");
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_send.c

48.5.2. Получение сообщений

Функция `mq_receive()` удаляет из очереди, куда ссылается дескриптор `mqdes`, самое старое сообщение с наивысшим приоритетом, после чего возвращает это сообщение в буфер, на который указывает аргумент `msg_ptr`.

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
```

При успешном завершении возвращает количество байтов в полученном сообщении; при ошибке возвращает `-1`

Вызывающий процесс использует аргумент `msg_len`, чтобы сообщить о том, сколько байтов доступно в буфере, на который указывает `msg_ptr`.

Вне зависимости от реального размера сообщения аргумент `msg_len` (и, как следствие, размер буфера, на который указывает `msg_ptr`) должен быть больше или равен атрибуту

очереди mq_msgsize; в противном случае mq_receive() завершится ошибкой EMSGSIZE. Если мы не знаем значение данного атрибута, то можем его получить с помощью функции mq_getattr() (в приложении, состоящем из взаимодействующих процессов, обычно можно обойтись и без этой функции, поскольку параметр mq_msgsize, как правило, устанавливается заранее).

Если аргумент msg_prio не равен NULL, то приоритет полученного сообщения копируется по адресу, на который указывает msg_prio.

При пустой очереди функция mq_receive() либо блокируется, пока не станет доступным хотя бы одно сообщение, либо сразу же завершается ошибкой EAGAIN, если был установлен флаг O_NONBLOCK (мы не увидим символ конца файла при отсутствии записывающих процессов, как это происходит при использовании каналов).

Программа из листинге 48.5, предоставляет интерфейс командной строки к функции mq_receive(). Формат команд для этой программы представлен в функции usageError().

Следующая сессия командной строки демонстрирует использование программ из листингов 48.4 и 48.5. Вначале создадим очередь и отправим в нее несколько сообщений с разными приоритетами:

```
$ ./pmsg_create -cx /mq
$ ./pmsg_send /mq msg-a 5
$ ./pmsg_send /mq msg-b 0
$ ./pmsg_send /mq msg-c 10
```

Затем последовательно выполним несколько команд, чтобы извлечь эти сообщения из очереди:

```
$ ./pmsg_receive /mq
Read 5 bytes; priority = 10
msg-c
$ ./pmsg_receive /mq
Read 5 bytes; priority = 5
msg-a
$ ./pmsg_receive /mq
Read 5 bytes; priority = 0
msg-b
```

Как видно из вывода, приведенного выше, сообщения были извлечены в порядке их приоритета.

На этот момент очередь является пустой. Если выполнить еще одно блокирующее извлечение, то операция будет приостановлена:

```
$ ./pmsg_receive /mq
Блокируется; чтобы завершить программу, нажимаем Ctrl+C
```

С другой стороны, если выполнить неблокирующее извлечение, то вызов немедленно вернет сообщение об ошибке:

```
$ ./pmsg_receive -n /mq
ERROR [EAGAIN/EWOULDBLOCK Resource temporarily unavailable] mq_receive
```

Листинг 48.5. Чтение сообщений из очереди POSIX

pmsg/pmsg_receive.c

```
#include <mqueue.h>
#include <fcntl.h>              /* Для определения O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
```

```
{
    fprintf(stderr, "Usage: %s [-n] mq-name\n", progName);
    fprintf(stderr, "    -n              Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;
    void *buffer;
    struct mq_attr attr;
    ssize_t numRead;

    flags = O_RDONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

    numRead = mq_receive(mqd, buffer, attr.mq_msgsize, &prio);
    if (numRead == -1)
        errExit("mq_receive");

    printf("Read %ld bytes; priority = %u\n", (long) numRead, prio);
    if (write(STDOUT_FILENO, buffer, numRead) == -1)
        errExit("write");
    write(STDOUT_FILENO, "\n", 1);
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_receive.c

48.5.3. Отправка и получение сообщений с ограниченным временем ожидания

Функции `mq_timedsend()` и `mq_timedreceive()` являются прямыми аналогами `mq_send()` и `mq_receive()`, но с одним отличием: если операция не может быть выполнена немедленно и если среди атрибутов сообщения не указан флаг `O_NONBLOCK`, аргумент `abs_timeout` определяет продолжительность блокирования вызова.

```
#define _XOPEN_SOURCE 600
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned int msg_prio, const struct timespec *abs_timeout);
```

Возвращает 0 при успешном завершении или -1 при ошибке

```
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                        unsigned int *msg_prio,
                        const struct timespec *abs_timeout);
```

Возвращает объем полученного сообщения (в байтах) или -1 при ошибке

Аргумент abs_timeout представляет собой структуру timespec (см. подраздел 23.4.2), которая задает время ожидания в виде количества секунд и наносекунд, прошедших с начала «эры UNIX». Для указания относительного времени ожидания можно воспользоваться функцией clock_gettime(), чтобы извлечь текущее значение часов CLOCK_REALTIME и затем добавить к нему необходимый период времени, получив тем самым структуру timespec, инициализированную должным образом.

Если вызов функции mq_timedsend() или mq_timedreceive() не завершается вовремя, то он возвращает ошибку ETIMEDOUT. Если в Linux аргументу abs_timeout присвоить значение NULL, то время ожидания будет неограниченным. Однако такая возможность не предусмотрена стандартом SUSv3, и портируемые приложения не могут на нее полагаться.

Функции mq_timedsend() и mq_timedreceive() изначально появились в стандарте POSIX.1d (1999) и доступны не во всех реализациях UNIX.

48.6. Оповещение о сообщении

Особенность, которая отличает очереди сообщений POSIX от их аналогов из System V, состоит в возможности получать асинхронные оповещения о появлении в ранее пустой очереди нового сообщения (то есть очередь перестает быть пустой). Это значит, что вместо выполнения блокирующего вызова mq_receive() или маркировки дескриптора очереди сообщений как неблокирующего с последующими регулярными «опросами» очереди процесс может подписаться на оповещения о приходе сообщений и заняться чем-нибудь другим. Оповещение можно получать либо в виде сигнала, либо путем вызова функции в отдельном потоке.

Оповещения, предоставляемые очередями сообщений POSIX, похожи на механизм оповещений для POSIX-таймеров, описанный в разделе 23.6 (оба этих программных интерфейса изначально появились в стандарте POSIX.1b).

Функция mq_notify() подписывает вызывающий процесс на оповещения, передающиеся ему при появлении сообщения в пустой очереди, на которую ссылается дескриптор mqdes.

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Аргумент `notification` определяет, как именно процесс будет уведомлен. Прежде чем переходить к подробностям, стоит сделать несколько замечаний, касающихся механизма оповещения.

- Только один процесс может быть подписан на получение оповещений из определенной очереди. Если на них уже подписан другой процесс, то дальнейшие попытки подписаться завершаться неудачно (вызов `mq_notify()` вернет ошибку `EBUSY`).
- Зарегистрированный процесс оповещается только о появлении сообщения в ранее пустой очереди. Если на момент регистрации очередь уже содержит сообщения, то оповещения начнут приходить лишь после того, как она опустеет и получит новое сообщение.
- После отправки оповещения подписка процесса аннулируется, в результате чего любой другой процесс может подписаться на данную очередь. Иными словами, если процесс желает и дальше получать оповещения, то должен каждый раз заново выполнять подписку, используя вызов `mq_notify()`.
- Зарегистрированный процесс оповещается только в том случае, если нет никакого другого процесса, заблокированного обращением к очереди с помощью вызова `mq_receive()`. При наличии такого процесса он сначала должен будет прочитать сообщение, а подписчик останется зарегистрированным.
- Процесс может вручную отменить подписку на оповещения из заданной очереди, вызвав функцию `mq_notify()` со значением `NULL` в качестве аргумента `notification`.

В подразделе 23.6.1 вы уже познакомились со структурой `sigevent`, которая используется в качестве типа аргумента `notification`. Здесь она представлена в упрощенном виде, только с теми полями, которые имеют отношение к функции `mq_notify()`:

```
union sigval {
    int sival_int;              /* Вспомогательное целочисленное значение */
    void *sival_ptr;            /* Указатель на данные оповещения */
};

struct sigevent {
    int sigev_notify;           /* Способ оповещения */
    int sigev_signo;            /* Сигнал оповещения для SIGEV_SIGNAL */
    union sigval sigev_value;   /* Значение, передающееся в обработчик
                                   сигнала или функцию в отдельном потоке */
    void (*sigev_notify_function) (union sigval);
    /* Функция, доставляющая оповещение в отдельном потоке */
    void *sigev_notify_attributes;  /* Указатель на 'pthread_attr_t' */
};
```

Поле `sigev_notify` этой структуры принимает одно из следующих значений.

- `SIGEV_NONE` — подписывает процесс на оповещения, но при появлении сообщения в ранее пустой очереди не уведомляет о данном факте. И, как обычно, после этого подписка отменяется.
- `SIGEV_SIGNAL` — оповещает процесс, генерируя сигнал, указанный в поле `sigev_signo`. Если это сигнал реального времени, то данные, передающиеся вместе с ним, указываются в поле `sigev_value` (см. подраздел 22.8.1). Их можно извлечь из поля `si_value` структуры `siginfo_t`, которая передается в обработчик сигнала или возвращается вызовами `sigwaitinfo()` или `sigtimedwait()`. В структуре `siginfo_t` также заполняются следующие поля: `si_code` (значение `SI_MESGQ`), `si_signo` (номер сигнала), `si_pid` (идентификатор процесса, пославшего сообщение) и `si_uid` (реальный ID пользователя, от имени которого было послано сообщение). В ряде реализаций поля `si_pid` и `si_uid` не устанавливаются.

- SIGEV_THREAD — оповещает процесс путем вызова в отдельном потоке функции, указанной в поле sigev_notify_function. Полю sigev_notify_attributes можно присвоить либо NULL, либо указатель на структуру pthread_attr_t, описывающую атрибуты нового потока (см. раздел 29.8). Объединение sigval, указанное в поле sigev_value, передается в качестве аргумента данной функции.

48.6.1. Получение оповещения в виде сигнала

В листинге 48.6 приводится пример оповещения, основанного на сигнале. Данная программа выполняет следующие шаги.

1. Открывает в неблокирующем режиме очередь сообщений, имя которой указано в командной строке (1), определяет значение атрибута mq_msgsize, подходящее для данной очереди (2), и выделяет буфер соответствующего размера, предназначенный для получения сообщения (3).
2. Блокирует сигнал-оповещение (SIGUSR1) и устанавливает для него обработчик (4).
3. Выполняет начальный вызов mq_notify(), чтобы подписать процесс на получение оповещений (5).
4. Входит в бесконечный цикл и в нем выполняет следующие действия:
   - делает вызов sigsuspend(), который разблокирует сигнал-оповещение и ждет, пока он не будет перехвачен (6). Окончание данного вызова свидетельствует о появлении оповещения. На этом этапе подписка на оповещения процесса уже является недействительной;
   - вызывает функцию mq_notify(), чтобы снова подписать данный процесс на оповещения (7);
   - выполняет цикл while, который опустошает очередь, пытаясь прочитать из нее как можно больше сообщений (8).

Листинг 48.6. Получение оповещения в виде сигнала

pmsg/mq_notify_sig.c

```
#include <signal.h>
#include <mqueue.h>
#include <fcntl.h>                /* Для определения O_NONBLOCK */
#include "tlpi_hdr.h"

#define NOTIFY_SIG SIGUSR1

static void
handler(int sig)
{
    /* Просто прерываем вызов sigsuspend() */
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    mqd_t mqd;
    struct mq_attr attr;
    void *buffer;
    ssize_t numRead;
    sigset_t blockMask, emptyMask;
    struct sigaction sa;
```

```
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

❶   mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

❷   if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

❸   buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

❹   sigemptyset(&blockMask);
    sigaddset(&blockMask, NOTIFY_SIG);
    if (sigprocmask(SIG_BLOCK, &blockMask, NULL) == -1)
        errExit("sigprocmask");
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(NOTIFY_SIG, &sa, NULL) == -1)
        errExit("sigaction");

❺   sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = NOTIFY_SIG;
    if (mq_notify(mqd, &sev) == -1)
        errExit("mq_notify");

    sigemptyset(&emptyMask);

    for (;;) {
❻       sigsuspend(&emptyMask);         /* Ждем сигнала-оповещения */

❼       if (mq_notify(mqd, &sev) == -1)
            errExit("mq_notify");

❽       while ((numRead = mq_receive(mqd, buffer, attr.mq_msgsize, NULL)) >= 0)
            printf("Read %ld bytes\n", (long) numRead);

        if (errno != EAGAIN)            /* Непредвиденная ошибка */
            errExit("mq_receive");
    }
}
```

pmsg/mq_notify_sig.c

Ряд аспектов программы из листинга 48.6 заслуживает дополнительного внимания.

- Вместо использования вызова `pause()` мы блокируем сигнал и ждем его с помощью вызова `sigsuspend()`; так можно исключить возможность потери сигнала, который доставляется, пока программа занимается другими делами (то есть когда она не заблокирована в ожидании сигнала) в цикле `for`. Если бы мы задействовали для этого функцию `pause()`, то следующий ее вызов был бы заблокирован, несмотря на уже доставленный сигнал.
- Очередь открывается в неблокирующем режиме; при поступлении оповещения мы применяем цикл `while` для прочтения сообщения из очереди. Опустошение очереди таким способом гарантирует, что процесс будет оповещен при появлении нового сообщения. Использование неблокирующего режима означает, что при опустошении очереди цикл `while` завершится неудачей (вызов `mq_receive()` вернет ошибку `EAGAIN`).

Такой подход аналогичен применению неблокирующего ввода/вывода в сочетании с оповещениями, срабатывающими при готовности (о чем мы поговорим в подразделе 59.1.1), и применяется по схожим причинам.

- Важно, чтобы в цикле `for` подписка на оповещение произошла *до* того, как из очереди будут прочитаны все сообщения. В противном случае может произойти следующая цепочка событий: все сообщения в очереди прочитаны и цикл `while` завершается; новое сообщение появляется в очереди; вызывается метод `mq_notify()`, который создает подписку на соответствующее оповещение. В результате не будет сгенерировано никакого дополнительного сигнала, поскольку очередь ни разу не была пустой. Следовательно, при следующем вызове `sigsuspend()` программа продолжит оставаться заблокированной.

48.6.2. Получение уведомлений в отдельном потоке

В листинге 48.7 показан пример оповещения с помощью потоков. В этой программе использован целый ряд архитектурных решений, которые применяются в листинге 48.6:

- при появлении оповещения программа, прежде чем опустошить очередь, возобновляет свою подписку (2);
- применяется неблокирующий режим; это позволяет полностью опустошить очередь после получения оповещения, не блокируя программу (5).

Листинг 48.7. Получение оповещений с помощью потока

pmsg/mq_notify_thread.c

```
#include <pthread.h>
#include <mqueue.h>
#include <fcntl.h>              /* Для определения O_NONBLOCK */
#include "tlpi_hdr.h"

static void notifySetup(mqd_t *mqdp);

static void                     /* Функция оповещения потока */
(1) threadFunc(union sigval sv)
{
    ssize_t numRead;
    mqd_t *mqdp;
    void *buffer;
    struct mq_attr attr;

    mqdp = sv.sival_ptr;

    if (mq_getattr(*mqdp, &attr) == -1)
        errExit("mq_getattr");
    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

(2) notifySetup(mqdp);

    while ((numRead = mq_receive(*mqdp, buffer, attr.mq_msgsize,
            NULL)) >= 0)
        printf("Read %ld bytes\n", (long) numRead);

    if (errno != EAGAIN)              /* Непредвиденная ошибка */
        errExit("mq_receive");
    free(buffer);
```

```
    pthread_exit(NULL);
}

static void
notifySetup(mqd_t *mqdp)
{
    struct sigevent sev;
③   sev.sigev_notify = SIGEV_THREAD;     /* Оповещаем через поток */
    sev.sigev_notify_function = threadFunc;
    sev.sigev_notify_attributes = NULL;
            /* Может быть указателем на структуру pthread_attr_t */
④   sev.sigev_value.sival_ptr = mqdp;   /* Аргумент функции threadFunc() */

    if (mq_notify(*mqdp, &sev) == -1)
        errExit("mq_notify");
}

int
main(int argc, char *argv[])
{
    mqd_t mqd;
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

⑤   mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

⑥   notifySetup(&mqd);
    pause();                        /* Ждем оповещений через функцию потока */
}
```

pmsg/mq_notify_thread.c

Стоит отметить следующие архитектурные особенности программы из листинга 48.7.

- Программа запрашивает оповещения через поток, указав значение `SIGEV_THREAD` в поле `sigev_notify` структуры `sigevent`, которая передается в функцию `mq_notify()`. Начальная функция потока, `threadFunc()`, указана в поле `sigev_notify_function` ③.
- Включив оповещения, главная программа приостанавливается на неопределенное время; оповещения передаются по таймеру путем вызова в отдельном потоке функции `threadFunc()` ①.
- Дескриптор очереди сообщений, `mqd`, можно было бы сделать видимым для функции `threadFunc()`, создав для него глобальную переменную. Но, чтобы проиллюстрировать альтернативу, мы выбрали другой подход: адрес дескриптора очереди помещается в поле `sigev_value.sival_ptr`, которое передается вызову `mq_notify()` ④. Позже, при вызове функции `hreadFunc()`, данный адрес будет передан ей в качестве аргумента.

  В поле sigev_value.sival_ptr следует хранить не сам дескриптор очереди сообщений (или его версию с приведенным типом), а указатель на него, так как данное значение не является массивом. Кроме того, стандарт SUSv3 не предусматривает никаких гарантий относительно происхождения или размера структуры данных, которая представляет тип mqd_t.

48.7. Возможности, характерные для Linux

Очереди POSIX-сообщений, реализованные в Linux, предоставляют ряд нестандартных, но полезных возможностей.

Отображение и удаление объектов очереди сообщений в командной строке

В главе 47 мы упоминали: объекты POSIX IPC реализованы в виде файлов, которые хранятся в виртуальной файловой системе, и их можно выводить и удалять с помощью команд `ls` и `rm`. Чтобы проделать это с очередями сообщений POSIX, нужно подключить соответствующую файловую систему, используя команду следующего вида:

```
# mount -t mqueue source target
```

Параметр `source` может быть каким угодно (обычно на его месте указывают строку `none`). Он ни на что не влияет, а всего лишь отображается командами `mount` и `df`, которые работают с файлом `/proc/mounts`. Параметр `target` — точка подключения файловой системы очереди сообщений. Процесс подключения файловой системы очереди сообщений и вывода ее содержимого показан в следующей сессии командной строки. Для начала создадим каталог и подключим к ней файловую систему:

```
$ su                        Для подключения требуются повышенные привилегии
Password:
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue
$ exit                      Завершаем сессию администратора
```

Теперь выведем запись файла `/proc/mounts`, которая касается нового подключения, а также права доступа к точке подключения:

```
$ cat /proc/mounts | grep mqueue
none /dev/mqueue mqueue rw 0 0
$ ls -ld /dev/mqueue
drwxrwxrwt 2 root root 40 Jul 26 12:09 /dev/mqueue
```

В выводе команды `ls` стоит обратить внимание на следующий аспект: при подключении файловой системы очереди сообщений выбранного каталога автоматически устанавливается закрепляющий бит (на это указывает буква `t` в поле с правами доступа). Это значит, что непривилегированный процесс может отключить только ту очередь, которой он владеет.

В завершение создадим очередь сообщений, выведем ее имя с помощью команды `ls` и удалим ее:

```
$ ./pmsg_create -c /newq
$ ls /dev/mqueue
newq
$ rm /dev/mqueue/newq
```

Получение сведений об очереди сообщений

Можно вывести содержимое объектов в файловой системе очереди сообщений. Каждый из таких объектов (виртуальных файлов) содержит информацию о соответствующей очереди:

```
$ ./pmsg_create -c /mq              Создаем очередь
$ ./pmsg_send /mq abcdefg           Записываем в очередь 7 байт
$ cat /dev/mqueue/mq
QSIZE:7         NOTIFY:0            SIGNO:0         NOTIFY_PID:0
```

Поле `QSIZE` обозначает общее количество байтов данных в очереди. Оставшиеся поля относятся к оповещениям. Если значение `NOTIFY_PID` не равно 0, то процесс с указанным идентификатором подписался на оповещения для этой очереди; сведения о типе оповещений хранятся в следующих полях:

- `NOTIFY` — значение, связанное с одной из констант `sigev_notify`: `0` (`SIGEV_SIGNAL`), `1` (`SIGEV_NONE`) или `2` (`SIGEV_THREAD`);

- при выборе типа оповещения SIGEV_SIGNAL поле SIGNO определяет, какой именно сигнал будет доставлен.

Информация, хранящаяся в этих полях, проиллюстрирована в следующей сессии командной строки:

```
$ ./mq_notify_sig /mq &          Оповещение с помощью сигнала SIGUSR1
                                 (номер 10 на платформе x86)
$ cat /dev/mqueue/mq
QSIZE:7      NOTIFY:0     SIGNO:10    NOTIFY_PID:18158
$ kill %1
[1]   Terminated    ./mq_notify_sig /mq
$ ./mq_notify_thread /mq &       Оповещение на основе отдельного потока
[2] 18160
$ cat /dev/mqueue/mq
QSIZE:7      NOTIFY:2     SIGNO:0     NOTIFY_PID:18160
```

Использование очередей сообщений с альтернативными моделями ввода/вывода

В Linux дескриптор очереди сообщений на самом деле является файловым дескриптором. Для его отслеживания можно задействовать мультиплексирующие системные вызовы ввода/вывода (select() и poll()) или программный интерфейс epoll (подробности о нем см. в главе 59). Это позволяет избежать трудностей, с которыми можно столкнуться при работе с очередями сообщений в System V, когда программа ожидает ввода как в очереди, так и в файловом дескрипторе. Однако данная возможность является нестандартной; спецификация SUSv3 не требует, чтобы дескрипторы очередей сообщений были реализованы в виде файловых дескрипторов.

48.8. Ограничения, относящиеся к очередям сообщений

В стандарте SUSv3 предусмотрено два ограничения, относящихся к очередям сообщений:

- MQ_PRIO_MAX — это ограничение определяет максимально возможный приоритет сообщения. Мы рассматривали его в подразделе 48.5.1;
- MQ_OPEN_MAX — система может установить данное ограничение для определения максимального количества очередей сообщений, доступных для открытия каждому отдельному процессу. Стандарт SUSv3 требует, чтобы это ограничение было равно как минимум _POSIX_MQ_OPEN_MAX (8). В Linux такое ограничение не определено. Вместо него применяется ограничение относительно количества файловых дескрипторов, которые, как вы уже знаете, представляют дескрипторы очередей сообщений (см. раздел 48.7). Иными словами, в Linux общесистемное ограничение и ограничение для отдельных процессов обозначают суммарное количество дескрипторов файлов и очередей сообщений. Больше подробностей о соответствующих ограничениях приводится в разделе 36.3, посвященном ограничению на ресурсы RLIMIT_NOFILE.

Помимо вышеупомянутых ограничений, описанных в стандарте SUSv3, Linux предоставляет несколько файлов внутри /proc, предназначенных для просмотра и (при наличии подходящих привилегий) изменения ограничений, относящихся к использованию очередей сообщений POSIX. Следующие три файла хранятся в каталоге /proc/sys/fs/mqueue.

- msg_max — определяет максимальное значение атрибута mq_maxmsg для новых очередей сообщений (то есть максимум для поля attr.mq_maxmsg, которое применяется при создании очереди с помощью mq_open()). Значение по умолчанию для данного ограничения — 10. Минимальное равно 1 (в ядрах Linux до версии 2.6.28 оно равнялось 10), а максимальное

определяется константой HARD_MSGMAX. Ее значение вычисляется по формуле (131,072 / sizeof(void *)) и на платформе Linux/x86-32 равно 32 768. Когда привилегированный процесс (CAP_SYS_RESOURCE) вызывает mq_open(), ограничение msg_max игнорируется, но значение HARD_MSGMAX по-прежнему остается максимальным для поля attr.mq_maxmsg.

- msgsize_max — определяет максимальное значение атрибута mq_msgsize для новых очередей сообщений, созданных непривилегированными процессами (то есть максимум для поля attr.mq_msgsize, которое используется при создании очереди с помощью mq_open()). Значение по умолчанию для данного ограничения — 8192. Минимальное равно 128 (или 8192 в ядрах Linux до версии 2.6.28), а максимальное — 1 048 576 (или INT_MAX в ядрах Linux до версии 2.6.28). Если функция mq_open() вызывается привилегированным процессом (CAP_SYS_RESOURCE), то данное ограничение игнорируется.
- queues_max — это общесистемное ограничение на количество очередей сообщений, которые можно создать. После достижения данного ограничения очереди сообщений могут создаваться только привилегированными процессами (CAP_SYS_RESOURCE). По умолчанию ограничение равно 256. Его можно изменять в пределах от 0 до INT_MAX.

Linux также представляет ограничение на ресурсы RLIMIT_MSGQUEUE, с помощью которого можно установить максимальный объем памяти, потребляемой всеми очередями сообщений, принадлежащими пользователю с заданным реальным идентификатором (см. раздел 36.3).

48.9. Резюме

Очереди сообщений POSIX позволяют процессам обмениваться отдельными блоками данных (сообщениями). Каждое сообщение обладает целочисленным приоритетом, определяющим его место в очереди (и порядок, в котором оно будет получено).

Дополнительная информация

В [Stevens, 1999] приводится альтернативный взгляд на очереди сообщений POSIX и описывается их реализация в пользовательском пространстве на основе файлов, отображенных в память. Отдельные подробности об очередях сообщений POSIX также доступны в [Gallmeister, 1995].

48.10. Упражнения

48.1. Измените программу из листинга 48.5 (pmsg_receive.c) так, чтобы она принимала время ожидания в качестве аргумента командной строки и задействовала функцию mq_timedreceive() вместо mq_receive().

48.2. Перепишите клиент-серверное приложение для обмена числовыми последовательностями из раздела 44.8, используя очереди POSIX-сообщений.

48.3. Напишите простую программу для обмена сообщениями (похожую на talk(1), но без интерфейса на основе curses), применяя очереди сообщений POSIX.

48.4. Измените программу из листинга 48.6 (mq_notify_sig.c) для демонстрации того, что оповещение о поступлении сообщений (подписка на которое оформляется с помощью вызова mq_notify()) приходит только один раз.

48.5. Замените обработчик сигнала в листинге 48.6 (mq_notify_sig.c) на вызов sigwaitinfo(). После получения структуры siginfo_t выведите ее содержимое на экран. Каким образом программа могла бы получить дескриптор очереди сообщений из этой структуры, возвращенной вызовом sigwaitinfo()?

48.6. Взгляните на листинг 48.7. Могли бы вы сделать buffer глобальной переменной и выделить для нее память лишь один раз в главной программе? Обоснуйте свой ответ.

49 Семафоры стандарта POSIX

Данная глава посвящена POSIX-семафорам, которые позволяют синхронизировать доступ к общим ресурсам между разными процессами и потоками выполнения. Предполагается, что читатель уже знаком с общей идеей семафоров и их назначением.

49.1. Краткий обзор

Стандартом SUSv3 предусмотрено два типа POSIX-семафоров.

- *Именованные семафоры* — имеют имя. Неродственные процессы могут получить доступ к одному семафору, вызвав функцию `sem_open()` с одним и тем же именем.
- *Анонимные семафоры* — не имеют имени; вместо этого помещаются в заранее оговоренный участок памяти. Семафоры данного типа доступны для совместного использования процессами или группой потоков. В первом случае они должны находиться в разделяемой памяти, а во втором — в пространстве, общем для потоков (например, в куче или в глобальной переменной).

POSIX-семафор представляет собой целое число, которое не может опускаться ниже нуля. Если процесс попытается сделать семафор отрицательным, то применяемый для этого вызов либо заблокируется, либо завершится ошибкой, сигнализирующей о недопустимости данной операции.

Ряд систем не предоставляет полной реализации POSIX-семафоров. Часто поддерживаются только анонимные семафоры, которые можно разделять между потоками. Именно так было в Linux 2.4; но с выходом ядра версии 2.6 и добавлением поддержки NPTL в glibc Linux поддерживает полную реализацию POSIX-семафоров.

> В случае с Linux 2.6 и NPTL-потоками работа с семафорами (инкрементация и декрементация) реализована в виде системного вызова `futex(2)`.

49.2. Именованные семафоры

Для работы с именованными семафорами предусмотрены следующие функции:

- `sem_open()` — открывает или создает новый семафор, инициализирует его (если она его создала) и возвращает дескриптор, который можно использовать в дальнейшем;
- `sem_post(sem)` и `sem_wait(sem)` — соответственно инкрементируют и декрементируют значение семафора;
- `sem_getvalue()` — возвращает текущее значение семафора;
- `sem_close()` — закрывает семафор, ранее открытый вызывающим процессом;
- `sem_unlink()` — удаляет имя семафора и делает его кандидатом на удаление; само удаление произойдет, когда семафор закроет все процессы.

Стандарт SUSv3 не уточняет, как именно должно быть реализовано именование семафоров. В некоторых системах они представляют собой обычные файлы, хранящиеся в специальном каталоге в стандартной файловой системе. В Linux они имеют вид небольших объектов разделяемой памяти POSIX, хранящихся в файловой системе `tmpfs`

(см. раздел 14.10), подключенной к каталогу /dev/shm, а их имена выглядят как sem.*name*. Данная файловая система обладает сохраняемостью на уровне ядра — объекты семафоров, хранящиеся в ней, существуют даже в том случае, если не открыты ни одним процессом, но теряются при выключении системы.

Именованные семафоры поддерживаются в ядре Linux, начиная с версии 2.6.

49.2.1. Открытие именованного семафора

Функция sem_open() открывает именованный семафор; если семафор не существует, он будет создан.

```
#include <fcntl.h>              /* Определяет константы вида O_* */
#include <sys/stat.h>           /* Определяет константы, описывающие режим */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );
```

Возвращает указатель на семафор или SEM_FAILED, если произошла ошибка

Аргумент name обозначает имя семафора и подчиняется правилам, описанным в разделе 47.1.

Аргумент oflag представляет собой битовую маску, которая определяет, какая операция выполняется: открывается существующий семафор или создается новый. Если oflag равен 0, то мы обращаемся к существующему. В случае указания в маске oflag флага O_CREAT будет создан новый семафор (при условии, что имя name еще не занято). Если одновременно указать флаги O_CREAT и O_EXCL, а семафор с именем name уже существует, то вызов sem_open() завершится неудачей.

При использовании функции sem_open() для открытия семафора ей нужно передать всего два аргумента. Но в случае указания флага O_CREAT еще два аргумента становятся обязательными: mode и value (если семафор с именем name уже существует, то оба этих аргумента игнорируются). Они имеют следующее назначение.

- Аргумент mode — это битовая маска, которая определяет права доступа к новому семафору. Здесь применяются те же битовые значения, что и для файлов (см. табл. 15.4); как и в случае с вызовом open(), к маске применяется атрибут umask (см. подраздел 15.4.6). Стандарт SUSv3 не описывает никаких флагов для аргумента oflag, определяющих режим доступа (O_RDONLY, O_WRONLY и O_RDWR). Во многих системах, в том числе и в Linux, при открытии семафора по умолчанию используется режим доступа O_RDWR, так как большинство приложений выполняют как чтение, так и изменение семафоров, что подразумевает применение функций sem_post() и sem_wait(). То есть каждой категории пользователей, которым нужно работать с семафорами (владельцу, группе и т. д.) следует выдать права на чтение и запись.
- Аргумент value является беззнаковым целым числом, определяющим начальное значение нового семафора. Создание и инициализация семафора происходит автоматически.

Вне зависимости от того, создается ли новый семафор или открывается уже существующий, функция sem_open() возвращает указатель на значение типа sem_t, которое будет использоваться в последующих вызовах для работы с семафором. В случае неудачи sem_open() возвращает ошибку SEM_FAILED (в большинстве реализаций она определена как ((sem_t *) 0) или ((sem_t *) –1); в Linux применяется первый вариант).

Стандарт SUSv3 гласит, что при попытке выполнения операций (`sem_post()`, `sem_wait()` и т. д.) с *копией* переменной `sem_t`, на которую указывает возвращенное из `sem_open()` значение, результаты получаются неопределенными. Иными словами, следующее использование переменной `sem2` является недопустимым:

```
sem_t *sp, sem2
sp = sem_open(...);
sem2 = *sp;
sem_wait(&sem2);
```

Потомок, созданный с помощью вызова `fork()`, наследует ссылки на все именованные семафоры, открытые его родителем. После этого родительский и дочерний процессы могут задействовать данные семафоры для синхронизации своих действий.

Пример программы

Программа, показанная в листинге 49.1, предоставляет простой интерфейс командной строки к функции `sem_open()`. В функции `usageError()` представлен формат команд для этой программы.

В следующей сессии командной строки демонстрируется использование данного примера. Сначала мы вызываем команду `umask`, чтобы полностью закрыть доступ для всех остальных пользователей. Затем создаем семафор, доступный лишь нам, и выводим содержимое виртуального каталога (доступного только в Linux), который хранит именованные семафоры.

```
$ umask 007
$ ./psem_create -cx /demo 666      666 означает право доступа на чтение
                                   и запись для всех пользователей
$ ls -l /dev/shm/sem.*
-rw-rw----  1 mtk users 16 Jul 6 12:09 /dev/shm/sem.demo
```

Вывод команды `ls` показывает: атрибут `umask` переопределил права доступа, разрешив чтение и запись остальным пользователям.

Если мы еще раз используем то же имя, чтобы создать доступный только нам семафор, то ничего не получится, поскольку данное имя уже занято.

```
$ ./psem_create -cx /demo 666
ERROR [EEXIST File exists] sem_open   Ошибка, потому что указан флаг O_EXCL
```

Листинг 49.1. Использование функции sem_open() для открытия или создания именованного POSIX-семафора

psem/psem_create.c

```
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name
            [octal-perms [value]]\n", progName);
    fprintf(stderr, "    -c    Create semaphore (O_CREAT)\n");
    fprintf(stderr, "    -x    Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;

    flags = 0;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':   flags |= O_CREAT;           break;
        case 'x':   flags |= O_EXCL;            break;
        default:    usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    /* По умолчанию используются права доступа rw-------,
       а начальное значение семафора равно 0 */
    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");
    value = (argc <= optind + 2) ? 0 : getInt(argv[optind + 2], 0, "value");

    sem = sem_open(argv[optind], flags, perms, value);
    if (sem == SEM_FAILED)
        errExit("sem_open");
    exit(EXIT_SUCCESS);
}
```

psem/psem_create.c

49.2.2. Закрытие семафора

Когда процесс открывает именованный семафор, между ними устанавливается связь на уровне записи, которая создается системой. Функция `sem_close()` удаляет эту связь (то есть закрывает семафор), освобождает все выделенные на нее ресурсы и уменьшает счетчик ссылок на семафор.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Когда процесс завершается или выполняет вызов `exec()`, открытые им именованные семафоры автоматически закрываются. Закрытие семафора не приводит к его удалению. Для этого нужно использовать функцию `sem_unlink()`.

49.2.3. Удаление именованного семафора

Функция `sem_unlink()` удаляет семафор с именем `name` и делает его кандидатом на удаление, которое происходит, когда все процессы перестают его использовать (удаление может случиться сразу же, если все другие процессы уже закрыли данный семафор).

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Применение функции sem_unlink() продемонстрировано в листинге 49.2.

Листинг 49.2. Использование функции sem_unlink() для удаления именованного POSIX-семафора

psem/psem_unlink.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sem-name\n", argv[0]);

    if (sem_unlink(argv[1]) == -1)
        errExit("sem_unlink");
    exit(EXIT_SUCCESS);
}
```

psem/psem_unlink.c

49.3. Операции с семафорами

POSIX-семафор — целое число, которое не может быть меньше нуля. Операции с такими объектами имеют следующие особенности:

- функции sem_post() и sem_wait(), изменяющие значения семафоров, делают это для каждого семафора отдельно и не поддерживают групповых операций;
- функции sem_post() и sem_wait() инкрементируют и декрементируют значения семафоров ровно на 1;
- POSIX-семафоры не поддерживают операции ожидания нулевого значения.

Из всего вышесказанного может создаться следующее впечатление: POSIX-семафоры являются менее функциональными, чем, например, их аналоги из System V. Но это не так — с их помощью можно добиться тех же результатов, что и с семафорами из других стандартов. В некоторых ситуациях может потребоваться чуть больше усилий, но обычно использование POSIX-семафоров приводит к уменьшению объема кода.

49.3.1. Декрементация семафора

Функция sem_wait() декрементирует (уменьшает на 1) значение семафора, на который указывает аргумент sem.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Если семафор имеет значение больше нуля, то функция `sem_wait()` сразу же завершится. Когда семафор равен 0, функция `sem_wait()` заблокируется до тех пор, пока его значение не станет положительным; когда это случится, он будет декрементирован, а функция `sem_wait()` завершит свою работу.

При прерывании вызова `sem_wait()` с помощью обработчика сигнала он завершится ошибкой `EINTR`; использование флага `SA_RESTART` в вызове `sigaction()` (когда этот обработчик создавался) не будет играть никакой роли (в некоторых реализациях UNIX флаг `SA_RESTART` приводит к автоматическому перезапуску данного вызова).

Программа, показанная в листинге 49.3, предоставляет интерфейс командной строки к функции `sem_wait()`. Ее применение будет продемонстрировано чуть позже.

Листинг 49.3. Использование функции sem_wait() для декрементации POSIX-семафора

psem/psem_wait.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    sem_t *sem;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sem-name\n", argv[0]);

    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_wait(sem) == -1)
        errExit("sem_wait");

    printf("%ld sem_wait() succeeded\n", (long) getpid());
    exit(EXIT_SUCCESS);
}
```

psem/psem_wait.c

Функция `sem_trywait()` является неблокирующей версией `sem_wait()`.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если декрементация не может быть выполнена немедленно, то вызов `sem_trywait()` завершается ошибкой `EAGAIN`.

Функция `sem_timedwait()` представляет собой еще одну разновидность `sem_wait()`. Она позволяет вызывающему процессу ограничить время блокирования вызова.

```
#define _XOPEN_SOURCE 600
#include <semaphore.h>

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если вызов `sem_timedwait()` не успевает выполнить декрементацию семафора за отведенное ему время, то завершается ошибкой `ETIMEDOUT`.

Аргумент `abs_timeout` представляет собой структуру `timespec` (см. подраздел 23.4.2), которая определяет время ожидания в виде количества секунд и наносекунд, прошедших с начала «эры UNIX». Чтобы указать относительное время ожидания, нужно получить текущее значение часов `CLOCK_REALTIME`, применяя вызов `clock_gettime()`, и добавить к нему необходимое количество секунд или наносекунд. В результате должна получиться структура `timespec`, пригодная для использования в функции `sem_timedwait()`.

Функция `sem_timedwait()`, будучи изначально разработанной для стандарта POSIX.1d (1999), доступна не во всех реализациях UNIX.

49.3.2. Инкрементация семафора

Функция `sem_post()` инкрементирует (увеличивает на 1) значение семафора, на который указывает аргумент `sem`.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если перед вызовом `sem_post()` семафор был равен 0, и при этом какой-то другой процесс (или поток) заблокирован в ожидании ее декрементации, то данный процесс возобновляет работу и выполняет функцию `sem_wait()`, декрементируя семафор. При блокировке в вызове `sem_wait()` нескольких процессов (или потоков) в случае использования стандартной циклической политики разделения времени невозможно определить, какой из них возобновит работу и выполнит декрементацию (POSIX-семафоры являются лишь средством синхронизации; они не предназначены для создания очередей).

> Согласно стандарту SUSv3 если процессы или потоки выполняются в рамках политики планирования реального времени, то первым возобновит работу тот из них, у которого наивысший приоритет (то есть ожидавший дольше всех).

Инкрементация POSIX-семафора связана с освобождением некоего общего ресурса, в результате этого становящегося доступным для другого процесса или потока.

Программа, показанная в листинге 49.4, предоставляет интерфейс командной строки к функции `sem_post()`. Ее применение будет продемонстрировано чуть ниже.

Листинг 49.4. Использование функции sem_post() для инкрементации POSIX-семафора

psem/psem_post.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    sem_t *sem;

    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);
```

```
    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_post(sem) == -1)
        errExit("sem_post");
    exit(EXIT_SUCCESS);
}
```

psem/psem_post.c

49.3.3. Получение текущего значения семафора

Функция `sem_getvalue()` возвращает текущее значение семафора, представленного указателем `sem`; результат передается в виде целого числа, на которое указывает аргумент `sval`.

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Если один или несколько процессов (или потоков) заблокированы в ожидании декрементации семафора, то значение, возвращаемое в аргументе `sval`, зависит от конкретной реализации. Стандарт SUSv3 предусматривает два варианта: 0 или отрицательное число, модуль которого равен количеству ожидающих процессов (или потоков), заблокированных в вызове `sem_wait()`. В Linux и ряде других систем используется первый вариант; второй подход тоже применяется в отдельных реализациях.

> Возвращение отрицательного аргумента `sval` при наличии заблокированных процессов может быть полезно, особенно для отладки, но стандарт SUSv3 не требует так делать, поскольку методики, применяемые частью систем для оптимизации работы POSIX-семафоров, не позволяют записывать количество процессов, которые оказываются при этом заблокированными.

Нужно отметить, что на момент завершения вызова `sem_getvalue()` значение, возвращенное в аргументе `sval`, может уже быть неактуальным. Программа, которая полагается на актуальность информации, предоставляемой функцией `sem_getvalue()`, подвержена состоянию гонки вида «время проверки к времени использования» (см. раздел 38.6).

Программа в листинге 49.5 задействует функцию `sem_getvalue()`, чтобы получить значение семафора, чье имя указано в виде аргумента командной строки, и затем направляет результат в стандартный вывод.

Листинг 49.5. Использование функции sem_getvalue() для получения значения POSIX-семафора

psem/psem_getvalue.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int value;
    sem_t *sem;

    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);
```

```
    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_getvalue(sem, &value) == -1)
        errExit("sem_getvalue");

    printf("%d\n", value);
    exit(EXIT_SUCCESS);
}
```

psem/psem_getvalue.c

Пример программы

Использование программ, представленных ранее в этой главе, показано на примере следующей сессии командной строки. Для начала мы создадим семафор, минимальное значение которого равно нулю, и затем запустим в фоновом режиме программу, которая попытается его декрементировать:

```
$ ./psem_create -c /demo 600 0
$ ./psem_wait /demo &
[1] 31208
```

Фоновая команда блокируется, поскольку значение семафора равно `0` и, следовательно, не может быть уменьшено. Получим значение семафора:

```
$ ./psem_getvalue /demo
0
```

Как видите, значение равно `0`. В некоторых системах результат мог бы быть равен `-1`; это сигнализировало бы о наличии одного процесса, ожидающего изменения семафора.

Теперь запустим команду инкрементации. Это приведет к завершению работы вызова `sem_wait()`, заблокированного в фоновой программе:

```
$ ./psem_post /demo
$ 31208 sem_wait() succeeded
```

В последней строке видно: приглашение командной строки перемешано с выводом фонового задания.

Чтобы увидеть следующее приглашение командной строки, мы нажимаем клавишу Enter. Это приводит к тому, что командная оболочка сначала сигнализирует о завершении фонового задания. Затем выполняем дальнейшие операции с семафором:

```
Нажимаем Enter
[1]- Done           ./psem_wait /demo
$ ./psem_post /demo                 Инкрементируем семафор
$ ./psem_getvalue /demo             Получаем значение семафора
1
$ ./psem_unlink /demo               Мы закончили работать с этим семафором
```

49.4. Анонимные семафоры

Анонимные семафоры (основанные на памяти) представляют собой переменные типа `sem_t`, которые хранятся в памяти, выделенной приложением. Чтобы сделать семафор доступным для процессов или потоков, его нужно поместить в общий для них участок памяти.

Для работы с именованными и анонимными семафорами используются одни и те же функции (`sem_wait()`, `sem_post()`, `sem_getvalue()` и т. д.). Но, кроме этого, вводятся еще два (обязательных) вызова:

- функция `sem_init()` инициализирует семафор и оповещает систему о том, как он будет разделяться: между разными процессами или между потоками одного процесса;
- функция `sem_destroy(sem)` уничтожает семафор.

Обе функции нельзя использовать в сочетании с именованными семафорами.

Сравнение анонимных и именованных семафоров

Анонимные семафоры позволяют избежать процедуры выбора имени. Это может быть удобно в следующих ситуациях.

- Семафор, который разделяется между потоками, не нуждается в имени. Создавая анонимный семафор в виде разделяемой переменной (глобальной или внутри кучи), мы автоматически делаем его доступным для всех потоков.
- Семафору, разделяемому между родственными процессами, тоже не нужно имя. Анонимный семафор, выделенный родительским процессом на общем участке памяти (например, в разделяемом анонимном отображении), автоматически наследуется потомком вместе с отображением в ходе выполнения вызова `fork()`.
- Если мы создаем динамическую структуру данных (например, двоичное дерево), каждому элементу которой нужен отдельный семафор, то выделение анонимных семафоров будет наиболее простым решением. Применение именованных потребовало бы разработки специального соглашения об использовании (уникальных) имен в случае с каждым элементом и управлении этими именами (например, когда они становятся ненужными, их следует удалять).

49.4.1. Инициализация анонимного семафора

Функция `sem_init()` инициализирует анонимный семафор, представленный указателем `sem`, с помощью значения аргумента `value`.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `pshared` определяет способ разделения семафора: между потоками или процессами.

- Если значение `pshared` равно нулю, то семафор будет разделен между потоками вызывающего процесса. В этом случае аргумент `sem` обычно представляет собой адрес либо глобальной переменной, либо переменной, находящейся в куче. Такие семафоры функционируют на уровне процесса и уничтожаются вместе с ним.
- Если значение `pshared` не равно нулю, то семафор будет разделен между разными процессами. В данном случае аргумент `sem` должен хранить адрес, находящийся внутри участка разделяемой памяти (это может быть объект разделяемой памяти POSIX, разделяемое отображение, созданное с помощью вызова `mmap()`, или разделяемый сегмент памяти System V). Семафор функционирует, пока существует участок разделяемой

памяти, на котором он находится. (Участки памяти, созданные большинством из этих способов, обладают сохраняемостью на уровне ядра; исключение составляет анонимное отображение, существующее, пока его использует хотя бы один процесс.) Потомок, созданный путем вызова `fork()`, наследует отображения родителя, поэтому получает доступ и к семафорам, разделяемым между процессами; с его помощью он может синхронизировать свои действия с родителем.

Аргумент `pshared` необходим по следующим причинам.

- Некоторые реализации не поддерживают семафоры, разделяемые между процессами. Если аргументу `pshared` в такой системе передать ненулевое значение, то вызов `sem_init()` вернет ошибку. До версии 2.6 и появления NPTL-потоков ядро Linux не поддерживало анонимные семафоры этого типа (в старой реализации многопоточности, LinuxThreads, передача аргументу `pshared` ненулевого значения приводила к ошибке `ENOSYS`).
- Если реализация поддерживает разделение семафоров между процессами и потоками, то выбор типа разделения может оказаться необходимым, так как для поддержки того или иного режима системе нужно выполнить определенные действия. Предоставление этой информации может также позволить системе оптимизировать работу с учетом типа разделения.

NPTL-реализация функции `sem_init()` игнорирует аргумент `pshared`, поскольку ни один из типов разделения не требует специальных действий. Тем не менее приложения, которые пишутся с учетом разных платформ или рассчитаны на будущие изменения, должны указывать для аргумента `pshared` подходящее значение.

> Стандарт SUSv3 отмечает: в случае неудачного завершения функция sem_init() возвращает -1, однако в нем ничего не сказано относительно значения, которое возвращается при успешном выполнении. Тем не менее в руководствах к большинству современных UNIX-систем говорится о том, что в случае успеха возвращается 0. (Одним из заметных исключением является Solaris, где описание возвращаемого значения похоже на представленное в стандарте SUSv3; однако, судя по исходному коду OpenSolaris, функция sem_init() при успешном выполнении тоже возвращает 0.) Стандарт SUSv4 исправляет ситуацию, явно определяя значение 0 на случай успешного выполнения sem_init().

У анонимных семафоров нет параметров, описывающих права доступа (то есть функция `sem_init()` не имеет аргумента, аналогичного `mode` в вызове `sem_open()`). Доступ к этим семафорам регулируется на основе привилегий, выданных процессу для использования соответствующего участка памяти.

Стандарт SUSv3 гласит: инициализация уже инициализированного анонимного семафора приводит к неопределенным последствиям. Иными словами, мы должны проектировать свои приложения так, чтобы только один процесс вызывал функцию `sem_init()` для инициализации семафора.

Как и в случае с именованными семафорами, стандарт SUSv3 говорит о том, что в случае работы с *копией* переменной `sem_t`, чей адрес был передан в функцию `sem_init()` в качестве аргумента `sem`, результаты получаются непредсказуемыми. Обращаться следует исключительно к «оригиналу» семафора.

Пример программы

В подразделе 30.1.2 была представлена программа (см. листинг 30.2), которая использует мьютексы для защиты критического участка кода, где два потока обращались к одной и той же глобальной переменной. Программа, показанная в листинге 49.6, решает ту же проблему, но с помощью анонимного семафора, разделяемого между потоками.

Листинг 49.6. Применение анонимного POSIX-семафора для регулирования доступа к глобальной переменной

psem/thread_incr_psem.c

```
#include <semaphore.h>
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static sem_t sem;

static void *         /* Выполняем 'arg' итераций, инкрементируя 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1)
            errExit("sem_wait");
        loc = glob;
        loc++;
        glob = loc;

        if (sem_post(&sem) == -1)
            errExit("sem_post");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    /* Инициализируем семафор, разделяемый между потоками, с помощью значения 1 */

    if (sem_init(&sem, 0, 1) == -1)
        errExit("sem_init");

    /* Создаем два потока, которые инкрементируют 'glob' */

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    /* Ждем завершения потоков */

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
```

```
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

psem/thread_incr_psem.c

49.4.2. Уничтожение анонимного семафора

Функция `sem_destroy()` уничтожает анонимный семафор `sem`, инициализированный ранее с помощью вызова `sem_init()`. Эта процедура является безопасной только в том случае, если доступа к семафору не ожидает ни один поток или процесс.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

После уничтожения участка памяти с анонимным семафором его можно снова инициализировать, используя функцию `sem_init()`.

Уничтожение анонимного семафора должно происходить до освобождения занимаемой им памяти. Например, если семафор является автоматически выделяемой переменной, то должен быть уничтожен до возвращения функции, в которой был создан. При нахождении семафора на участке разделяемой памяти POSIX его нужно удалить после того, как он перестает использоваться каким-либо процессом, но перед удалением объекта разделяемой памяти с помощью функции `shm_unlink()`.

Часть систем позволяют пропустить вызов `sem_destroy()` без каких-либо проблем. Но существуют реализации, в которых это может привести к утечке ресурсов. Чтобы исключить такую возможность, портируемые приложения должны вызывать `sem_destroy()`.

49.5. Сравнение POSIX-семафоров с мьютексами из библиотеки Pthreads

И POSIX-семафоры, и Pthreads-мьютексы могут использоваться для синхронизации потоков в рамках одного и того же процесса. Они мало чем отличаются с точки зрения производительности, однако мьютексы являются более предпочтительными, поскольку поддерживают атрибут, указывающий на владельца, что располагает к лучшему структурированию кода (только поток, закрывший мьютекс, может его открыть). Для сравнения, семафор можно инкрементировать из потока, который его не инкрементировал. Такая гибкость может стать причиной неудачной архитектуры приложения (именно поэтому семафоры иногда называют аналогом инструкции `goto` в параллельном программировании).

Существует один сценарий, в котором мьютексы не подходят для использования в многопоточном приложении и могут быть заменены семафорами. Функция `sem_post()` является безопасной для вызова в асинхронных сигналах (см. табл. 21.1), поэтому ее можно задействовать внутри обработчика сигнала для синхронизации с другим потоком. Мьютексы этого не позволяют, так как соответствующие функции библиотеки Pthreads

небезопасны с точки зрения асинхронных сигналов. Однако данное преимущество семафоров редко оказывается востребованным, потому что асинхронные сигналы рекомендуется принимать в вызовах наподобие `sigwaitinfo()`, а не в самих обработчиках (см. подраздел 33.2.4).

49.6. Ограничения, связанные с семафорами

В стандарте SUSv3 описано два ограничения, налагаемых на семафоры.

- `SEM_NSEMS_MAX` — это максимальное количество POSIX-семафоров, которые могут быть у одного процесса. В стандарте SUSv3 указано, что данное значение должно быть не меньше 256. В Linux количество семафоров ограничивается только объемом доступной памяти.
- `SEM_VALUE_MAX` — это максимальное значение, которого может достигнуть POSIX-семафор. Семафоры могут быть равны любому числу от 0 до `SEM_VALUE_MAX`. Стандарт SUSv3 требует, чтобы данное ограничение было не меньше 32 767; в Linux он равен `INT_MAX` (2 147 483 647 в Linux/x86-32).

49.7. Резюме

POSIX-семафоры позволяют синхронизировать действия потоков или процессов. Они бывают двух видов: именованные и анонимные. Именованный семафор идентифицируется по имени и может разделяться между любыми процессами, у которых есть права на его открытие. Анонимный не имеет имени; для его разделения процессы или потоки должны поместить его на участок памяти, являющийся для них общим. В случае с процессами это может быть объект разделяемой памяти POSIX, а с потоками — глобальная переменная.

Интерфейс POSIX-семафоров отличается простотой. Выделение семафоров и работа с ними происходит на индивидуальном уровне, а инкрементация и декрементация изменяет их значение на 1.

POSIX-семафоры имеют ряд преимуществ перед своими аналогами из System V, но их сложнее переносить на другие системы. Для синхронизации в рамках многопоточных приложений более предпочтительными обычно являются мьютексы.

Дополнительная информация

В [Stevens, 1999] приводится альтернативный взгляд на POSIX-семафоры и описывается их реализация в пользовательском пространстве на основе других IPC-механизмов (очередей FIFO, файлов, отображенных в память, и семафоров System V). Использование POSIX-семафоров в многопоточных приложениях демонстрируется в [Butenhof, 1996].

49.8. Упражнение

49.1. Отредактируйте программу из листинга 49.3 (`psem_wait.c`), заменив `sem_wait()` функцией `sem_timedwait()`. Программа должна принимать дополнительный аргумент командной строки, который определяет относительное время ожидания вызова `sem_timedwait()` (в секундах).

50 Разделяемая память POSIX

В предыдущих главах мы уже познакомились с разделяемыми файловыми отображениями, которые позволяют неродственным процессам взаимодействовать с помощью общих участков памяти (см. подраздел 45.4.2). Однако использование таких отображений для организации межпроцессного взаимодействия требует создания файла на диске, даже если мы не заинтересованы в постоянном резервном хранилище для разделяемого участка. Помимо неудобств, связанных с созданием файла, это приводит к дополнительному расходу ресурсов на ввод/вывод.

В связи с данными недостатками в стандарт POSIX.1b был добавлен новый программный интерфейс для работы с разделяемой памятью, которому и посвящена данная глава.

50.1. Краткий обзор

Разделяемая память POSIX позволяет неродственным процессам иметь общий отображаемый участок без необходимости создавать соответствующий отображенный файл. Этот механизм поддерживается в ядре Linux, начиная с версии 2.4.

Стандарт SUSv3 не содержит никаких подробностей реализации разделяемой памяти POSIX. В частности, в нем отсутствуют требования к использованию файловой системы (реальной или виртуальной) для идентификации объектов разделяемой памяти, хотя многие UNIX-системы задействуют для этого средства файловой системы. В ряде систем имена таких объектов представляют собой файлы, хранящиеся в специальном каталоге в рамках стандартной файловой системы. Linux использует для этого файловую систему `tmpfs` (см. раздел 14.10), подключенную к каталогу `/dev/shm`. Она обладает сохраняемостью на уровне ядра; это значит, что объекты разделяемой памяти, которые в ней хранятся, существуют независимо от того, открыты ли они каким-либо процессом, но теряются при выключении системы.

> Общий объем памяти на всех разделяемых участках в системе ограничен размером связанной с ними файловой системы tmpfs. Она обычно подключается во время загрузки и имеет некий стандартный размер (например, 256 Мбайт), который при необходимости может быть изменен администратором путем повторного подключения с помощью команды вида mount -o remount,size=<количество_байтов>.

Для использования объектов разделяемой памяти POSIX нужно выполнить два шага.

1. Воспользоваться функцией `shm_open()`, чтобы открыть объект с заданным именем (правила именования объектов разделяемой памяти POSIX описаны в разделе 47.1). Эта функция является аналогом системного вызова `open()`. Она либо создает новый объект разделяемой памяти, либо открывает уже существующий. В качестве результата `shm_open()` возвращает файловый дескриптор, ссылающийся на объект.
2. Передать файловый дескриптор, полученный на предыдущем шаге, в вызов `mmap()`, в аргументе `flags` которого указан флаг `MAP_SHARED`. Это отобразит объект разделяемой памяти на виртуальное адресное пространство процесса. По аналогии с другими

способами применения вызова `mmap()` можно закрыть файловый дескриптор, не влияя на само отображение. Но иногда данный дескриптор лучше держать открытым, чтобы использовать его в последующих вызовах `fstat()` и `ftruncate()` (см. раздел 50.2).

Поскольку для обращения к объекту разделяемой памяти применяется файловый дескриптор, для работы с данным объектом не требуется специальных функций; вместо этого можно применять уже имеющиеся в UNIX системные вызовы (например, `ftruncate()`).

50.2. Создание объектов разделяемой памяти

Функция `shm_open()` создает новый или открывает уже существующий объект разделяемой памяти. Ее аргументы аналогичны тем, которые используются в вызове `open()`.

```
#include <fcntl.h>          /* Определяет константы вида O_* */
#include <sys/stat.h>       /* Определяет константы аргумента mode */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Возвращает файловый дескриптор или `-1`, если произошла ошибка

Аргумент `name` идентифицирует объект разделяемой памяти, который нужно создать или открыть. Аргумент `oflag` — это битовая маска, влияющая на работу вызова. Поддерживаемые ею значения перечислены в табл. 50.1.

Таблица 50.1. Битовые значения для аргумента oflag функции shm_open()

Флаг	Описание
O_CREAT	Создает объект, если он не был создан ранее
O_EXCL	Выполняет исключительно создание объекта, если указан флаг O_CREAT
O_RDONLY	Открывает только для чтения
O_RDWR	Открывает только для записи
O_TRUNC	Усекает объект до нулевой длины

Одно из назначений аргумента `oflag` заключается в определении того, нужно ли создавать объект разделяемой памяти перед его открытием. Если этот аргумент не включает в себя флаг `O_CREAT`, то открывается уже существующий объект. В противном случае при отсутствии объекта с таким именем он будет создан. Совместное использование флагов `O_EXCL` и `O_CREAT` гарантирует, что создателем объекта будет вызывающий процесс; в случае существования объекта мы получим ошибку `EEXIST`.

Аргумент `oflag` также определяет, какой доступ будет у вызывающего процесса к объекту разделяемой памяти; для этого применяется одна из констант: `O_RDONLY` или `O_RDWR`.

Последнее значение, `O_TRUNC`, приводит к установке нулевой длины успешно открытому объекту.

В Linux усечение выполняется даже в случае, когда объект открыт только для чтения. Но в стандарте SUSv3 отмечается, что совместное использование флагов O_TRUNC и O_RDONLY приводит к неопределенным результатам, поэтому портируемые приложения не могут полагаться на такое поведение.

При создании объекта разделяемой памяти пользователь и группа, которые им владеют, определяются на основе реальных пользовательского и группового идентификаторов процесса, вызвавшего `shm_open()`, а права доступа к объекту устанавливаются в соответствии со значением, указанным в битовой маске `mode`. Битовые значения аргумента `mode` аналогичны применяемым для файлов (см. табл. 15.4). Как и в случае с системным вызовом `open()`, к правам доступа, заданным в битовой маске, применяется атрибут `umask` (см. подраздел 15.4.6). Отличие от вызова `open()` состоит в том, что аргумент `mode` всегда является обязательным; если мы не создаем новый объект, то он должен быть равен `0`.

Для файлового дескриптора, возвращаемого вызовом `shm_open()`, устанавливается флаг `FD_CLOEXEC` (см. раздел 27.4); благодаря этому он автоматически закрывается, когда процесс выполняет `exec()` (точно так же при выполнении `exec()` удаляются отображения).

Сразу после создания новый объект разделяемой памяти имеет нулевую длину. То есть перед вызовом `mmap()` обычно используется функция `ftruncate()`, которая устанавливает размер объекта (см. раздел 5.8). Ее можно вызвать и после `mmap()`, чтобы уменьшить или увеличить объект разделяемой памяти (при этом следует учитывать замечания, приведенные в подразделе 45.4.3).

При расширении объекта разделяемой памяти дополнительные байты изначально заполняются нулями.

В любой момент к файловому дескриптору, возвращенному функцией `shm_open()`, можно применить вызов `fstat()` (см. раздел 15.1). Данное действие позволит получить структуру `stat`, поля которой содержат информацию об объекте разделяемой памяти, включая его размер (`st_size`), права доступа (`st_mode`), владельца (`st_uid`) и группу (`st_gid`); это те поля, которые должны содержаться в структуре `stat` согласно стандарту SUSv3, однако, помимо них, Linux возвращает различные дополнительные сведения, включая данные о времени.

Информация о правах доступа и владельце объекта разделяемой памяти может быть изменена путем вызовов `fchmod()` и, соответственно, `fchown()`.

Пример программы

В листинге 50.1 представлен простой пример использования вызовов `shm_open()`, `ftruncate()` и `mmap()`. Эта программа создает объект разделяемой памяти, размер которого задается с помощью аргумента командной строки, и отображает его на виртуальное адресное пространство процесса. (Этап отображения является избыточным, так как мы не собираемся ничего делать с разделяемой памятью; он здесь для демонстрации применения вызова `mmap()`.) Программа позволяет задействовать аргументы командной строки, чтобы выбрать флаги (`O_CREAT` и `O_EXCL`) для функции `shm_open()`.

В следующем примере мы используем данную программу для создания объекта разделяемой памяти размером 10 000 байт, а затем запускаем команду, чтобы показать этот объект в `/dev/shm`:

```
$ ./pshm_create -c /demo_shm 10000
$ ls -l /dev/shm
total 0
-rw-------    1 mtk      users         10000 Jun 20 11:31 demo_shm
```

Листинг 50.1. Создание объекта разделяемой памяти POSIX

pshm/pshm_create.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"
```

```
static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] shm-name
            size [octal-perms]\n", progName);
    fprintf(stderr, "    -c    Create shared memory (O_CREAT)\n");
    fprintf(stderr, "    -x    Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt, fd;
    mode_t perms;
    size_t size;
    void *addr;

    flags = O_RDWR;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':   flags |= O_CREAT;           break;
        case 'x':   flags |= O_EXCL;            break;
        default:    usageError(argv[0]);
        }
    }

    if (optind + 1 >= argc)
        usageError(argv[0]);

    size = getLong(argv[optind + 1], GN_ANY_BASE, "size");
    perms = (argc <= optind + 2) ? (S_IRUSR | S_IWUSR) :
                getLong(argv[optind + 2], GN_BASE_8, "octal-perms");

    /* Создаем объект разделяемой памяти и устанавливаем его размер */

    fd = shm_open(argv[optind], flags, perms);
    if (fd == -1)
        errExit("shm_open");

    if (ftruncate(fd, size) == -1)
        errExit("ftruncate");

    /* Отображаем объект разделяемой памяти */

    addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    exit(EXIT_SUCCESS);
}
```

pshm/pshm_create.c

50.3. Использование объектов разделяемой памяти

В листингах 50.2 и 50.3 демонстрируется применение объектов разделяемой памяти для передачи данных от одного процесса другому. Первая программа копирует строку в существующий объект разделяемой памяти; имя объекта указывается в первом аргументе

командной строки, а копируемая строка — во втором. Но перед выполнением этих операций программа задействует вызов `ftruncate()`, чтобы размер объекта был равен размеру строки, которую нужно копировать.

Листинг 50.2. Копирование данных в объект разделяемой памяти POSIX

pshm/pshm_write.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    size_t len;                 /* Размер объекта разделяемой памяти */
    char *addr;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name string\n", argv[0]);

    fd = shm_open(argv[1], O_RDWR, 0);   /* Открываем существующий объект */
    if (fd == -1)
        errExit("shm_open");

    len = strlen(argv[2]);
    if (ftruncate(fd, len) == -1)  /* Изменяем размер объекта, чтобы вместить строку */
        errExit("ftruncate");
    printf("Resized to %ld bytes\n", (long) len);

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)          /* Дескриптор 'fd' больше не нужен */
        errExit("close");

    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);   /* Копируем строку в разделяемую память */
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_write.c

Программа из листинга 50.3 направляет в стандартный вывод строку, хранящуюся в существующем объекте разделяемой памяти, чье имя указано в виде аргумента командной оболочки. Выполнив функцию `shm_open()`, программа использует вызов `fstat()`, чтобы определить размер разделяемой памяти; полученное значение передается в вызов `mmap()`, который отображает объект. В конце строка выводится с помощью операции `write()`.

Листинг 50.3. Копирование данных из объекта разделяемой памяти POSIX

pshm/pshm_read.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    int fd;
    char *addr;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);

    fd = shm_open(argv[1], O_RDONLY, 0); /* Открываем существующий объект */
    if (fd == -1)
        errExit("shm_open");

    /* Используем размер объекта разделяемой памяти в качестве длины
       аргумента для вызова mmap() и количества байтов для операции write() */

    if (fstat(fd, &sb) == -1)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)          /* Дескриптор 'fd' больше не нужен */
        errExit("close");

    write(STDOUT_FILENO, addr, sb.st_size);
    printf("\n");
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_read.c

Использование программ из листингов 50.2 и 50.3 продемонстрировано в следующей сессии командной строки. Сначала создадим объект разделяемой памяти нулевой длины (воспользовавшись программой из листинга 50.1).

```
$ ./pshm_create -c /demo_shm 0
$ ls -l /dev/shm              Проверяем размер объекта
total 4
-rw-------    1 mtk    users     0 Jun 21 13:33 demo_shm
```

Затем скопируем строку в объект разделяемой памяти с помощью программы из листинга 50.2:

```
$ ./pshm_write /demo_shm 'hello'
$ ls -l /dev/shm              Показываем, что размер объекта изменился
total 4
-rw-------    1 mtk    users     5 Jun 21 13:33 demo_shm
```

Вывод, представленный выше, показывает: программа изменила размер объекта разделяемой памяти так, чтобы тот мог вместить заданную строку.

Теперь задействуем программу из листинга 50.3 для вывода строки из нашего объекта:

```
$ ./pshm_read /demo_shm
hello
```

Для координации доступа к разделяемой памяти обычно используются какие-нибудь средства синхронизации. В примере сессии, показанной выше, вся координация сводилась

к тому, что пользователь запускал программы последовательно, одну за другой. Вместо этого рекомендуется применять механизм синхронизации (например, семафоры).

50.4. Удаление объектов разделяемой памяти

Стандарт SUSv3 требует, чтобы объекты разделяемой памяти имели сохраняемость как минимум на уровне ядра — то есть они должны существовать до тех пор, пока не будут удалены, или до перезагрузки системы. Когда объект разделяемой памяти больше не нужен, его следует удалить с помощью функции `shm_unlink()`.

```
#include <sys/mman.h>

int shm_unlink(const char *name);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Функция `shm_unlink()` удаляет объект разделяемой памяти с именем `name`. Удаление не влияет на отображения объекта (которые будут существовать, пока соответствующие не завершатся или не выполнят вызов `munmap()`), но не позволят его открывать путем дальнейших вызовов `shm_open()`. Когда все процессы удалят свои отображения, объект будет уничтожен, а его содержимое утеряно.

Программа, представленная в листинге 50.4, применяет функцию `shm_unlink()` для удаления объекта разделяемой памяти, указанного в виде аргумента командной строки.

Листинг 50.4. Использование функции shm_unlink() для удаления объекта разделяемой памяти POSIX

pshm/pshm_unlink.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);
    if (shm_unlink(argv[1]) == -1)
        errExit("shm_unlink");
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_unlink.c

50.5. Сравнение программных интерфейсов для работы с разделяемой памятью

Мы уже успели рассмотреть разные методики разделения участков памяти между неродственными процессами:

- разделяемые файловые отображения (см. подраздел 45.4.2);
- объекты разделяемой памяти POSIX (которым посвящена данная глава).

Многие замечания, приведенные ниже в этом разделе, в равной степени относятся и к анонимным отображениям (см. раздел 45.7), применяемым для разделения памяти между родственными процессами, связанными вызовом fork().

Ниже перечислены особенности, характерные для обеих методик.

- Они обеспечивают быстрое межпроцессное взаимодействие; для синхронизации доступа к разделяемому участку обычно используется семафор (или аналогичный механизм).
- Разделяемый участок памяти, отображенный на виртуальное адресное пространство процесса, ведет себя так же, как и другие участки памяти процесса.
- Размещение участков разделяемой памяти внутри виртуального адресного пространства процесса происходит похожим образом. Информация обо всех видах участков разделяемой памяти хранится в файле `/proc/PID/maps` (доступном только в Linux).
- Если участок разделяемой памяти отображается не на фиксированный адрес, то нужно сделать так, чтобы все ссылки на адреса внутри этого участка вычислялись в качестве сдвигов (а не указателей), поскольку местоположение отображения в адресном пространстве может варьироваться в зависимости от конкретного процесса.
- Функции для работы с участками виртуальной памяти, описанные в главе 46, можно применять к участкам, созданным любой из этих методик.

Представленные методики имеют и различия. В их числе можно выделить тот факт, что содержимое разделяемого файлового отображения синхронизируется с исходным отображенным файлом; то есть данные, хранящиеся на общем участке памяти, не будут утрачены при перезагрузке системы.

Выбор между двумя этими интерфейсами зависит от того, нужно ли нам постоянное резервное хранилище. В случае положительного ответа лучше выбрать разделяемое файловое отображение. Но, если такое хранилище не требуется, то объекты разделяемой памяти POSIX позволяют избежать дополнительных расходов, связанных с использованием файлов на диске.

50.6. Резюме

Объекты разделяемой памяти POSIX применяются для разделения участков памяти между неродственными процессами и не требуют создания файлов на диске. Для этого вместо вызова `open()`, который обычно выполняется перед `mmap()`, используется функция `shm_open()`. Она создает виртуальный файл в рамках файловой системы, находящейся в памяти; для работы с ним можно задействовать традиционные системные вызовы, рассчитанные на файловые дескрипторы. В частности, чтобы задать размер объекта разделяемой памяти, подойдет вызов `ftruncate()` (так как изначально размер равен нулю).

На данный момент мы рассмотрели две методики разделения участков памяти между неродственными процессами: разделяемые файловые отображения и объекты разделяемой памяти POSIX. В нескольких аспектах они довольно похожи, но между ними существует и различие, которое сводится к тому, нужно ли нам постоянное резервное хранилище.

51 Блокировка файлов

В предыдущих главах мы уже рассматривали различные методики, позволяющие синхронизировать действия разных процессов: сигналы (главы 20 и 22) и семафоры (глава 49). В этой главе мы познакомимся с дополнительными средствами синхронизации, специально предназначенными для работы с файлами.

51.1. Краткий обзор

Приложениям часто приходится считывать из файлов данные, изменять их и записывать обратно. Это не вызывает никаких проблем, если файл используется только одним процессом. Но все становится намного сложнее, когда несколько процессов одновременно изменяют один и тот же файл. Представьте, к примеру, что для обновления файла, содержащего порядковый номер, каждый процесс выполняет следующие шаги.

1. Чтение порядкового номера из файла.
2. Использование этого номера для каких-то задач внутри приложения.
3. Инкрементация порядкового номера и запись его обратно в файл.

Проблема заключается вот в чем: при отсутствии механизма синхронизации два процесса могут выполнить вышеописанные действия одновременно, и это приведет к последствиям, проиллюстрированным на рис. 51.1 (подразумевается, что начальное значение порядкового номера равно 1000).

Проблема очевидна: по завершении данных действий файл будет содержать значение 1001 вместо корректного 1002 (это пример состояния гонки). Чтобы избежать такой ситуации, нужен некий вид межпроцессной синхронизации.

Для синхронизации этих действий можно было бы воспользоваться, например, семафорами, однако блокировка файлов обычно является более предпочтительной, поскольку ядро автоматически связывает блокировку с файлом.

> В [Stevens & Rago, 2005] утверждается: первая реализация блокировки файлов в UNIX датируется 1980 годом, а также отмечается, что вызов `fcntl()`, на котором мы сосредоточимся в этой главе, появился в 1984 году в System V Release 2.

В данной главе мы рассмотрим два разных программных интерфейса для блокировки файлов:

- вызов `flock()`, блокирующий весь файл целиком;
- вызов `fcntl()`, блокирующий отдельные участки файла.

Первый изначально появился в системе BSD, а второй — в System V.

Общий принцип использования этих вызовов выглядит следующим образом.

1. Блокировка файла.
2. Выполнение ввода/вывода.
3. Разблокировка файла, чтобы другой процесс мог его заблокировать.

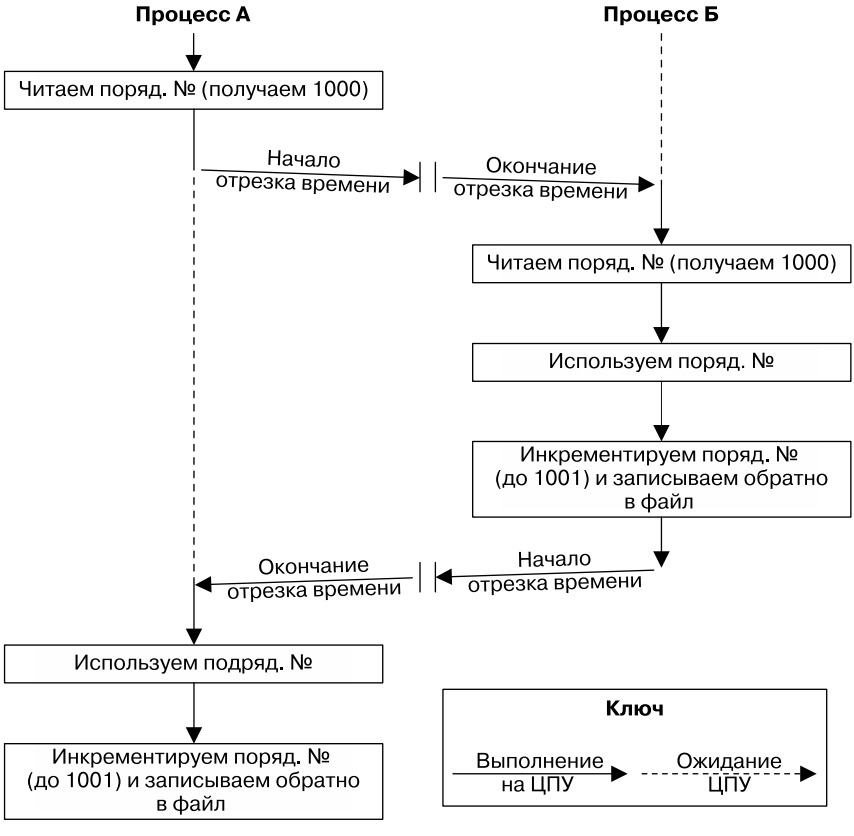


Рис. 51.1. Два процесса одновременно обновляют файл без синхронизации

Обычно блокировка применяется в сочетании с вводом/вывод, но также можно применять ее в качестве средства синхронизации. Процессы могут соблюдать соглашение, в соответствии с которым частичная или полная блокировка файла сигнализирует о доступе процесса к какому-то другому общему ресурсу (например, к участку разделяемой памяти).

Сочетание блокировки файлов со стандартным вводом/выводом

Функции библиотеки `stdio` выполняют буферизацию в пользовательском пространстве, в связи с чем следует быть осторожными, применяя их вместе с методиками блокировки, описанными в данной главе. Дело в том, что буфер ввода может заполниться до создания блокировки, а буфер вывода может быть сброшен до ее удаления. Существует несколько способов, позволяющих избежать этих проблем:

- выполнять файловый ввод/вывод с помощью `read()` и `write()` (и аналогичных системных вызовов) вместо библиотеки `stdio`;
- сбрасывать поток `stdio` сразу же после создания блокировки, повторяя эту процедуру перед ее удалением;
- полностью отключить буферизацию ввода/вывода, используя вызов `setbuf()` или аналогичный (возможно, ценой ухудшения производительности).

Необязательная и строгая блокировка

В оставшейся части данной главы блокировка будет разделяться на необязательную и строгую. По умолчанию она является *необязательной* — другой процесс может ее просто проигнорировать. Чтобы такой подход как-то работал, процессы должны взаимодействовать, создавая блокировку до выполнения ввода/вывода. Для сравнения, система *строгой* блокировки заставляет процесс, выполняющий ввода/вывод, подчиняться чужим блокировкам. Мы еще вернемся к этому различию в разделе 51.4.

51.2. Блокировка файла с помощью вызова flock()

Вызов `flock()` является лишь подмножеством вызова `fcntl()`, но мы все равно остановимся на нем отдельно, поскольку он до сих пор используется в ряде приложений и имеет некоторые семантические отличия, касающиеся наследования и удаления блокировок.

```
#include <sys/file.h>

int flock(int fd, int operation);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Системный вызов `flock()` блокирует весь файл целиком. Сам файл, который нужно заблокировать, передается аргументу `fd` в виде открытого дескриптора. Аргумент `operation` обозначает одну из операций: `LOCK_SH`, `LOCK_EX` или `LOCK_UN` (табл. 51.1).

По умолчанию вызов `flock()` блокируется, если другой процесс уже поместил несовместимую блокировку на заданный файл. Чтобы это предотвратить, можно применить к значениям аргумента `operation` побитовое ИЛИ (`|`). В таком случае, если другой процесс удерживает несовместимую блокировку для того же файла, то вызов `flock()` не заблокируется, а сразу же вернет `-1` и укажет в переменной `errno` ошибку `EWOULDBLOCK`.

Таблица 51.1. Значения аргумента operation из вызова flock()

Значение	Описание
LOCK_SH	Применяет разделяемую блокировку к файлу, на который указывает fd
LOCK_EX	Применяет эксклюзивную блокировку к файлу, на который указывает fd
LOCK_UN	Разблокирует файл, на который указывает fd
LOCK_NB	Выполняет неблокирующий запрос блокировки

Разделяемую блокировку файла может одновременно удерживать неограниченное количество процессов. Для сравнения, эксклюзивную блокировку в любой отдельный момент времени может удерживать только один процесс (иными словами, это не дает другим процессам создавать любые блокировки — как разделяемые, так и эксклюзивные). В табл. 51.2 собраны правила совместимости для блокировок, созданных с помощью `flock()`. Предполагается, что процесс А первым разместил блокировку, а данные правила определяют, сможет ли затем процесс Б выполнить ее.

Таблица 51.2. Совместимость типов блокировки вызова flock()

Процесс А	**Процесс Б**	
	LOCK_SH	LOCK_EX
LOCK_SH	Да	Нет
LOCK_EX	Нет	Нет

Процесс может разместить эксклюзивную или разделяемую блокировку вне зависимости от режима доступа (чтение, запись или чтение + запись).

Существующую разделяемую блокировку можно преобразовать в эксклюзивную (и наоборот), выполнив еще один вызов `flock()` и указав соответствующее значение для аргумента `operation`. Эта процедура заблокируется, если другой процесс уже удерживает разделяемую блокировку для данного файла (разве что было задано значение `LOCK_NB` — тогда этого не произойдет).

Атомарность операции преобразования блокировки *не* гарантируется. В ходе ее выполнения сначала удаляется существующая блокировка, а затем устанавливается новая. Между этими двумя действиями другой процесс может успеть получить несовместимую блокировку. В данном случае преобразование будет либо заблокировано, либо, если указан режим `LOCK_NB`, завершено с ошибкой, а процесс потеряет исходную блокировку (такая ситуация была характерна для оригинальной версии `flock()` в системе BSD и до сих пор встречается в ряде других реализаций UNIX).

> Вызов flock() хоть и не является частью стандарта SUSv3, но поддерживается в большинстве реализаций UNIX. Некоторые из них требуют подключения заголовочного файла <fcntl.h> или <sys/fcntl.h> вместо <sys/file.h>. Поскольку данный вызов берет свое начало в системе BSD, блокировки, устанавливаемые им, иногда называют файловыми BSD-блокировками.

Применение вызова `flock()` демонстрируется в листинге 51.1. Эта программа блокирует файл и, перейдя в режим сна на заданное количество секунд, разблокирует его обратно. Она принимает три аргумента командной строки. Первый из них представляет файл, подлежащий блокировке. Второй определяет ее тип (разделяемая или эксклюзивная) и необходимость включения флага `LOCK_NB` (неблокирующий режим). Третий аргумент обозначает количество секунд, которое должно пройти между блокировкой и разблокировкой; он не является обязательным и по умолчанию равен 10 секундам.

Листинг 51.1. Использование вызова flock()

filelock/t_flock.c

```
#include <sys/file.h>
#include <fcntl.h>
#include "curr_time.h"                  /* Объявление currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd, lock;
    const char *lname;

    if (argc < 3 || strcmp(argv[1], "--help") == 0 ||
            strchr("sx", argv[2][0]) == NULL)
        usageErr("%s file lock [sleep-time]\n"
```

```
                "     'lock' is 's' (shared) or 'x' (exclusive)\n"
                "         optionally followed by 'n' (nonblocking)\n"
                "     'sleep-time' specifies time to hold lock\n", argv[0]);

    lock = (argv[2][0] == 's') ? LOCK_SH : LOCK_EX;
    if (argv[2][1] == 'n')
        lock |= LOCK_NB;

    fd = open(argv[1], O_RDONLY);   /* Открываем файл, который нужно заблокировать */
    if (fd == -1)
        errExit("open");

    lname = (lock & LOCK_SH) ? "LOCK_SH" : "LOCK_EX";

    printf("PID %ld: requesting %s at %s\n", (long) getpid(), lname, currTime("%T"));

    if (flock(fd, lock) == -1) {
        if (errno == EWOULDBLOCK)
            fatal("PID %ld: already locked - bye!", (long) getpid());
        else
            errExit("flock (PID=%ld)", (long) getpid());
    }

    printf("PID %ld: granted    %s at %s\n", (long) getpid(), lname, currTime("%T"));

    sleep((argc > 3) ? getInt(argv[3], GN_NONNEG, "sleep-time") : 10);

    printf("PID %ld: releasing  %s at %s\n", (long) getpid(),
            lname, currTime("%T"));
    if (flock(fd, LOCK_UN) == -1)
        errExit("flock");

    exit(EXIT_SUCCESS);
}
```

filelock/t_flock.c

С помощью программы из листинга 51.1 можно выполнить ряд экспериментов, помогающих изучить поведение вызова `flock()`. Отдельные примеры показаны в следующей сессии командной строки. Сначала создадим файл, после чего запустим экземпляр нашей программы, который будет находиться в фоне, удерживая блокировку на протяжении 60 секунд:

```
$ touch tfile
$ ./t_flock tfile s 60 &
[1] 9777
PID 9777: requesting    LOCK_SH at 21:19:37
PID 9777: granted       LOCK_SH at 21:19:37
```

Теперь запустим еще один экземпляр той же программы, который успешно запрашивает и снимает разделяемую блокировку:

```
$ ./t_flock tfile s 2
PID 9778: requesting    LOCK_SH at 21:19:49
PID 9778: granted       LOCK_SH at 21:19:49
PID 9778: releasing     LOCK_SH at 21:19:51
```

Но если запустить еще один экземпляр данной программы, который попытается получить эксклюзивную блокировку в неблокирующем режиме, то его запрос будет немедленно отклонен:

```
$ ./t_flock tfile xn
PID 9779: requesting    LOCK_EX at 21:20:03
PID 9779: already locked - bye!
```

При попытке получить эксклюзивную блокировку в блокирующем режиме программа будет заблокирована. Ее запрос будет удовлетворен только после того, как фоновый процесс, уснувший на 60 секунд, снимет свою блокировку:

```
$ ./t_flock tfile x
PID 9780: requesting    LOCK_EX at 21:20:21
PID 9777: releasing     LOCK_SH at 21:20:37
PID 9780: granted       LOCK_EX at 21:20:37
PID 9780: releasing     LOCK_EX at 21:20:47
```

51.2.1. Семантика наследования и снятия блокировок

Как показано в табл. 51.1, файловую блокировку можно снять, задействуя вызов `flock()`, указав в качестве аргумента `operation` флаг `LOCK_UN`. Кроме того, блокировка будет автоматически снята при закрытии соответствующего файлового дескриптора. Однако не все так просто. Блокировка, полученная с помощью вызова `flock()`, связана с описанием открытого файла (см. раздел 5.4), а не с его дескриптором или самим файлом (индексным дескриптором файла). Это значит, что при дублировании файлового дескриптора (используя вызовы `dup()` и `dup2()` или вызов `fcntl()` с флагом `F_DUPFD`) его копия будет ссылаться на ту же блокировку. Например, при полученной блокировке файла, на который указывает аргумент `fd`, для ее снятия потребуется следующий код (проверка на ошибки опускается):

```
flock(fd, LOCK_EX);          /* Получаем блокировку с помощью 'fd'*/
newfd = dup(fd);             /* 'newfd' ссылается на ту же блокировку, что и 'fd' */
flock(newfd, LOCK_UN);       /* Снимаем блокировку, полученную с помощью 'fd' */
```

Если мы получим блокировку с помощью определенного файлового дескриптора, продублируем его один или несколько раз и не выполним вручную операцию снятия блокировки, то она будет снята только после закрытия всех дескрипторов-дубликатов.

Но если для получения второго дескриптора (и связанного с ним описания открытого файла), ссылающегося на тот же файл, использовать вызов `open()`, то этот дескриптор будет восприниматься вызовом `flock()` как нечто отдельное. Например, процесс, выполняющий следующий код, блокируется на втором вызове `flock()`:

```
fd1 = open("a.txt", O_RDWR);
fd2 = open("a.txt", O_RDWR);
flock(fd1, LOCK_EX);
flock(fd2, LOCK_EX);     /* Исключается из-за блокировки, установленной для 'fd1' */
```

Таким образом, процесс может освободить себя от блокировки файла с помощью вызова `flock()`. Как мы увидим позже, этого не может случиться с блокировками записей, полученными благодаря применению вызова `fcntl()`.

Дочерний процесс, создаваемый нами путем `fork()`, получает копии файловых дескрипторов своего родителя, которые (как и в случае с дескрипторами, продублированными через вызов `dup()` и ему подобные) ссылаются на те же атрибуты открытого файла и, следовательно, на те же блокировки. Следующий код, например, заставляет дочерний процесс удалить блокировку родителя:

```
flock(fd, LOCK_EX);          /* Родитель получает блокировку */
if (fork() == 0)             /* Если это потомок... */
flock(fd, LOCK_UN);          /* Снимаем блокировку, разделяемую с родителем */
```

Такая семантика может быть удобной для (автоматической) передачи файловой блокировки от родительского процесса дочернему: после вызова `fork()` родитель закрывает свой дескриптор, после чего блокировка переходит под полный контроль потомка. Как вы увидите позже, этого нельзя добиться при использовании блокировок записей, полученных с помощью вызова `fcntl()`.

Блокировки, созданные с применением `flock()`, сохраняются на протяжении всей работы вызова `exec()` (разве что на описание открытого файла ссылается единственный дескриптор, для которого установлен флаг `FD_CLOEXEC`).

Семантика, описанная выше, применяется в Linux и соответствует классической реализации вызова `flock()` в системах BSD. В некоторых UNIX-системах вызов `flock()` основан на `fcntl()`; позже вы увидите, что поведение этих вызовов при наследовании и снятии блокировок отличается. Поскольку взаимодействие блокировок, созданных с помощью вызовов `flock()` и `fcntl()`, является неопределенным, для каждого конкретного файла блокировки следует создавать каким-то одним способом.

51.2.2. Ограничения вызова flock()

Создание блокировок с использованием вызова `flock()` имеет несколько ограничений.

- Файлы можно блокировать только целиком. Такая грубая блокировка может негативно сказаться на параллельной работе взаимодействующих процессов. Например, если их несколько и все хотят получить доступ к разным участкам одного файла, то теоретически они могут работать параллельно, однако вызов `flock()` исключил бы такую возможность.
- Вызов `flock()` позволяет устанавливать только необязательные блокировки.
- Многие реализации файловой системы NFS не распознают блокировки, выданные вызовом `flock()`.

Вызов `fcntl()`, который мы рассмотрим в следующем разделе, реализует модель, лишенную этих ограничений.

51.3. Блокировка записей с помощью вызова fcntl()

Используя вызов `fcntl()` (см. раздел 5.2), блокировку можно установить как для всего файла целиком, так и для любой его части, даже если ее размер равен одному байту. Такой подход обычно называют *блокировкой записей*, хотя это не совсем верно, поскольку файлы в UNIX-системе не имеют границ (присущих записям) и представляют собой байтовые последовательности. Понятие записи в случае с файлом определяется исключительно самим приложением.

Обычно вызов `fcntl()` применяется для блокировки байтов в диапазоне, который соответствует границам записи внутри файла, определенным на уровне приложения; отсюда и термин *блокировка записей*. Термины *диапазон байтов*, *участок файла*, *сегмент файла* применяются не так часто, но более точно описывают этот вид блокировки файлов (это единственный способ, описанный в оригинальной спецификации POSIX.1 и стандарте SUSv3, в связи с чем его иногда называют POSIX-блокировкой).

> Стандарт SUSv3 требует, чтобы блокировка записей поддерживалась для обычных файлов, и допускает ее поддержку другими файловыми объектами. Как правило, эту блокировку имеет смысл применять только к обычным файлам (поскольку в случае с другими файловыми объектами понятие диапазона байтов теряет свое первоначальное значение), но в Linux это можно делать для любого файлового дескриптора.

На рис. 51.2 показано, как можно синхронизировать доступ двух процессов к одному и тому же участку файла, используя блокировку записей (предполагается, что все запросы выполняются в блокирующем режиме и приостанавливаются, если блокировка удерживается другим процессом).

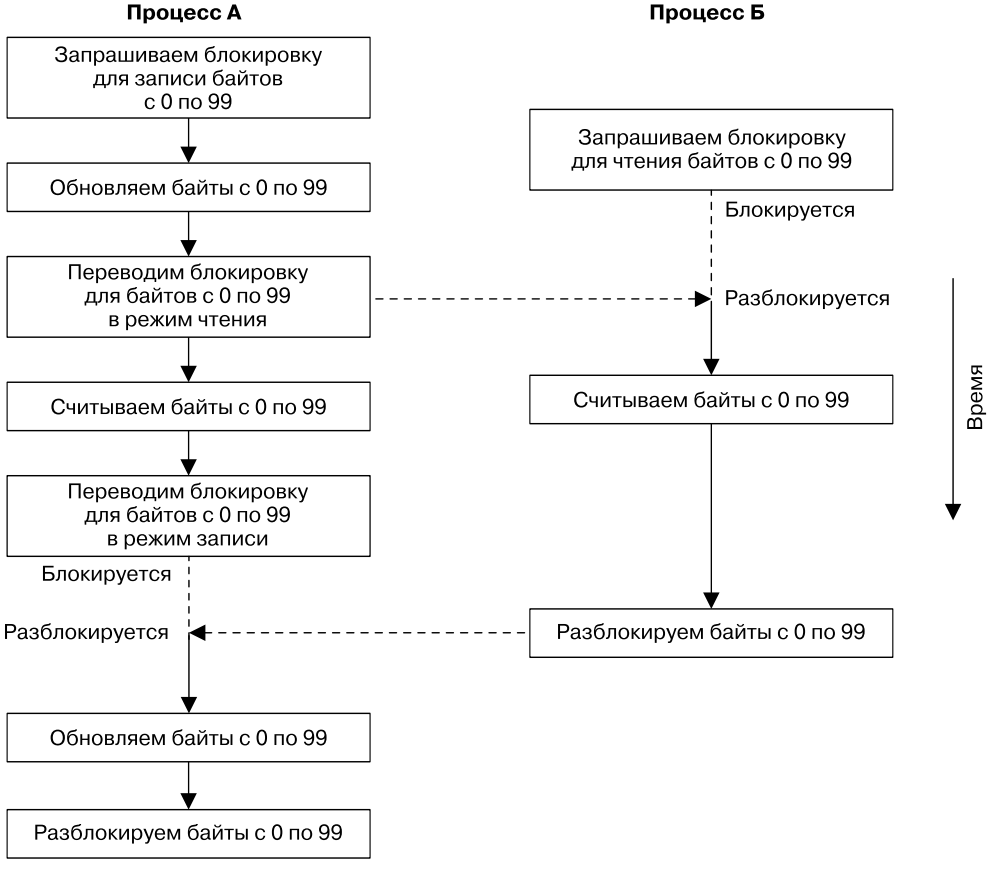


Рис. 51.2. Синхронизация доступа к одному и тому же участку файла с помощью блокировки записей

В целом создание или удаление файловой блокировки с применением вызова `fcntl()` выглядит следующим образом:

```
struct flock flockstr;

/* Инициализируем поля 'flockstr', чтобы описать блокировку,
   которую нужно установить или удалить */


fcntl(fd, cmd, &flockstr);  /* Устанавливаем блокировку, описанную в 'fl' */
```

Аргумент `fd` представляет собой открытый дескриптор, ссылающийся на файл, который мы хотим заблокировать.

Прежде чем перейти к аргументу `cmd`, сначала познакомимся со структурой `flock`.

51.3.1. Структура flock

Структура `flock` описывает блокировку, которую нужно установить или снять. Она имеет следующее определение:

```
struct flock {
    short l_type;       /* Тип блокировки: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;     /* Как интерпретируется поле 'l_start': SEEK_SET,
                           SEEK_CUR, SEEK_END */
    off_t l_start;      /* Начало блокировки (сдвиг) */
    off_t l_len;        /* Количество блокируемых байтов; 0 означает «до конца файла» */
    pid_t l_pid;        /* Процесс, не дающий нам установить
                           блокировку (только для F_GETLK) */
};
```

Поле `l_type` обозначает тип блокировки, которую мы хотим установить. Оно может принимать одно из значение, перечисленных в табл. 51.3.

С точки зрения семантики блокировки, предназначенные для чтения (`F_RDLCK`) и записи (`F_WRLCK`), соответствуют разделяемым и эксклюзивным блокировкам, устанавливаемым вызовом `flock()`, и подчиняются тем же правилам совместимости (см. табл. 51.2). Блокировку `F_RDLCK`, относящуюся к определенному участку файла, может удерживать любое количество процессов, но только один процесс может владеть блокировкой `F_WRLCK` (которая к тому же исключает блокировки любых типов, принадлежащих другим процессам). Использование значения `F_UNLCK` для аргумента `l_type` аналогично операции `LOCK_UN` для вызова `flock()`.

Таблица 51.3. Типы блокировок, создаваемые вызовом fcntl()

Тип блокировки	Описание
F_RDLCK	Устанавливает блокировку для чтения
F_WRLCK	Устанавливает блокировку для записи
F_UNLCK	Удаляет существующую блокировку

Для установки блокировки `F_RDLCK` файл должен быть открыт для чтения. Аналогично `F_WRLCK` требует, чтобы файл был открыт для записи. Для размещения блокировок обоих видов файл следует открывать в режиме чтения-записи (`O_RDWR`). Попытка установить блокировку, несовместимую с режимом доступа к файлу, приведет к ошибке `EBADF`.

Поля `l_whence`, `l_start` и `l_len` в совокупности определяют диапазон байтов, которые нужно заблокировать. Первые два из них являются аналогами аргументов `whence` и `offset` для вызова `lseek()` (см. раздел 4.7). Поле `l_start` обозначает сдвиг внутри файла, вычисляемый относительно:

- начала файла, если аргумент `l_whence` равен `SEEK_SET`;
- текущего сдвига в файле, если аргумент `l_whence` равен `SEEK_CUR`;
- конца файла, если аргумент `l_whence` равен `SEEK_END`.

В двух последних случаях аргумент `l_start` может иметь отрицательное значение, если итоговая позиция находится не перед началом файла (байтом 0).

Поле `l_len` содержит содержит целое число, обозначающее количество байтов, которые нужно заблокировать, начиная с позиции, заданной с помощью `l_whence` и `l_start`. Теоретически заблокировать можно и несуществующие байты, выходящие за пределы конца файла, но байты, размещаемые до его начала, не могут быть заблокированы.

Начиная с версии 2.4.21 ядро Linux позволяет передавать аргументу `l_len` отрицательные значения. Это значит, что заблокировать нужно `l_len` байт, размещенных до позиции, заданной с помощью `l_whence` и `l_start` (то есть байты в диапазоне от `(l_start – abs(l_len))` до `(l_start - 1)`). Такая возможность не является обязательной, но допускается стандартом SUSv3. Она также поддерживается в некоторых других реализациях UNIX.

В целом приложения должны блокировать как можно меньший диапазон байтов. Это улучшает параллельную работу других процессов, одновременно пытающихся заблокировать разные участки одного и того же файла.

> Понятие минимального диапазона в отдельных ситуациях необходимо уточнять. Блокировка записей в сочетании с вызовами mmap() может привести к нежелательным последствиям в сетевых файловых системах, таких как NFS и CIFS. Дело в том, что mmap() отображает файлы постранично. Если файловая блокировка выровнена по странице, то никаких проблем не возникает, так как она покроет собой весь участок, соответствующий «грязной» странице. В противном же случае возникает состояние гонки: при изменении любой части отображенной страницы ядро может выполнить запись на участок, не покрытый блокировкой.

Передача аргументу `l_len` значения `0` будет сигнализировать о том, что нужно заблокировать все байты, начиная с позиции, заданной с помощью `l_start` и `l_whence`, и до конца файла, независимо от того, насколько он вырастет. Это удобно в тех случаях, когда мы не знаем наперед, сколько байтов будет добавлено к файлу. Чтобы заблокировать весь файл целиком, аргументу `l_whence` можно передать значение `SEEK_SET`, а аргументам `l_start` и `l_len` присвоить `0`.

51.3.2. Аргумент cmd

При работе с блокировками файлов аргументу `cmd` вызова `fcntl()` можно передать три разных значения. Первые два из них задействуют для установки и снятия блокировок.

- `F_SETLK` — устанавливает (поле `l_type` равно `F_RDLCK` или `F_WRLCK`) или снимает (поле `l_type` равно `F_UNLCK`) блокировку для байтов, заданных с помощью `flockstr`. Если другой процесс удерживает несовместимую блокировку для любой части заданного участка, то вызов `fcntl()` завершается ошибкой `EAGAIN`. В некоторых реализациях UNIX в такой ситуации используется ошибка `EACCES`. Оба варианта допускаются стандартом SUSv3, поэтому портируемые приложения должны проверять оба значения.
- `F_SETLKW` — делает то же самое, что и `F_SETLK`, но при удержании другим процессом несовместимой блокировки для любой части заданного участка вызов блокируется, пока блокировка не будет установлена. Если мы обрабатываем сигналы и не указали при этом флаг `SA_RESTART` (см. раздел 21.5), то операция `F_SETLKW` может быть прервана (что приведет к ее завершению с ошибкой `EINTR`). Данным обстоятельством можно воспользоваться, установив время ожидания для запроса блокировки с помощью вызова `alarm()` или `setitimer()`.

  Стоит отметить: вызов `fcntl()` блокирует либо весь заданный участок целиком, либо ничего. Он не станет блокировать только те байты, которые доступны на текущий момент.

  Оставшаяся операция вызова `fcntl()` используется для определения того, можно ли заблокировать заданный участок:
- `F_GETLK` — проверяет вероятность установки блокировки, заданной с помощью структуры `flockstr`, но этим и ограничивается. Поле `l_type` должно быть равно `F_RDLCK` или `F_WRLCK`. По завершении вызова структура `flockstr` содержит конечный результат — информацию о том, можно или нельзя установить данную блокировку. Если она возможна (то есть

на заданном участке нет несовместимых блокировок), то полю `l_type` присваивается значение `F_UNLCK`, а остальные поля не меняются. При обнаружении на участке файла одной или нескольких несовместимых блокировок структура `flockstr` будет содержать сведения только об *одной* из них (о какой именно — предсказать невозможно), включая ее тип (`l_type`), диапазон байтов (`l_start` и `l_len`; поле `l_whence` всегда равно `SEEK_SET`) и идентификатор процесса, который удерживает блокировку (`l_pid`).

Обратите внимание: использование значения `F_GETLK` в сочетании с последующей операцией `F_SETLK` или `F_SETLKW` может привести к состоянию гонки. На момент выполнения второго вызова данные, возвращенные операцией `F_GETLK`, могут быть неактуальными. Поэтому значение `F_GETLK` не такое полезное, каким кажется на первый взгляд. Даже если эта операция сигнализирует о возможности установки блокировки, то все равно нужно быть готовыми к тому, что операция `F_SETLK` или `F_SETLKW` вернет ошибку.

Библиотека GNU языка C предоставляет собственную реализацию функции lockf(), которая является всего лишь упрощенным интерфейсом к вызову fcntl() (стандарт SUSv3 описывает функцию lockf(), но не уточняет, как она соотносится с fcntl(); однако в большинстве UNIX-систем функция lockf() реализована поверх fcntl()). Вызов вида lockf(fd, operation, size) аналогичен вызову fcntl(), в котором поля l_whence, l_start и l_len равны соответственно SEEK_CUR, 0 и size; то есть функция lockf() блокирует последовательность байтов, начиная с текущего сдвига в файле. Аргумент operation функции lockf() аналогичен аргументу cmd вызова fcntl(), хотя для установки, снятия и проверки наличия блокировок используются другие константы. Функция lockf() устанавливает только эксклюзивные блокировки (позволяющие запись). Больше подробностей см. на странице lockf(3) руководства.

51.3.3. Подробности об установке и снятии блокировок

Стоит отметить следующие моменты относительно установки и снятия блокировок, созданных с помощью вызова `fcntl()`.

- Разблокировка участка файла всегда завершается успешно и без задержек, даже если этот участок не содержал блокировки.
- В любой заданный момент времени процесс может удерживать только одну блокировку, относящуюся к определенному участку файла. Установка новой блокировки для уже заблокированного участка либо ничего не изменит (при совпадении типов новой и существующей блокировок), либо автоматически переведет существующую блокировку в другой режим. Во втором случае, если режим меняется с чтения на запись, нужно быть готовым к тому, что такой вызов может завершиться ошибкой (`F_SETLK`) или заблокироваться (`F_SETLKW`). Этим `fcntl()` отличается от функции `flock()`, в которой преобразование блокировок не является атомарной операцией.
- Процесс не может освободить себя от блокировки участка файла, даже если попытается установить блокировки через разные дескрипторы, ссылающиеся на один и тот же файл (этим вызов `fcntl()` отличается от функции `flock()`; мы еще вернемся к данной проблеме в подразделе 51.3.5).
- Установка блокировки, находящейся в другом режиме, посреди уже заблокированного участка приведет к получению трех разных блокировок: по обе стороны от новой блокировки появляются два участка меньше размера, сохраняющие предыдущий режим (рис. 51.3). И наоборот: получение второй блокировки в том же режиме, которая является смежной с уже существующей или перекрывает ее, приводит к созданию одной объединенной блокировки, покрывающей оба заблокированных участка. Возможны и другие вариации. Например, разблокировка небольшого диапазона посреди заблокированного участка приводит к созданию двух блокировок меньшего размера

по обе стороны от этого диапазона. Если новая блокировка перекрывает существующую и имеет другой режим, то «захватывает» ее байты, уменьшая ее размер.

- Закрытие файлового дескриптора в случае с заблокированными участками файла имеет несколько необычную семантику, которая будет описана в подразделе 51.3.5.

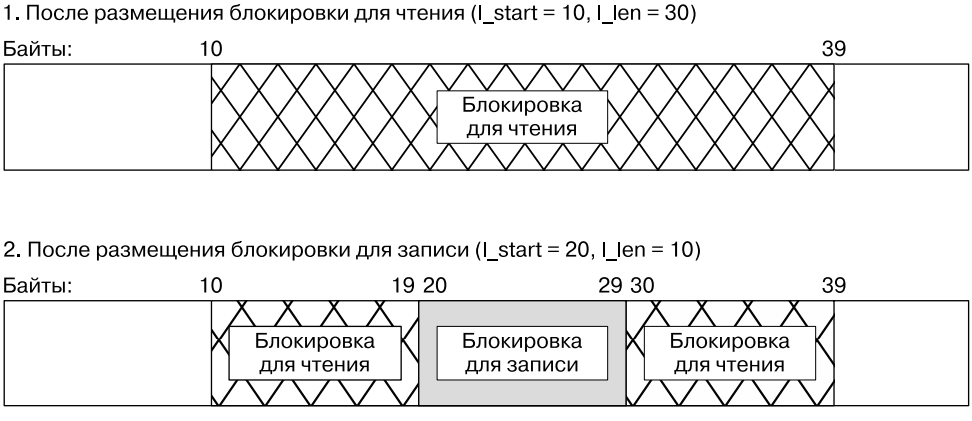


Рис. 51.3. Расщепление существующей блокировки в режиме чтения блокировкой в режиме записи, установленной тем же процессом

51.3.4. Взаимная блокировка

Используя флаг `F_SETLKW`, нужно учитывать возможность сценария, проиллюстрированного на рис. 51.4. В данной ситуации блокируется каждый второй запрос на получение блокировки; так происходит из-за блокировки, установленной другим процессом. Этот сценарий называется *взаимной блокировкой*. Теоретически он может привести к тому, что оба процесса будут заблокированы навсегда. Однако ядро учитывает такую возможность, проверяя каждый новый запрос на получение блокировки, сделанный операцией `F_SETLKW`, убеждаясь в том, что он не приведет к взаимной блокировке. Если опасность существует, то ядро выбирает один из заблокированных процессов и делает так, чтобы его вызов `fcntl()` был разблокирован и завершился с ошибкой `EDEADLK`. (В Linux выбирается процесс, который последним сделал вызов `fcntl()`, но такое поведение не является обязательным с точки зрения стандарта SUSv3 и может отличаться в других реализациях UNIX или будущих версиях Linux. Любой процесс, выполняющий операцию `F_SETLKW`, должен быть готов к обработке ошибки `EDEADLK`.)

Ядро способно обнаружить взаимную блокировку даже в ситуации, когда блокировки применяются к разным файлам, в результате чего процессы блокируются по кругу. (Это можно описать так: процесс А пытается получить блокировку для участка, заблокированного процессом Б; тот, в свою очередь, пытается получить блокировку, удерживаемую процессом В; а процесс В ждет получения блокировки, удерживаемой процессом А.)

51.3.5. Пример: программа для интерактивной блокировки

Программа, представленная в листинге 51.2, является интерактивной и позволяет экспериментировать с блокировкой записей. Она принимает всего один аргумент командной строки — имя файла, который мы хотим заблокировать. С ее помощью можно проверить

многие приведенные выше утверждения относительно того, как работает частичная блокировка. Данная программа предназначена для интерактивного использования и принимает команды следующего вида:

```
cmd lock start length [ whence ]
```

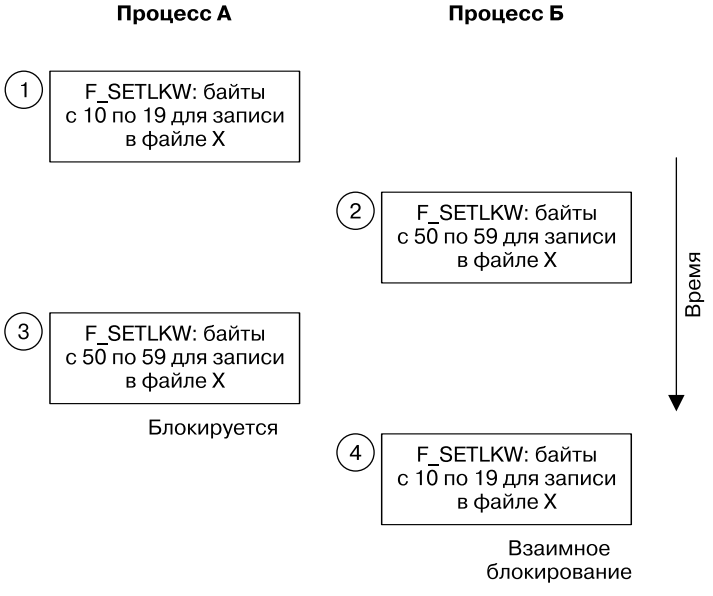


Рис. 51.4. Взаимная блокировка двух процессов, которые не дают друг другу выполнить запрос на получение блокировки

Аргумент `cmd` поддерживает значения `g` (`F_GETLK`), `s` (`F_SETLK`) или `w` (`F_SETLKW`). Остальные аргументы применяются для инициализации структуры `flock`, передаваемой в вызов `fcntl()`. Аргумент `lock` представляет собой значение поля `l_type` и может быть равен `r` (`F_RDLCK`), `w` (`F_WRLCK`) или `u` (`F_WRLCK`). Аргументы `start` и `length` являются целыми числами, представляющими значения полей `l_start` и `l_len`. Последний аргумент, `whence`, является необязательным и представляет значение поля `l_whence`; он может быть равен `s` (`SEEK_SET` — по умолчанию), `c` (`SEEK_CUR`) или `e` (`SEEK_END`). Причина, по которой мы приводим поля `l_start` и `l_len` к типу `long long`, когда передаем их в вызов `printf()`, описана в разделе 5.10.

Листинг 51.2. Эксперимент с блокировкой записей

filelock/i_fcntl_locking.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
#define MAX_LINE 100

static void
displayCmdFmt(void)
{
    printf("\n    Format: cmd lock start length [whence]\n\n");
    printf("     'cmd' is 'g' (GETLK), 's' (SETLK), or 'w' (SETLKW)\n");
    printf("     'lock' is 'r' (READ), 'w' (WRITE), or 'u' (UNLOCK)\n");
```

```
    printf("     'start' and 'length' specify byte range to lock\n");
    printf("     'whence' is 's' (SEEK_SET, default), 'c' (SEEK_CUR), "
            "or 'e' (SEEK_END)\n\n");
}

int
main(int argc, char *argv[])
{
    int fd, numRead, cmd, status;
    char lock, cmdCh, whence, line[MAX_LINE];
    struct flock fl;
    long long len, st;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open (%s)", argv[1]);

    printf("Enter ? for help\n");

    for (;;) {     /* Просим ввести команду блокирования и выполняем ее */
        printf("PID=%ld> ", (long) getpid());
        fflush(stdout);

        if (fgets(line, MAX_LINE, stdin) == NULL)    /* Конец файла */
            exit(EXIT_SUCCESS);
        line[strlen(line) - 1] = '\0';   /* Удаляем '\n' в конце */

        if (*line == '\0')
            continue;                    /* Пропускаем пустые строки */

        if (line[0] == '?') {
            displayCmdFmt();
            continue;
        }

        whence = 's';        /* Значение по умолчанию для 'whence' */
        numRead = sscanf(line, "%c %c %lld %lld %c", &cmdCh, &lock,
                         &st, &len, &whence);
        fl.l_start = st;
        fl.l_len = len;

        if (numRead < 4 || strchr("gsw", cmdCh) == NULL ||
            strchr("rwu", lock) == NULL || strchr("sce", whence) == NULL) {
            printf("Invalid command!\n");
            continue;
        }

        cmd = (cmdCh == 'g') ? F_GETLK : (cmdCh == 's')
            ? F_SETLK : F_SETLKW;
        fl.l_type = (lock == 'r') ? F_RDLCK : (lock == 'w')
            ? F_WRLCK : F_UNLCK;
        fl.l_whence = (whence == 'c') ? SEEK_CUR :
                      (whence == 'e') ? SEEK_END : SEEK_SET;

        status = fcntl(fd, cmd, &fl);   /* Выполняем запрос... */
```

```
        if (cmd == F_GETLK) {           /* ...и смотрим, что получилось */
            if (status == -1) {
                errMsg("fcntl - F_GETLK");
            } else {
                if (fl.l_type == F_UNLCK)
                    printf("[PID=%ld] Lock can be placed\n",
                        (long) getpid());
                else              /* Заблокировано кем-то другим */
                    printf("[PID=%ld] Denied by %s lock on %lld:%lld "
                            "(held by PID %ld)\n", (long) getpid(),
                             (fl.l_type == F_RDLCK) ? "READ" : "WRITE",
                             (long long) fl.l_start,
                             (long long) fl.l_len, (long) fl.l_pid);
            }
        } else {                /* F_SETLK, F_SETLKW */
            if (status == 0)
                printf("[PID=%ld] %s\n", (long) getpid(),
                        (lock == 'u') ? "unlocked" : "got lock");
            else if (errno == EAGAIN || errno == EACCES)    /* F_SETLK */
                printf("[PID=%ld] failed (incompatible lock)\n",
                        (long) getpid());
            else if (errno == EDEADLK)        /* F_SETLKW */
                printf("[PID=%ld] failed (deadlock)\n", (long) getpid());
            else
                errMsg("fcntl - F_SETLK(W)");
        }
    }
}
```

filelock/i_fcntl_locking.c

В сессиях командной строки, представленных ниже, демонстрируется использование двух экземпляров программы из листинга 51.2, с помощью которых устанавливаются блокировки для одного и того же файла размером 100 байт (`tfile`). На рис. 51.5 показано состояние удовлетворенных и отложенных запросов на получение блокировки на разных этапах сессии (см. комментарии ниже).

Для начала запустим первый экземпляр программы из листинга 51.2 (процесс А), установив блокировку для чтения байтов файла с 0 по 39:

```
Terminal window 1
$ ls -l tfile
-rw-r--r-- 1 mtk users                  100 Apr 18 12:19 tfile
$ ./i_fcntl_locking tfile
Enter ? for help
PID=790> s r 0 40
[PID=790] got lock
```

Теперь запустим второй экземпляр программы (процесс Б) и установим блокировку для чтения байтов с 70 и до конца файла:

```
Terminal window 2
$ ./i_fcntl_locking tfile
Enter ? for help
PID=800> s r -30 0 e
[PID=800] got lock
```

К этому моменту все должно выглядеть так, как на рис. 51.5, *а*, где процессы А (ID процесса — 790) и Б (ID процесса — 800) удерживают блокировки для разных участков файла.

Теперь мы возвращаемся к процессу А, где пытаемся добавить блокировку с возможностью записи для всего файла. Сначала мы используем операцию F_GETLK для проверки того, можно ли убрать блокировку и выдать сообщение, что существует конфликтная блокировка. Затем мы пытаемся установить блокировку с помощью операции F_SETLK, которая также не работает. Наконец, мы пытаемся добавить блокировку с F_SETLKW.

```
PID=790> g w 0 0
[PID=790] Denied by READ lock on 70:0 (held by PID 800)
PID=790> s w 0 0
[PID=790] failed (incompatible lock)
PID=790> w w 0 0
```

К данному моменту все должно выглядеть так, как на рис. 51.5, *б*, где процессы А и Б удерживают блокировки для разных участков файла и при этом процесс А имеет отложенный запрос на блокирование всего файла.

Продолжаем работу с процессом Б, пытаясь заблокировать весь файл. Для начала воспользуемся операцией F_GETLK, чтобы узнать, возможно ли установить такую блокировку (то есть нет ли на этом участке конфликтующих блокировок). Затем попытаемся выполнить блокировку с помощью операции F_SETLKW.

```
PID=800> g w 0 0
[PID=800] Denied by READ lock on 0:40
(held by PID 790)
PID=800> w w 0 0
[PID=800] failed (deadlock)
```

На рис. 51.5, *в*, показано, что произошло, когда процесс Б запросил блокировку с возможностью записи для всего файла целиком: взаимная блокировка. На этом этапе ядро выбрало один запрос, который должен завершиться с ошибкой, — в данном случае запрос процесса Б; его вызов fcntl() вернет ошибку EDEADLK.

Продолжаем работать с процессом Б, удаляя все блокировки, удерживаемые им для нашего файла:

```
PID=800> s u 0 0
[PID=800] unlocked
[PID=790] got lock
```

Последняя строчка вывода говорит о том, что запрос процесса А наконец может быть удовлетворен.

Важно понимать: несмотря на отмену запроса, который приводил к взаимной блокировке, процесс Б все еще удерживает другую свою блокировку и запрос на получение блокировки процессом А все еще ожидает своей очереди. Запрос процесса А будет удовлетворен только после того, как процесс Б уберет свою оставшуюся блокировку и сделает актуальной ситуацию, показанную на рис. 51.5, *г*.

51.3.6. Пример: библиотека функций для установки блокировок

В листинге 51.3 собран набор функций для блокировки, пригодный для использования в других программах.

- Функция lockRegion() задействует операцию F_SETLK, чтобы установить блокировку для открытого файла, на который указывает дескриптор fd. Аргумент type определяет тип блокировки (F_RDLCK или F_WRLCK). Аргументы whence, start и len задают диапа-

зон блокируемых байтов; они предоставляют значения для полей структуры `flockstr` (с помощью которой устанавливается блокировка) с аналогичными именами.

- Функция `lockRegionWait()` похожа на `lockRegion()`, но выполняет блокирующий запрос; то есть вместо `F_SETLK` она применяет флаг `F_SETLKW`.
- Функция `regionIsLocked()` проверяет, можно ли установить блокировку для заданного файла. Ее аргументы такие же, как и у функции `lockRegion()`. Она возвращает `0` (`false`), если ни один другой процесс не удерживает блокировку, которая конфликтует с указанным в вызове запросом. Если один или несколько процессов удерживают конфликтующие блокировки, то она возвращает ненулевое значение (то есть `true`) — идентификатор одного из этих процессов.

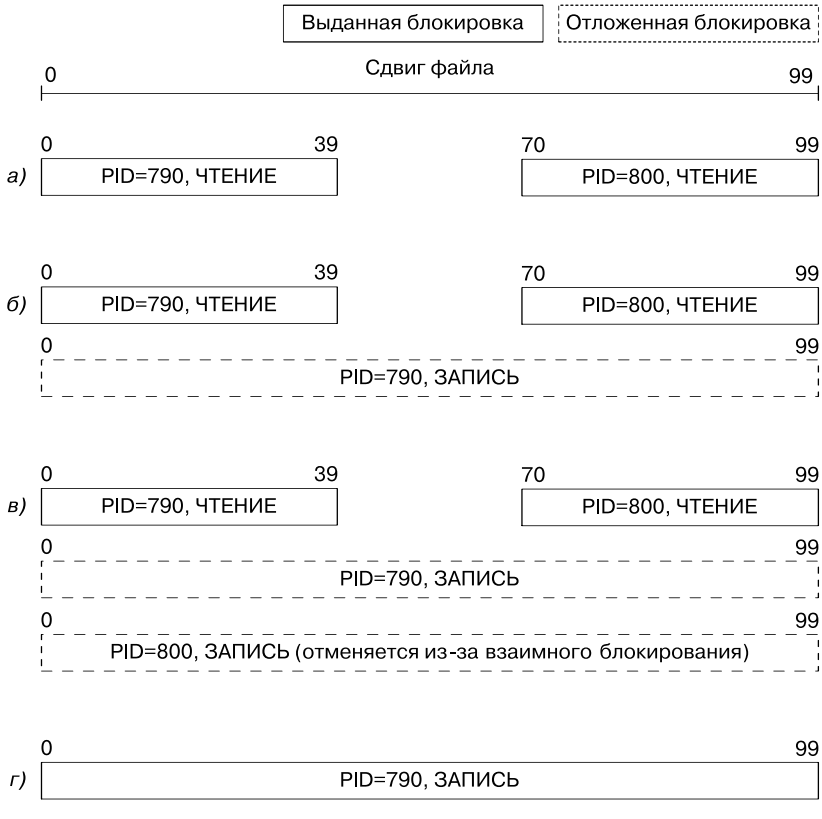


Рис. 51.5. Состояние удовлетворенных и отложенных запросов при выполнении программы i_fcntl_locking.c

Листинг 51.3. Функции для блокировки участков файла

filelock/region_locking.c

```
#include <fcntl.h>
#include "region_locking.h"     /* Объявляет определенные здесь функции */

/* Блокирует участок файла (частная; публичный интерфейс находится ниже) */

static int
lockReg(int fd, int cmd, int type, int whence, int start, off_t len)
```

```
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;
    return fcntl(fd, cmd, &fl);
}

int             /* Устанавливает блокировку для участка файла
                   с помощью неблокирующей операции F_SETLK */
lockRegion(int fd, int type, int whence, int start, int len)
{
    return lockReg(fd, F_SETLK, type, whence, start, len);
}

int             /* Устанавливает блокировку для участка файла
                   с помощью блокирующей операции F_SETLKW */
lockRegionWait(int fd, int type, int whence, int start, int len)
{
    return lockReg(fd, F_SETLKW, type, whence, start, len);
}

/* Проверяет, доступен ли участок файла для блокирования. Если да,
   возвращает 0; если другой процесс удерживает несовместимую блокировку,
   возвращает его PID; в случае ошибки возвращает -1. */

pid_t
regionIsLocked(int fd, int type, int whence, int start, int len)
{
    struct flock fl;

    fl.l_type = type;
    fl.l_whence = whence;
    fl.l_start = start;
    fl.l_len = len;
    if (fcntl(fd, F_GETLK, &fl) == -1)
        return -1;
    return (fl.l_type == F_UNLCK) ? 0 : fl.l_pid;
}
```

filelock/region_locking.c

51.3.7. Производительность блокировок и их ограничения

Стандарт SUSv3 разрешает устанавливать фиксированные, общесистемные ограничения максимального количества блокировок для записей, которые могут быть получены. При его достижении функция `fcntl()` завершается ошибкой `ENOLCK`. В Linux такого ограничения не существует; мы ограничены лишь объемом доступной памяти (похожая ситуация наблюдается и во многих других UNIX-системах).

Вопрос «Насколько быстро можно установить и снять блокировку записи?» не имеет однозначного ответа, поскольку скорость этих операций зависит от структуры, используемой ядром для хранения таких блокировок, и от местоположения нашей блокировки внутри нее. Мы рассмотрим данную структуру чуть ниже, но сначала следует остановиться на требованиях, которые к ней предъявляются:

- ядро должно иметь возможность объединить новую блокировку с любыми существующими (удерживаемыми тем же процессом), если они имеют тот же режим и располагаются по любую сторону от нее;
- новая блокировка может полностью заменить собой одну или несколько существующих блокировок, удерживаемых вызывающим процессом. Все они должны быть доступны, чтобы ядро могло легко их найти;
- при создании новой блокировки посреди существующей, которая имеет другой режим, процедура разделения этой существующей блокировки (см. рис. 51.3) должна быть простой.

Структура данных ядра, используемая для хранения информации о блокировках, спроектирована специально, чтобы удовлетворить эти требования. У каждого открытого файла есть связанный список блокировок, содержимое которого упорядочено по идентификатору процесса, а затем по начальному сдвигу. Пример такого списка показан на рис. 51.6.

В данном списке также хранятся блокировки, созданные с помощью вызова `flock()`, и сведения об аренде открытого файла (мы коснемся темы аренды файлов в разделе 51.5, во время обсуждения файла /proc/locks). Однако блокировки таких типов обычно куда менее многочисленны и, следовательно, шансов повлиять на производительность у них тоже меньше, поэтому мы не станем останавливаться на них отдельно.

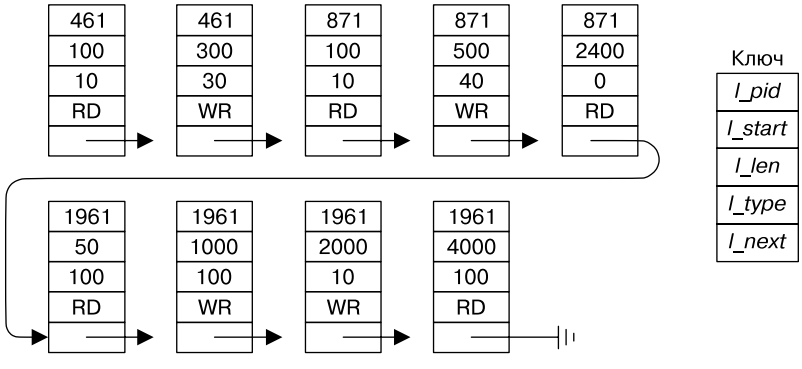


Рис. 51.6. Пример списка блокировок записей для одного файла

Каждый раз при добавлении в эту структуру данных новой блокировки ядро должно проверять наличие конфликтов с любыми существующими блокировками, установленными для того же файла. Поиск конфликтов выполняется последовательно, начиная с первого элемента списка. Большое количество блокировок может быть произвольным образом распределено между множеством процессов, поэтому время, которое уходит на добавление или удаление блокировок, можно считать прямо пропорциональным их общему количеству для конкретного файла.

51.3.8. Семантика наследования и снятия блокировок

Семантика наследования и снятия блокировок, созданных с помощью вызова `fcntl()`, существенно отличается от аналогичной семантики для блокировок на основе `flock()`. Обратите внимание на следующие моменты.

- Блокировки записей не наследуются дочерним процессом после выполнения `fork()`. Для сравнения, в случае с вызовом `flock()` потомок наследует ссылки на блокировки

своего родителя и может их снять; если это произойдет, родитель тоже потеряет данные блокировки.

- Блокировки записей сохраняются на протяжении выполнения вызова `exec()` (при этом стоит учитывать влияние флага `FD_CLOEXEC`, описанное ниже).
- Все потоки одного процесса разделяют один и тот же набор блокировок.
- Блокировки записей связаны как с процессом, так и с индексным дескриптором (см. раздел 5.4). Неудивительно, что при завершении процесса снимаются все его блокировки. Менее очевидное следствие такой связи — *все* блокировки, относящиеся к определенному файлу, снимаются, когда процесс закрывает соответствующий файловый дескриптор; при этом неважно, использовался ли он для получения блокировок. Например, в следующем коде вызов `close(fd2)` снимает блокировку, удерживаемую для файла `testfile` вызывающим процессом, хотя она была установлена через файловый дескриптор `fd1`:

```
struct flock fl;

fl.l_type = F_WRLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 0;
fd1 = open("testfile", O_RDWR);
fd2 = open("testfile", O_RDWR);

if (fcntl(fd1, cmd, &fl) == -1)
    errExit("fcntl");
close(fd2);
```

Семантика, описанная в последнем пункте, действует независимо от способа получения тех или иных дескрипторов, ссылающихся на один и тот же файл, и от того, как они были закрыты. Например, для получения копии дескриптора открытого файла используются вызовы `dup()`, `dup2()` и `fcntl()`. А для его закрытия, помимо вызова `close()`, можно также применить операцию `exec()` с флагом `FD_CLOEXEC` или вызов `dup2()`, который закрывает дескриптор, указанный во втором аргументе, если тот является открытым.

Семантика наследования и снятия блокировок, установленных с помощью вызова `fcntl()`, — архитектурный изъян. Например, она делает проблематичным использование таких блокировок из библиотечных пакетов, так как функции библиотеки не могут исключить возможность того, что вызывающий процесс закроет дескриптор, который ссылается на заблокированный файл, и тем самым удалит установленную ими блокировку. В качестве альтернативы блокировку можно было бы связать с файловым, а не с индексным дескриптором. Но текущая семантика блокировки записей давно является устоявшейся и стандартизированной. К сожалению, это существенно ограничивает применение вызова `fcntl()`.

> В случае с вызовом `flock()` блокировка связывается только с дескриптором открытого файла и продолжает существовать, пока ее вручную не снимут или не будут закрыты все дескрипторы, ссылающиеся на описание открытого файла.

51.3.9. Зависание блокировок и приоритет отложенных запросов на их получение

Когда несколько процессов вынуждены ждать своей очереди, чтобы установить блокировку для участка, уже кем-то заблокированного, возникает несколько вопросов.

Может ли процесс, пытающийся установить блокировку для записи, зависнуть из-за других процессов, которые устанавливают блокировки для чтения того же участка?

В Linux (как и во многих других реализациях UNIX) это действительно может произойти, вызывая зависание блокировки для записи (возможно, даже навсегда).

Существуют ли какие-либо правила очередности получения блокировок в ситуации, когда их пытаются установить сразу несколько процессов? Например, выдаются ли они по принципу «первым пришел — первым ушел»? И зависят ли эти правила от типа блокировки, запрашиваемой каждым из процессов (то есть имеют ли блокировки для чтения приоритет перед блокировками для записи или наоборот)? В Linux действуют такие правила:

- порядок, в котором удовлетворяются отложенные запросы, является неопределенным. Если несколько процессов ожидают своей очереди для установки блокировки, порядок их выполнения зависит от планировщика ядра;
- записывающие блокировки не имеют приоритета перед считывающими и наоборот.

Эти утверждения могут оказаться неактуальными в других системах. В некоторых реализациях UNIX запросы на получение блокировок обслуживаются по принципу «первым пришел — первым ушел», а чтение имеет более высокий приоритет по сравнению с записью.

51.4. Строгая блокировка

Блокировки, которые мы обсуждали до сих пор, являлись *необязательными*. Это значит, что процесс может выполнять обычный ввод/вывод, не прибегая к вызову `fcntl()` (или `flock()`). Ядро не сможет помешать возникновению подобной ситуации. При использовании необязательных блокировок приложение само должно:

- установить для файла подходящего владельца (или группу) и права доступа, чтобы не дать неизвестному процессу выполнять для файла ввод/вывод;
- убедиться в том, что процессы, из которых состоит приложение, взаимодействуют друг с другом, получая перед выполнением ввода/вывода подходящие блокировки.

Linux, как и многие другие реализации UNIX, позволяет делать блокировки записей, создаваемые вызовом `fcntl()`, *строгими*. Это значит, что каждая операция ввода/вывода проверяется на совместимость с любыми блокировками, удерживаемыми другими процессами для того же участка файла.

> Режим необязательной блокировки иногда называют дискреционным, а строгий режим — принуждающим. Строгая блокировка не предусмотрена стандартом SUSv3, но доступна в большинстве современных UNIX-систем (с рядом отличий в реализации).

Чтобы задействовать строгую блокировку в Linux, ее нужно сначала включить на уровне файловой системы, в которой хранятся интересующие файлы. Для этого систему следует подключить, используя параметр `-o mand` (доступный только в Linux):

```
# mount -o mand /dev/sda10 /testfs
```

Внутри программы того же результата можно добиться с помощью флага `MS_MANDLOCK` при вызове `mount(2)` (см. подраздел 14.8.1).

Чтобы проверить, подключена ли файловая система с поддержкой строгой блокировки, можно проанализировать вывод команды `mount(8)`, запущенной без параметров:

```
# mount | grep sda10
/dev/sda10 on /testfs type ext3 (rw,mand)
```

Строгая блокировка файла становится возможной, когда бит установки группового идентификатора включен, а групповое выполнение, наоборот, выключено. В старых

UNIX-системах такое сочетание битов с правами доступа ничего не значило и попросту игнорировалось. Это позволило включать строгую блокировку без необходимости переписывания существующих программ или добавления новых системных вызовов. В командной оболочке данный вид блокировки можно включить следующим образом:

```
$ chmod g+s,g-x /testfs/file
```

Чтобы сделать то же самое в программе, файлу нужно назначить походящие права доступа, воспользовавшись вызовом `hmod()` или `fchmod()` (см. подраздел 15.4.7).

При выполнении команды `ls(1)` для файла, для которого была включена строгая блокировка, в столбце с правами доступа, касающимися группового выполнения, будет указано значение `S`:

```
$ ls -l /testfs/file
-rw-r-Sr--    1 mtk      users      0 Apr 22 14:11 /testfs/file
```

Строгая блокировка поддерживается всеми стандартными системами в Linux и UNIX, однако ее сопровождение может отсутствовать в некоторых сетевых файловых системах или системах, не относящихся к семейству UNIX. Например, система VFAT компании Microsoft не поддерживает бит установки группового идентификатора, поэтому строгую блокировку в ней использовать нельзя.

Влияние строгой блокировки на операции ввода/вывода

Допустим, для файла включена строгая блокировка. Что произойдет, если системный вызов, который передает данные (то есть выполняет операции вроде `read()` или `write()`), сталкивается с конфликтующей блокировкой (то есть пытается записать или прочитать данные с участка, заблокированного для чтения или записи)? Ответ зависит от того, в каком режиме был открыт файл — блокирующем или неблокирующем. В первом случае системный вызов блокируется, а во втором — немедленно завершается ошибкой `EAGAIN`. По похожему принципу работают вызовы `truncate()` и `ftruncate()`, когда пытаются добавить или удалить данные с участка, покрытого блокировкой (для чтения или записи), принадлежащей другому процессу.

При открытии файла в блокирующем режиме (то есть если в вызове `open()` не был указан флаг `O_NONBLOCK`) системные вызовы для ввода/вывода могут привести к взаимной блокировке. Взгляните на пример, представленный на рис. 51.7: два процесса, открывшие один и тот же файл в блокирующем режиме, получают блокировки для записи на разные участки этого файла и затем каждый из них пытается выполнить запись на участок, заблокированный другим процессом. Ядро выходит из данной ситуации так же, как в случае с двумя вызовами `fcntl()`, блокирующими друг друга (см. подраздел 51.3.1): выбирает один заблокированный процесс и делает так, чтобы его операция `write()` завершилась ошибкой `EDEADLK`.

Любая попытка открыть файл с флагом `O_TRUNC` сразу же завершается неудачей (ошибкой `EAGAIN`), если в любой части того же файла другой процесс удерживает блокировку для чтения или записи.

Если к *любой* части файла применяется строгая блокировка для чтения или записи, то для него невозможно создать разделяемое отображение в память (то есть выполнить `mmap()` с флагом `MAP_SHARED`). И наоборот: если файл каким-либо образом используется в разделяемом отображении, то строгую блокировку невозможно применить *ни к одной* его части. В обоих случаях соответствующий системный вызов немедленно завершается ошибкой `EAGAIN`. Причина этого ограничения становится понятной, если проанализировать то, как реализованы отображения в память. В подразделе 45.4.2 мы увидели, что разделяемые файловые отображения применяют файл как для чтения, так и для записи (в частности, запись конфликтует с любым видом блокировки для того же файла). Кроме того, данный

ввод/вывод выполняется подсистемой управления памятью, не имеющей ни малейшего представления о местоположении каких-либо файловых блокировок. Следовательно, чтобы не дать отображению обновить файл, для которого установлена строгая блокировка, ядро производит простую проверку: в момент вызова `mmap()` оно ищет любые блокировки в файле, подлежащем отображению (обратная процедура выполняется для вызова `fcntl()`).

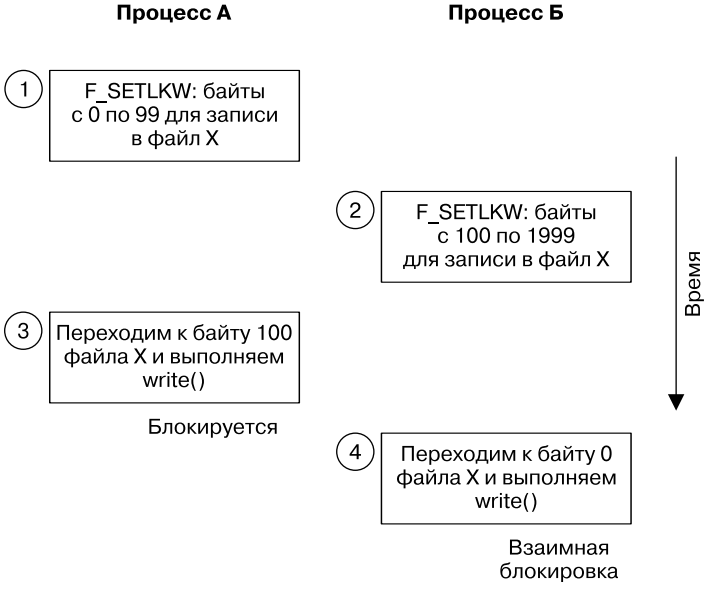


Рис. 51.7. Взаимная блокировка в случае поддержки строгих блокировок

Неочевидные аспекты строгой блокировки

Строгая блокировка не настолько удобна, как можно было бы ожидать. К тому же она имеет ряд недостатков.

- Файл, для которого удерживается строгая блокировка, может быть удален другими процессами, поскольку для этого нужно всего лишь нужно обладать подходящими правами доступа к его родительскому каталогу.
- Прежде чем применять строгую блокировку к публично доступному файлу, стоит тщательно все обдумать, так как ее нельзя переопределить даже привилегированным процессом. Злоумышленник может совершить DoS-атаку, постоянно удерживая блокировку (в большинстве случаев файл можно снова сделать доступным, отключив бит установки группового идентификатора, но иногда это не представляется возможным — например, если строгая блокировка привела к зависанию системы).
- Использование строгой блокировки влечет за собой снижение производительности. При каждой операции ввода/вывода, направленной на файл с поддержкой таких блокировок, ядро вынуждено проверять наличие конфликтов. Если файл содержит большое количество блокировок, то эта проверка может существенно замедлить системные вызовы ввода/вывода.
- Строгая блокировка также влияет на архитектуру приложения. Следует учитывать, что любая операция ввода/вывода может вернуть ошибку `EAGAIN` (для неблокирующего ввода/вывода) или `EDEADLK` (для блокирующего ввода/вывода).

- Ряд случаев состояния гонки в текущей реализации ядра Linux приводит к тому, что системные вызовы, выполняющие ввод/вывод, в отдельных ситуациях могут завершиться успешно даже при наличии строгих блокировок, которые должны были бы их отклонить.

  В целом применения строгих блокировок лучше избегать.

51.5. Файл /proc/locks

Список блокировок, удерживаемых в системе, можно просмотреть в файле `/proc/locks` (доступном только в Linux). Ниже показан пример его содержимого (в данном случае в нем находятся сведения о четырех блокировках):

```
$ cat /proc/locks
1: POSIX  ADVISORY  WRITE 458 03:07:133880 0 EOF
2: FLOCK  ADVISORY  WRITE 404 03:07:133875 0 EOF
3: POSIX  ADVISORY  WRITE 312 03:07:133853 0 EOF
4: FLOCK  ADVISORY  WRITE 274 03:07:81908 0 EOF
```

Файл `/proc/locks` хранит информацию о блокировках, созданных с применением вызовов `flock()` и `fcntl()`. Каждая запись состоит из восьми полей (слева направо).

1. Порядковый номер в наборе блокировок, удерживаемых для заданного файла (см. подраздел 51.3.4).
2. Тип блокировки. Значения `FLOCK` указывают на блокировки, созданные с помощью вызова `flock()`, а `POSIX` — на те, что созданы с использованием `fcntl()`.
3. Режим блокировки: `ADVISORY` или `MANDATORY`.
4. Тип блокировки: `READ` или `WRITE` (относится к разделяемым и эксклюзивным блокировкам на основе вызова `fcntl()`).
5. Идентификатор процесса, удерживающего блокировку.
6. Три числа, разделенные двоеточиями, обозначают файл, для которого удерживается блокировка. Это мажорные и минорные номера устройства в текущей файловой системе, а также индексный дескриптор файла.
7. Начальный байт блокировки. В случае с блокировками на основе вызова `flock()` данный столбец всегда равен 0.
8. Конечный байт блокировки. Значение `EOF` показывает, что блокировка доходит до самого конца файла (то есть при вызове `fcntl()` поле `l_len` было равно 0). В случае с блокировками, основанными на `flock()`, этот столбец всегда равен `EOF`.

С помощью информации из файла `/proc/locks` можно определить, какими процессами и для каких файлов удерживаются те или иные блокировки. То, как это делается, показано в следующей сессии командной строки на примере блокировки 3 из списка, приведенного выше. Эта блокировка удерживается процессом с идентификатором 312 для индексного дескриптора 133853 на устройстве, чьи мажорный и минорный идентификаторы равны 3 и 7. Для начала воспользуемся командой `ps(1)`, чтобы вывести сведения о процессе, чей PID равен 312:

```
$ ps -p 312
PID TTY          TIME CMD
312 ?        00:00:00 atd
```

Как видите, программой, удерживающей блокировку, оказался демон `atd`, который выполняет пакетные задания по расписанию.

Чтобы найти заблокированный файл, поищем для начала в каталоге `/dev` устройство с идентификатором 3:7. Им оказался файл `/dev/sda7`:

```
$ ls -li /dev/sda7 | awk '$6 == "3," && $7 == 10'
  1311 brw-rw----    1 root   disk    3,  7 May 12 2006 /dev/sda7
```

Теперь определим точку подключения устройства `/dev/sda7` и найдем ту часть файловой системы, которой принадлежит индексный дескриптор с номером 133853:

```
$ mount | grep sda7
/dev/sda7 on / type reiserfs (rw)   Устройство подключено к каталогу /
$ su                                Поэтому поиск выполняется по всем каталогам
Password:
# find / -mount -inum 133853        Ищем индексный дескриптор 133853
/var/run/atd.pid
```

Параметр `-mount` не дает команде `find` заходить в подкаталоги, к которым подключены другие файловые системы.

В завершение выведем содержимое заблокированного файла:

```
# cat /var/run/atd.pid
312
```

Демон `atd` удерживает блокировку для файла `/var/run/atd.pid`, содержащего ID процесса, в котором этот демон выполняется. Это делается для того, чтобы не дать запустить больше одного экземпляра программы `atd`. Такой подход будет рассмотрен в разделе 51.6.

Файл `/proc/locks` также позволяет получить сведения об отложенных запросах на получение блокировок:

```
$ cat /proc/locks
1: POSIX ADVISORY WRITE 11073 03:07:436283 100 109
1: -> POSIX ADVISORY WRITE 11152 03:07:436283 100 109
2: POSIX MANDATORY WRITE 11014 03:07:436283 0 9
2: -> POSIX MANDATORY WRITE 11024 03:07:436283 0 9
2: -> POSIX MANDATORY READ 11122 03:07:436283 0 19
3: FLOCK ADVISORY WRITE 10802 03:07:134447 0 EOF
3: -> FLOCK ADVISORY WRITE 10840 03:07:134447 0 EOF
```

Строки, где сразу после номера блокировки указаны символы `->`, представляют собой запросы на получение блокировок с соответствующими номерами. Здесь мы имеем один отложенный запрос на получение блокировки 1 (необязательной, созданной вызовом `fcntl()`), два отложенных запроса на получение блокировки 2 (строгой, созданной вызовом `fcntl()`) и один отложенный запрос на получение блокировки 3 (созданной с помощью `flock()`).

> Файл /proc/locks также содержит информацию об аренде файлов, установленной любыми процессами в системе. Аренда файлов — уникальный для Linux механизм, доступный в ядрах версии 2.4 и выше. Когда процесс арендует файл, он получает уведомления (на основе сигналов), если другие процессы пытаются выполнить для данного файла операции open() или truncate() (последняя учитывается, так как это единственный системный вызов, способный изменить содержимое файла, не открывая его). Механизм аренды файлов предоставляется для поддержки уступающих блокировок, применяющихся в протоколе SMB, и для аналогичной системы в протоколе NFS, которая называется делегированием. Больше подробностей об аренде файлов содержится в описании операции F_SETLEASE на странице fcntl(2) руководства.

51.6. Выполнение только одного экземпляра программы

Некоторые программы (в частности, многие демоны) должны следить за тем, чтобы они были запущены в единственном экземпляре. Обычно это достигается следующим образом:

демон создает файл в стандартном каталоге и применяет к нему блокировку для записи. Он удерживает ее на протяжении всего своего существования и удаляет прямо перед завершением. Если попытаться запустить другой экземпляр того же демона, то он не сможет получить блокировку для соответствующего файла и автоматически завершится, понимая: один его экземпляр уже выполняется в системе.

> Многие сетевые серверы используют другой принцип; чтобы узнать, выполняется ли в системе другой экземпляр сервера, они проверяют, занят ли их стандартный порт (см. раздел 57.10).

Для хранения таких файлов обычно применяется каталог `/var/run`. Как вариант местоположение файла может определяться конфигурацией демона.

Обычно демон записывает в заблокированный файл идентификатор своего процесса, так что в качестве расширения файла часто используется `.pid` (например, демон `syslogd` создает файл `/var/run/syslogd.pid`). Это удобно, если нужно найти PID демона, а также обеспечивает дополнительную меру предосторожности: можно проверить, существует ли процесс с таким идентификатором, воспользовавшись вызовом `kill(pid, 0)`, как показано в разделе 20.5. (В старых реализациях UNIX, которые не поддерживали блокирование файлов, данный подход применялся в качестве неидеального, но достаточно практичного способа определения, действительно ли предыдущий экземпляр демона все еще работает, или он просто не сумел удалить этот файл перед завершением.)

Процедура создания и блокировки файла с идентификатором процесса может иметь множество мелких вариаций. Листинг 51.4 основан на принципе, описанном в книге [Stevens, 1999], и предоставляет функцию `createPidFile()`, которая инкапсулирует вышеописанные действия. Вызов данной функции в общем случае выглядит так:

```
if (createPidFile("mydaemon", "/var/run/mydaemon.pid", 0) == -1)
    errExit("createPidFile");
```

В функции `createPidFile()` есть один неочевидный нюанс: использование вызова `ftruncate()` для удаления любой строки, которая могла находиться в файле до того. Это делается на случай, если предыдущий экземпляр демона не смог удалить файл — возможно, в результате системного сбоя. При возникновении такой ситуации, если идентификатор нового процесса слишком маленький, можно не полностью перезаписать старое содержимое файла. Например, если идентификатор равен 789, мы запишем в файл строку `789\n`, но предыдущий экземпляр демона мог записать значение `12345\n`. Без предварительного усечения файла результат выглядел бы как `789\n5\n`. Иногда удаление любой существующей строки может и не понадобиться, но такой подход более аккуратен и исключает любую путаницу.

Аргументу `flags` можно передать константу `CPF_CLOEXEC`, заставляющую вызов `createPidFile()` установить для файлового дескриптора флаг `FD_CLOEXEC` (см. раздел 27.4). Это может пригодиться в серверных программах, которые перезапускают себя с помощью вызова `exec()`. Если не закрыть дескриптор во время данного вызова, то перезапущенный сервер будет считать, что в системе уже выполняется один его экземпляр.

Листинг 51.4. Создание PID-файла, который дает запустить только один экземпляр программы
filelock/create_pid_file.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include "region_locking.h"   /* Для lockRegion() */
#include "create_pid_file.h"  /* Обьявляем createPidFile() и определяем CPF_CLOEXEC */
#include "tlpi_hdr.h"

#define BUF_SIZE 100          /* Достаточно большой для хранения PID в виде строки */
```

```
/* Открываем/создаем файл с именем 'pidFile', блокируем его, при необходимости
   устанавливаем флаг FD_CLOEXEC для его дескриптора, записываем в него PID и (если
   вызывающий процесс в этом заинтересован) возвращаем дескриптор, ссылающийся
   на этот файл. Вызывающий процесс отвечает за удаление файла 'pidFile' прямо
   перед завершением работы. Аргумент 'progName' должен содержать имя вызывающей
   программы (argv[0] или нечто похожее); используется исключительно в диагностических
   целях. Если мы не можем открыть файл 'pidFile' или сталкиваемся с какой-то другой
   ошибкой, выводим в терминале соответствующее диагностическое сообщение. */

int
createPidFile(const char *progName, const char *pidFile, int flags)
{
    int fd;
    char buf[BUF_SIZE];

    fd = open(pidFile, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("Could not open PID file %s", pidFile);

    if (flags & CPF_CLOEXEC) {

        /* Устанавливаем файловому дескриптору флаг FD_CLOEXEC */

        flags = fcntl(fd, F_GETFD);               /* Извлекаем флаги */
        if (flags == -1)
            errExit("Could not get flags for PID file %s", pidFile);

        flags |= FD_CLOEXEC;                      /* Включаем FD_CLOEXEC */

        if (fcntl(fd, F_SETFD, flags) == -1)      /* Обновляем флаги */
            errExit("Could not set flags for PID file %s", pidFile);
    }

    if (lockRegion(fd, F_WRLCK, SEEK_SET, 0, 0) == -1) {
        if (errno == EAGAIN || errno == EACCES)
            fatal("PID file '%s' is locked; probably "
                    "'%s' is already running", pidFile, progName);
        else
            errExit("Unable to lock PID file '%s'", pidFile);
    }

    if (ftruncate(fd, 0) == -1)
        errExit("Could not truncate PID file '%s'", pidFile);
    snprintf(buf, BUF_SIZE, "%ld\n", (long) getpid());
    if (write(fd, buf, strlen(buf)) != strlen(buf))
        fatal("Writing to PID file '%s'", pidFile);
    return fd;
}
```

filelock/create_pid_file.c

51.7. Устаревшие способы блокировки

Старые реализации UNIX, не поддерживавшие файловых блокировок, использовали *специальные* методики блокировки. И хотя после появления вызова `fcntl()` необходимость в них полностью отпала, мы рассмотрим их в данном разделе, поскольку они все еще встречаются в некоторых старых программах. Все они, по сути, носят рекомендательный характер.

open(file, O_CREAT | O_EXCL,...) плюс unlink(file)

Стандарт SUSv3 требует, чтобы вызов `open()` с флагами `O_CREAT` и `O_EXCL` выполнял поэтапную проверку наличия файла и в случае его отсутствия автоматически его создавал (см. раздел 5.1). То есть если два процесса проверяют, существует ли файл, и используют при этом данные флаги, то только один из них выполнит данную операцию успешно (другой процесс получит ошибку `EEXIST` из вызова `open()`). Такой подход в сочетании с системным вызовом `unlink()` создает основу для механизма блокировки. Ее получение выполняется путем успешного открытия файла с помощью флагов `O_CREAT` и `O_EXCL`, за которым сразу же следует вызов `close()`. Снятие блокировки заключается в выполнении вызова `unlink()`. Несмотря на свою практичность, эта методика имеет несколько ограничений.

- Если вызов `open()` завершается неудачно, сигнализируя о том, что блокировкой владеет другой процесс, нужно повторять процедуру в каком-то подобии цикла: либо непрерывно (это будет нагружать процессор), либо с интервалами (это приведет к задержке между снятием блокировки и моментом ее получения). Вызов `fcntl()` позволяет указать флаг `F_SETLKW`, чтобы заблокировать процесс до тех пор, пока блокировка не освободится.
- Получение и снятие блокировок с помощью вызовов `open()` и `unlink()` подразумевает выполнение операций файловой системы, которые уступают в скорости блокировкам записей. (Например, одна из систем автора этой книги, имеющая архитектуру x86-32 и работающая под управлением Linux 2.6.31, тратит на получение и снятие одного миллиона блокировок в файловой системе `ext3` 44 секунды; блокировки записей в тех же условиях уменьшают время получения и снятия до 2,5 секунды.)
- Если процесс случайно завершится, не удалив за собой заблокированный файл, то блокировка не будет снята. Для устранения этой проблемы существуют специальные методики (например, проверив время последнего изменения файла и идентификатор процесса, можно узнать, работает ли данный процесс до сих пор), но ни одна из них не обеспечивает абсолютную надежность. Для сравнения, блокировки записей снимаются автоматически при завершении процесса.
- При размещении нескольких блокировок (то есть когда используется больше одного заблокированного файла) взаимная блокировка не определяется автоматически. Если процессы заблокируют друг друга, то уже не смогут выйти из этого состояния (каждый из них зависнет, проверяя доступность нужной ему блокировки). Для сравнения, в случае с блокировками записей, созданных с помощью вызова `fcntl()`, взаимную блокировку определяет ядро.
- Файловая система NFS версии 2 не поддерживает семантику флага `O_EXCL`. В Linux 2.4 NFS-клиенты тоже не имеют корректной поддержки этого флага, даже в NFS 3 и выше.

link(file, lockfile) плюс unlink(lockfile)

Тот факт, что системный вызов `link()` завершается ошибкой, если заданная ссылка уже существует, тоже использовался в качестве механизма блокировки (и тоже в сочетании с вызовом `unlink()`, выполнявшим процедуру снятия блокировки). Данный подход состоял в следующем: каждый процесс, которому нужно получить блокировку, создает временный файл с уникальным именем, содержащим, как правило, идентификатор процесса (а также имя компьютера, если заблокированный файл создается в сетевой файловой системе). Для получения блокировки в заранее оговоренном каталоге создается ссылка на этот временный файл (семантика жестких ссылок требует нахождения обоих путей в одной и той же файловой системе). Успешное выполнение вызова `link()` сигнализирует о получении блокировки. Возникшая ошибка (`EEXIST`) говорит о том, что блокировка удерживается другим процессом и следует повторить попытку чуть позже. Данный подход имеет те же недостатки, что и применение вызова `open(file, O_CREAT | O_EXCL,...)`, описанное выше.

open(file, O_CREAT | O_TRUNC | O_WRONLY, 0) плюс unlink(file)

Тот факт, что при существовании заданного файла вызов `open()` с флагом `O_TRUNC` завершается неудачей, тоже брался за основу механизма блокировки. Для получения блокировки используется следующий код для создания нового файла (без проверки на ошибки):

```
fd = open(file, O_CREAT | O_TRUNC | O_WRONLY, (mode_t) 0);
close(fd);
```

Успешное выполнение вызова `open()` (если заданного файла не существует) сигнализирует о получении блокировки. Возникшая ошибка (`EACCES`) говорит о том, что блокировка удерживается другим процессом и следует повторить попытку позже. Кроме ограничений, характерных для ранее описанных методик, такой подход имеет еще один недостаток: его нельзя использовать в программах с привилегиями администратора, так как вызов `open()` всегда будет завершаться успешно, независимо от прав доступа, установленных для файла.

51.8. Резюме

Блокировки позволяют процессам синхронизировать свои действия с файлами. Linux предоставляет два системных вызова для управления блокировками: `flock()` (происходит из BSD-систем) и `fcntl()` (происходит из System V). И хотя оба они доступны в большинстве реализаций UNIX, только `fcntl()` входит в стандарт SUSv3.

Системный вызов `flock()` блокирует весь файл целиком. Он может устанавливать блокировки двух типов: разделяемые (совместимые с разделяемыми блокировками, которые удерживаются другими процессами) и эксклюзивные (не дающие другим процессам устанавливать блокировки любых типов).

Системный вызов `fcntl()` блокирует участок файла, размер которого может варьироваться от одного байта до объема всего файла (блокировка записей). Можно устанавливать блокировки двух типов: для чтения и для записи; по своей семантике они похожи на разделяемые и эксклюзивные блокировки, устанавливаемые вызовом `flock()`. Если блокирующий запрос на получение блокировки (`F_SETLKW`) приводит к взаимной блокировке, то ядро выбирает один из вовлеченных процессов и завершает его вызов `fcntl()` с ошибкой `EDEADLK`.

Блокировки, установленные с помощью вызовов `flock()` и `fcntl()`, не знают друг о друге (если только первый не реализован на основе второго, что встречается в некоторых системах). Они имеют разную семантику касательно наследования при вызове `fork()` и снятия при закрытии файловых дескрипторов.

Файл `/proc/locks` (доступный только в Linux) содержит список блокировок, удерживаемых всеми процессами в системе.

Дополнительная информация

Подробное обсуждение блокировки записей вызовом `fcntl()` можно найти в книгах [Stevens & Rago, 2005] и [Stevens, 1999]. В [Bovet & Cesati, 2005] описаны детали реализации вызовов `flock()` и `fcntl()`. [Tanenbaum, 2007] и [Deitel et al., 2004] содержат описание понятия взаимной блокировки, включая способы ее обнаружения, обхода и предотвращения.

51.9. Упражнения

51.1. Поэкспериментируйте с запуском нескольких экземпляров программы из листинга 51.1 (`t_flock.c`), чтобы определить следующие аспекты работы вызова `flock()`.

- Может ли процесс, пытающийся установить эксклюзивную блокировку для файла, зависнуть из-за группы других процессов, устанавливающих для того же файла разделяемые блокировки?
- Представьте: к файлу применена эксклюзивная блокировка и другие процессы ждут своей очереди, чтобы применить к этому файлу разделяемые и эксклюзивные блокировки. Существуют ли какие-то правила, определяющие, какой процесс получит следующую блокировку при снятии предыдущей? Например, имеют ли разделяемые блокировки приоритет перед эксклюзивными или наоборот? Применяются ли они по принципу «первым пришел — первым ушел»?
- Если у вас есть доступ к какой-то другой UNIX-системе, поддерживающей вызов `flock()`, то попробуйте определить, какие правила в ней действуют.

51.2. Напишите программу, которая определяет, поддерживает ли вызов `flock()` обнаружение взаимной блокировки в ситуациях, когда с ее помощью блокируются два разных файла из двух разных процессов.

51.3. Напишите программу, проверяющую утверждение относительно семантики наследования и снятия блокировок, приведенное в подразделе 51.2.1.

51.4. Поэкспериментируйте с запуском программ из листингов 51.1 (`t_flock.c`) и 51.2 (`i_fcntl_locking.c`), чтобы понять, влияют ли друг на друга блокировки, установленные с помощью вызовов `flock()` и `fcntl()`. Если у вас есть доступ к другой реализации UNIX, то проведите тот же эксперимент и в ней.

51.5. В подразделе 51.3.4 отмечалось: в Linux время, требуемое для добавления или проверки наличия блокировки, зависит от местоположения блокировки в списке, относящемся к конкретному файлу. Чтобы проверить это, напишите две программы.

- Первая программа должна применить к файлу, скажем, 40 001 блокировку. Все они должны быть установлены через один байт, то есть 0, 2, 4, 6 и т. д., вплоть до, скажем, 80 000. Выполнив данную процедуру, процесс должен заснуть.
- Пока первая программа бездействует, вторая должна выполнить в цикле, скажем, 10 000 итераций, пытаясь с помощью операции `F_SETLK` заблокировать один из байтов, блокировка для которого была установлена в предыдущем шаге (эти попытки будут неизменно завершаться неудачей). При каждом выполнении программа должна пытаться заблокировать `N * 2` байт файла.

Используя встроенную команду `time`, измерьте время выполнения второй программы при `N`, равном 0, 10 000, 20 000, 30 000 и 40 000. Соответствуют ли результаты ожидаемому линейному поведению?

51.6. Поэкспериментируйте с программой из листинга 51.2 (`i_fcntl_locking.c`), чтобы проверить утверждения, сделанные в подразделе 51.3.6 относительно зависания блокировок и приоритета блокировки записей вызовом `fcntl()`.

51.7. Если у вас есть доступ к другим реализациям UNIX, задействуйте программу из листинга 51.2 (`i_fcntl_locking.c`). Проверьте, сможете ли определить правила блокировки записей вызовом `fcntl()`, касающиеся блокировки записывающих процессов и порядка, в котором удовлетворяются множественные запросы на получение блокировок.

51.8. Используйте программу из листинга 51.2 (`i_fcntl_locking.c`) для демонстрации того, что ядро обнаруживает циклическую взаимную блокировку на основе трех (или более) процессов, блокирующих один и тот же файл.

51.9. Напишите две программы (или одну, применяющую дочерний процесс), чтобы воспроизвести сценарий взаимной блокировки с помощью строгих блокировок, описанных в разделе 51.4.

51.10. Прочитайте справочную страницу утилиты `lockfile(1)`, которая поставляется вместе с приложением `procmail`. Напишите ее упрощенную версию.

52 Сокеты: введение

Сокеты — это механизм межпроцессного взаимодействия, который позволяет обмениваться данными между приложениями, выполняемыми как локально, так и на разных компьютерах, соединенных по сети. Первая широко распространенная реализация программного интерфейса сокетов появилась в 4.2BSD в 1983 году и с тех пор была перенесена практически во все UNIX-системы и большинство систем других семейств.

> Программный интерфейс сокетов формально описан в стандарте POSIX.1g, который был утвержден в 2000 году после примерно десяти лет рассмотрения. Позже ему на смену пришла спецификация SUSv3.

Эта и следующие главы описывают различные аспекты использования сокетов.

- Данная глава представит общие принципы программного интерфейса сокетов — фундамент, который вам потребуется при чтении остального материала. Здесь вы не найдете никаких примеров кода. Практические аспекты применения сокетов в UNIX- и интернет-доменах будут представлены позже.
- Глава 53 посвящена сокетам домена UNIX, позволяющим взаимодействовать приложениям в рамках одной системы.
- Глава 54 познакомит с различными концепциями компьютерных сетей и ключевыми возможностями сетевых протоколов TCP/IP. Применение этим знаниям вы найдете в следующих главах.
- Глава 55 описывает сокеты интернет-доменов, которые позволяют приложениям, находящимся на разных компьютерах, взаимодействовать по сети TCP/IP.
- В главе 56 мы обсудим архитектуру серверов, использующих сокеты.
- Глава 57 охватывает различные продвинутые темы, включая дополнительные возможности ввода/вывода сокетов, более подробный взгляд на протокол TCP и применение параметров сокета для получения и изменения его атрибутов.

Все указанные главы нацелены на то, чтобы помочь читателю хорошо подготовиться к работе с сокетами. Данная тема (особенно сетевое взаимодействие) сама по себе является огромным разделом в программировании, которому посвящены целые книги. Источники информации для дальнейшего изучения сокетов перечислены в разделе 55.15.

52.1. Краткий обзор

При использовании клиент-серверной архитектуры взаимодействие клиентов с помощью сокетов происходит следующим образом:

- каждое приложение создает сокет — «устройство», позволяющее им общаться друг с другом. Каждая из сторон должна иметь собственный сокет;
- сервер привязывает свой сокет к общеизвестному адресу (имени), чтобы клиенты могли его найти.

Сокет создается с применением системного вызова `socket()`; вся дальнейшая работа с сокетом выполняется с помощью дескриптора, возвращенного этим вызовом:

```
fd = socket(domain, type, protocol);
```

В следующих подразделах мы рассмотрим сокеты доменов и их виды. Во всех приложениях, которые приводятся в данной книге, аргумент `protocol` всегда равен 0.

Домены взаимодействия

Сокеты существуют внутри домена взаимодействия, определяющего:

- способ идентификации сокета (то есть формат его «адреса»);
- диапазон взаимодействия (то есть находятся ли приложения в одной системе или на разных компьютерах, соединенных по сети).

Современные операционные системы поддерживают как минимум домены следующих видов:

- UNIX-домен (`AF_UNIX`) позволяет взаимодействовать приложениям, находящимся на одном компьютере (в спецификации POSIX.1g синонимом `AF_UNIX` является константа `AF_LOCAL`, хотя в стандарте SUSv3 она не предусмотрена);
- IPv4-домен (`AF_INET`) позволяет взаимодействовать приложениям, которые выполняются на разных компьютерах, соединенных по сети на основе протокола IPv4 (Internet Protocol version 4);
- IPv6-домен (`AF_INET6`) позволяет взаимодействовать приложениям, выполняемым на разных компьютерах, соединенных по сети на основе протокола IPv4 (Internet Protocol version 6). Протокол IPv6 должен прийти на смену IPv4, хотя распространен все еще значительно меньше, чем предшественник.

Характеристики этих доменов собраны в табл. 52.1.

В некоторых примерах кода вместо AF_UNIX можно встретить константы с такими именами, как PF_UNIX. В данном контексте AF означает семейство адресов (англ. address family), а PF — семейство протоколов (англ. protocol family). Изначально предполагалось, что одно семейство протоколов может поддерживать разные семейства адресов. Но на практике этого никто никогда не делал, а все существующие реализации определяют константы вида PF_ в качестве синонимов для констант AF_ (в стандарт SUSv3 входят только последние). В этой книге всегда используется префикс AF_. Подробную информацию о происхождении указанных констант можно найти в разделе 4.2 книги [Stevens et al., 2004].

Таблица 52.1. Домены сокетов

Домен	Взаимодействие выполняется	Взаимодействие между приложениями	Формат адреса	Структура адреса
AF_UNIX	Внутри ядра	На одном компьютере	Путь	sockaddr_un
AF_INET	Через IPv4	На компьютерах, соединенных по сети IPv4	32-разрядный адрес IPv4 + 16-разрядный номер порта	sockaddr_in
AF_INET6	Через IPv6	На компьютерах, соединенных по сети IPv6	128-разрядный адрес IPv6 + + 16-разрядный номер порта	sockaddr_in6

Виды сокетов

Любая реализация предоставляет как минимум два вида сокетов: потоковые и датаграммные. Эти две разновидности поддерживаются как в UNIX-, так и в интернет-доменах. Свойства данных видов сокетов собраны в табл. 52.2.

Таблица 52.2. Виды сокетов и их свойства

Свойство	**Вид сокета**	
	Потоковый	**Датаграммный**
Надежная доставка	Да	Нет
Сохранение границ сообщений	Нет	Да
Ориентированные на соединение	Да	Нет

Потоковые сокеты (`SOCK_STREAM`) предоставляют надежный, двунаправленный канал взаимодействия на основе байтового потока. Ниже поясняются термины, использованные в данном описании:

- *надежный* — мы гарантируем одно из двух: либо передаваемые данные будут доставлены невредимыми приложению-адресату в том виде, в котором они были посланы (при отсутствии сбоев в сетевом канале и получателе), либо мы получим уведомление о возможном сбое при передаче;
- *двунаправленный* — данные могут передаваться между сокетами в любом направлении;
- *байтовый поток* — как и в случае с именованными каналами, сообщения не имеют границ (см. раздел 44.1).

Потоковый сокет напоминает применение двух каналов, обеспечивающих двунаправленное взаимодействие между двумя приложениями, с той лишь разницей, что сокеты (интернет-доменов) позволяют взаимодействовать по сети.

Потоковые сокеты работают парами, соединяясь друг с другом, поэтому их называют *ориентированными на соединение*. *Одноранговым* называют сокет, находящийся на другом конце соединения; адрес такого сокета и приложение, которое его использует, тоже называют одноранговыми. Иногда в качестве синонима приводят термин *«удаленный»* (или *«внешний»*). Аналогичным образом приложение, сокет и адрес на этом конце соединения иногда называют локальными. Потоковый сокет может иметь только одну пару.

Датаграммные сокеты (`SOCK_DGRAM`) позволяют обмениваться данными в виде сообщений, которые называются *датаграммами*. Сокеты этого вида сохраняют границы сообщений, но не обеспечивают надежную передачу данных. Сообщения могут приходить не в том порядке, дублироваться или вовсе теряться.

Датаграммные сокеты являются общим случаем сетевого взаимодействия, *не требующего соединения*. В процессе работы им не нужно соединяться с другими сокетами (этим они отличаются от потоковых). В подразделе 52.6.2 вы увидите, что датаграммные сокеты на самом деле могут быть соединены друг с другом, но по своей семантике данная процедура некоторым образом отличается от соединения потоковых сокетов.

В интернет-доменах датаграммные сокеты задействуют протокол UDP (от англ. User Datagram Protocol — протокол пользовательских датаграмм), а потоковые — TCP (от англ. Transmission Control Protocol — протокол управления передачей). Для краткости мы будем использовать термины *«UDP-сокет»* и *«TCP-сокет»*.

Системные вызовы для работы с сокетами

Ниже перечислены ключевые системные вызовы для работы с сокетами.

- Системный вызов `socket()` создает новый сокет.
- Системный вызов `bind()` привязывает сокет к адресу. Обычно он используется сервером для привязки сокета к общеизвестному адресу, чтобы клиенты могли его найти.
- Системный вызов `listen()` позволяет потоковому сокету принимать входящие соединения от других сокетов.
- Системный вызов `accept()` принимает соединение от удаленного приложения в «слушающий» потоковый сокет и опционально возвращает адрес удаленного сокета.
- Системный вызов `connect()` устанавливает соединение с другим сокетом.

> В большинстве архитектур Linux (за исключением Alpha и IA-64) все системные вызовы для работы с сокетами реализованы в виде библиотечных функций-оберток вокруг одного-единственного системного вызова, socketcall() (это один из остатков старой реализации сокетов, которая была выполнена в виде отдельного от Linux проекта). Тем не менее в настоящей книге все данные функции называются системными вызовами, поскольку именно так они были изначально реализованы в системе BSD, равно как и во многих других разновидностях UNIX.

Ввод/вывод через сокеты может быть выполнен с помощью традиционных операций `read()` и `write()` или же специальных системных вызовов (таких как `end()`, `recv()`, `sendto()` и `recvfrom()`). По умолчанию все эти вызовы блокируются, если ввод/вывод нельзя выполнить немедленно. Неблокирующие чтение и запись тоже возможны, если включить флаг состояния `O_NONBLOCK`, используя операции `F_SETFL` для вызова `fcntl()` (см. раздел 5.3).

> В Linux можно воспользоваться вызовом ioctl(fd, FIONREAD, &cnt), чтобы получить количество непрочитанных байтов, доступных в потоковом сокете, на который ссылается дескриптор fd. В случае с датаграммным сокетом эта операция возвращает количество байтов в следующем непрочитанном сообщении (это значение может быть равным нулю, если следующей датаграммы не существует или она имеет нулевую длину). Эта возможность не предусмотрена стандартом SUSv3.

52.2. Создание сокета: socket()

Системный вызов `socket()` создает новый сокет.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Возвращает файловый дескриптор или `-1`, если произошла ошибка

Аргументы `domain` и `type` обозначают соответственно домен соединения сокета и его тип. Последний обычно принимает одно из двух значений: `SOCK_STREAM` (для создания потокового сокета) или `SOCK_DGRAM` (для создания датаграммного).

Для сокетов, описываемых в данной книге, аргумент `protocol` всегда равен `0`. Ненулевые значения применяются для других типов сокетов, которые здесь не затрагиваются. Например, в случае с сырыми сокетами (`SOCK_RAW`) он равен `IPPROTO_RAW`.

При успешном выполнении вызов `socket()` возвращает файловый дескриптор, который будет использоваться для работы с новым сокетом в последующих системных вызовах.

Начиная с версии 2.6.27, ядро Linux позволяет задействовать аргумент type альтернативным способом, предоставляя два нестандартных флага, которые могут применяться к типу сокета с помощью побитового ИЛИ. Значение SOCK_CLOEXEC заставляет ядро включить для нового файлового дескриптора флаг FD_CLOEXEC, используемый по тому же принципу, что и флаг O_CLOEXEC в вызове open() (см. подраздел 4.3.1). Значение SOCK_NONBLOCK заставляет ядро установить для описания файла флаг O_NONBLOCK, делая все последующие операции ввода/вывода с сокетом неблокирующими. Это позволяет избежать дополнительных вызовов fcntl() для достижения того же результата.

52.3. Привязывание сокета к адресу: bind()

Системный вызов `bind()` привязывает сокет к заданному адресу.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `sockfd` представляет собой файловый дескриптор, полученный из предыдущего вызова `socket()`. Аргумент `addr` является указателем на структуру, описывающую адрес привязки сокета. Тип структуры, передаваемой в этом аргументе, зависит от домена сокета. Аргумент `addrlen` обозначает размер структуры с адресом; он имеет тип `socklen_t`, который согласно стандарту SUSv3 должен быть целым числом.

Обычно серверный сокет привязывается к общеизвестному адресу; клиентам, подключающимся к серверу, о нем известно заранее.

Серверный сокет можно не привязывать к общеизвестному адресу. Например, в случае с интернет-доменом сервер может пропустить операцию bind() и сразу сделать вызов listen(), что заставит ядро выбрать для соответствующего сокета динамический порт (они будут описаны в подразделе 54.6.1). Позже сервер может использовать функцию getsockname() (см. раздел 57.5) для извлечения адреса из своего сокета. В данном случае сервер должен опубликовать полученный адрес, чтобы клиенты могли найти его сокет. Это можно сделать, зарегистрировав адрес сервера в централизованной службе каталогов, к которой затем подключаются клиенты (например, в системе Sun RPC эта проблема решается с помощью сервера portmapper). Сокет самой службы каталогов, естественно, должен быть доступен по общеизвестному адресу.

52.4. Универсальные структуры для хранения адресов сокетов: struct sockaddr

На аргументах `addr` и `addrlen` вызова `bind()` следует остановиться отдельно. В табл. 52.1 вы можете видеть, что во всех доменах применяются адреса в разных форматах. Например, сокеты домена UNIX задействуют пути к файлам, тогда как в интернет-доменах адрес состоит из IPC-адреса и номера порта. Для каждого домена предусмотрена отдельная

структура данных, хранящая адрес сокета. Но ввиду того, что системные вызовы наподобие bind() являются универсальными и охватывают все домены, они должны иметь возможность принимать адреса любых типов. Для этого в программном интерфейсе сокетов объявлена универсальная структура данных, struct sockaddr. Ее единственное назначение — привести различные адреса, использующиеся в разных доменах, к единому типу, который можно передавать в системные вызовы для работы с сокетами. Структура sockaddr обычно имеет следующий вид:

```
struct sockaddr {
    sa_family_t sa_family;     /* Семейство адресов (константы вида AF_*) */
    char        sa_data[14];   /* Адрес сокета (размер зависит от домена) */
};
```

Эта структура служит шаблоном для всех других хранящих адреса определенных доменов; все они начинаются с поля family, которое соотносится с полем sa_family структуры sockaddr (согласно стандарту SUSv3 тип данных sa_family_t представляет собой целое число). Значения поля family должно быть достаточно для определения размера и формата адреса, хранящегося в остальной части структуры.

В некоторых реализациях UNIX структура sockaddr содержит дополнительное поле sa_len, обозначающее ее общий размер. Стандарт SUSv3 не требует наличия этого поля; к тому же оно не поддерживается программным интерфейсом сокетов в Linux.

Если определить макрос проверки возможностей _GNU_SOURCE, то библиотека glibc будет использовать расширение компилятора gcc для прототипирования системных вызовов в заголовочном файле <sys/socket.h>, исключая тем самым необходимость приведения типов (struct sockaddr *). Однако в портируемых приложениях полагаться на эту возможность нельзя (в других системах компилятор будет выводить соответствующие предупреждения).

52.5. Потоковые сокеты

Принцип работы потоковых сокетов можно объяснить на примере телефонной сети.

1. Системный вызов socket(), создающий сокет, аналогичен подключению телефонного аппарата. Чтобы приложения могли взаимодействовать друг с другом, каждое из них должно создать свой сокет.
2. Взаимодействие с помощью потоковых сокетов аналогично телефонному звонку. Прежде чем начать общение, приложения должны соединить свои сокеты. Это делается следующим образом.
   - Одно приложение делает вызов bind(), чтобы привязать свой сокет к общеизвестному адресу, и затем вызывает listen() для уведомления ядра о своей готовности принимать входящие соединения. Возвращаясь к нашей аналогии: чтобы другие люди могли нам звонить, у нас должен быть телефонный номер, зарегистрированный на АТС.
   - Другое приложение устанавливает соединение с помощью вызова connect(), указывая адрес сокета, к которому оно хочет подключиться. Данное действие аналогично набору телефонного номера.
   - Затем приложение, вызвавшее listen(), принимает соединение, используя вызов accept(). Это похоже на то, как мы снимаем телефонную трубку, когда слышим звонок. Вызов accept() блокируется, если сделать его до того, как другое приложение выполнит connect() («в ожидании звонка»).
3. Подключившись, можно передавать данные в обоих направлениях (аналогично ведению диалога по телефону), пока одно из приложений не закроет соединение с помощью вызова close(). Взаимодействие выполняется с использованием традиционных

системных вызовов `read()` и `write()` или же ряда специальных операций для работы с сокетами (таких как `send()` и `recv()`), которые предоставляют дополнительные возможности.

Применение системных вызовов для работы с потоковыми сокетами проиллюстрировано на рис. 52.1.

Активные и пассивные сокеты

Потоковые сокеты часто делят на активные и пассивные.

- Сокет, созданный с помощью вызова `socket()`, по умолчанию является *активным*. Его можно использовать в вызове `connect()`, чтобы установить соединение с пассивным. Эта процедура называется *активным открытием*.
- *Пассивным* (или *слушающим*) называется сокет, который в результате вызова `listen()` способен принимать входящие соединения. Процедура приема входящих соединений называется *пассивным открытием*.

В большинстве приложений, использующих потоковые сокеты, сервер выполняет пассивное открытие, а клиент — активное. Мы руководствуемся данным правилом в последующих разделах, поэтому приложение, выполняющее активное открытие сокета, будем называть просто клиентом. Аналогично вместо словосочетания приложение, которое выполняет пассивное открытие сокета» будет задействован термин «сервер».

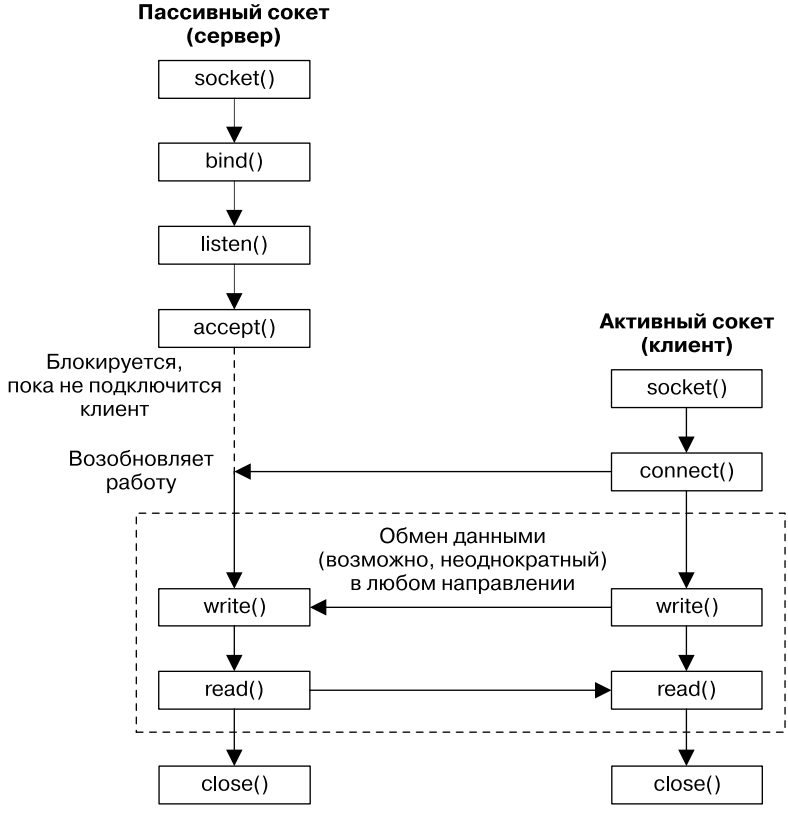


Рис. 52.1. Общая схема системных вызовов, применяемых для работы с потоковыми сокетами

52.5.1. Ожидание входящих соединений: listen()

Системный вызов `listen()` делает потоковый сокет, на который указывает файловый дескриптор `sockfd`, *пассивным*. Впоследствии этот сокет будет использоваться для приема соединений от других (активных) сокетов.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Вызов `listen()` нельзя применять к подключенным сокетам, для которых уже была успешно выполнена операция `connect()`, или к возвращенным вызовами `accept()`.

Чтобы понять назначение аргумента `backlog`, следует отметить: клиент может вызвать `connect()` до того, как сервер выполнит вызов `accept()`. Например, это может случиться из-за того, что он занят работой с какими-то другими клиентами. Данная ситуация приводит к возникновению *отложенного соединения*, показанного на рис. 52.2.

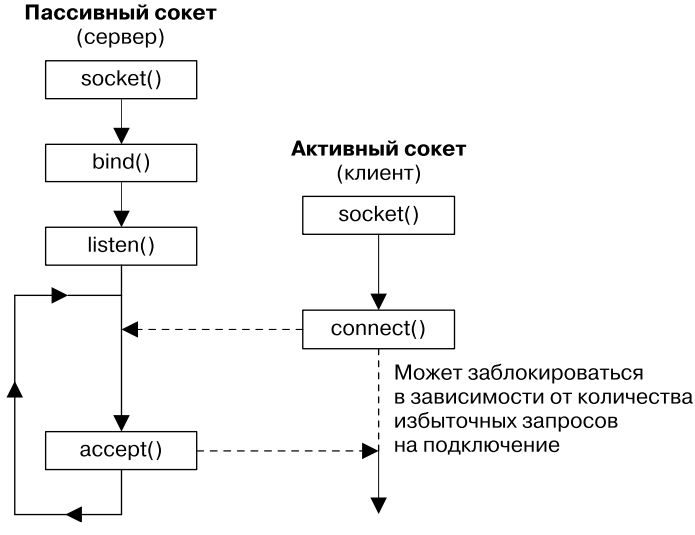


Рис. 52.2. Отложенное соединение с сокетом

Ядро должно записывать сведения о каждом отложенном запросе на подключение, чтобы впоследствии выполнить необходимые вызовы `accept()`. Аргумент `backlog` позволяет ограничить количество таких отложенных соединений. Запросы, находящиеся в рамках допустимого значения, завершаются успешно и без каких-либо задержек (в случае с TCP-сокетами все немного сложнее; в этом вы убедитесь в подразделе 57.6.4). Остальные запросы блокируются до тех пор, пока соединение не будет принято (с помощью `accept()`) и, следовательно, удалено из очереди отложенных соединений.

SUSv3 позволяет устанавливать «потолок» для аргумента `backlog` и разрешает округлять до него любые значения, которые его превышают. В стандарте также говорится о том, что реализация должна объявить это ограничение в виде константы `SOMAXCONN` в заголовочном файле `<sys/socket.h>`. В Linux она равна 128. Но, начиная с версии 2.4.25,

ядро Linux позволяет корректировать ее значение во время выполнения программы с помощью файла /proc/sys/net/core/somaxconn (доступного только в Linux). В более старых версиях ядра данную константу нельзя было изменить.

> В оригинальной реализации сокетов в системе BSD верхнее ограничение для аргумента backlog составляло 5; эту цифру иногда можно встретить в старом коде. Все современные реализации устанавливают менее жесткое ограничение, что продиктовано требованиями сетевых серверов, обслуживающих большое количество клиентов, используя TCP-сокеты.

52.5.2. Прием соединения: accept()

Системный вызов `accept()` принимает входящее соединение на слушающем потоковом сокете, на который указывает файловый дескриптор `sockfd`. Если вызов `accept()` не обнаруживает ожидающих соединений, то блокируется и ждет, пока не поступит соответствующий запрос.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Возвращает файловый дескриптор или `-1`, если произошла ошибка

Ключом к пониманию вызова `accept()` является тот факт, что он создает *новый* сокет, который затем подключается к удаленному сокету, выполнившему вызов `connect()`. Файловый дескриптор для подключенного сокета возвращается в виде результата выполнения функции `accept()`. Слушающий сокет (`sockfd`) остается открытым и может использоваться для приема последующих соединений. Типичное серверное приложение создает один слушающий сокет, привязывает его к общеизвестному адресу, после чего обрабатывает с его помощью все клиентские запросы.

Остальные аргументы вызова `accept()` возвращают адрес удаленного сокета. Аргумент `addr` указывает на структуру, применяемую для возвращения адреса сокета. Тип данного аргумента зависит от домена сокета (как и в случае с вызовом `bind()`).

Аргумент `addrlen` служит для возвращения результата. Он указывает на целое число. Перед выполнением вызова оно должно быть инициализировано с помощью размера буфера, на который указывает `addr`. Благодаря этому ядро знает, сколько места доступно для возвращения адреса сокета. При возвращении вызова `accept()` данному числу присваивается значение, описывающее количество байтов, скопированных в буфер.

Если вас не интересует адрес удаленного сокета, то аргументам `addr` и `addrlen` следует присвоить значения `NULL` и `0` соответственно (при желании вы можете получить этот адрес позже, воспользовавшись системным вызовом `getpeername()`, описанным в разделе 57.5).

52.5.3. Соединение с удаленным сокетом: connect()

Системный вызов `connect()` соединяет активный сокет, на который указывает файловый дескриптор `sockfd`, со слушающим сокетом, чей адрес задан в виде аргументов `addr` и `addrlen`.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргументы addr и addrlen указываются таким же образом, как и в вызове bind().

Если вызов connect() завершается неудачей и мы хотим повторить попытку соединения, то портируемая процедура для этого согласно стандарту SUSv3 выглядит так: нужно закрыть имеющийся сокет, создать вместо него новый и с его помощью попытаться соединиться еще раз.

52.5.4. Операции ввода/вывода с потоковыми сокетами

Двунаправленный канал взаимодействия двух конечных точек обеспечивается за счет пары соединенных между собой сокетов. На рис. 52.3 показано, как это выглядит в UNIX-домене.

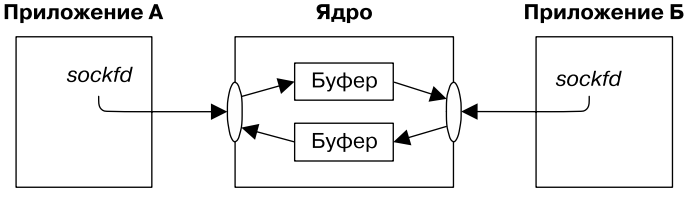


Рис. 52.3. Потоковые сокеты домена UNIX предоставляют двунаправленный канал взаимодействия

По своей семантике операции ввода/вывода с потоковыми сокетами похожи на аналогичные операции с именованными каналами.

- Для выполнения ввода/вывода служат системные вызовы read() и write() (или специальные вызовы send() и recv(), описанные в разделе 57.3). Поскольку сокеты являются двунаправленными, обе эти операции можно использовать на любом конце соединения.
- Сокет может быть закрыт с помощью системного вызова close() или в результате завершения приложения. После этого, если удаленная программа попытается прочитать данные на другом конце соединения, она получит символ конца файла (когда закончатся все данные в буфере). При попытке записи в данный сокет удаленная программа получит сигнал SIGPIPE, а системный вызов завершится ошибкой EPIPE. Как уже отмечалось в разделе 44.2, в такой ситуации сигнал SIGPIPE обычно игнорируется, а информация о закрытом соединении определяется по ошибке EPIPE.

52.5.5. Закрытие соединения: close()

Обычно для закрытия соединения на основе потоковых сокетов используется вызов close(). Если на сокет указывают несколько файловых дескрипторов, то он будет закрыт вместе с последним из них.

Представьте, что после закрытия соединения удаленное приложение сталкивается со сбоем, не может прочитать или корректно обработать данные, которые ему были отправлены. В таком случае невозможно узнать о произошедшей ошибке. Удостовериться в том, что данные были успешно прочитаны и обработаны, можно, создав в приложении некий протокол подтверждения. Обычно это выражается в возвращении специального сообщения, подтверждающего успешное выполнение операции.

В разделе 57.2 будет описан системный вызов shutdown(), который обеспечивает более гибкое управление процедурой закрытия соединений на основе потоковых сокетов.

52.6. Датаграммные сокеты

Принцип их работы можно объяснить на примере почтового сервиса.

1. Системный вызов `socket()` является аналогом установки почтового ящика (здесь имеются в виду ящики, традиционные для ряда стран: они устанавливаются в пригородах и используются как для получения, так и для отправки писем). Любое приложение, которое хочет отправлять или получать датаграммы, должно создать датаграммный сокет с помощью вызова `socket()`.
2. Чтобы иметь возможность принимать датаграммы (письма) от других приложений, нужно привязать свой сокет к общеизвестному адресу, воспользовавшись вызовом `bind()`. Обычно это делает сервер, а клиент инициирует взаимодействие, отправляя по данному адресу свою датаграмму (в некоторых доменах, включая UNIX-домен, клиент, желающий получать датаграммы, посланные сервером, тоже может задействовать вызов `bind()` для привязки своего сокета к определенному адресу).
3. Для отправки датаграммы приложение делает вызов `sendto()`. Он в качестве одного из своих аргументов принимает адрес удаленного сокета, которому предназначено сообщение. Это похоже на то, как мы записываем на конверте адрес получателя, прежде чем опустить его в почтовый ящик.
4. Чтобы получить датаграмму, приложение делает вызов `recvfrom()`, который может заблокироваться, если она еще не пришла. Поскольку данный вызов позволяет определить адрес отправителя, при желании можно послать ответное сообщение (это пригодится в ситуациях, когда отправитель привязан к адресу, о котором мы не знаем заранее — что является типичной ситуацией). Здесь наша аналогия немного хромает, поскольку в реальности отправителю не обязательно писать на конверте свой собственный адрес.
5. Когда сокет больше не нужен, приложение закрывает его с помощью вызова `close()`.

Как и в случае с почтовой службой, если послать с одного адреса на другой несколько датаграмм (писем), нет никакой гарантии их прихода в том же порядке (при условии, что они вообще придут). Хотя датаграммы имеют одну особенность, которой нет в почтовых службах: исходные сетевые протоколы иногда повторяют передачу пакетов данных, из-за чего одна и та же датаграмма может быть получена несколько раз.

Применение системных вызовов для работы с датаграммными сокетами проиллюстрировано на рис. 52.4.

52.6.1. Обмен датаграммами: recvfrom() и sendto()

Системные вызовы `recvfrom()` и `sendto()` принимают и отправляют датаграммы на датаграммный сокет.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buffer, size_t length,
                 int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

Возвращает количество полученных байтов, `0`, если обнаружен конец файла, или `-1` при ошибке

```
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Возвращает количество отправленных байтов или `-1` при ошибке

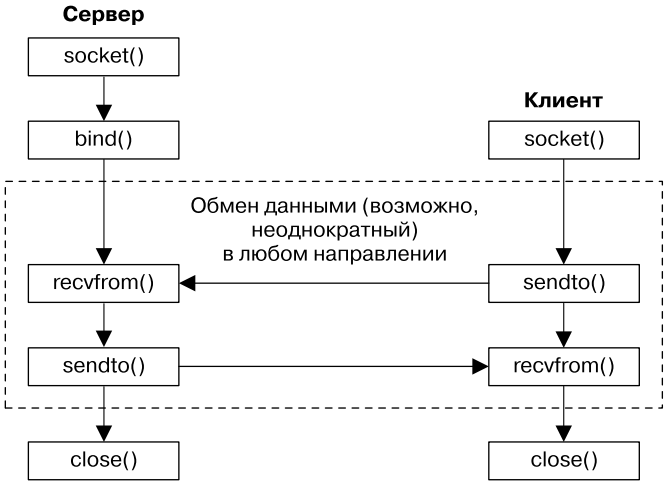


Рис. 52.4. Список системных вызовов, которые используются для работы с датаграммными сокетами

Возвращаемое значение и первые три аргумента этих системных вызовов такие же, как у операций `read()` и `write()`.

Четвертый аргумент, `flags`, представляет собой битовую маску, которая определяет специальные возможности ввода/вывода, связанные с сокетами. Мы вернемся к данной теме в разделе 57.3, когда познакомимся с системными вызовами `recv()` и `send()`. Если эти возможности не нужны, то аргументу `flags` можно передать `0`.

Аргументы `src_addr` и `addrlen` используются, чтобы получить или задать адрес удаленного сокета — объекта взаимодействия.

В вызове `recvfrom()` аргументы `src_addr` и `addrlen` возвращают адрес удаленного сокета, с помощью которого была отправлена датаграмма. (Они аналогичны аргументам `addr` и `addrlen` в вызове `accept()`, возвращающим адрес подключающегося удаленного сокета.) Аргумент `src_addr` является указателем на структуру с адресом, совместимым с текущим доменом. По аналогии с вызовом `accept()`, аргумент `addrlen` может хранить результат выполнения. Изначально ему следует присвоить размер структуры, на которую указывает `src_addr`; во время возвращения он будет содержать количество байтов, записанных в эту структуру.

Если адрес отправителя неинтересен, то аргументам `src_addr` и `addrlen` можно передать значение `NULL`. В таком случае в контексте получения датаграммы вызовы `recvfrom()` и `recv()` эквивалентны. Для ее чтения можно воспользоваться и вызовом `read()`, который ведет себя так же, как `recv()` с аргументом `flags`, равным 0.

Независимо от значения аргумента `length` вызов `recvfrom()` извлекает из датаграммного сокета ровно одно сообщение. Если размер сообщения превышает заданное значение, то оно автоматически урезается до `length` байт.

> При использовании системного вызова recvmsg() (см. раздел 57.13.2) усечение датаграммы можно обнаружить с помощью флага MSG_TRUNC, который возвращается в поле msg_flags структуры msghdr. Подробности см. на странице recvmsg(2) руководства.

В случае с вызовом `sendto()` аргументы `dest_addr` и `addrlen` описывают сокет, которому будет отправлена датаграмма. Они применяются по тому же принципу, что и в вызове `connect()`. Аргумент `dest_addr` представляет собой структуру с адресом, совместимым

с текущим доменом. Ему передается адрес сокета-получателя. Его длина указывается с помощью аргумента `addrlen`.

> Linux позволяет отправлять датаграммы нулевой длины, задействуя вызов `sendto()`, однако не все реализации UNIX это поддерживают.

52.6.2. Использование вызова connect() в сочетании с датаграммными сокетами

Датаграммные сокеты не нуждаются в соединении, однако при работе с ними тоже можно применять системный вызов `connect()`. Он заставляет ядро записать адрес удаленного сокета. Такие сокеты называются *соединенными*. Стандартные датаграммные сокеты, к которым не был применен вызов `connect()`, называют *несоединенными*.

После соединения датаграммного сокета:

- датаграммы, отправляемые с помощью вызова `write()` (или `send()`), автоматически передаются заданному удаленному сокету. Как и в случае с вызовом `sendto()`, каждая операция `write()` приводит к созданию отдельной датаграммы;
- мы можем читать только датаграммы, отправленные заданным удаленным сокетом.

Стоит отметить: для датаграммных сокетов вызов `connect()` имеет асимметричный эффект. Утверждения, приведенные выше, применимы только к сокетам, для которых был сделан вызов `connect()`, и не относятся к удаленным сокетам на другом конце соединения (если только удаленное приложение не вызвало `connect()` для своего сокета).

Удаленный адрес соединенного датаграммного сокета можно изменить с помощью повторного вызова `connect()`. Можно также полностью разорвать связь с удаленным сокетом, указав значение `AF_UNSPEC` в качестве семейства адресов в соответствующей структуре (например, в домене UNIX это поле `sun_family`). Однако стоит помнить, что многие реализации UNIX не поддерживают использование значения `AF_UNSPEC` для данных целей.

> Стандарт SUSv3 довольно расплывчато описывает процесс разрыва соединения. В нем говорится, что для этого вызову connect() следует передать некий «нулевой адрес», однако определение данного термина не дается. Стандарт SUSv4 недвусмысленно требует применять значение AF_UNSPEC.

Очевидное преимущество определения удаленного адреса для датаграммного сокета заключается в том, что таким образом для передачи данных можно использовать более простые системные вызовы ввода/вывода. Вызов `sendto()` с аргументами `dest_addr` и `addrlen` можно заменить операцией `write()`. В основном это может пригодиться в приложениях, которым нужно отправлять множество датаграмм одному и тому же удаленному сокету (что довольно типично для ряда датаграммных клиентов).

> В некоторых реализациях TCP/IP привязка датаграммного сокета к удаленному адресу приводит к улучшению производительности (см. [Stevens et al., 2004]). В Linux эта разница малозначительна.

52.7. Резюме

Сокеты обеспечивают взаимодействие приложений, размещенных локально или на разных компьютерах, соединенных по сети.

Сокет существует в рамках домена взаимодействия, который определяет диапазон обмена данными и формат адреса для идентификации сокетов. Стандарт SUSv3 поддерживает следующие домены: UNIX (`AF_UNIX`), IPv4 (`AF_INET`) и IPv6 (`AF_INET6`).

Большинство приложений использует сокеты одного из двух видов: потоковые или датаграммные. Потоковые (`SOCK_STREAM`) устанавливают между двумя конечными точками надежный двунаправленный канал взаимодействия на основе байтового потока. Датаграммные сокеты (`SOCK_DGRAM`) обеспечивают ненадежное взаимодействие, не нуждающееся в соединении и ориентированное на передачу сообщений.

На серверной стороне потоковый сокет обычно создается с помощью вызова `socket()` и привязывается к общеизвестному адресу с использованием вызова `bind()`. Затем сервер вызывает `listen()`, чтобы сокет мог принимать входящие соединения. Каждое клиентское соединение принимается путем вызова `accept()`, который возвращает файловый дескриптор нового сокета, подключенного к клиентской стороне. Для создания потокового сокета на клиенте обычно применяется вызов `socket()`. Затем для установки соединения между двумя потоковыми сокетами вызывается `connect()` и указывается общеизвестный адрес. После этого данные могут быть переданы в любом направлении с помощью вызовов `read()` и `write()`. После явного или опосредованного закрытия всех файловых дескрипторов, ссылающихся на тот или иной потоковый сокет, соединение разрывается.

Для создания датаграммного сокета на серверной стороне и привязки его к общеизвестному адресу используются также вызовы `socket()` и `bind()`. Поскольку датаграммные сокеты не нуждаются в соединении, сервер может принимать с их помощью отправленные любым клиентом датаграммы. Для их получения подходит операция `read()` или специальный системный вызов `recvfrom()`, который возвращает адрес отправителя. На стороне клиента датаграммный сокет создается с помощью все того же вызова `socket()`, а для отправки датаграммы по определенному адресу (принадлежащему серверу) применяется вызов `sendto()`. Можно воспользоваться системным вызовом `connect()`, чтобы связать датаграммный сокет с удаленным адресом. После этого получателя можно больше не указывать, благодаря чему для отправки датаграмм будет достаточно операции `write()`.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 55.15.

53 Сокеты: домен UNIX

Эта глава посвящена сокетам домена UNIX, которые позволяют взаимодействовать процессам на одном и том же компьютере. Мы обсудим использование как потоковых, так и датаграммных сокетов. Кроме того, опишем применение прав доступа к файлам для управления доступом к сокетам домена UNIX, создание пары соединенных сокетов с помощью вызова `socketpair()` и абстрактное пространство имен сокетов в Linux.

53.1. Адреса сокетов в домене UNIX: struct sockaddr_un

Адрес сокета в домене UNIX представляет собой путь к файлу, а структура, предназначенная для его хранения, имеет следующий вид:

```
struct sockaddr_un {
    sa_family_t sun_family;    /* Всегда равно AF_UNIX */
    char sun_path[108];        /* Путь к сокету с нулевым символом в конце */
};
```

> Префикс sun_ в полях структуры sockaddr_un не имеет никакого отношения к компании Sun Microsystems; это всего лишь сокращение от socket unix.

В стандарте SUSv3 не уточняется размер поля `sun_path`. В ранних реализациях системы BSD для него применялись 108 и 104 байта, а в одной современной системе (HP-UX 11) действует значение 92. Портируемые приложения должны быть созданы из расчета наименьшего размера, используя при записи в это поле вызов `snprintf()` или `strncpy()`, чтобы избежать переполнения буфера.

Для привязки сокета UNIX-домена к адресу нужно инициализировать структуру `sockaddr_un`, передать (приведенный) указатель на нее аргументу `addr` вызова `bind()` и указать `addrlen` в качестве размера этой структуры, как показано в листинге 53.1.

Листинг 53.1. Привязка сокета домена UNIX

```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;

sfd = socket(AF_UNIX, SOCK_STREAM, 0);            /* Создаем сокет */
if (sfd == -1)
    errExit("socket");

memset(&addr, 0, sizeof(struct sockaddr_un));     /* Очищаем структуру */
addr.sun_family = AF_UNIX;                        /* Адрес домена UNIX */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);

if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

Применение вызова `memset()` в листинге 53.1 гарантирует, что все поля структуры равны `0` (данным аспектом пользуется последующий вызов `strncpy()`, финальный аргумент которого на единицу меньше, чем размер поля `sun_path`; благодаря этому в конце данного поля всегда будет находиться нулевой символ). Обнуление всей структуры целиком с помощью вызова `memset()` вместо инициализации отдельных полей гарантирует, что ни одно поле, даже если оно нестандартное и предоставляется только текущей реализацией, не будет пропущено.

> У вызова memset() есть альтернатива из мира BSD, функция bzero(), тоже обнуляющая содержимое структуры. В стандарте SUSv3 она упоминается в связке с функцией bcopy() (которая является аналогом memmove()); обе эти функции считаются устаревшими, а вместо них рекомендуется использовать вызовы memset() и memmove(). Функции bzero() и bcopy() не вошли в стандарт SUSv4.

Привязывая сокет домена UNIX, вызов `bind()` создает запись в файловой системе (следовательно, каталог, являющийся частью пути к сокету, должен быть доступен для чтения и записи). Владелец файла определяется на основе стандартных правил (см. подраздел 15.3.1). Сам файл помечается как сокет. Если применить к нему вызов `stat()`, то в поле `st_mode` возвращенной структуры `stat` (точнее, в той его части, которая хранит тип файла) будет указано значение `S_IFSOCK` (см. раздел 15.1). При вводе команды `ls -l` в первом столбце будет значиться тип `s`, указывающий на сокет домена UNIX, а команда `ls -F` добавляет к пути сокета знак равенства (`=`).

> Сокеты домена UNIX идентифицируются с помощью путей, однако операции ввода/вывода, которые к ним применяются, не затрагивают исходное устройство.

Необходимо отметить несколько моментов, касающихся привязки сокетов UNIX-домена.

- Сокет нельзя привязать к существующему пути (вызов `bind()` завершится ошибкой `EADDRINUSE`).
- Обычно сокет привязывают к полному пути для фиксации его местоположения в файловой системе. Использование относительных путей допустимо, но не рекомендуется, поскольку подразумевается, что клиент, который хочет подключиться к сокету, знает текущий каталог приложения, выполнившего вызов `bind()`.
- Сокет можно привязать только к одному пути; и наоборот — путь может быть привязан только к одному сокету.
- Сокет нельзя открыть с помощью вызова `open()`.
- Когда сокет больше не нужен, его запись в файловой системе (путь) можно удалить, используя такие вызовы, как `unlink()` или `remove()` (обычно так и следует делать).

В большинстве примеров, которые здесь приводятся, сокеты домена UNIX привязываются к файлам в каталоге `/tmp`, поскольку он присутствует в любой системе и является доступным для записи. Это облегчает запуск программ и избавляет от необходимости изменять пути к сокетам.

Однако имейте в виду, что в большинстве случаев данный выбор не самый лучший. Как уже отмечалось в разделе 38.7, создание файлов в публичных каталогах, доступных для записи (таких как `/tmp`) может стать причиной различного рода уязвимостей. Например, создавая в каталоге `/tmp` файл, имя которого совпадает с сокетом какого-то приложения, мы фактически осуществляем простую DoS-атаку. Реальные приложения должны привязывать свои сокеты домена UNIX к абсолютным путям на основе достаточно защищенных каталогов.

53.2. Потоковые сокеты в домене UNIX

В данном разделе представлено простое клиент-серверное приложение, использующее потоковые сокеты в домене UNIX. Клиентская программа (листинг 53.4) устанавливает соединение с сервером и передает ему данные со своего стандартного ввода. Серверная программа (листинг 53.3) принимает клиентские соединения и направляет все данные, посланные клиентами, в стандартный вывод. Это простой пример *итерационного* сервера — программы, которая обслуживает своих клиентов последовательно, один за другим (серверная архитектура будет рассмотрена более подробно в главе 56).

В листинге 53.2 приводится заголовочный файл, применяемый обеими программами.

Листинг 53.2. Заголовочный файл для программ us_xfr_sv.c и us_xfr_cl.c

sockets/us_xfr.h

```
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100
```

sockets/us_xfr.h

На следующих страницах сначала будет представлен исходный код сервера и клиента, а затем мы обсудим подробности их реализации и рассмотрим пример их совместного использования.

Листинг 53.3. Простой сервер на основе потоковых сокетов домена UNIX

sockets/us_xfr_sv.c

```
#include "us_xfr.h"

#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        errExit("socket");

  /* Формируем адрес сервера, привязываем к нему сокет и делаем этот сокет слушающим */
    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *) &addr,
        sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
```

```
    if (listen(sfd, BACKLOG) == -1)
        errExit("listen");

    for (;;) {   /* Последовательно обрабатываем клиентские соединения */

    /* Принимаем соединение. Оно будет назначено новому сокету, 'cfd'; слушающий

       сокет ('sfd') остается открытым и может принимать последующие соединения. */
        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
            errExit("accept");

    /* Направляем данные подключенного сокета в стандартный вывод,
       пока не обнаружим конец файла */

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");

        if (numRead == -1)
            errExit("read");

        if (close(cfd) == -1)
            errMsg("close");
    }
}
```

sockets/us_xfr_sv.c

Листинг 53.4. Простой клиент на основе потоковых сокетов домена UNIX

sockets/us_xfr_cl.c

```
#include "us_xfr.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);  /* Создаем клиентский сокет */
    if (sfd == -1)
        errExit("socket");

    /* Формируем адрес сервера и выполняем соединение */
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        errExit("connect");
    /* Копируем в сокет стандартный ввод */
    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");
```

```
    if (numRead == -1)
        errExit("read");
    exit(EXIT_SUCCESS);  /* При обнаружении конца файла сокет закрывается */
}
```

sockets/us_xfr_cl.c

Серверная программа, представленная в листинге 53.3, выполняет следующие действия:

- создает сокет;
- удаляет любой существующий файл с путем, к которому мы хотим привязать наш сокет;
- формирует структуру с адресом серверного сокета, привязывает сокет к этому адресу и делает его слушающим;
- входит в бесконечный цикл для обработки клиентских запросов. Каждая итерация цикла состоит из следующих шагов:
  - принятие соединения и получение для него нового сокета, `cfd`;
  - чтение всех данных из подключенного сокета и запись их в стандартный вывод;
  - закрытие подключенного сокета `cfd`.

Работу сервера нужно завершать вручную (например, с помощью сигнала).

Клиентская программа из листинга 53.4 выполняет следующие действия:

- создает сокет;
- формирует структуру с адресом сервера и подключается к сокету по данному адресу;
- выполняет цикл, который копирует стандартный ввод в соединение сокета. При обнаружении конца файла в своем стандартном вводе клиент завершает работу, в результате чего его сокет закрывается. При этом сервер, считывая данные с сокета на другом конце соединения, видит символ конца файла.

Пример использования указанных программ показан в следующей сессии командной строки. Начнем с запуска сервера в фоновом режиме:

```
$ ./us_xfr_sv > b &
[1] 9866
$ ls -lF /tmp/us_xfr        Анализируем файл сокета с помощью команды ls
srwxr-xr-x    1 mtk      users       0 Jul 18 10:48 /tmp/us_xfr=
```

Теперь запустим клиентскую программу, предварительно создав тестовый файл, который будет применяться в качестве ее ввода:

```
$ cat *.c > a
$ ./us_xfr_cl < a                Клиент принимает ввод из файла
```

На данном этапе дочерний процесс уже завершился. Завершим работу сервера и проверим, совпадает ли его вывод с вводом клиента:

```
$ kill %1                        Завершаем сервер
[1]+ Terminated   ./us_xfr_sv >b  Командная оболочка сигнализирует о завершении сервера
$ diff a b
$
```

Команда `diff` ничего не выводит. Это говорит о том, что входящий и исходящий файлы идентичны.

Обратите внимание: путь к сокету продолжает существовать даже после завершения работы сервера. Именно поэтому перед операцией `bind()` сервер удаляет любой файл с тем же путем, что у сокета, используя вызов `remove()` (при наличии подходящих прав доступа данный вызов удаляет файл любого типа, даже если тот не является сокетом). Если этого не сделать и если сокет с тем же путем уже был создан в результате предыдущего запуска сервера, то операция `bind()` завершится ошибкой.

53.3. Датаграммные сокеты в домене UNIX

В обобщенном описании датаграммных сокетов, приведенном в разделе 52.6, утверждалось, что они не обеспечивают надежную передачу данных. Однако это относилось к взаимодействию по сети. В домене UNIX датаграммное взаимодействие происходит внутри ядра и является надежным. Все сообщения доставляются в правильном порядке и без дублирования.

Максимальный размер датаграммы в домене UNIX

Стандарт SUSv3 не уточняет максимальный размер датаграмм, которые можно отправлять, используя сокеты в домене UNIX. Linux поддерживает довольно большие датаграммы. Ограничение устанавливается с помощью параметра сокета `SO_SNDBUF` и различных файлов в каталоге `/proc` (см. справочную страницу `socket(7)`). Однако в отдельных реализациях UNIX действует более строгое ограничение — например, 2048 байт. Портируемым приложениям, применяющим датаграммные сокеты в домене UNIX, рекомендуется существенно ограничивать максимальный размер датаграмм.

Пример программы

В листингах 53.6 и 53.7 представлено простое клиент-серверное приложение, которое задействует датаграммные сокеты в домене UNIX. В обеих этих программах применяется заголовочный файл из листинга 53.5.

Листинг 53.5. Заголовочный файл, используемый в программах ud_ucase_sv.c и ud_ucase_cl.c

sockets/ud_ucase.h

```
#include <sys/un.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10       /* Максимальный размер сообщений,
                             которыми обмениваются клиент и сервер */
#define SV_SOCK_PATH "/tmp/ud_ucase"
```

sockets/ud_ucase.h

Сначала серверная программа (см. листинг 53.6) создает сокет и привязывает его к общеизвестному адресу (предварительно удалив любой существующий путь, который совпадает с этим адресом). Затем она входит в бесконечный цикл, где принимает датаграммы от клиентов, используя вызов `recvfrom()`, переводит полученный текст в верхний регистр и возвращает результат обратно клиенту; адрес клиента извлекается с помощью все того же вызова `recvfrom()`.

Клиентская программа (см. листинг 53.7) создает сокет и привязывает его к адресу, чтобы сервер мог отправить свой ответ. Адрес клиента делается уникальным за счет добавления к пути идентификатора его процесса. Клиент входит в цикл, отправляя серверу каждый из своих аргументов командной строки в виде отдельного сообщения. Сделав это, он считывает ответ сервера и направляет его в стандартный вывод.

Листинг 53.6. Простой датаграммный сервер в домене UNIX

sockets/ud_ucase_sv.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
```

```
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);    /* Создаем серверный сокет */
    if (sfd == -1)
        errExit("socket");

    /* Формируем общеизвестный адрес и привязываем к нему серверный сокет */
    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &svaddr,
          sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

  /* Принимаем сообщения, переводим в верхний регистр и возвращаем результат клиенту */
    for (;;) {
        len = sizeof(struct sockaddr_un);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);

        if (numBytes == -1)
            errExit("recvfrom");

        printf("Server received %ld bytes from %s\n", (long) numBytes, claddr.sun_path);

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *)
               &claddr, len) != numBytes)
            fatal("sendto");
    }
}
```

sockets/ud_ucase_sv.c

Листинг 53.7. Простой датаграммный клиент в домене UNIX

sockets/ud_ucase_cl.c

```
#include "ud_ucase.h"
int

main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msg...\n", argv[0]);
```

```
 /* Создаем клиентский сокет; привязываем его к уникальному пути (основанному на PID) */
    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&claddr, 0, sizeof(struct sockaddr_un));
    claddr.sun_family = AF_UNIX;
    snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());
    if (bind(sfd, (struct sockaddr *) &claddr,
           sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    /* Формируем адрес сервера */

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

   /* Отправляем серверу сообщения; направляем ответы в стандартный вывод */

    for (j = 1; j < argc; j++) {
        msgLen = strlen(argv[j]);    /* Может превысить BUF_SIZE */
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_un)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");
        printf("Response %d: %.*s\n", j, (int) numBytes, resp);
    }

    remove(claddr.sun_path);     /* Удаляем путь к клиентскому сокету */
    exit(EXIT_SUCCESS);
}
```

sockets/ud_ucase_cl.c

Использование серверной и клиентской программ показано на примере следующей сессии командной строки:

```
$ ./ud_ucase_sv &
[1] 20113
$ ./ud_ucase_cl hello world          Отправляем серверу два сообщения
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 1: HELLO
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 2: WORLD
$ ./ud_ucase_cl 'long message'       Отправляем серверу одно более длинное сообщение
Server received 10 bytes from /tmp/ud_ucase_cl.20151
Response 1: LONG MESSA
$ kill %1                            Завершаем работу сервера
```

С помощью второго запуска клиентской программы мы показали, что сообщение, размер которого превышает значение аргумента `length` в вызове `recvfrom()` (в данном случае это константа `BUF_SIZE`, определенная в листинге 53.5 и равная 10), автоматически усекается. Как видите, усечение произошло, ведь сервер вывел всего 10 байт, тогда как сообщение, посланное клиентом, было 12-байтным.

53.4. Права доступа к сокетам домена UNIX

Права доступа к файлу сокета и его владелец определяют, какие процессы могут с ним взаимодействовать:

- чтобы подключиться к потоковому сокету домена UNIX, необходимо иметь возможность записывать в его файл;
- чтобы послать сообщение датаграммному сокету домена UNIX, нужно иметь право на запись в его файл.

Кроме того, следует владеть правами на выполнение (поиск) во всех каталогах, составляющих путь к сокету.

По умолчанию полный доступ к сокету (созданному с помощью вызова `bind()`) имеют его владелец (пользователь), группа и другие пользователи. Чтобы это изменить, перед `bind()` можно сделать вызов `umask()`, который отключит нежелательные права доступа.

Отдельные системы игнорируют права доступа к файлу сокета (что допускается стандартом SUSv3). Следовательно, портируемые приложения не могут управлять доступом к сокету с помощью данных прав, хотя для этой цели можно использовать права доступа к каталогу, в котором находится файл сокета.

53.5. Создание соединенной пары сокетов: socketpair()

Иногда одному процессу может понадобиться создать пару сокетов и соединить их. Это можно сделать, используя два вызова `socket()`, один вызов `bind()`, после которых следует либо цепочка из `listen()`, `connect()` и `accept()` (для потоковых сокетов), либо `connect()` (для датаграммных сокетов). Но всего перечисленного можно добиться с помощью единственного вызова `socketpair()`.

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Системный вызов `socketpair()` можно использовать только в домене UNIX; то есть аргумент `domain` должен быть равен `AF_UNIX` (это ограничение действует в большинстве реализаций и является логичным, так как пара сокетов создается на одном и то же компьютере). Тип сокета, `type`, должен быть равен либо `SOCK_DGRAM`, либо `SOCK_STREAM`. Аргументу `protocol` следует передать значение `0`. Массив `sockfd` возвращает файловые дескрипторы, ссылающиеся на два соединенных сокета.

Если передать аргументу `type` значение `SOCK_STREAM`, то мы получим аналог двунаправленного канала (или *потокового канала*). Каждый сокет можно использовать как для чтения, так и для записи; данные по каналам можно передавать в любом направлении (в системе BSD вызов `pipe()` реализован в виде обертки для `socketpair()`).

По способу применения пара сокетов обычно мало чем отличается от именованного канала. После вызова `socketpair()` процесс может создать потомка, воспользовавшись операцией `fork()`. Тот унаследует файловые дескрипторы родителя, в том числе и ссылающиеся на пару сокетов. Таким образом, родитель и потомок могут применять данный подход для межпроцессного взаимодействия.

Одно из отличий вызова `socketpair()` от выполнения аналогичной операции вручную заключается в том, что сокеты не привязываются ни к какому адресу. Это позволяет избежать целого ряда уязвимостей в безопасности, поскольку другие процессы таких сокетов не видят.

53.6. Абстрактное пространство имен сокетов в Linux

Так называемое *абстрактное пространство имен* является одной из уникальных возможностей Linux, которая позволяет привязывать сокет домена UNIX к имени, не имеющему отношения к файловой системе. Это дает несколько потенциальных преимуществ:

- можно не беспокоиться о возможных конфликтах с существующими файлами;
- после завершения работы с сокетом не нужно удалять соответствующий файл. Абстрактное имя автоматически исчезает вместе с закрытием сокета;
- нет нужды создавать для сокета отдельный файл. Это может пригодиться в среде `chroot` или при отсутствии прав на запись в файловую систему.

Чтобы создать абстрактную привязку, нужно сделать первый байт поля `sun_path` нулевым (`\0`). Это отличает абстрактные имена сокетов от традиционных, принятых в домене UNIX и состоящих из одного или несколько ненулевых байтов, в конце которых находится нулевой символ. Остальные байты поля `sun_path` определяют абстрактное имя сокета. Данное имя не предусматривает завершающего символа, поэтому учитывается каждый байт.

Создание абстрактной привязки для сокета демонстрируется в листинге 53.8.

Листинг 53.8. Создание абстрактной привязки для сокета

Из файла **sockets/us_abstract_bind.c**

```
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_un addr;
    char *str;

    memset(&addr, 0, sizeof(struct sockaddr_un));

    /* Очищаем структуру с адресом */
    addr.sun_family = AF_UNIX;              /* Адрес домена UNIX */

    /* Бит addr.sun_path[0] уже был обнулен вызовом memset() */
    strncpy(&addr.sun_path[1], "xyz", sizeof(addr.sun_path) - 2);
             /* Абстрактное имя "xyz", за которым следуют нулевые байты */

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sockfd == -1)
        errExit("socket");

    if (bind(sockfd, (struct sockaddr *) &addr,
            sizeof(sa_family_t) + strlen(str) + 1) == -1)
        errExit("bind");
```

```
    sleep(60);
    exit(EXIT_SUCCESS);
}
```

Из файла **sockets/us_abstract_bind.c**

Идентификация абстрактных имен сокетов с помощью начального нулевого байта может иметь необычные последствия. Представьте, что переменная `name` указывает на строку нулевой длины; попытаемся привязать сокет домена UNIX к полю `sun_path`, инициализированному следующим образом:

```
strncpy(addr.sun_path, name, sizeof(addr.sun_path) - 1);
```

В Linux таким манером мы непреднамеренно создадим для сокета абстрактную привязку. Этот код можно считать ошибочным. В других реализациях UNIX последующий вызов `bind()` завершится ошибкой.

53.7. Резюме

Сокеты UNIX-домена позволяют приложениям взаимодействовать на одном и том же компьютере. Эти сокеты могут быть потоковыми и датаграммными.

Сокеты домена UNIX идентифицируются по имени в файловой системе. Доступ к сокету может регулироваться с помощью прав доступа к соответствующему файлу.

Системный вызов `socketpair()` создает пару сокетов домена UNIX, соединенных между собой. Это позволяет избежать сразу нескольких системных вызовов для операций создания, привязки и подключения. В своем использовании такие сокеты обычно похожи на именованный канал: создав пару сокетов, процесс генерирует поток, который наследует ссылающиеся на них дескрипторы.

Абстрактное пространство имен сокетов (доступное только в Linux) позволяет привязывать сокет домена UNIX к имени, не имеющему отношения к файловой системе.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 55.14.

53.8. Упражнения

53.1. В разделе 53.3 отмечалось, что датаграммные сокеты домена UNIX являются надежными. Напишите программу, которая показывает следующее: отправитель блокируется, если шлет сообщения быстрее, чем получатель может их прочитать, и остается заблокированным, пока получатель читает отложенные датаграммы.

53.2. Перепишите программы `us_xfr_sv.c` (см. листинг 53.3) и `us_xfr_cl.c` (см. листинг 53.4) с помощью абстрактного пространства имен сокетов (см. раздел 53.6).

53.3. Заново реализуйте клиент-серверное приложение для генерирования числовых последовательностей (см. раздел 44.8), задействуя потоковые сокеты домена UNIX.

53.4. Представьте: мы создали два датаграммных сокета в домене UNIX, привязанных к путям `/somepath/a` и `/somepath/b`, и соединили их друг с другом. Что произойдет, если мы создадим третий датаграммный сокет и попытаемся с его помощью передать сообщение сокету `/somepath/a` (с помощью вызова `sendto()`)? Напишите программу, которая отвечает на этот вопрос. По возможности проверьте, как будет вести себя данная программа в других UNIX-системах.

54 Сокеты: основы сетей TCP/IP

В этой главе вы познакомитесь с основными концепциями компьютерных сетей и сетевых протоколов TCP/IP. Понимание данной темы является обязательным для эффективного использования сокетов интернет-домена, описанных в следующей главе.

Здесь мы начнем упоминать различные RFC-документы (от англ. Request for Comments — «рабочие предложения»). В них находится формальное описание каждого сетевого протокола, который будет обсуждаться в данной книге. Больше о них можно узнать в разделе 54.7; там же перечислены RFC-документы, имеющие особую важность с точки зрения представленных здесь тем.

54.1. Интерсети

Интерсеть (или *интернет* с маленькой буквы *«и»*) соединяет разные компьютерные сети, позволяя взаимодействовать входящим в них компьютерам. Иными словами, это сеть компьютерных сетей. Составные части интерсетей называют *подсетями*. Интерсеть призвана инкапсулировать разные физические сети, представляя всем подключенным к ней компьютерам унифицированную сетевую архитектуру. Это, к примеру, означает, что для идентификации всех узлов интерсети используются адреса в едином формате.

Несмотря на существование множества межсетевых протоколов, именно технология TCP/IP стала доминирующей, вытеснив даже коммерческие решения, которые когда-то применялись в локальных и глобальных сетях. Интерсеть на основе протокола TCP/IP, соединяющая миллионы компьютеров по всему миру, называется *Интернетом* (с прописной буквы *«И»*).

Первая широко распространенная реализация TCP/IP была разработана для системы 4.2BSD в 1983 году. Непосредственно из данного кода происходит несколько других реализаций, хотя некоторые версии TCP/IP, в том числе и та, что применяется в Linux, были созданы с нуля, а код системы BSD использовался в качестве эталонного образца.

> Начало стека TCP/IP было положено в проекте, спонсируемом Управлением перспективных исследовательских проектов Министерства обороны США (англ. US Department of Defense Advanced Research Projects Agency — ARPA, а позже DARPA), целью которого являлась разработка сетевой компьютерной архитектуры для одной из первых глобальных сетей под названием ARPANET. Если быть точным, используемая там технология называлась пакетом межсетевых протоколов DARPA, но широкой публике она известна как семейство протоколов TCP/IP, или просто TCP/IP.
>
> Краткая история Интернета и TCP/IP приведена на веб-странице http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet.

Простая интерсеть представлена на рис. 54.1. На этой диаграмме компьютер `tekapo` играет роль *маршрутизатора* — узла, соединяющего одну подсеть с другой, передавая данные между ними. Помимо действующего межсетевого протокола, маршрутизатор

должен понимать протоколы второго (канального) уровня, используемые в каждой соединяемой подсети (они могут быть разными).

Маршрутизатор имеет несколько сетевых интерфейсов, по одному для каждой подсети, к которой он подключен. Такие компьютеры (это могут быть не только маршрутизаторы) называются *многоадресными* (маршрутизатор можно считать многоадресным узлом, переправляющим пакеты из одной подсети в другую). Как понятно из названия, у каждого интерфейса многоадресного узла есть отдельный адрес (для каждой подсети, к которой подключен данный узел).

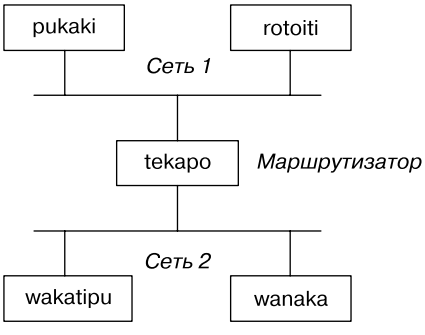


Рис. 54.1. Интерсеть использует маршрутизатор для соединения двух подсетей

54.2. Сетевые протоколы и уровни

Сетевой протокол — набор правил, определяющий способ передачи информации по сети. Такие протоколы в целом делятся на *уровни*, которые «наслаиваются» друг на друга, добавляя все новые возможности.

Семейство протоколов TCP/IP представляет собой многоуровневый протокол (рис. 54.2). Он состоит из протокола IP (от англ. Internet Protocol) и разных других протоколов, функционирующих поверх него (код, реализующий эти уровни, обычно называют *стеком протоколов*). Свое название стек TPC/IP получил ввиду того факта, что TCP (от англ. Transmission Control Protocol) является основным протоколом транспортного уровня.

> На рис. 54.2 нет целого ряда протоколов, основанных на TCP/IP, поскольку они не имеют отношения к данной главе. Протокол ARP (от англ. Address Resolution Protocol) предназначен для связывания интернет-адресов с физическими адресами (например, Ethernet). Протокол ICMP (от англ. Internet Control Message Protocol) служит для уведомления об ошибках и управления информацией внутри сети (он применяется в программе ping, с помощью которой обычно проверяют работоспособность и доступность узла в сети TCP/IP, а также в утилите traceroute, которая отслеживает путь, проделанный IP-пакетом внутри сети). Протокол IGMP (от англ. Internet Group Management Protocol) используется в обычных узлах и маршрутизаторах для поддержки многоадресной передачи IP-датаграмм.

Одним из принципов, который делает представленную многоуровневую структуру такой мощной и гибкой, является *прозрачность* — каждый уровень инкапсулирует все операции и всю сложность более низких уровней. Например, приложение, применяющее TCP, должно знать только о стандартном программном интерфейсе для работы с сокетами и о том, что это надежный транспортный протокол на основе байтовых потоков.

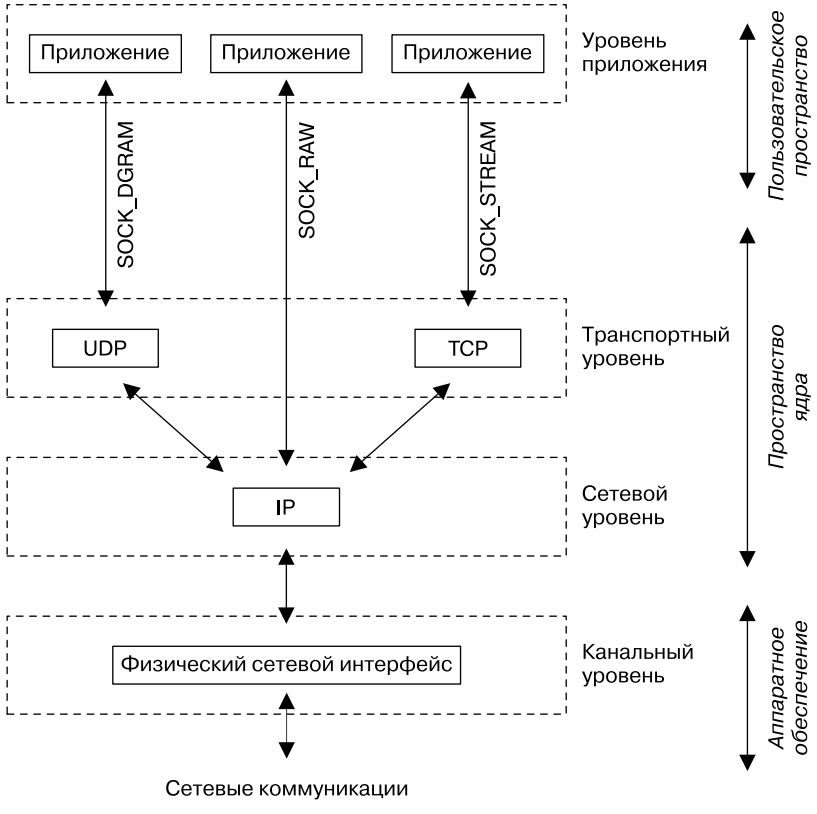


Рис. 54.2. Протоколы семейства TCP/IP

Принцип работы TCP в данном случае неважен (при рассмотрении параметров сокетов в разделе 57.9 вы увидите, что это не всегда так; в ряде случаев приложение должно знать о некоторых подробностях работы исходного транспортного протокола). Неважен также принцип работы протокола IP или канального уровня. Для приложений все выглядит так, будто они взаимодействуют друг с другом напрямую через программный интерфейс сокетов; это продемонстрировано на рис. 54.3, где пунктирные горизонтальные линии обозначают маршруты между соответствующими программами, а также протоколами TCP и IP на обоих узлах.

Инкапсуляция

Это важный аспект многоуровневого сетевого протокола. Пример инкапсуляции протоколов семейства TCP/IP показан на рис. 54.4. Ключевая идея заключается в том, что при поступлении с более высокого уровня информация (например, данные программы, TCP-сегмент или IP-датаграмма) никак не интерпретируется, а просто помещается в пакет того типа, который используется в текущем протоколе, маркируется соответствующим заголовком и передается на уровень ниже. При переходе снизу вверх происходит обратный процесс: данные распаковываются.

> На рис. 54.4 этого не видно, но принцип инкапсуляции распространяется на канальный уровень, где IP-датаграммы упаковываются в так называемые сетевые кадры. То же самое можно сказать и об уровне приложения, где данные могут быть упакованы каким-то особым образом.

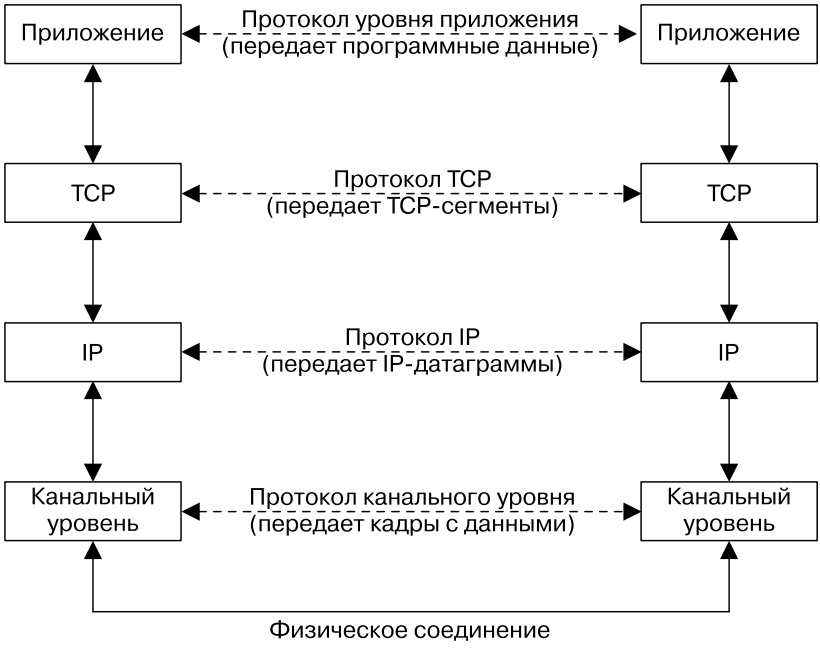


Рис. 54.3. Многоуровневое взаимодействие на основе протоколов TCP/IP

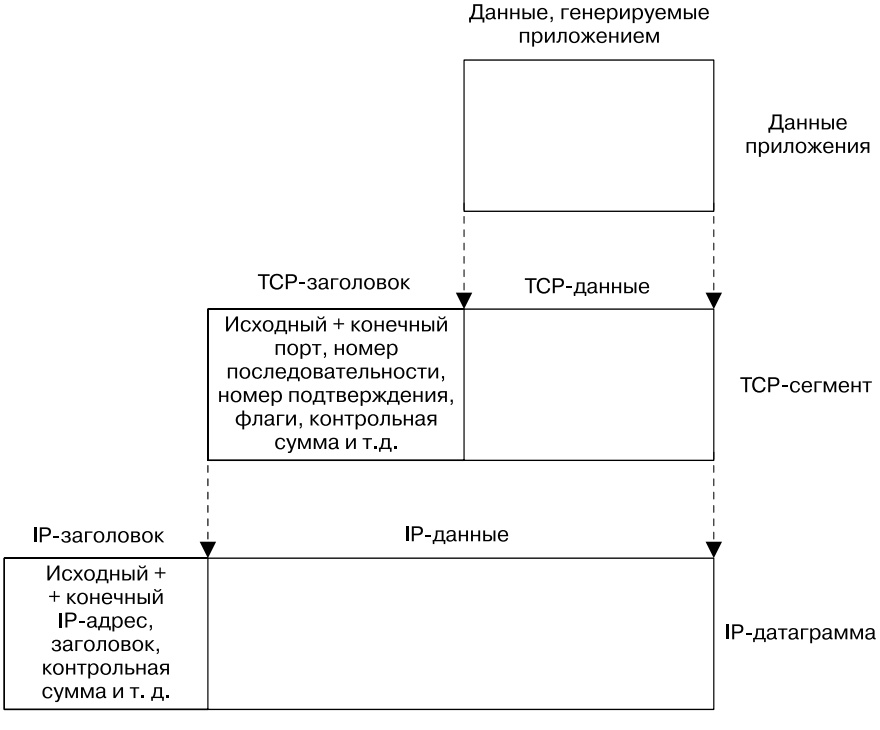


Рис. 54.4. Инкапсуляция на разных уровнях TCP/IP

54.3. Канальный уровень

Самым низким уровнем на рис. 54.2 является *канальный*; он состоит из драйвера устройства и физического интерфейса (сетевой карты), соединенного с реальными сетевыми коммуникациями (например, с телефонным проводом, коаксиальным или оптическим кабелем). Этот уровень предназначен для физической передачи данных внутри сети.

Передавая информацию, канальный уровень упаковывает датаграммы из сетевого уровня в блоки под названием *«кадры»*. Помимо самих данных, каждый кадр содержит заголовок с дополнительными сведениями — например, адрес получателя и размер кадра. Канальный уровень передает кадры по физическому соединению и обрабатывает подтверждение со стороны получателя (подтверждения используются не во всех канальных протоколах). Этот уровень может брать на себя обнаружение ошибок, повторную передачу и управление потоком данных. Некоторые протоколы канального уровня способны также разбивать сетевые пакеты на отдельные кадры и затем собирать их обратно на стороне получателя.

С точки зрения прикладного программирования канальный протокол в целом можно игнорировать, так как все его аспекты реализованы на уровне драйвера и аппаратного обеспечения.

Одной из особенностей канального уровня, играющей важную роль в понимании протокола IP, является максимальный размер передаваемого блока информации (англ. maximum transmission unit, MTU). В настоящем контексте это максимальный размер кадра. Данное значение может отличаться в разных протоколах канального уровня.

Команда netstat -i выводит список сетевых интерфейсов в системе, включая их MTU.

54.4. Сетевой уровень: IP

Над канальными протоколами находится *сетевой уровень*, который занимается доставкой пакетов (данных) от исходного узла к конечному. Этот уровень выполняет целый ряд разных задач, включая следующие:

- разбиение данных на небольшие фрагменты, подходящие для передачи через канальный уровень (если нужно);
- маршрутизацию данных по интерсети;
- вспомогательные функции для транспортного уровня.

В стеке TCP/IP основным протоколом является сетевой уровень IP. Реализация IP, которая появилась в системе 4.2BSD, имела версию 4 (IPv4). В начале 1990-х была разработана новая, шестая версия данного протокола (IPv6). Наиболее заметная разница между ними заключается в том, что для определения подсетей и узлов в IPv4 используются 32-разрядные адреса, а в IPv6 — 128-разрядные (это делает доступным куда более широкий диапазон адресов). IPv4 все еще преобладает на просторах Интернета, однако в ближайшие несколько лет он должен быть вытеснен протоколом IPv6. Обе версии IP поддерживают транспортные протоколы более высокого уровня, в том числе TCP и UDP.

Тридцатидвухразрядное пространство имен, которое используется в сетях IPv4, теоретически позволяет поддерживать миллиарды узлов, однако из-за способа структурирования и выделения адресов их реальное количество оказалось значительно меньшим. Именно потенциальная нехватка IPv4-адресов послужила одной из главных причин создания IPv6.

Краткую историю протокола IPv6 можно найти на странице www.laynetworks.com/IPv6.htm.

В связи с существованием протоколов IPv4 и IPv6 напрашивается вопрос: «А что насчет IPv5?» Такой версии никогда не существовало. Заголовок каждой IP-датаграммы включает

четырехразрядное поле для хранения номера версии (IPv4-датаграммы всегда хранят в этом поле значение 4), а номер 5 был выделен для экспериментального протокола Internet Stream Protocol (его вторая версия, известная как ST-II, описана в документе RFC 1819). Данный протокол, ориентированный на соединения, был создан в 1970-х и предназначался для передачи голоса, видео и распределенного моделирования. В связи с тем что номер 5 уже был занят, наследником протокола IPv4 стала версия 6.

На рис. 54.2 показан *сырой* сокет (`SOCK_RAW`), который позволяет приложениям взаимодействовать непосредственно на уровне IP. Мы не станем рассматривать здесь сырые сокеты, поскольку большинство приложений задействует транспортные протоколы (TCP или UDP). Больше подробностей можно узнать в главе 28 книги [Stevens et al., 2004]. Одним из показательных примеров использования сырых сокетов является консольная программа `sendip` (http://www.earth.li/projectpurple/progs/sendip.html), позволяющая создавать и передавать IP-датаграммы с произвольным содержимым (с опциональной поддержкой UDP-датаграмм и TCP-сегментов).

Протокол IP передает датаграммы

Протокол передает данные в виде датаграмм (пакетов). Каждая из них, передаваемая от одного узла к другому, перемещается по сети независимо и может иметь уникальный маршрут. IP-датаграмма содержит заголовок, размер которого варьируется от 20 до 60 байт. В нем хранится адрес получателя, чтобы датаграмма могла быть направлена в пункт назначения, а также адрес отправителя для определения получателем ее происхождения.

Отправляющий узел может подменить исходный адрес пакета. Это является основой для простой DoS-атаки по протоколу TCP, известной как SYN-флуд (англ. SYN-flood). [Lemon, 2002] описывает детали такой атаки и меры защиты от нее, предпринимаемые современными реализациями TCP.

Реализация протокола IP может устанавливать ограничение на максимальный размер датаграммы. Существует также минимальный размер, который реализация обязана поддерживать; он равен *размеру буфера повторной сборки* IP. В IPv4 это ограничение установлено на уровне 576 байт, а в IPv6 — 1500 байт.

Протокол IP не поддерживает соединения и является ненадежным

В описании протокола IP отмечается: он *не поддерживает соединения*, так как в нем нет понятия виртуальной цепи, связывающей два узла. Кроме того, этот протокол является *ненадежным*: он делает «все возможное» для передачи датаграмм от отправителя к получателю, но не гарантирует поступление пакетов в порядке их отправки, а также то, что они не будут продублированы и вообще дойдут. Протокол IP также не обеспечивает восстановление данных после ошибки (пакеты с ошибочными заголовками просто отклоняются). Надежность должна предоставляться с помощью транспортного протокола (например, TCP) или самого приложения.

Протокол IPv4 вычисляет контрольную сумму заголовка, что позволяет определять связанные с ним ошибки. Однако ошибки, которые происходят с данными, передающимися внутри пакетов, остаются без внимания. Разработчики IPv6 отказались от контрольной суммы заголовка и возложили ответственность за проверку ошибок и надежность на протоколы более высокого уровня. (Протокол UDP тоже поддерживает контрольные суммы; в IPv4 эта возможность является опциональной, а в IPv6 — обязательной; вычисление контрольных сумм в протоколе TCP является обязательным вне зависимости от версии IP.)

Дублирование IP-датаграмм может возникнуть из-за методик, применяемых некоторыми протоколами канального уровня для надежной передачи данных, или в результате прохождения

IP-датаграмм через сеть, не основанную на стеке TCP/IP, поддерживающую повторную передачу пакетов.

Протокол IP не исключает фрагментации датаграмм

IPv4-датаграммы могут достигать 65 535 байт. Протокол IPv6 поддерживает датаграммы размером 65 575 байт (40 байт для заголовка, 65 535 — для данных) и позволяет передавать сообщения еще большего размера (так называемые *джамбограммы* — англ. jumbograms).

Ранее мы отмечали: большинство протоколов канального уровня налагает ограничение на максимальный размер блоков данных (MTU). Например, в широко используемой архитектуре Ethernet он равен 1500 байтам (что намного меньше максимального размера IP-датаграммы). Кроме того, в протоколе IP существует понятие *пути MTU*. Это минимальное значение MTU на всех канальных уровнях маршрута, пройденного от источника к конечной цели (в Ethernet значение MTU часто равно минимальному MTU на маршруте).

Если датаграмма превышает MTU, то протокол IP разбивает ее на фрагменты, чьи размеры подходят для передачи по сети. В точке назначения они собираются обратно в исходную датаграмму (каждый такой фрагмент сам является датаграммой с полем, содержащим сдвиг, который позволяет определить его местоположение в оригинальном сообщении).

Эта фрагментация хоть и не видна протоколам более высокого уровня, но все равно считается нежелательной (см. [Kent & Mogul, 1987]). Дело в том, что протокол IP не поддерживает повторную передачу, а итоговую датаграмму можно собрать только в том случае, если до пункта назначения дошли все ее фрагменты. В результате при потере хотя бы одного фрагмента или в случае возникновения ошибки передачи мы теряем всю датаграмму целиком. В некоторых ситуациях это может привести к существенному увеличению процента потерянных данных (в случае с протоколами более высокого уровня, такими как UDP, не выполняющими повторную передачу пакетов) или снижению скорости (в случае с такими протоколами, как TCP, поддерживающими повторную передачу). Современные реализации TCP используют специальный алгоритм *(обнаружение пути MTU)* для определения маршрута MTU между двумя узлами и соответствующего разбиения данных, передаваемых протоколу IP, чтобы тот, в свою очередь, сразу передавал датаграммы подходящего размера. UDP не предоставляет подобного механизма; в подразделе 54.6.2 вы увидите, каким образом приложения на основе приведенного протокола могут справляться с потенциальной фрагментацией пакетов на уровне IP.

54.5. IP-адреса

IP-адрес состоит из двух частей: идентификатора сети, в которой находится узел, и идентификатора самого узла.

Адреса протокола IPv4

В IPv4 адрес занимает 32 бита (см. рис. 54.5). Его обычно записывают в виде десятичных чисел (всего 4 байта), разделенных точками, например, `204.152.189.116`.

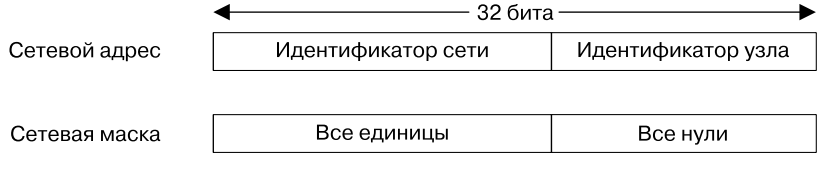


Рис. 54.5. Структура сетевого адреса и соответствующей сетевой маски в IPv4

Когда организация подает заявку на получение диапазона адресов формата IPv4, она получает 32-разрядный сетевой адрес и соответствующую 32-разрядную *маску сети*. В двоичном виде крайние левые биты этой маски состоят из единиц; остальные биты заполнены нулями. Единица указывает на то, где в адресе находится идентификатор сети, тогда как нули определяют, какая часть адреса доступна для назначения уникальных идентификаторов сетевым узлам. Размер первой части маски определяется во время назначения адреса. Поскольку идентификатор сети всегда находится слева, для описания диапазона адресов достаточно записи следующего вида:

```
204.152.189.0/24
```

`/24` означает, что часть адреса, содержащая идентификатор сети, состоит из первых 24 бит, а оставшиеся 8 бит отводятся для идентификатора узла. Ту же самую маску можно было бы определить как `255.255.255.0`.

Организация, владеющая этим диапазоном, имеет право назначить своим компьютерам 254 уникальных интернет-адреса — с `204.152.189.1` по `204.152.189.254`. Существует два адреса, которые нельзя назначить: первый — тот, чей идентификатор узла полностью состоит из нулей (которые используются для идентификации самой сети), а второй имеет адрес узла, полностью состоящий из единиц (что является *широковещательным адресом подсети* — в нашем случае это `204.152.189.255`).

В протоколе IPv4 определенные адреса имеют специальное назначение. Например, `127.0.0.1` обычно является адресом, замкнутым на себя, и назначается узлу с именем `localhost` (эту роль может играть любой IPv4-адрес в подсети 127.0.0.0/8, однако `127.0.0.1` является общепринятым выбором). Датаграмма, посланная по данному адресу, на самом деле никогда не попадает в сеть, а возвращается и принимается тем же узлом, который ее отправил. Этот адрес принято задействовать для тестирования клиент-серверных программ в локальной среде. В программах на языке C он определен в виде константы `INADDR_LOOPBACK`.

Константа `INADDR_ANY` в IPv4 обозначает так называемый *универсальный адрес*. Он используется приложениями, привязывающими сокет интернет-домена к узлу с множественной адресацией. Если приложение, находящееся на таком узле, привяжет свой сокет только к одному IP-адресу, то сможет принимать UDP-датаграммы или TCP-соединения, направленные только на данный адрес. Однако такие приложения обычно должны принимать датаграммы или соединения, в которых указан любой из адресов соответствующего узла, и применение универсального адреса делает это возможным. В стандарте SUSv3 для константы INADDR_ANY не предусмотрено какого-то определенного значения, но в большинстве реализаций она равна `0.0.0.0` (все нули).

Обычно IPv4-адреса делят на *подсети*. Идентификатор узла после этого становится меньше, а в освободившуюся часть адреса записывается идентификатор подсети (рис. 54.6). Способ разделения битов идентификатора узла определяется местным сетевым администратором. Так делается в связи с тем, что организации часто не хотят размещать все свои компьютеры в единой сети. Вместо этого может использоваться набор подсетей («внутренних интерсетей»), каждая из которых определяется с помощью сочетания собственного ID и идентификатора всей сети. Это сочетание часто называют *расширенным сетевым идентификатором*. Маска подсети выполняет ту же функцию, что и маска сети, описанная ранее; для ее представления можно задействовать похожую запись, определяющую диапазон адресов, входящих в некую подсеть.

Представьте, например, что идентификатор нашей сети равен `204.152.189.0/24` и мы решили разделить этот диапазон адресов, выделив из 8-разрядного идентификатора узла 4 бита на идентификатор подсети. В результате маска подсети будет состоять из 28 единиц, за которыми следуют четыре нуля, а подсеть с идентификатором 1 будет иметь вид `204.152.189.16/28`.

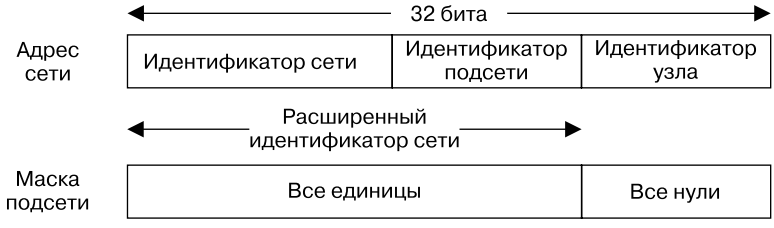


Рис. 54.6. Разделение на подсети в IPv4

Адреса протокола IPv6

Протоколы IPv6 и IPv4 имеют похожую структуру адресов. Ключевое различие заключается в том, что в IPv6 адреса состоят из 128 бит, и несколько первых битов отводится под *префикс формата*, который указывает на тип адреса (мы не станем детально рассматривать разные типы адресов; подробности см. в приложении A книги [Stevens et al., 2004] и в документе RFC 3513).

IPv6-адреса обычно записываются в виде последовательности шестнадцатеричных чисел, разделенных двоеточиями. Например:

```
F000:0:0:0:0:0:A:1
```

IPv6-адреса часто включают в себя несколько нулей, идущих подряд; для обозначения данной последовательности можно использовать два двоеточия (`::`). Таким образом, адрес, приведенный выше, можно записать в следующем виде:

```
F000::A:1
```

IPv6 допускает применение в адресе только одного двойного двоеточия; несоблюдение этого правила приведет к непредсказуемым результатам.

Протокол IPv6, как и IPv4, поддерживает замкнутый на себя адрес (127 нулей, за которыми следует одна единица, — то есть `::1`) и универсальный адрес (все нули — `0::0` или `::`).

Чтобы приложения, основанные на IPv6, могли взаимодействовать с узлами, поддерживающими только IPv4, были разработаны адреса, совместимые с IPv4. Их формат показан на рис. 54.7.

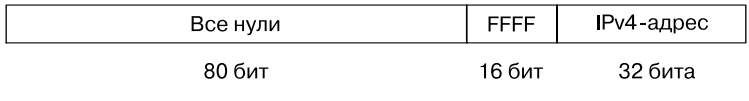


Рис. 54.7. Формат адресов, совместимых с IPv4

При записи такого адреса часть, совместимая с IPv4 (то есть последние 4 байта), записывается в виде десятичных чисел, разделенных точками. Таким образом, в IPv6 адрес `204.152.189.116` будет иметь вид `::FFFF:204.152.189.116`.

54.6. Транспортный уровень

В стеке TCP/IP существует два широко используемых транспортных протокола:

- *протокол пользовательских датаграмм* (англ. User Datagram Protocol, UDP), предназначенный для датаграммных сокетов;

- *протокол управления передачей* (англ. Transmission Control Protocol, TCP), предназначенный для потоковых сокетов.

Прежде чем перейти к их рассмотрению, следует сначала ознакомиться с таким понятием, как номера портов, которое применяется в обоих этих протоколах.

54.6.1. Номера портов

Задача транспортного протокола состоит в предоставлении сквозного канала, позволяющего взаимодействовать приложениям, расположенным на разных компьютерах (или иногда на одном компьютере). Чтобы ее выполнить, данному протоколу нужно как-то различать разные приложения на одном и том же узле. В TCP и UDP это делается с помощью *16-разрядного номера порта*.

Общеизвестные, зарегистрированные и привилегированные порты

Часть общеизвестных номеров портов строго привязаны к определенным приложениям (также известным как *сервисы*). Например, демон `ssh` (защищенная командная оболочка) использует общеизвестный порт 22, а протоколу HTTP, через который взаимодействуют веб-серверы и браузеры, назначен номер 80. Распределением портов в диапазоне от 0 до 1023 занимается Администрация адресного пространства Интернет (англ. Internet Assigned Numbers Authority, IANA, http://www.iana.org/). Назначение такого номера основывается на одобренной сетевой спецификации (обычно в виде документа RFC).

IANA также ведет учет *зарегистрированных портов*, выделяемых разработчикам приложений с соблюдением менее строгих правил (это значит, что программа не должна гарантировать доступность указанных портов для выполнения заявленных задач). Диапазон зарегистрированных портов, которыми заведует IANA, представлен номерами с 1024 по 41 951 (не все номера в данном диапазоне зарегистрированы).

Актуальный список общеизвестных и зарегистрированных номеров портов находится на https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

В большинстве реализаций TCP/IP (в том числе и в Linux) номера портов в диапазоне от 0 до 1023 являются привилегированными — то есть только привилегированные процессы (`CAP_NET_BIND_SERVICE`) могут подключаться к этим портам. Данное обстоятельство не дает обычному пользователю написать программу, которая, например, подменяет демон `ssh`, чтобы перехватить пароли (иногда привилегированные порты называют *зарезервированными*).

Один и тот же порт можно использовать отдельно в TCP и UDP, однако на практике общеизвестные номера портов обычно назначают сразу обоим протоколам, даже если сервис работает только по одному из них (часто так и происходит). Это позволяет избежать путаницы с назначением портов в разных протоколах.

Динамические порты

Если приложение само не выберет определенный номер порта (то есть когда сокет не привязывается к конкретному порту с помощью `bind()`), то протоколы TCP и UDP назначат ему временный *динамический порт*. В данном случае приложение (обычно клиент) не заботит номер порта, который оно использует, однако порт все же должен быть назначен: таким образом транспортные протоколы смогут идентифицировать конечные точки взаимодействия. Это требуется и для того, чтобы удаленная программа на другом конце канала знала, как именно ей взаимодействовать с приложением. Протоколы TCP и UDP также назначают динамический порт при попытке привязать сокет к порту с номером 0.

С точки зрения организации IANA номера в диапазоне с 49 152 по 65 535 являются динамическими или приватными; они могут применяться локальными приложениями и назначаться в качестве динамических портов. Однако в некоторых системах для таких портов выделены другие диапазоны. В Linux этот диапазон определяется двумя номерами, содержащимися в файле `/proc/sys/net/ipv4/ip_local_port_range` (при желании их можно поменять).

54.6.2. Протокол пользовательских датаграмм (UDP)

UDP привносит в протокол IP ровно две возможности: номера портов и контрольные суммы для данных, которые позволяют обнаруживать ошибки в переданной информации.

Как и IP, протокол UDP не поддерживает соединения и является в такой же степени ненадежным. Если приложению, работающему поверх UDP, требуется некая надежность, то оно должно реализовать ее самостоятельно. Но, несмотря на это, протокол UDP иногда оказывается более предпочтительным по сравнению с TCP; подробную информацию по теме см. в разделе 57.12.

> Контрольные суммы, применяемые в UDP и TCP, имеют длину всего 16 бит и являются дополнительным средством, которое может не обнаружить ошибки определенного типа. Следовательно, они не обеспечивают абсолютно надежного обнаружения ошибок. Нагруженные интернет-серверы сталкиваются с одной необнаруженной ошибкой передачи данных в среднем раз в несколько дней ([Stone & Partridge, 2000]). Приложения, нуждающиеся в более жестком контроле целостности данных, могут использовать протокол уровня защищенных сокетов (англ. Secure Sockets Layer, SSL), обеспечивающий не только защищенное взаимодействие, но и более тщательную проверку на ошибки. Одним из вариантов также является реализация собственного механизма мониторинга ошибок.

Выбор размера UDP-датаграмм, позволяющего избежать фрагментации на уровне IP

В разделе 54.4 мы описали механизм фрагментации в протоколе IP и отметили, что обычно его лучше избегать. Протокол TCP делает это автоматически, тогда как в UDP для фрагментации достаточно передать датаграмму, чей размер превышает значение MTU на локальном канальном уровне.

Приложения, основанные на UDP, обычно не знают о размере MTU на маршруте между исходным и конечным узлами. Для обхода IP-фрагментации обычно выбирается консервативный подход, состоящий в выборе размера IP-датаграммы, не превышающего минимальный размер буфера сборки в IPv4, который равен 476 байтам (это значение, скорее всего, должно быть меньше MTU заданного маршрута). Восемь байт уходит на заголовок UDP, и еще минимум 20 нужно для IP-заголовка, что оставляет для самой UDP-датаграммы всего 548 байт. На практике многие приложения, основанные на UDP, выбирают для своих датаграмм еще более низкое ограничение — 512 байт (см. [Stevens, 1994]).

54.6.3. Протокол управления передачей (TCP)

Данный протокол предоставляет надежный двунаправленный канал взаимодействия двух точек (то есть приложений), основанный на соединении и реализованный в виде байтового потока (рис. 54.8). Для обеспечения всех этих возможностей протокол TCP

должен выполнять действия, описанные в настоящем подразделе (подробное описание возможностей можно найти в книге [Stevens, 1994]).

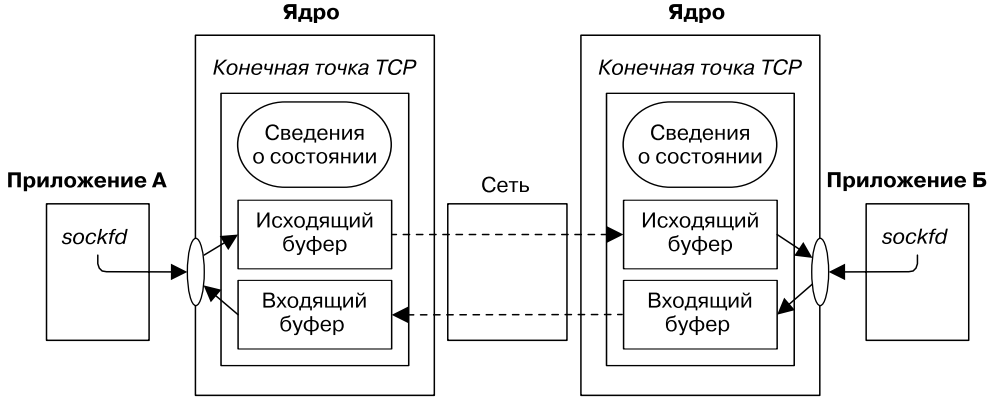


Рис. 54.8. Сокеты, соединенные по TCP

Мы задействуем термин *«конечная точка»* для обозначения информации об одном из концов TCP-соединения, хранящейся в ядре (часто эту точку, принадлежащую клиентскому приложению, мы будем называть просто «TCP-клиентом»). Данная информация включает в себя исходящий и входящий буферы для текущего конца соединения, а также состояние, применяемое для синхронизации работы двух конечных точек (более подробно о состоянии мы поговорим в подразделе 57.6.3, когда познакомимся с диаграммой перехода состояний в протоколе TCP). Далее в этой книге конечные точки TCP-соединения, которые используются на разных концах канала, основанного на потоковых сокетах, для передачи данных в определенном направлении, будут называться *принимающим* и *отправляющим* концами.

Установка соединения

Прежде чем начать взаимодействие, протокол TCP устанавливает соединение между двумя конечными точками. Во время этой процедуры отправитель и получатель могут обменяться параметрами, которые они хотят назначить соединению.

Упаковка данных в сегменты

Данные разбиваются на сегменты, каждый из которых содержит контрольную сумму, что позволяет обнаруживать ошибки сквозной передачи. Каждый сегмент передается в отдельной IP-датаграмме.

Подтверждение, повторная передача и время ожидания

Когда TCP-сегмент доходит до пункта назначения без ошибок, получатель посылает отправителю подтверждение, информируя его об успешной доставке данных. При получении сегмента с ошибками он отклоняется, а отправка подтверждения не происходит. Чтобы как-то реагировать на ситуации, когда сегмент не доходит или отклоняется, в момент его передачи отправитель устанавливает таймер. Если время ожидания истекло, а подтверждение все еще не пришло, то сегмент отправляется повторно.

Время, уходящее на передачу сегмента и получение подтверждения, варьируется в зависимости от пропускной способности сети и ее загруженности, поэтому протокол TCP применяет алгоритм автоматического корректирования времени ожидания.

Иногда принимающая сторона задерживает отправку подтверждения, пытаясь по возможности «прицепить» его к какому-то ответу, который направляется непосредственно отправителю (каждый TCP-сегмент содержит поле подтверждения, которое можно использовать для этих целей). Данная методика называется отложенным подтверждением, и цель ее заключается в уменьшении количества отправляемых пакетов и снижении загруженности отправляющего и принимающего узлов.

Порядок передачи

Каждому байту, который передается по TCP-соединению, назначается порядковый номер. Он определяет местоположение байта в потоке данных внутри соединения (каждый из двух потоков в соединении выполняет отдельную нумерацию). Передаваемый TCP-сегмент содержит порядковый номер своего первого байта.

Назначение порядкового номера каждому сегменту имеет несколько целей.

- Это позволяет получателю собрать TCP-сегменты в корректном порядке и передать их на уровень приложения в виде байтового потока (в любой момент времени между отправителем и получателем может находиться несколько TCP-сегментов, порядок следования которых был изменен).
- Сообщение о подтверждении, отсылаемое получателем обратно отправителю, может использовать порядковый номер для определения того, какой именно TCP-сегмент был получен.
- Получатель может задействовать порядковый номер для удаления продублированных сегментов. Дублирование может возникнуть либо на уровне IP-датаграмм, либо в результате работы алгоритма повторной передачи в протоколе TCP, который может еще раз послать успешно доставленный сегмент, если соответствующее подтверждение было утеряно или пришло слишком поздно.

Исходный порядковый номер (англ. initial sequence number, ISN) для потока начинается не с нуля. Он генерируется специальным алгоритмом, который инкрементирует ISN, назначенный новому соединению (чтобы старые сегменты из предыдущих соединений не могли быть перепутаны с новыми). Данный алгоритм генерирует сложноугадываемый номер ISN. Это 32-разрядное число, сбрасываемое к нулю при достижении максимального значения.

Управление потоком

Механизм управления потоком позволяет избежать ситуаций, когда быстрый отправитель «забрасывает» пакетами медленного получателя. Для его реализации в протоколе TCP предусмотрен буфер, который хранит входящие данные (каждая сторона TCP-соединения сообщает размер своего буфера в момент подключения). По мере передачи данные накапливаются в этом буфере и удаляются из него, когда приложение их считывает. Вместе с каждым подтверждением получатель шлет отправителю сведения о том, сколько свободного места осталось во входящем буфере (то есть сколько байтов отправитель может передать). Механизм управления потоком в протоколе TCP задействует так называемый *алгоритм скользящего окна*, который следит за тем, чтобы общий объем неподтвержденных сегментов между отправителем и получателем не превышал `N` байтов (предложенный размер окна). Если входящий буфер получателя

полностью заполняется, то окно считается закрытым, а отправляющая сторона останавливает передачу.

Получатель может переопределить стандартный размер входящего буфера, используя параметр сокета SO_RCVBUF (см. страницу socket(7) руководства).

Контроль над перегрузкой: алгоритмы замедленного старта и предотвращения перегрузки

Алгоритмы такого контроля, применяющиеся в TCP, предназначены для того, чтобы не дать быстрому отправителю переполнить сеть. Если отправляющая сторона передает пакеты быстрее, чем промежуточный маршрутизатор может их перенаправить, то это приводит к потере пакетов. Такая ситуация, в свою очередь, может вызвать серьезное ухудшение производительности, если отправляющая сторона продолжит передавать данные с той же скоростью. Алгоритмы контроля над перегрузкой играют важную роль в двух случаях.

- *После установки соединения.* На данном этапе (или когда возобновляется передача информации в ранее неактивном соединении) отправитель мог бы сразу послать максимальное количество сегментов, которое влезет в окно, афишируемое получателем (на самом деле именно это и делали ранние реализации протокола TCP). Но если сеть не может справиться с таким потоком сегментов, то отправитель сразу же рискует переполнить сеть.
- *При обнаружении перегрузки.* При обнаружении отправителем перегрузки он должен снизить скорость передачи. Это обнаружение основывается на предположении о том, что процент потери пакетов в результате ошибок передачи является крайне низким; следовательно, если пакет утерян, то в качестве причины подразумевается перегрузка.

Стратегия контроля над перегрузкой в протоколе TCP состоит из двух алгоритмов: замедленного старта и предотвращения перегрузки.

Алгоритм замедленного старта заставляет TCP начинать передачу данных на низкой скорости, но позволяет экспоненциально увеличивать ее по мере получения подтверждений от принимающей стороны. За счет медленного начала протокол TCP пытается избежать переполнения сети слишком быстрым отправителем. Но если не контролировать экспоненциальный рост скорости передачи, то отправитель рано или поздно получит возможность перегрузить сеть. Чтобы этого избежать и ограничить потенциальный рост скорости, протокол TCP использует *алгоритм предотвращения перегрузки.*

Когда включено предотвращение перегрузки, отправитель сразу после установки соединения начинает с небольшого *окна*, ограничивающего объем подтвержденных данных, доступных для передачи. По мере получения подтверждений от удаленного узла это окно экспоненциально расширяется. Но, достигнув некоего потолка, который, как считается, находится недалеко от пропускной способности сети, рост переходит в линейную стадию. (Приблизительная пропускная способность сети выводится из вычислений, основанных на скорости передачи в момент фиксирования перегрузки, или же назначается явно во время установки соединения.) При этом объем данных, передаваемый отправляющей стороной, дополнительно ограничивается скользящим окном, которое афишируется получателем, а также локальным исходящим буфером протокола TCP.

Сочетание этих двух алгоритмов позволяет отправителю быстро увеличивать скорость передачи, используя всю доступную пропускную способность сети, но не

превышая ее. В результате удается быстро достичь состояния равновесия, когда отправитель передает пакеты с той же скоростью, с которой он получает подтверждение от получателя.

54.7. Документы, выносимые на рассмотрение (RFC)

Все интернет-протоколы, которые мы обсуждаем в нашей книге, оформлены в виде RFC-документов, содержащих их формальную спецификацию. Они публикуются на веб-сайте http://www.rfc-editor.org/, финансируемом организацией Internet Society (http://www.internetsociety.org/). Документы, описывающие интернет-стандарты, разрабатываются под эгидой организации Internet Engineering Task Force (IETF, www.ietf.org). Она представляет собой сообщество сетевых архитекторов, операторов, поставщиков услуг и исследователей, заинтересованных в развитии и стабильной работе Интернета. Членом IETF может стать любой человек, которому близка эта тематика.

Непосредственное отношение к материалу, рассматриваемому в данной книге, имеют следующие RFC-документы (их перевод можно найти на сайте rfc2.ru):

- RFC 791, *Протокол IP*. Дж. Постел (ред.), 1981;
- RFC 950, *Стандартные процедуры организации подсетей IP*. Дж. Могул, Дж. Постел, 1985;
- RFC 793, *Протокол управления передачей (TCP)*. Дж. Постел (ред.), 1981;
- RFC 768, *Протокол датаграмм клиента (UDP)*. Дж. Постел (ред.), 1980;
- RFC 1122, *Требования к хостам Internet. Коммуникационные уровни*. Р. Брэйден (ред.), 1989.

RFC 1122 дополняет (и исправляет) ряд более старых документов, описывающие протоколы семейства TCP/IP. Это один из двух RFC-документов, которые часто встречаются под названием «RFC-требования к узлам». Второй имеет номер 1123 и посвящен прикладным протоколам, таким как telnet, FTP и SMTP.

54.8. Резюме

TCP/IP представляет собой многоуровневый пакет сетевых протоколов. На самом нижнем его уровне находится протокол IP, передающий данные в виде датаграмм. IP не поддерживает соединения; это значит, что для передачи каждой отдельной датаграммы может выбираться особый сетевой маршрут. Протокол IP является ненадежным: он не дает никаких гарантий относительно порядка, в котором будут получены датаграммы, отсутствия дубликатов и даже просто успешной передачи данных. Если вам требуется надежность, то она должны быть предоставлена протоколом более высокого уровня (например, TCP) или самим приложением.

Оригинальной версией протокола IP является IPv4. В начале 1990-х была разработана новая версия, IPv6. Наиболее заметная ее особенность заключается в использовании 128 бит для представления сетевых адресов (вместо 32, как IPv4). Это позволяет подключать к глобальной сети Интернет значительно большее количество компьютеров. На сегодняшний день IPv4 остается наиболее распространенной версией протокола IP, хотя ожидается, что в ближайшие годы будет вытеснена IPv6.

Поверх IP функционируют различные протоколы транспортного уровня, наиболее востребованными из которых являются UDP и TCP. Первый передает датаграммы и является ненадежным. Второй гарантирует надежность, поддерживает соединения и основывается на байтовых потоках. Протокол TCP берет на себя все нюансы, связанные с установкой и разрывом соединения. Прежде чем передавать данные по IP, он упаковывает их в сегменты, каждый из которых получает уникальный порядковый номер; это позволяет подтверждать их доставку и собирать в правильном порядке на стороне получателя. Кроме того, TCP управляет потоком данных и обеспечивает контроль над перегрузкой, не давая быстрому отправителю полностью загрузить медленного получателя или переполнить сеть.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 55.14.

55 Сокеты: домены сети Интернет

Познакомившись с основными принципами работы сокетов и протоколов семейства TCP/IP, можно готовы приступить к написанию сетевых программ на основе доменов IPv4 (`AF_INET`) и IPv6 (`AF_INET6`).

Как отмечалось в главе 54, идентификатор сокета в интернет-домене состоит из IP-адреса и номера порта. Компьютеры работают с данной информацией в двоичном виде, однако люди предпочитают иметь дело с именами, а не с числами. Поэтому в настоящей главе будут описаны методики, позволяющие идентифицировать сетевые узлы и их порты с помощью имен. Мы также рассмотрим библиотечные функции для получения IP-адреса(-ов) заданного узла и номер порта, связанный с определенной службой. В рамках этой темы познакомимся с системой доменных имен (англ. Domain Name System, DNS), которая представляет собой распределенную базу данных, связывающую сетевые имена компьютеров с их IP-адресами и наоборот.

55.1. Сокеты интернет-домена

Потоковые сокеты интернет-домена реализованы поверх протокола TCP. Они предоставляют надежный двунаправленный канал данных на основе байтового потока.

Датаграммные сокеты интернет-домена реализованы поверх протокола UDP. Они похожи на сокеты домена UNIX, но имеют следующие особенности:

- в отличие от датаграммных сокетов UNIX-домена, UDP-сокеты не являются надежными: пакеты могут теряться, дублироваться или приходить не в том порядке, в котором они были отправлены;
- отправка пакетов через датаграммный сокет UNIX-домена блокируется, если очередь данных принимающего сокета заполнена. Для сравнения: если UDP-датаграмма грозит переполнить очередь получателя, она просто автоматически отклоняется.

55.2. Порядок байтов в сети

IP-адреса и номера портов являются целыми числами. Одна из проблем, с которой можно столкнуться при передаче этих значений по сети, такова: различные аппаратные платформы хранят байты целого числа в разном порядке (при условии, что их больше одного). Как показано на рис. 55.1, существуют платформы, где целое значение начинается со старшего байта (то есть с наименьшего адреса); иногда их называют *big endian* («тупоконечными»). Платформы, в которых целые значения начинаются с младшего байта, называют *little endian*, или «остроконечными». (Эти названия заимствованы из сатирического романа Джонатана Свифта «Путешествия Гулливера» [1726 год], там они использовались для обозначения враждующих политических фракций, разбивающих вареные яйца с разных концов.) Наиболее заметным примером архитектуры little endian является x86 (можно

также вспомнить архитектуру VAX компании Digital, на компьютерах которой активно применялась система BSD). В большинстве других платформ байты хранятся в порядке от старшего к младшему. Есть также несколько архитектур, способных переключаться между этими форматами. Порядок следования байтов на конкретном компьютере называют *локальным*.

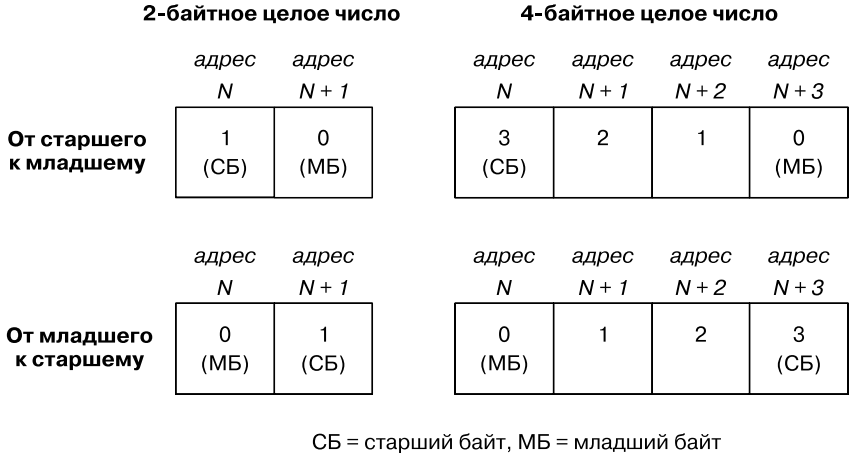


Рис. 55.1. Разный порядок следования байтов в 2- и 4-байтных целых числах

Поскольку номера портов и IP-адреса приходится передавать между разными узлами сети, каждый из которых должен понимать их значение, возникает необходимость в некоем стандартном порядке следования байтов. Этот порядок называется *сетевым*, и байты в нем размещаются от старшего к младшему.

Позже в данной главе мы рассмотрим различные функции для приведения имен сетевых узлов (например, www.kernel.org) и служб (например, `http`) к соответствующему цифровому представлению. Обычно они возвращают целые числа в сетевом порядке следования байтов, которые можно скопировать непосредственно в подходящие поля структуры для хранения адреса сокета.

Но иногда с целочисленными константами для IP-адресов и номеров портов приходится работать напрямую. Например, чтобы явно указать номер порта в коде нашей программы, получить его в качестве аргумента командной строки или воспользоваться константами `INADDR_ANY` и `INADDR_LOOPBACK` при определении IPv4-адреса. В языке C эти значения представлены в том формате, который используется на текущем компьютере. Прежде чем задействовать их в структуре для хранения адреса сокета, следует привести их к сетевому порядку следования байтов.

Для преобразования целых чисел между стандартным сетевым форматом и представлением, применяемым на текущем компьютере, предусмотрены функции `tons()`, `htonl()`, `ntohs()` и `ntohl()` (обычно они реализованы в виде макросов).

```
#include <arpa/inet.h>

uint16_t htons(uint16_t host_uint16);
```

Возвращает значение `host_uint16`, приведенное к сетевому порядку следования байтов

```
uint32_t htonl(uint32_t host_uint32);
```

Возвращает значение `host_uint32`, приведенное к сетевому порядку следования байтов

```
uint16_t ntohs(uint16_t net_uint16);
```

Возвращает значение `net_uint16`, приведенное к порядку следования байтов, которое применяется на текущем компьютере

```
uint32_t ntohl(uint32_t net_uint32);
```

Возвращает значение `net_uint32`, приведенное к порядку следования байтов, используемое на текущем компьютере

Когда-то эти функции имели прототипы следующего вида:

```
unsigned long htonl(unsigned long hostlong);
```

Это объясняет происхождение их имен: в данном случае `htonl` расшифровывается как *host to network long* (локальный формат в сетевом, тип `long`). В большинстве старых систем, где были реализованы сокеты, короткие целые числа занимали 16 бит, а длинные — 32 бита. Сейчас же все иначе (как минимум в случае с короткими целыми числами), поэтому прототипы, приведенные выше, более точно определяют типы данных, с которыми имеют дело указанные функции, хотя сами имена остались без изменений. Типы `uint16_t` и `uint32_t` предназначены для хранения 16- и 32-разрядных беззнаковых целых значений.

Строго говоря, реальная необходимость в этих четырех функциях возникает только в системах, где локальный порядок следования байтов отличается от сетевого. Тем не менее их следует использовать в любом случае, чтобы обеспечить портируемость программ между разными аппаратными архитектурами. В системах, где локальный порядок следования байтов совпадает с сетевым, приведенные функции просто возвращают передаваемые им аргументы.

55.3. Представление данных

При написании сетевых программ следует помнить: в каждой компьютерной архитектуре принято по-своему представлять некоторые типы данных. Мы уже отмечали, что целочисленные значения могут храниться с разным порядком следования байтов. Но существуют и другие особенности. Например, тип данных `long` в языке C может занимать как 32, так и 64 бита в зависимости от системы. В случае со структурами все только усложняется, ведь платформы задействуют различные правила выравнивания полей структуры по границам адреса, из-за чего величина сдвига между полями может варьироваться.

Из-за указанных отличий в представлении данных приложения, передающие информацию по сети между двумя гетерогенными (несовместимыми) системами, должны использовать некое общее соглашение относительно кодирования и декодирования этой информации. Процесс приведения данных к стандартному формату, подходящему для передачи по сети, называется *маршалингом* (англ. marshalling — «упорядочивание»). Существуют разные стандарты маршалинга, такие как XDR (External Data Representation — внешнее представление данных; см. RFC 1014), ASN.1-BER (Abstract Syntax Notation 1, см. http://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx), CORBA и XML. Обычно эти стандарты определяют строгий формат для каждого типа данных (описывая, например, порядок следования байтов и количество применяемых битов). Во время кодирования в подходящий формат данные маркируются с помощью одного или нескольких дополнительных полей, которые определяют их тип (и, возможно, длину).

Однако вместо маршалинга часто используется более простая методика: все передаваемые данные переводятся в текстовый вид, а отдельные элементы разделяются специальными символами (обычно символом новой строки). Одним из преимуществ такого подхода является то, что для отладки приложения можно задействовать программу `telnet`. Для этого достаточно следующей команды:

```
$ telnet host port
```

Затем можно вводить строчки текста, которые будут посланы приложению, и просматривать полученные ответы. Данная методика будет продемонстрирована в разделе 55.11.

Проблема, связанная с различиями в представлении информации в гетерогенных системах, касается не только сетевого взаимодействия, но и любых механизмов обмена данными между системами. Например, те же проблемы существуют при передаче файлов, хранящихся на диске или магнитной ленте. Просто сетевое программирование на сегодняшний день является наиболее распространенным контекстом, в котором можно столкнуться с подобной проблемой.

Для работы с данными, передаваемыми с помощью потокового сокета и закодированными в виде разбитого на отдельные строки текста, удобно использовать функцию `readLine()`, представленную в листинге 55.1.

```
#include "read_line.h"

ssize_t readLine(int fd, void *buffer, size_t n);

                        Возвращает либо количество байтов, скопированных в buffer
                        (не считая завершающего нулевого символа), либо 0,
                        если обнаружен конец файла, либо -1 при ошибке
```

Функция `readLine()` считывает байты из файла, указанного дескриптором `fd`, пока не обнаруживает символ новой строки. Входящая последовательность байтов сохраняется в участке памяти, на который ссылается аргумент `buffer`; размер этого участка должен быть не меньше `n` байт. Возвращаемая строка всегда содержит в конце нулевой символ; таким образом, объем полученных данных не превышает `(n - 1)` байт. В случае успеха функция `readLine()` возвращает количество байтов, помещенных в `buffer`, не считая конечного нулевого символа.

Листинг 55.1. Построчное чтение данных

sockets/read_line.c

```
#include <unistd.h>
#include <errno.h>
#include "read_line.h"              /* Объявление readLine() */

ssize_t
readLine(int fd, void *buffer, size_t n)
{
    ssize_t numRead;  /* Сколько байтов было прочитано */
    size_t totRead;   /* Общее количество прочитанных байтов на этот момент */
    char *buf;
    char ch;

    if (n <= 0 || buffer == NULL) {
        errno = EINVAL;
        return -1;
    }
```

```
    buf = buffer; /* Арифметика указателей не поддерживается для "void *" */

    totRead = 0;
    for (;;) {
        numRead = read(fd, &ch, 1);

        if (numRead == -1) {
            if (errno == EINTR)   /* Прерывание --> перезапускаем read() */
                continue;
            else
                return -1;          /* Какая-то другая ошибка */

        } else if (numRead == 0) {  /* Конец файла */
            if (totRead == 0)       /* Ничего не прочитано; возвращаем 0 */
                return 0;
            else
            /* Прочитано какое-то количество байтов; добавляем '\0' */
                break;

        } else {        /* На данном этапе 'numRead' должно быть равно 1 */
            if (totRead < n - 1) {  /* Отклоняем лишние байты: > (n - 1) */
                totRead++;
                *buf++ = ch;
            }

            if (ch == '\n')
                break;
        }
    }
    *buf = '\0';
    return totRead;
}
```

sockets/read_line.c

Если количество байтов, прочитанных до обнаружения новой строки, равно или превышает `(n - 1)`, то функция `readLine()` отклоняет лишние данные (включая сам символ новой строки). В случае размещения символа новой строки среди первых `(n - 1)` байт он попадает в итоговую строку (таким образом, для определения, является ли строка усеченной, достаточно проверить, присутствует ли символ новой строки перед нулевым символом). Мы используем данный подход, чтобы прикладные протоколы, принимающие ввод построчно, не рассматривали одну длинную строку как совокупность нескольких строк. Это могло бы нарушить протокол, поскольку приложения на обоих концах соединения потеряли бы синхронность. В качестве альтернативного подхода можно было бы заставить функцию `readLine()` считывать только те байты, которых достаточно для заполнения буфера, а оставшиеся данные вплоть до символа новой строки передать следующему вызову `readLine()`. В этом случае вызывающему процессу пришлось бы учитывать возможность частичного чтения строки.

55.4. Адреса интернет-сокетов

Существует два вида адресов для сокетов интернет-домена: IPv4 и IPv6.

Адреса сокетов в формате IPv4: struct sockaddr_in

IPv4-адрес сокета хранится в структуре `sockaddr_in`, определенной в заголовочном файле `<netinet/in.h>`:

```
struct in_addr {                /* 4-байтный адрес IPv4 */
    in_addr_t s_addr;           /* Беззнаковое целое число */
};

struct sockaddr_in {            /* IPv4-адрес сокета */
    sa_family_t sin_family;     /* Семейство адресов (AF_INET) */
    in_port_t sin_port;         /* Номер порта */
    struct in_addr sin_addr;    /* IPv4-адрес */
    unsigned char __pad[X];     /* Сведения о размере структуры 'sockaddr' (16 байт) */
};
```

В разделе 52.4 мы увидели, что обобщенная структура `sockaddr` начинается с поля, определяющего домен сокета. В структуре `sockaddr_in` это поле называется `sin_family` и всегда хранит значение `AF_INET`. Поля `sin_port` и `sin_addr` обозначают номер порта и IP-адрес и соблюдают сетевой порядок следования байтов. Типы данных `in_port_t` и `in_addr_t` представляют собой целые беззнаковые числа длиной 16 и соответственно 32 бита.

Адреса сокетов в формате IPv6: struct sockaddr_in6

В протоколе IPv6, как и в IPv4, адрес сокета состоит из IP-адреса и номера порта. Разница заключается в том, что в IPv6 адрес занимает 126 бит вместо 32. IPv6-адрес сокета хранится в структуре `sockaddr_in6`, определенной в заголовочном файле `<netinet/in.h>`:

```
struct in6_addr {               /* Структура с IPv6-адресом */
    uint8_t s6_addr[16];        /* 16 байт == 128 бит */
};

struct sockaddr_in6 {           /* IPv6-адрес сокета */
    sa_family_t sin6_family;    /* Семейство адресов (AF_INET6) */
    in_port_t sin6_port;        /* Номер порта */
    uint32_t sin6_flowinfo;     /* Сведения о потоке данных в IPv6 */
    struct in6_addr sin6_addr;  /* IPv6-адрес */
    uint32_t sin6_scope_id;     /* Идентификатор диапазона (с Linux 2.4) */
};
```

Поле `sin_family` хранит значение `AF_INET6`. Поля `sin6_port` и `sin6_addr` представляют собой номер порта и IP-адрес (тип данных `uint8_t`, который используется в структуре `in6_addr`, является 8-разрядным целым беззнаковым числом). Остальные поля, `sin6_flowinfo` и `sin6_scope_id`, выходят за рамки этой книги; в нашем случае они всегда равны 0. Все поля в структуре `sockaddr_in6` соблюдают сетевой порядок следования байтов.

> IPv6-адреса описаны в документе RFC 4291. Подробности об управлении потоком данных в IPv6 (sin6_flowinfo) можно найти в приложении A книги [Stevens et al., 2004] и в документах RFC 2460 и 3697. RFC 3493 и 4007 содержат информацию о поле sin6_scope_id.

Протокол IPv6, как и IPv4, поддерживает универсальный адрес и замыкание на себя. Однако использование данных возможностей усложнено тем фактом, что IPv6-адрес хранится в массиве (а не с помощью скалярного типа). Для демонстрации этого воспользуемся универсальным адресом (`0::0`). Константа `IN6ADDR_ANY_INIT`, которая ему соответствует, имеет следующее определение:

```
#define IN6ADDR_ANY_INIT { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 } }
```

> В Linux некоторые аспекты заголовочного файла отличаются от описания, приведенного в книге. В частности, структура in6_addr содержит объединение, представляющее 128-разрядный IPv6-адрес в виде 16 байт, восьми 2-байтных или четырех 32-байтных чисел. Из-за наличия этого поля определение константы IN6ADDR_ANY_INIT в библиотеке glibc включает в себя еще один дополнительный набор вложенных фигурных скобок, не показанный в нашем описании.

Константу IN6ADDR_ANY_INIT можно использовать в инициализаторе при объявлении переменной, однако нельзя указывать в правой части операции присваивания, поскольку синтаксис языка C не разрешает применять в таких операциях константы-структуры. Вместо этого необходимо задействовать заранее определенную переменную in6addr_any, которая в языке C инициализируется следующим образом:

```
const struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
```

Таким образом, чтобы присвоить данной структуре универсальный адрес, нужно выполнить следующий код:

```
struct sockaddr_in6 addr;

memset(&addr, 0, sizeof(struct sockaddr_in6));
addr.sin6_family = AF_INET6;
addr.sin6_addr = in6addr_any;
addr.sin6_port = htons(SOME_PORT_NUM);
```

Для обозначения адреса, замкнутого на себя (::1), в IPv6 предусмотрены константа IN6ADDR_LOOPBACK_INIT и переменная in6addr_loopback.

В отличие от IPv4, протокол IPv6 обеспечивает сетевой порядок следования байтов в своих константах и инициализаторах переменных. Но, как видно в вышеприведенном коде, все равно необходимо проверять порядок следования байтов в номере порта.

При одновременном использовании протоколов IPv4 и IPv6 на одном и том же компьютере они имеют общее пространство номеров портов. Это значит, что если, к примеру, приложение привязывает IPv6-сокет к TCP-порту под номером 2000 (с помощью универсального адреса), то IPv4-сокет с протоколом TCP не может подключиться к тому же порту. (Реализация TCP/IP следит за тем, чтобы сокеты на других сетевых узлах могли подключаться к нашему сокету вне зависимости от того, в рамках какого протокола они действуют — IPv4 или IPv6.)

Структура sockaddr_storage

Вместе с выходом IPv6 в программном интерфейсе сокетов появилась новая обобщенная структура под названием sockaddr_storage. Она имеет размер, достаточный для хранения адреса любого вида сокетов (то есть к этому типу можно привести любую структуру для хранения адресов сокетов). В частности, данная структура позволяет прозрачно хранить адреса сокетов в форматах IPv4 и IPv6, что позволяет избавиться от проверки версии протокола IP в нашем коде. В Linux структура sockaddr_storage имеет следующее определение:

```
#define __ss_aligntype uint32_t      /* В 32-разрядных архитектурах */
struct sockaddr_storage {
    sa_family_t ss_family;
    __ss_aligntype __ss_align;       /* Автоматическое выравнивание */
    char __ss_padding[SS_PADSIZE];   /* Программные данные размером 128 байт */
};
```

55.5. Краткий обзор функций для преобразования сетевых адресов и имен служб

Как мы уже говорили, компьютеры представляют IP-адреса и номера портов в двоичном виде, однако человеку легче запоминать имена, а не числа. К тому же символьное имя позволяет обеспечить косвенное обращение к сетевому узлу: изменение исходного числового значения не помешает использовать то же самое имя.

Имя сетевого узла — символьный идентификатор системы, которая подключена к сети (и, возможно, имеет несколько IP-адресов). *Имя службы* — символьное представление номера порта.

Адрес узла может быть представлен в виде двоичного значения, символьного имени или IP-формата (в IPv4 это десятичные числа, разделенные точками, а в IPv6 — шестнадцатеричная строка).

Номер порта можно представить в двоичном виде или с помощью символьного имени службы.

Для преобразования этих форматов существуют различные библиотечные функции. В данном разделе приводится их краткий обзор, а в следующем вы детально познакомитесь с современными программными интерфейсами (`inet_ntop()`, `inet_pton()`, `getaddrinfo()`, `getnameinfo()` и т. д.). В разделе 55.13 мы затронем устаревшие интерфейсы (`inet_aton()`, `inet_ntoa()`, `gethostbyname()`, `getservbyname()` и т. д.).

Преобразование IPv4-адресов между двоичным и символьным форматами

Функции `inet_aton()` и `inet_ntoa()` приводят IPv4-адрес, записанный в виде разделенных точками десятичных чисел, в двоичный формат (и наоборот). Мы упоминаем их потому, что они встречаются в старом коде. На сегодняшний день они считаются устаревшими. Функции для выполнения аналогичных преобразований, которые следует использовать в современных программах, описаны ниже.

Преобразование IPv4- и IPv6-адресов между двоичным и символьным форматами

Функции `inet_pton()` и `inet_ntop()` аналогичны функциям `inet_aton()` и `inet_ntoa()`, но поддерживают работу с IPv6-адресами. Они позволяют переводить IPv4- и IPv6-адреса в *презентационный* формат (десятичные числа, разделенные точками, или шестнадцатеричную строку) и из него.

Людям привычнее иметь дело с именами, чем с числами, в связи с чем данные функции используются в программах лишь изредка. Одно из применений `inet_ntop()` состоит в создании версии IP-адреса, которую можно записать в журнал. Иногда это является более подходящим вариантом, чем конвертация IP-адреса в имя узла, поскольку:

- преобразование IP-адреса в имя сетевого узла требует обращения к DNS-серверу, на что может уйти некоторое время;
- в ряде случаев в DNS-сервере может отсутствовать PTR-запись, связывающая IP-адрес с соответствующим именем узла.

Данные функции обладают куда более простым программным интерфейсом по сравнению с `getaddrinfo()` и `getnameinfo()` (выполняющими преобразование между двоичным представлением и соответствующими символьными именами), поэтому мы опишем их в первую очередь (в разделе 55.6). Это позволит сразу же продемонстрировать некоторые рабочие примеры использования сокетов в интернет-домене.

Преобразование имен узлов и служб в двоичный вид и обратно

Функция `getaddrinfo()` является современным аналогом устаревших функций `gethostbyname()` и `getservbyname()`. Если передать ей имена узла и службы, то она вернет набор структур, содержащих соответствующие IP-адреса (их может быть несколько) и номер порта в двоичном виде. В отличие от `gethostbyname()`, функция `getaddrinfo()` умеет прозрачно работать с адресами в форматах IPv4 и IPv6, поэтому ее можно использовать для написания программ, которые не зависят от версии протокола IP. Данную функцию следует применять во всех новых программах для преобразования имен узлов и служб в двоичное представление.

Функция getnameinfo() выполняет обратное преобразование, превращая IP-адрес и номер порта в соответствующие имена узла и службы.

С помощью функций getaddrinfo() и getnameinfo() можно также переводить IP-адрес в презентационный формат и обратно.

Прежде чем приступать к подробному обсуждению этих функций в разделе 55.10, сначала необходимо рассмотреть службу DNS (см. раздел 55.8) и файл /etc/services (см. раздел 55.9). DNS позволяет совместно работающим серверам обслуживать распределенные базы данных, которые привязывают IP-адреса к именам узлов и наоборот. Существование таких служб, как DNS, необходимо для работы Интернета, поскольку централизованное управление всеми существующими именами узлов было бы попросту невозможным. Файл /etc/services привязывает номера портов к символьным именам служб.

55.6. Функции inet_pton() и inet_ntop()

Функции inet_pton() и inet_ntop() позволяют преобразовывать IPv4- и IPv6-адреса между двоичным и презентационным (десятичные числа, разделенные точками, или шестнадцатеричная строка) форматами. Буква p в именах этих функций обозначает *presentation* («презентационный»), а n — *network* («сеть»).

```
#include <arpa/inet.h>

int inet_pton(int domain, const char *src_str, void *addrptr);
```

Возвращает 1 при успешном завершении, 0, если строка src_str не соответствует презентационному формату, или -1 при ошибке

```
const char *inet_ntop(int domain, const void *addrptr, char *dst_str,
                      size_t len);
```

Возвращает указатель на dst_str или NULL при ошибке

Презентационный вид представляет собой строку, понятную для человека. Например:

- 204.152.189.116 (IPv4-адрес в виде десятичных чисел, разделенных точками);
- ::1 (IPv6-адрес в виде шестнадцатеричных значений, разделенных двоеточиями);
- ::FFFF:204.152.189.116 (IPv4-адрес, совместимый с IPv6).

Функция inet_pton() переводит презентационную строку, переданную в аргументе src_str, в двоичный IP-адрес с сетевым порядком следования байтов. Аргумент domain должен быть равен AF_INET или AF_INET6. Полученный адрес помещается в структуру, на которую указывает аргумент addrptr; тип структуры должен быть либо in_addr, либо in6_addr, в зависимости от значения domain.

Функция inet_ntop() выполняет обратное преобразование. Повторим: аргумент domain должен быть равен AF_INET или AF_INET6, а указатель addrptr должен ссылаться на структуру типа in_addr или in6_addr, которую мы хотим преобразовать. Итоговая строка с нулевым символом в конце помещается в буфер, на который указывает dst_str. Аргумент *len* должен содержать размер этого буфера. В случае успеха функция inet_ntop() возвращает dst_str. Если размер len слишком маленький, то inet_ntop() возвращает NULL, а глобальной переменной errno присваивается ENOSPC.

Чтобы подобрать подходящий размер буфера, на который указывает аргумент dst_str, можно использовать две константы, определенные в заголовочном файле <netinet/in.h>.

Речь идет о максимальной длине (включая завершающий нулевой байт) презентационной строки для IPv4- и IPv6-адресов:

```
#define INET_ADDRSTRLEN 16   /* Максимальный размер строки с десятичными числами,
                                разделенными точками (IPv4) */
#define INET6_ADDRSTRLEN 46  /* Максимальный размер шестнадцатеричной строки (IPv6) */
```

Примеры использования `inet_pton()` и `inet_ntop()` приведены в следующем разделе.

55.7. Пример клиент-серверного приложения (на основе датаграммных сокетов)

В этом разделе мы возьмем клиентскую и серверную программы для изменения регистра, показанные в разделе 53.3, и перепишем их с учетом использования датаграммных сокетов в домене `AF_INET6`. Исходный код будет представлен с минимальными комментариями, поскольку по своей структуре он похож на ранее рассмотренные программы. Главное отличие новой версии заключается в определении и инициализации структуры с адресом сокета в формате IPv6, которую мы описали в разделе 55.4.

И клиент, и сервер задействуют заголовочный файл, показанный в листинге 55.2. В этом файле определяется номер порта сервера и максимальный размер сообщений, которыми клиент и сервер могут обмениваться между собой.

Листинг 55.2. Заголовочный файл, применяемый программами i6d_ucase_sv.c и i6d_ucase_cl.c

sockets/i6d_ucase.h

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10        /* Максимальный размер сообщений,
                              которыми обмениваются клиент и сервер */
#define PORT_NUM 50002     /* Номер серверного порта */
```

sockets/i6d_ucase.h

В листинге 55.3 показана серверная программа. Сервер использует функцию `inet_ntop()` для приведения адреса клиента (полученного с помощью вызова `recvfrom()`) в презентационный вид.

Клиентская программа, представленная в листинге 55.4, содержит два заметных нововведения по сравнению со старой версией, рассчитанной на домен UNIX (см. листинг 53.7). Во-первых, клиент интерпретирует аргумент командной строки как адрес сервера в формате IPv6 (остальные аргументы воспринимаются как отдельные датаграммы, которые нужно послать серверу). С помощью функции `inet_pton()` данный адрес приводится в двоичный вид. Во-вторых, клиент не привязывает свой сокет ни к какому адресу. Как отмечалось в подразделе 54.6.1, если у сокета в интернет-домене нет адреса, то ядро автоматически привязывает его к динамическому порту в текущей системе. Это можно наблюдать в следующей сессии командной строки, где сервер и клиент запускаются на одном и том же компьютере:

```
$ ./i6d_ucase_sv &
[1] 31047
$ ./i6d_ucase_cl ::1 ciao        Отправляем серверу в локальной системе
Server received 4 bytes from (::1, 32770)
Response 1: CIAO
```

Этот вывод говорит о том, что вызов `recvfrom()` сумел получить адрес клиентского сокета, в том числе номер его динамического порта, хотя клиент и не выполнял операцию `bind()`.

Листинг 55.3. Сервер для изменения регистра на основе датаграммных сокетов и IPv6

sockets/i6d_ucase_sv.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];
    char claddrStr[INET6_ADDRSTRLEN];

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_addr = in6addr_any;          /* Универсальный адрес */
    svaddr.sin6_port = htons(PORT_NUM);

    if (bind(sfd, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_in6)) == -1)
        errExit("bind");

    /* Получаем сообщения, переводим их в верхний регистр и возвращаем клиенту */

    for (;;) {
        len = sizeof(struct sockaddr_in6);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                        (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        if (inet_ntop(AF_INET6, &claddr.sin6_addr, claddrStr,
                    INET6_ADDRSTRLEN) == NULL)
            printf("Couldn't convert client address to string\n");
        else
            printf("Server received %ld bytes from (%s, %u)\n",
                   (long) numBytes, claddrStr, ntohs(claddr.sin6_port));

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *)
                  &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/i6d_ucase_sv.c

Листинг 55.4. Клиент для изменения регистра на основе датаграммных сокетов и IPv6

sockets/i6d_ucase_cl.c

```
#include "i6d_ucase.h"
int

main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host-address msg...\n", argv[0]);

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);  /* Создаем клиентский сокет */
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_port = htons(PORT_NUM);
    if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0)
        fatal("inet_pton failed for address '%s'", argv[1]);

    /* Отправляем сообщения серверу; направляем ответы в стандартный вывод */
    for (j = 2; j < argc; j++) {
        msgLen = strlen(argv[j]);
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                   sizeof(struct sockaddr_in6)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");
        printf("Response %d: %.*s\n", j - 1, (int) numBytes, resp);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/i6d_ucase_cl.c

55.8. Система доменных имен (DNS)

В разделе 55.10 мы подробно рассмотрим функции `getaddrinfo()` и `getnameinfo()`, которые позволяют получать IP-адрес из имени сетевого узла и наоборот. Но перед этим нужно объяснить, как служба DNS хранит связи между именами узлов и IP-адресами.

До появления DNS имена сетевых узлов и IP-адреса связывались с помощью локального файла `/etc/hosts`, содержащего записи следующего вида:

```
# IP-address       canonical hostname        [aliases]
127.0.0.1          localhost
```

Функция `gethostbyname()` (предшественница `getaddrinfo()`) получала IP-адрес на основе этого файла, находя совпадение для либо канонического имени узла (то есть

официального и первичного имени компьютера), либо одного из псевдонимов (которые опционально могут быть перечислены через пробел).

Однако такой подход плохо масштабируется и является непригодным, если в сети большое количество узлов (как в Интернете с его миллиардами компьютеров).

Для решения данной проблемы была разработана служба DNS. Ее ключевые идеи заключаются в следующем.

- Имена сетевых узлов организованы иерархически (рис. 55.2). Каждый *узел* в этой иерархии имеет метку (имя), длина которой может достигать 63 символов. На вершине иерархии находится безымянный узел — «анонимный корневой сервер».
- *Доменное имя* узла состоит из совокупности всех имен на пути от данного узла к корневому серверу, разделенных точками. Например, `google.com` — доменное имя узла `google`.
- *Полностью определенное имя домена* (англ. fully qualified domain name, FQDN), такое как `www.kernel.org.`, задает место узла в иерархии. Подобные имена заканчиваются точкой, хотя во многих случаях она опускается.
- Ни одна организация или система не управляет всей иерархией целиком. Вместо этого существует иерархия DNS-серверов, каждый из которых отвечает за какую-то ее часть (*зону*). Обычно каждая зона имеет *первичный DNS-сервер* и один или несколько *вторичных*. Последние нужны на случай, если первичный выйдет из строя. Зоны, в свою очередь, тоже могут быть поделены на более мелкие участки. Когда в зоне появляется новый узел или изменяется привязка имени узла к адресу, администратор должен обновить базу данных на соответствующем локальном DNS-сервере (остальные серверы в иерархии не нуждаются в ручном обновлении баз данных).

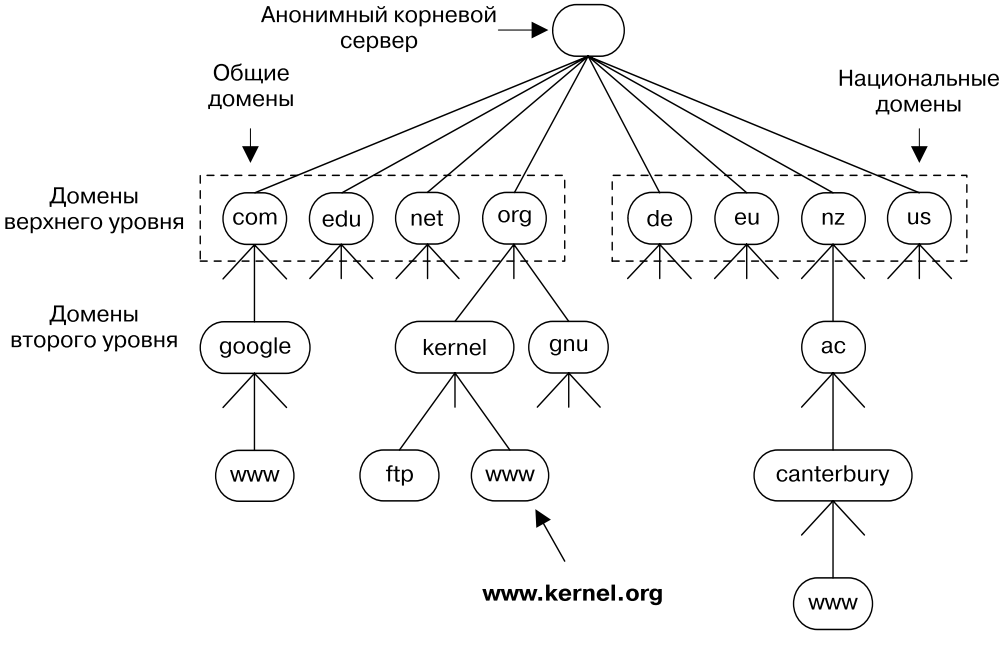


Рис. 55.2. Часть иерархии DNS

Реализация DNS-сервера named(8), которая используется в Linux, основана на проекте BIND (Berkeley Internet Name Domain) и поддерживается организацией Internet Systems Consortium

(http://www.isc.org/). Управление этим демоном осуществляется из файла /etc/named.conf (см. страницу named.conf(5) руководства). Основным источником информации о DNS и BIND можно считать книгу [Albitz & Liu, 2006]. Сведения о DNS также можно найти в главе 14 [Stevens, 1994], главе 11 [Stevens et al., 2004] и главе 24 [Comer, 2000].

- Функция `getaddrinfo()`, которую вызывают для получения IP-адреса для доменного имени, использует набор библиотечных функций, взаимодействующих с локальным DNS-сервером. Если этот сервер не может предоставить необходимую информацию, то `getaddrinfo()` обращается к другим DNS-серверам в рамках той же иерархии. Иногда данный процесс может длиться довольного долго, поэтому DNS-серверы применяют кэширование, чтобы избежать лишнего сетевого взаимодействия в случае с часто запрашиваемыми доменными именами.

Применение подхода, описанного выше, позволяет службе DNS оперировать огромными пространствами имен, для управления которыми не требуется централизованный механизм.

Рекурсивные и итеративные запросы

DNS-запросы делятся на две категории: *рекурсивные* и *итеративные*. При рекурсивном запросе сервер должен выполнить всю работу целиком, включая взаимодействие с другими DNS-серверами, если того требует ситуация. Функция `getaddrinfo()`, которая вызывается приложением в локальной системе, выполняет рекурсивный запрос к локальному DNS-серверу, а тот уже совершает процедуру разрешения доменного имени итеративно, если у него нет информации, необходимой для получения IP-адреса.

Для того чтобы объяснить принцип работы итеративного запроса, рассмотрим пример. Допустим, локальному DNS-серверу поступил запрос на разрешение имени `www.otago.ac.nz`. Тот первым делом свяжется с одним из немногочисленных корневых серверов, о которых (как и любой другой DNS-сервер) обязан знать. (Список этих серверов можно получить с помощью команды `dig . NS` или на веб-странице www.root-servers.org.) В случае с именем `www.otago.ac.nz` корневой сервер обратится к одному из локальных DNS-серверов в зоне `nz`, а тот уже попытается найти сервер с именем `www.otago.ac.nz`. Он будет направлен к серверу `ac.nz`. Затем локальный сервер выполнит запрос к `ac.nz` и будет направлен к серверу `otago.ac.nz`, который, наконец, вернет IP-адрес, связанный с именем `www.otago.ac.nz`.

При передаче функции `gethostbyname()` неполного доменного имени она сначала попытается его дополнить. Правила дополнения определяются в файле `/etc/resolv.conf` (см. страницу `resolv.conf(5)` руководства). По умолчанию `gethostbyname()` попытается как минимум использовать доменное имя локальной системы. Например, если ввести команду `ssh octavo` в системе с именем `oghma.otago.ac.nz`, итоговый DNS-запрос будет выполнен для имени `octavo.otago.ac.nz`.

Домены верхнего уровня

Сетевые узлы на уровень ниже корневого представляют собой так называемые *домены верхнего уровня* (ДВУ), за которыми следуют *домены второго уровня* и т. д. ДВУ делятся на две категории: *общие* и *национальные*.

Когда-то существовало всего семь общих доменов верхнего уровня, большинство из которых считались международными. На рис. 55.2 показаны четыре из них. Оставшиеся три — `int`, `mil` и `gov`; два последних зарезервированы для правительства США. Сравнительно недавно количество ДВУ расширилось (появились такие домены, как `info`, `name` и `museum`).

Каждая страна имеет национальный домен верхнего уровня, соответствующий стандарту ISO 3166-1 и состоящий из двух символов. Несколько из них показано на рис. 55.2: `de` (Германия, *Deutschland*), `eu` (наднациональный географический ДВУ Европейского Союза, *European*

Union), nz (Новая Зеландия, *New Zealand*) и us (Соединенные Штаты Америки, *United States of America*). Некоторые страны разделили свои ДВУ на подмножества доменов второго уровня по принципу общих доменов. Например, Новая Зеландия имеет домены ac.nz (учебные заведения), co.nz (коммерческие компании) и govt.nz (правительственные учреждения).

55.9. Файл /etc/services

Как отмечалось в подразделе 54.6.1, учетом общеизвестных номеров портов занимается организация IANA. У каждого порта есть соответствующее *имя службы*. Поскольку номера портов управляются централизованно и не подвержены таким частым изменениям, как IP-адреса, для хранения информации о них не требуется аналог DNS-сервера. Вместо этого номера портов и имена служб записываются в файл /etc/services. Для преобразования между ними используются функции getaddrinfo() и getnameinfo().

Каждая строка в файле /etc/services состоит из трех полей:

```
# Service name   port/protocol     [aliases]
echo             7/tcp             Echo      # echo service
echo             7/udp             Echo
ssh              22/tcp                      # Secure Shell
ssh              22/udp
telnet           23/tcp                      # Telnet
telnet           23/udp
smtp             25/tcp                      # Simple Mail Transfer Protocol
smtp             25/udp
domain           53/tcp                      # Domain Name Server
domain           53/udp
http             80/tcp                      # Hypertext Transfer Protocol
http             80/udp
ntp              123/tcp                     # Network Time Protocol
ntp              123/udp
login            513/tcp                     # rlogin(1)
who              513/udp                     # rwho(1)
shell            514/tcp                     # rsh(1)
syslog           514/udp                     # syslog
```

Поле protocol обычно содержит либо tcp, либо udp. У службы также могут быть псевдонимы (aliases), разделяемые пробелами. Помимо вышеперечисленного строка может содержать комментарий, начинающийся с символа #.

Как уже отмечалось ранее, в протоколах UDP и TCP номер порта обозначает разные сущности, но, согласно политике IANA, в обоих случаях номер порта привязывается к одной и той же службе, даже если она использует только один из протоколов. Например, службы telnet, ssh, HTTP и SMTP работают поверх TCP, но им также выделен и соответствующий UDP-порт. Аналогично служба NTP задействует только UDP, но в ее распоряжении также находится TCP-порт 123. В отдельных случаях служба может работать с обоими протоколами, например DNS и echo. Наконец, существуют редкие примеры того, как один порт в UDP и TCP привязан к разным службам. Среди них можно выделить порт 514, который в протоколе TCP отводится для службы rsh, а в TCP — для демона syslog (см. раздел 37.5). Дело в том, что указанные номера портов были распределены еще до того, как вступили в силу современные правила IANA.

Файл /etc/services содержит записи, связывающие номера с именами. Это не механизм резервирования: наличие в данном файле номера порта не гарантирует, что его можно будет привязать к конкретной службе.

55.10. Преобразование имен узлов и служб, не зависящее от протокола

Функция `getaddrinfo()` переводит имена узлов и служб в IP-адреса и номера портов. Она была добавлена в стандарт POSIX.1g в качестве (реентерабельной) замены устаревшим функциям `gethostbyname()` и `getservbyname()` (использование `getaddrinfo()` вместо `gethostbyname()` позволяет не беспокоиться о том, с какой версией протокола IP работает наша программа).

Функция `getnameinfo()` выполняет обратную операцию. Она переводит структуру с адресом сокета (IPv4 или IPv6) в строку, содержащую имена соответствующих узла и службы. Эта функция является (реентерабельным) аналогом устаревших функций `gethostbyaddr()` и `getservbyport()`.

> Подробное описание функций getaddrinfo() и getnameinfo(), а также детали их реализации представлены в главе 11 книги [Stevens et al., 2004]. Кроме того, они описаны в документе RFC 3493.

55.10.1. Функция getaddrinfo()

Если передать функции `getaddrinfo()` имена узла и службы, то она вернет список структур с адресами сокетов, каждая из которых содержит IP-адрес и номер порта.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Возвращает `0` при успешном завершении
или ненулевое значение при ошибке

В качестве ввода функция `getaddrinfo()` принимает аргументы `host`, `service` и `hints`. Аргумент `host` содержит либо имя сетевого узла, либо строку в формате IPv4 или IPv6, состоящую из десятичных чисел и точек или шестнадцатеричного значения (если быть точным, функция `getaddrinfo()` поддерживает более общий формат IPv4-адреса, представленный в подразделе 55.13.1). Аргумент `service` включает в себя либо имя службы, либо десятичный номер порта. Аргумент `hints` (подробно рассмотрен чуть ниже) указывает на структуру `addrinfo`, в которой описываются дальнейшие критерии для выбора структур с адресами сокетов, возвращаемых с помощью `result`.

Для возвращения результата функция `getaddrinfo()` динамически выделяет список связей со структурами `addrinfo` и делает так, чтобы аргумент `result` на него указывал. Каждая из этих структур содержит указатель на структуру с адресом сокета соответствующих *узла* и *службы* (см. рис. 55.3). Структура `addrinfo` имеет следующий вид:

```
struct addrinfo {
    int ai_flags;              /* Флаги ввода (константы вида AI_*) */
    int ai_family;             /* Семейство адресов */
    int ai_socktype;           /* Тип: SOCK_STREAM или SOCK_DGRAM */
    int ai_protocol;           /* Протокол сокета */
    size_t ai_addrlen;         /* Размер структуры, на которую указывает ai_addr */
    char *ai_canonname;        /* Каноническое имя узла */
    struct sockaddr *ai_addr;  /* Указатель на структуру с адресом сокета */
```

```
    struct addrinfo *ai_next;  /* Следующая структура в связном списке */
};
```

Аргумент `result` возвращает список структур, так как критерию, заданному с помощью аргументов `host`, `service` и `hints`, может соответствовать сразу несколько комбинаций из имен узлов и служб. Например, это относится к компьютерам, у которых больше одного сетевого интерфейса. Кроме того, если поле `hints.ai_socktype` равно 0, то можно получить две структуры для сокетов: одну для `SOCK_DGRAM`, а другую — для `SOCK_STREAM` (при условии, что служба доступна как по UDP, так и по TCP).

Поля структур `addrinfo`, которые возвращаются через аргумент `result`, описывают свойства соответствующих структур с адресами сокетов. Поле `ai_family` информирует о типе адреса и может быть равно либо `AF_INET`, либо `AF_INET6`. Поле `ai_socktype` обозначает протокол службы (TCP или UDP) и может принимать значения `SOCK_STREAM` и `SOCK_DGRAM`. Поле `ai_protocol` возвращает значение протокола для соответствующего семейства адресов и типа сокета (поля `ai_family`, `ai_socktype` и `ai_protocol` хранят значения, передающиеся в качестве аргументов вызова `socket()` при создании сокета для заданного адреса). Поле `ai_addrlen` обозначает размер структуры с адресом сокета, на которую указывает `ai_addr` (в байтах). Поле `in_addr` отмечает структуру с адресом сокета (`in_addr` для IPv4 или `in6_addr` для IPv6). Поле `ai_flags` игнорируется (оно используется для аргумента `hints`). Поле `ai_canonname` задействуется только в первой структуре `addrinfo` и только если в поле `hints.ai_flags` указан флаг `AI_CANONNAME` (см. ниже).

По аналогии с `gethostbyname()`, для возвращения результата функции `getaddrinfo()` иногда приходится обращаться к DNS-серверу, что может занять некоторое время. То же самое касается функции `getnameinfo()`, описанной в подразделе 55.10.4.

Применение функции `getaddrinfo()` будет продемонстрировано в разделе 55.11.

Аргумент hints

Аргумент `hints` задает подробные критерии для выбора структур с адресами сокетов, которые возвращает функция `getaddrinfo()`. При его использовании в структуре `addrinfo` можно задать только поля `ai_flags`, `ai_family`, `ai_socktype` и `ai_protocol`. Остальные поля игнорируются и при необходимости должны быть инициализированы нулями или значением `NULL`.

Поле `hints.ai_family` определяет домен для итоговых структур с адресами сокетов. Оно может быть равно `AF_INET` или `AF_INET6` (или какой-то другой константе вида `AF_*`, если это поддерживается на уровне реализации). Для получения структуры всех типов можно указать в данном поле значение `AF_UNSPEC`.

Поле `hints.ai_socktype` определяет тип сокета, для которого будет использоваться возвращаемая структура с адресом. При указании в нем значения `SOCK_DGRAM` запрос будет выполнен для UDP-службы, а соответствующая структура возвращена с помощью аргумента `result`. В случае указания `SOCK_STREAM` поиск будет выполняться среди TCP-служб. Если поле `hints.ai_socktype` равно 0, то подходящим будет считаться сокет любого типа.

Поле `hints.ai_protocol` определяет протокол сокета для итоговой структуры с адресом. В наших примерах оно всегда равно `0`, поскольку нам подходит любой протокол.

Поле `hints.ai_flags` представляет собой битовую маску, которая влияет на поведение функции `getaddrinfo()`. Маска формируется путем применения побитового ИЛИ к следующим значениям.

- `AI_ADDRCONFIG` — возвращает IPv4-адреса, только если в локальной системе имеется хотя бы один IPv4-адрес (не считая адреса, замкнутого на себя). То же самое касается адресов формата IPv6.
- `AI_ALL` — см. ниже описание константы `AI_V4MAPPED`.
- `AI_CANONNAME` — если аргумент `host` не равен `NULL`, то возвращает указатель на строку с каноническим именем узла с нулевым символом в конце. Указатель возвращается

в буфере, на который ссылается поле `ai_canonname` структуры `addrinfo`, полученной в аргументе `result`.

- `AI_NUMERICHOST` — делает так, что аргумент `host` интерпретируется как адрес, имеющий числовое представление. Это позволяет предотвратить поиск IP-адреса в ситуациях, когда он не требуется, поскольку данная операция может занимать существенное время.
- `AI_NUMERICSERV` — интерпретирует аргумент `service` как номер порта. Данный флаг предотвращает обращение к любой службе разрешения имен, потому что при установке аргументу `service` числового значения данный этап становится лишним.
- `AI_PASSIVE` — возвращает структуры с адресами сокетов, подходящие для пассивного открытия (то есть для слушающего сокета). В этом случае аргумент `host` должен быть равен `NULL`, а IP-адрес, хранящийся в итоговой структуре в аргументе `result`, будет универсальным (`INADDR_ANY` или `IN6ADDR_ANY_INIT`). Если данный флаг не установлен, то итоговые структуры с адресами подойдут для использования в вызовах `connect()` и `sendto()`. Если аргумент `host` равен `NULL`, то полученный адрес сокета будет замкнутым на себя (`INADDR_LOOPBACK` или `IN6ADDR_LOOPBACK_INIT` в зависимости от домена).
- `AI_V4MAPPED` — если в поле `ai_family` структуры `hint` было указано значение `AF_INET6` и если не было найдено ни одного IPv6-адреса, то в качестве результата должны вернуться структуры с IPv4-адресами, совместимыми с IPv6. При указанном сочетании флагов `AI_ALL` и `AI_V4MAPPED` в аргумент `result` попадут структуры с IPv6- и IPv4-адресами (последние будут совместимы с IPv6).

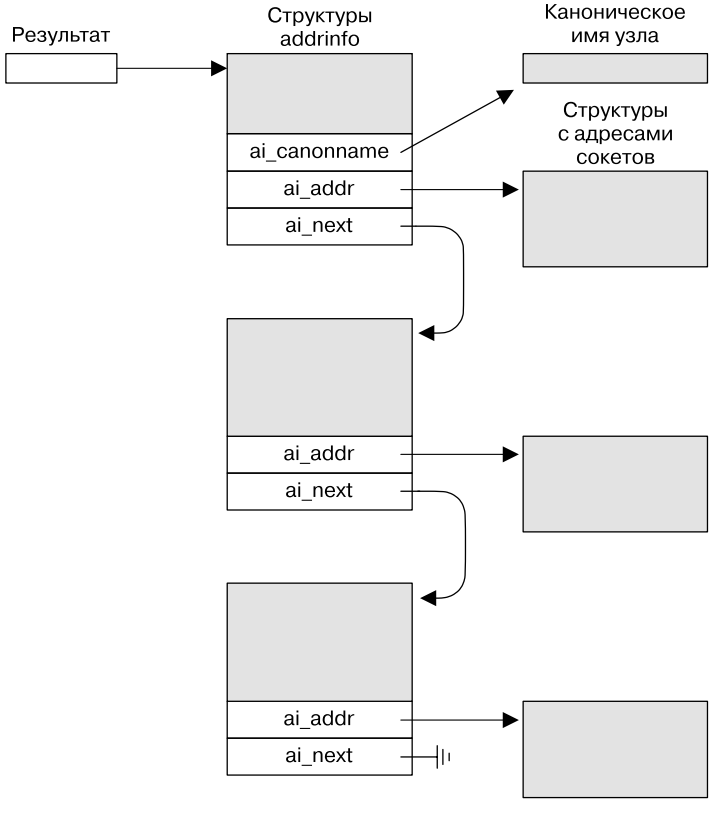


Рис. 55.3. Структуры, выделяемые и возвращаемые функцией getaddrinfo()

Как уже отмечалось выше, при использовании флага AI_PASSIVE аргумент hint может быть равен NULL. Данное значение можно передать и аргументу service — в этом случае номер порта, возвращаемый в структурах с адресами, будет равен 0 (то есть нас интересуют только адреса). Однако аргументы host и service не могут быть равны NULL одновременно.

Если нам не нужно задавать какие-либо из вышеописанных критериев, то аргументу hints можно передать NULL; в этом случае подразумевается, что его поля ai_socktype и ai_protocol равны 0, ai_flags равен (AI_V4MAPPED | AI_ADDRCONFIG), а ai_family равен AF_UNSPEC (в библиотеке glibc специально предусмотрено отклонение от стандарта SUSv3, согласно которому поле ai_flags считается равным 0, если аргумент hints равен NULL).

55.10.2. Удаление списков со структурами addrinfo: freeaddrinfo()

Функция getaddrinfo() динамически выделяет память для всех структур, на которые указывает result (см. рис. 55.3). Следовательно, когда эти структуры больше не нужны, вызывающему процессу нужно освободить занимаемые ими ресурсы. Для выполнения данной процедуры за один вызов предусмотрена функция freeaddrinfo().

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *result);
```

Если нужно сохранить копию одной из структур addrinfo или связанную с ней структуру с адресом сокета, то перед вызовом freeaddrinfo() их нужно продублировать.

55.10.3. Выявление ошибок: gai_strerror()

В случае ошибки функция getaddrinfo() возвращает один из ненулевых кодов, перечисленных в табл. 55.1.

Таблица 55.1. Ошибки, возвращаемые функциями getaddrinfo() и getnameinfo()

Ошибка	Описание
EAI_ADDRFAMILY	У узла host нет адресов в семействе hints.ai_family (не входит в стандарт SUSv3, но поддерживается большинством реализаций; только для getaddrinfo())
EAI_AGAIN	Временный сбой при разрешении имени (нужно попробовать позже)
EAI_BADFLAGS	В поле hints.ai_flags был указан некорректный флаг
EAI_FAIL	При доступе к DNS-серверу произошла необратимая ошибка
EAI_FAMILY	Семейство адресов, заданное в поле hints.ai_family, не поддерживается
EAI_MEMORY	Сбой при выделении памяти
EAI_NODATA	С именем host не связан ни один адрес (не входит в стандарт SUSv3, но поддерживается большинством реализаций; только для getaddrinfo())
EAI_NONAME	Неизвестные имена host или service, оба аргумента host и service равны NULL, либо же был задан флаг AI_NUMERICSERV, а строка, на которую указывает service, не содержит число

Ошибка	Описание
EAI_OVERFLOW	Переполнение буфера в аргументе
EAI_SERVICE	Заданная служба service не поддерживается типом hints.ai_socktype (только для getaddrinfo())
EAI_SOCKTYPE	Заданный тип hints.ai_socktype не поддерживается (только для getaddrinfo())
EAI_SYSTEM	Системная ошибка, возвращенная в глобальной переменной errno

Функция `gai_strerror()` возвращает строку с описанием одной из ошибок, перечисленных в табл. 55.1 (эти описания обычно более короткие, чем те, что представлены выше).

```
#include <netdb.h>

const char *gai_strerror(int errcode);
```

Возвращает указатель на строку, содержащую сообщение об ошибке

Строку, возвращенную функцией `gai_strerror()`, можно использовать как часть сообщения об ошибке, выводимого приложением.

55.10.4. Функция getnameinfo()

Функция `getnameinfo()` является противоположностью функции `getaddrinfo()`. При передаче ей структуры с адресом сокета (в формате IPv4 или IPv6) она вернет строку с именами соответствующих узла и службы или их числовой эквивалент, если не удается найти соответствия.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host,
                size_t hostlen, char *service, size_t servlen, int flags);
```

Возвращает `0` при успешном завершении или ненулевое значение при ошибке

Аргумент `addr` представляет собой указатель на структуру с адресом сокета, которую нужно преобразовать. Размер ее задается с помощью аргумента `addrlen`. Обычно значения этих аргументов можно получить из вызовов `accept()`, `recvfrom()`, `getsockname()` или `getpeername()`.

Итоговые имена узла и службы возвращаются в виде строк с нулевым символом в конце, на которые указывают аргументы `host` и `service`. Эти строки хранятся в буферах и должны быть выделены вызывающим процессом; их размеры следует передавать с помощью аргументов `hostlen` и `servlen`. В заголовочном файле `<netdb.h>` определены две константы, помогающие подобрать размеры данных буферов. Константа `NI_MAXHOST` обозначает максимальный размер строки с именем узла (в байтах). Она имеет значение 1025. Константа `NI_MAXSERV` обозначает максимальный размер строки с именем службы (в байтах) и равна 32. Обе константы не входят в стандарт SUSv3, но поддерживаются во всех реализациях UNIX, которые предоставляют функцию `getnameinfo()` (начиная с версии glibc 2.8, для получения

их определений необходимо задать один из следующих макросов проверки возможностей: _BSD_SOURCE, _SVID_SOURCE или _GNU_SOURCE).

Если нам не нужно имя узла, то аргументу host можно присвоить значение NULL, а hostlen передать 0. Аналогично, если нас не интересует имя службы, то аргументам service и servlen можно присвоить значения NULL и соответственно 0. При этом хотя бы один аргумент, host или service, должен иметь ненулевое значение (то же самое касается соответствующего аргумента, обозначающего длину).

Последний аргумент, flags, представляет собой битовую маску, которая влияет на поведение функции getnameinfo(). Маска формируется путем применения побитового ИЛИ к следующим значениям.

- NI_DGRAM — по умолчанию функция getnameinfo() возвращает имя, связанное со службой *потокового* (TCP) сокета. Обычно это не важно, ведь, как отмечалось в разделе 55.9, для TCP- и UDP-портов почти всегда используются одни и те же имена служб. Однако есть несколько случаев, когда имена все же отличаются; в таких ситуациях флаг NI_DGRAM приводит к возвращению имени службы датаграммного (UDP) сокета.
- NI_NAMEREQD — по умолчанию, если имя узла не удается найти, то вместо него возвращается числовой адрес. Указание флага NI_NAMEREQD повлечет ошибку EAI_NONAME.
- NI_NOFQDN — по умолчанию возвращается полное доменное имя узла. Если указать флаг NI_NOFQDN, то получим только первую часть имени, то есть имя конкретного компьютера (при условии, что он подключен к нашей локальной сети).
- NI_NUMERICHOST — делает так, что строка с адресом возвращается в аргументе host. Это бывает удобно в том случае, если мы хотим избежать запроса к DNS-серверу, который может занять довольно много времени.
- NI_NUMERICSERV — способствует возвращению строки с десятичным представлением номера порта в аргументе service. Это может пригодиться, когда мы знаем, что заданный порт не соответствует имени службы (например, если это динамический порт, присвоенный сокету ядром), и нам не хочется лишний раз выполнять поиск по файлу /etc/services.

При успешном завершении функция getnameinfo() возвращает 0, в случае ошибки — один из ненулевых кодов, описанных в табл. 55.1.

55.11. Пример клиент-серверного приложения на основе потоковых сокетов

Знания, приобретенные в предыдущих разделах, позволяют нам рассмотреть простое клиент-серверное приложение на основе TCP-сокетов. Задача, выполняемая этим приложением, аналогична той, которую решает клиент-серверная программа из раздела 44.8, использующая очередь FIFO: передача клиентам уникальной последовательности чисел (или диапазонов таких последовательностей).

Целые числа могут иметь разное представление на серверном и клиентском компьютерах, поэтому мы переводим их в строки с символом новой строки в конце, а для их чтения применяем функцию readLine() (см. листинг 55.1).

Общий заголовочный файл

И сервер, и клиент задействуют заголовочный файл, представленный в листинге 55.5, который, в свою очередь, подключает другие файлы и определяет номер TCP-порта для нашего приложения.

Серверная программа

Серверная программа, представленная в листинге 55.6, выполняет следующие шаги.

- Начинает последовательность либо с 1, либо с числа, переданного в виде аргумента командной строки (1).
- Игнорирует сигнал `SIGPIPE` (2), возникающий, когда сервер пытается выполнить запись в сокет, другой конец которого был закрыт; вместо этого операция `write()` завершается ошибкой `EPIPE`.
- Вызывает функцию `getaddrinfo()` (4) для получения списка структур с адресами сокетов, подключенных к TCP-порту `PORT_NUM` (вместо того чтобы задавать фиксированный номер порта, обычно лучше использовать имя службы). Мы указываем флаг `AI_PASSIVE` (3) для привязки итогового сокета к универсальному адресу (см. раздел 54.5). Как следствие, сервер сможет принимать запросы на соединение, направленные на любой из его адресов (если их у него несколько).
- Входит в цикл, перебирающий структуры с адресами сокетов, полученными в предыдущем шаге (5). Цикл завершается, когда программа находит адрес, с помощью которого можно успешно создать и привязать сокет (7).
- Устанавливает параметр `SO_REUSEADDR` для сокета, созданного в предыдущем шаге (6). Мы вернемся к обсуждению этого параметра в разделе 57.10, где будет отмечено, что TCP-сервер обычно должен устанавливать `SO_REUSEADDR` для своего слушающего сокета.
- Делает сокет слушающим (8).
- Входит в бесконечный цикл `for` (9), который обслуживает клиентов итерационным способом (см. главу 56). Сервер не принимает очередной клиентский запрос, пока не закончит обслуживание текущего. Для каждого клиента сервер выполняет следующие шаги.
  - Принимает новое соединение (10). Передает второму и третьему аргументу вызова `accept()` ненулевые указатели, чтобы получить адрес клиента (IP-адрес плюс номер порта) и направить его в стандартный вывод (11).
  - Считывает клиентское сообщение (12), которое заканчивается символом новой строки, и определяет, сколько последовательных номеров запрашивает клиент. Сервер переводит эту строку в целое число и сохраняет результат в переменной `reqLen` (13).
  - Возвращает клиенту текущий элемент последовательности (`seqNum`) в виде строки с нулевым символом в конце (14). Клиент может считать, что для него была выделена последовательность номеров в диапазоне от `seqNum` до `(seqNum + reqLen - 1)`.
  - Обновляет текущий элемент последовательности, добавляя к `seqNum` значение `reqLen` (15).

Листинг 55.5. Заголовочный файл, используемый программами is_seqnum_sv.c и is_seqnum_cl.c

sockets/is_seqnum.h

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <signal.h>
#include "read_line.h"          /* Объявление readLine() */
#include "tlpi_hdr.h"

#define PORT_NUM "50000"        /* Номер порта для сервера */

#define INT_LEN 30              /* Размер строки, достаточный для хранения наибольшего
                                   целого числа (включая завершающий символ '\n') */
```

sockets/is_seqnum.h

Листинг 55.6. Итерационный сервер, который взаимодействует с клиентами с помощью потокового сокета

sockets/is_seqnum_sv.c

```
#define _BSD_SOURCE         /* Получаем определения NI_MAXHOST и NI_MAXSERV
                               из файла <netdb.h> */
#include <netdb.h>
#include "is_seqnum.h"

#define BACKLOG 50

int
main(int argc, char *argv[])
{
    uint32_t seqNum;
    char reqLenStr[INT_LEN];     /* Длина запрашиваемой последовательности */
    char seqNumStr[INT_LEN];     /* Начало выделенной последовательности */
    struct sockaddr_storage claddr;
    int lfd, cfd, optval, reqLen;
    socklen_t addrlen;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
#define ADDRSTRLEN (NI_MAXHOST + NI_MAXSERV + 10)
    char addrStr[ADDRSTRLEN];
    char host[NI_MAXHOST];
    char service[NI_MAXSERV];

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [init-seq-num]\n", argv[0]);

(1) seqNum = (argc > 1) ? getInt(argv[1], 0, "init-seq-num") : 0;

(2) if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");

    /* Вызываем getaddrinfo(), чтобы получить список адресов,
       к которым можем попытаться привязать наш сокет */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_family = AF_UNSPEC;    /* Поддержка IPv4 или IPv6 */
(3) hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;
                    /* Универсальный IP-адрес; имя службы имеет числовой формат */
(4) if (getaddrinfo(NULL, PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Перебираем полученный список, пока не находим структуру с адресом,
       подходящую для создания и привязывания сокета */
    optval = 1;
(5) for (rp = result; rp != NULL; rp = rp->ai_next) {
        lfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (lfd == -1)
            continue;        /* В случае ошибки пробуем следующий адрес */
(6)     if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &optval,
              sizeof(optval)) == -1)
            errExit("setsockopt");
```

```
❼      if (bind(lfd, rp->ai_addr, rp->ai_addrlen) == 0)
           break;                       /* Успех */

   /* Вызов bind() завершился неудачно; закрываем этот сокет и пробуем следующий адрес */
       close(lfd);
   }

   if (rp == NULL)
       fatal("Could not bind socket to any address");

❽  if (listen(lfd, BACKLOG) == -1)
       errExit("listen");

   freeaddrinfo(result);

❾  for (;;) {            /* Обслуживаем клиентов итерационно */

       /* Принимаем соединение со стороны клиента и получаем его адрес */
       addrlen = sizeof(struct sockaddr_storage);
❿      cfd = accept(lfd, (struct sockaddr *) &claddr, &addrlen);
       if (cfd == -1) {
           errMsg("accept");
           continue;
       }

⓫      if (getnameinfo((struct sockaddr *) &claddr, addrlen,
                   host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0)
           snprintf(addrStr, ADDRSTRLEN, "(%s, %s)", host, service);
       else
           snprintf(addrStr, ADDRSTRLEN, "(?UNKNOWN?)");
       printf("Connection from %s\n", addrStr);

       /* Считываем запрос клиента, возвращаем в ответ элемент последовательности */
⓬      if (readLine(cfd, reqLenStr, INT_LEN) <= 0) {
           close(cfd);
           continue;        /* Ошибка при чтении; пропускаем запрос */
       }

⓭      reqLen = atoi(reqLenStr);
       if (reqLen <= 0) {   /* Отслеживаем клиентов, которые ведут себя некорректно */
           close(cfd);
           continue;        /* Некорректный запрос; пропускаем его */
       }

⓮      snprintf(seqNumStr, INT_LEN, "%d\n", seqNum);
       if (write(cfd, seqNumStr, strlen(seqNumStr)) != strlen(seqNumStr))
           fprintf(stderr, "Error on write");

⓯       seqNum += reqLen;     /* Обновляем элемент последовательности */
       if (close(cfd) == -1)  /* Закрываем соединение */
           errMsg("close");
   }
}
```

sockets/is_seqnum_sv.c

Клиентская программа

Клиентская программа, показанная в листинге 55.7, принимает два аргумента. Первый, обозначающий имя узла, на котором работает сервер, является обязательным.

Второй аргумент, определяющий длину последовательности, запрашиваемой клиентом, можно пропустить. По умолчанию длина равна 1. Клиент выполняет следующие шаги.

- Вызывает функцию `getaddrinfo()`, чтобы получить список структур с адресами, подходящими для подключения к TCP-серверу, привязанному к заданному узлу (1). В качестве номера порта клиент указывает значение `PORT_NUM`.
- Входит в цикл (2), который перебирает адреса, полученные в предыдущем шаге, пока не найдет тот, что позволяет успешно создать (3) сокет и подключиться с его помощью к серверу (4). Так как клиент не привязывает свой сокет, в результате вызова `connect()` ядро автоматически назначает ему динамический порт.
- Отправляет целое число, обозначающее длину нужной клиенту последовательности (5). Это число передается в строковом виде с символом новой строки в конце.
- Считывает номер последовательности, переданный сервером (который тоже имеет строковый формат и символ новой строки в конце) (6), и направляет его в стандартный вывод (7).

Запустив сервер и клиент на одном и том же компьютере, мы увидим следующее:

```
$ ./is_seqnum_sv &
[1] 4075
$ ./is_seqnum_cl localhost          Клиент 1: запрашивает один элемент
                                    последовательности
Connection from (localhost, 33273)  Сервер выводит адрес и порт клиента
Sequence number: 0                  Клиент выводит элемент
                                    последовательности
$ ./is_seqnum_cl localhost 10       Клиент 2: запрашивает десять элементов
                                    последовательности
Connection from (localhost, 33274)
Sequence number: 1
$ ./is_seqnum_cl localhost          Клиент 3: запрашивает один элемент
                                    последовательности
Connection from (localhost, 33275)
Sequence number: 11
```

Теперь покажем, как с помощью утилиты `telnet` можно выполнить отладку этого приложения:

```
$ telnet localhost 50000            Наш сервер использует этот номер порта
                                    Пустая строка, которую выводит telnet
Trying 127.0..0.1...
Connection from (localhost, 33276)
Connected to localhost.
Escape character is '^]'.
1                                   Вводим длину запрашиваемой последовательности
12                                  telnet выводит элемент последовательности
Connection closed by foreign host.  и обнаруживает, что сервер закрыл соединение
```

В сессии командной строки, приведенной выше, можно видеть, что ядро циклически перебирает номера динамических портов (другие реализации демонстрируют похожее поведение). В Linux такое поведение является результатом оптимизации, призванной сократить количество запросов к таблице ядра, хранящей локальные привязки сокетов. При достижении максимального значения ядро опять начинает выделять доступные номера, начиная с наименьшего элемента диапазона (который, в свою очередь, определяется в файле /proc/sys/net/ipv4/ip_local_port_range, доступном только в Linux).

Листинг 55.7. Клиент, использующий потоковые сокеты

sockets/is_seqnum_cl.c

```
#include <netdb.h>
#include "is_seqnum.h"

int
main(int argc, char *argv[])
{
    char *reqLenStr;            /* Длина запрашиваемой последовательности */
    char seqNumStr[INT_LEN];    /* Начало выделенной последовательности */
    int cfd;
    ssize_t numRead;
    struct addrinfo hints;
    struct addrinfo *result, *rp;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s server-host [sequence-len]\n", argv[0]);

    /* Вызываем getaddrinfo(), чтобы получить список адресов,
       к которым можем попытаться привязать наш сокет */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;        /* Поддержка IPv4 или IPv6 */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_NUMERICSERV;

(1) if (getaddrinfo(argv[1], PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Перебираем полученный список, пока не находим структуру с адресом,
       подходящую для успешного соединения с сокетом */
(2) for (rp = result; rp != NULL; rp = rp->ai_next) {
(3)     cfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (cfd == -1)
            continue;       /* В случае ошибки пробуем следующий адрес */

(4)     if (connect(cfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;              /* Успех */

     /* Сбой соединения: закрываем этот сокет и пробуем следующий адрес */
        close(cfd);
    }

    if (rp == NULL)
        fatal("Could not connect socket to any address");

    freeaddrinfo(result);

/* Отправляем длину запрашиваемой последовательности с символом новой строки в конце */
(5) reqLenStr = (argc > 2) ? argv[2] : "1";
    if (write(cfd, reqLenStr, strlen(reqLenStr)) != strlen(reqLenStr))
        fatal("Partial/failed write (reqLenStr)");
    if (write(cfd, "\n", 1) != 1)
        fatal("Partial/failed write (newline)");

    /* Считываем и выводим число, полученное от сервера */
(6) numRead = readLine(cfd, seqNumStr, INT_LEN);
```

```
    if (numRead == -1)
        errExit("readLine");
    if (numRead == 0)
        fatal("Unexpected EOF from server");

❼   printf("Sequence number: %s", seqNumStr);     /* Включает в себя '\n' */

    exit(EXIT_SUCCESS);                           /* Закрывает 'cfd' */
}
```

sockets/is_seqnum_cl.c

55.12. Библиотека для работы с сокетами интернет-домена

Здесь мы воспользуемся функциями, представленными в разделе 55.10, чтобы реализовать библиотеку для выполнения задач, которые обычно необходимо решать при работе с сокетами интернет-домена (данная библиотека инкапсулирует многие из тех шагов, которые мы показали на примере программ в разделе 55.11). Так как эти функции используют вызовы `getaddrinfo()` и `getnameinfo()`, не зависящие от протокола, их можно применять как для IPv4, так и для IPv6.

Заголовочный файл с объявлением данных функций представлен в листинге 55.8. Многие из них имеют похожие аргументы.

- Аргумент `host` содержит строку с именем сетевого узла или его числовым адресом (для IPv4 это десятичные числа, разделенные точкой, а для IPv6 — шестнадцатеричная строка). Можно также передать аргументу `host` нулевой указатель, чтобы использовать IP-адрес, замкнутый на себя.
- Аргументу `service` можно передать либо имя службы, либо порт, заданный в виде десятичной строки.
- Аргумент `type` обозначает тип сокета: `SOCK_STREAM` или `SOCK_DGRAM`.

Листинг 55.8. Заголовочный файл для программы inet_sockets.c

sockets/inet_sockets.h

```
#ifndef INET_SOCKETS_H
#define INET_SOCKETS_H     /* Не даем подключить заголовок больше одного раза */

#include <sys/socket.h>
#include <netdb.h>

int inetConnect(const char *host, const char *service, int type);
int inetListen(const char *service, int backlog, socklen_t *addrlen);
int inetBind(const char *service, int type, socklen_t *addrlen);
char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                char *addrStr, int addrStrLen);

#define IS_ADDR_STR_LEN 4096
    /* Рекомендуемая длина строкового буфера, который нужно передать
       в inetAddressStr(). Должна быть больше чем (NI_MAXHOST + NI_MAXSERV + 4) */
#endif
```

sockets/inet_sockets.h

Функция `inetConnect()` создает сокет типа `type` и подключает его к адресу, заданному с помощью аргументов `host` и `service`. Эта функция предназначена для TCP- и UDP-клиентов, которым нужно подключиться к серверному сокету.

```
#include "inet_sockets.h"

int inetConnect(const char *host, const char *service, int type);
```

Возвращает файловый дескриптор или -1 при ошибке

Файловый дескриптор нового сокета возвращается в качестве результата выполнения функции.

Функция `inetListen()` создает слушающий сокет (`SOCK_STREAM`), привязанный к универсальному IP-адресу и TCP-порту, заданному с помощью аргумента `service`. Она предназначена для использования в TCP-серверах.

```
#include "inet_sockets.h"

int inetListen(const char *service, int backlog, socklen_t *addrlen);
```

Возвращает файловый дескриптор или -1 при ошибке

Файловый дескриптор нового сокета возвращается в качестве результата выполнения функции.

Аргумент `backlog` определяет допустимое количество отложенных соединений (так же, как `listen()`).

Если аргумент `addrlen` является ненулевым указателем, то участок памяти, на который он ссылается, будет содержать итоговый размер структуры с адресом сокета, связанного с возвращенным файловым дескриптором. Это значение позволяет выделить буфер подходящего размера, в который будет помещен адрес сокета; позже, если понадобится адрес подключающегося клиента, можно передать его функции `accept()`.

Функция `inetBind()` создает сокет типа `type` и привязывает его к универсальному IP-адресу и порту, заданному с помощью аргументов `service` и `type` (тип сокета определяет протокол службы — TCP или UDP). Эта функция в основном предназначена для создания и привязки сокетов к определенному адресу в UDP-серверах и их клиентах.

```
#include "inet_sockets.h"

int inetBind(const char *service, int type, socklen_t *addrlen);
```

Возвращает файловый дескриптор или -1 при ошибке

Файловый дескриптор нового сокета возвращается в качестве результата выполнения функции.

Функция `inetBind()`, как и `inetListen()`, возвращает размер структуры с адресом заданного сокета, используя участок памяти, на который ссылается аргумент `addrlen`. Это может пригодиться, если нужно выделить буфер, впоследствии передаваемый в вызов `recvfrom()`, — так можно получить адрес сокета, передающего датаграмму. (Большинство действий, которые нужно выполнить для применения функций `inetListen()` и `inetBind()`, совпадают; они реализованы в виде единого вызова под названием `inetPassiveSocket()`.)

Функция `inetAddressStr()` приводит адрес интернет-сокета к презентационному виду.

```
#include "inet_sockets.h"

char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                     char *addrStr, int addrStrLen);
```

Возвращает указатель на `addrStr` (строку, хранящую имена узла и службы)

На основе структуры с адресом сокета, которая передается в аргумент `addr` и имеет размер `addrlen`, функция `inetAddressStr()` возвращает строку с нулевым символом в конце, содержащую соответствующие имя узла и номер порта в следующем формате:

```
(hostname, port-number)
```

Эта строка возвращается в буфере, на который указывает аргумент `addrStr`. Вызывающий процесс должен задать размер данного буфера с помощью `addrStrLen`. Полученная строка урезается, если ее размер превышает `(addrStrLen - 1)` байт. Рекомендуемый размер буфера `addrStr` определен в константе `IS_ADDR_STR_LEN`; этого значения должно быть достаточно для того, чтобы вместить любые строки, которые можно получить. В качестве результата функция `inetAddressStr()` возвращает `addrStr`.

Реализация функций, описанных в данном разделе, представлена в листинге 55.9.

Листинг 55.9. Библиотека для работы с сокетами интернет-домена

sockets/inet_sockets.c

```
#define _BSD_SOURCE  /* Для получения определений NI_MAXHOST и NI_MAXSERV из <netdb.h> */

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "inet_sockets.h"   /* Объявляет функции, определяемые здесь */
#include "tlpi_hdr.h"

int
inetConnect(const char *host, const char *service, int type)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;    /* Поддерживает IPv4 или IPv6 */
    hints.ai_socktype = type;
    s = getaddrinfo(host, service, &hints, &result);
    if (s != 0) {
        errno = ENOSYS;
        return -1;
    }

    /* Перебираем полученный список, пока не найдем структуру с адресом,
       с помощью которого можно успешно подключиться к сокету */
    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
```

```
            continue;      /* В случае ошибки пробуем следующий адрес */
        if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                  /* Успех */

    /* Ошибка соединения: закрываем этот сокет и пробуем следующий адрес */
        close(sfd);
    }
    freeaddrinfo(result);
    return (rp == NULL) ? -1 : sfd;
}

static int       /* Публичные интерфейсы: inetBind() и inetListen() */
inetPassiveSocket(const char *service, int type, socklen_t *addrlen,
                  Boolean doListen, int backlog)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, optval, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = type;
    hints.ai_family = AF_UNSPEC;    /* Поддерживает IPv4 или IPv6 */
    hints.ai_flags = AI_PASSIVE;    /* Используем универсальный IP-адрес */
    s = getaddrinfo(NULL, service, &hints, &result);
    if (s != 0)
        return -1;

    /* Перебираем полученный список, пока не найдем структуру с адресом,
       с помощью которого можно успешно создать и привязать сокет */
    optval = 1;
    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;      /* В случае ошибки пробуем следующий адрес */

        if (doListen) {
            if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval,
                    sizeof(optval)) == -1) {
                close(sfd);
                freeaddrinfo(result);
                return -1;
            }
        }

        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
            break;                      /* Успех */

        /* Сбой в вызове bind(): закрываем этот сокет и пробуем следующий адрес */
        close(sfd);
    }

    if (rp != NULL && doListen) {
        if (listen(sfd, backlog) == -1) {
            freeaddrinfo(result);
            return -1;
        }
    }
```

```
    if (rp != NULL && addrlen != NULL)
        *addrlen = rp->ai_addrlen;
        /* Возвращаем размер структуры с адресом */
    freeaddrinfo(result);
    return (rp == NULL) ? -1 : sfd;
}

int
inetListen(const char *service, int backlog, socklen_t *addrlen)
{
    return inetPassiveSocket(service, SOCK_STREAM, addrlen, TRUE, backlog);
}

int
inetBind(const char *service, int type, socklen_t *addrlen)
{
    return inetPassiveSocket(service, type, addrlen, FALSE, 0);
}

char *
inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
               char *addrStr, int addrStrLen)
{
    char host[NI_MAXHOST], service[NI_MAXSERV];
    if (getnameinfo(addr, addrlen, host, NI_MAXHOST,
                    service, NI_MAXSERV, NI_NUMERICSERV) == 0)
        snprintf(addrStr, addrStrLen, "(%s, %s)", host, service);
    else
        snprintf(addrStr, addrStrLen, "(?UNKNOWN?)");
    addrStr[addrStrLen - 1] = '\0';  /* Не забываем добавить нулевой символ в конце */
    return addrStr;
}
```

sockets/inet_sockets.c

55.13. Сравнение сокетов в UNIX- и интернет-доменах

При написании приложений, которые взаимодействуют по сети, использование сокетов интернет-домена является обязательным. Но если наши приложения находятся на одном и том же компьютере, то можно применять сокеты как в интернет-, так и в UNIX-домене. Так какой же домен лучше выбрать в таком случае и чем при этом следует руководствоваться?

Применение сокетов интернет-домена обычно является наиболее простым вариантом, потому что это позволяет запускать программы как локально, так и на разных узлах сети. Однако существует несколько причин, которые могут склонить нас к выбору UNIX-домена.

- В ряде систем сокеты UNIX-домена работают быстрее, чем сокеты интернет-домена.
- Для управления доступом к сокетам UNIX-домена можно применять права доступа к каталогу (и к файлу, если речь идет о Linux). Таким образом можно позволить подключаться к слушающему сокету или отправлять датаграммы только приложениям, которые имеют заданного владельца или входят в определенную группу. Так мы получаем простой механизм аутентификации клиентов. В интернет-домене для организации чего-то подобного требуются дополнительные действия.
- При использовании сокетов UNIX-домена мы передаем дескрипторы открытых файлов и учетные данные отправителя (см. подраздел 57.13.3).

55.14. Дополнительная информация

Существует множество печатных и электронных ресурсов, посвященных TCP/IP и программным интерфейсам сокетов.

- [Stevens at al., 2004] — ключевая книга по сетевому программированию с помощью программного интерфейса сокетов. [Snader, 2000] содержит дополнительные полезные руководства на данную тему.
- [Stevens, 1994] и [Wright & Stevens, 1995] описывают стек TCP/IP в подробностях. [Comer, 2000], [Comer & Stevens, 1999], [Comer & Stevens, 2000], [Kozierok, 2005] и [Goralski, 2009] этот материал тоже хорошо освещен.
- [Tanenbaum, 2002] содержит общие сведения об истории компьютерных сетей.
- [Herbert, 2004] подробно описывает стек TCP/IP в Linux 2.6.
- Руководство GNU языка C (находится на http://www.gnu.org/) содержит обширную информацию о программном интерфейсе сокетов.
- На сайте IBM Redbook находится документ под названием TCP/IP Tutorial and Technical Overview, в котором подробно описаны основные понятия, относящиеся к сетям, внутренности TCP/IP, программный интерфейс сокетов и множество связанных с этим тем. Документ доступен для свободной загрузки на http://www.redbooks.ibm.com/.
- [Gont, 2008] и [Gont, 2009 (b)] рассматривают протоколы IPv4 и TCP с точки зрения безопасности.
- Новостная рассылка comp.protocols.tcp-ip посвящена вопросам, связанным с сетевыми протоколами семейства TCP/IP.
- В книге [Sarolahti & Kuznetsov, 2002] описывается управление перегрузкой и другие нюансы реализации TCP в Linux.
- Информацию, относящуюся в первую очередь к Linux, можно найти на следующих страницах руководства: `socket(7)`, `ip(7)`, `raw(7)`, `tcp(7)`, `udp(7)` и `packet(7)`.
- См. также списки RFC-документов в разделе 54.7.

55.15. Резюме

Сокеты интернет-домена позволяют приложениям, находящимся на разных компьютерах, взаимодействовать друг с другом по сети TCP/IP. Адреса этих сокетов включают в себя IP-адрес и номер порта. В IPv4 числовые адреса занимают 32 бита, а в IPv6 — 128 бит. Датаграммные сокеты интернет-домена работают по протоколу UDP и предоставляют ненадежный канал взаимодействия, ориентированный на обмен сообщениями и не поддерживающий соединения. Потоковые сокеты интернет-домена работают по протоколу TCP; они обеспечивают надежный, двунаправленный, основанный на байтовом потоке канал обмена данными между двумя приложениями, соединенными друг с другом.

Разные компьютерные архитектуры используют различные правила представления типов данных. Например, в целом числе байты могут следовать от старшего к младшему и наоборот; к тому же количество байтов, выделяемых для типов `int` или `long`, может варьироваться в зависимости от компьютера. Ввиду этих отличий при передаче данных между гетерогенными компьютерами, соединенными по сети, нужно применять некое универсальное представление, не зависящее от архитектуры. Мы отметили, что для решения этой проблемы существуют разные стандарты маршалинга; а также описали методику, которая применяется во многих приложениях и состоит в переводе всех передаваемых данных в текстовый вид, где поля разделяются специальными символами (обычно символом новой строки).

Мы рассмотрели ряд функций для преобразования между строковыми представлениями IP-адресов (в IPv4 это десятичные числа, разделенные точками, а в IPv6 — шестнадцатеричная строка) и их двоичными эквивалентами. Хотя обычно более предпочтительным является использование имен узлов и служб, так как по сравнению с числами их легче запоминать; к тому же они остаются актуальными, даже если поменять соответствующие IP-адреса. Мы обсудили различные функции для перевода имен узлов и служб в их числовое представление (и наоборот). На сегодняшний день стандартной функцией для указанных операций является `getaddrinfo()`, но в имеющемся коде можно встретить ее устаревшие аналоги — `gethostbyname()` и `getservbyname()`.

Вслед за преобразованием имен узлов мы плавно подошли к обсуждению DNS — иерархической службы каталогов, реализованной в виде распределенной базы данных. Преимущество ее таково: информация об именах не централизована. Вместо этого существуют отдельные локальные зоны, администраторы которых отвечают за обновление соответствующих элементов иерархии. Чтобы получить IP-адрес для какого-либо имени узла, DNS-серверы должны взаимодействовать друг с другом.

55.16. Упражнения

55.1. Функция `readLine()`, показанная в листинге 55.1, не очень эффективна при чтении больших объемов данных, поскольку для получения каждого символа требуется отдельный системный вызов. Более рациональным подходом было бы считывать символы блоками, записывать их в буфер и только потом извлекать их построчно. Такой интерфейс может состоять из двух функций. Первую из них можно назвать `readLineBufInit(fd, &rlbuf)`; она инициализирует структуру данных, на которую указывает `rlbuf`. В этой структуре предусмотрено место для буфера с символами, информации о его размере и указателя на следующий «непрочитанный» символ в буфере. В нем также хранится копия файлового дескриптора, переданного в виде аргумента `fd`. Вторая функция, `readLineBuf(&rlbuf)`, возвращает из буфера `rlbuf` следующую строку. При необходимости она может прочитать дополнительный блок данных из файлового дескриптора, сохраненного в структуре `rlbuf`. Реализуйте обе эти функции. Внедрите их в программы `is_seqnum_sv.c` (листинг 55.6) и `is_seqnum_cl.c` (листинг 55.7).

55.2. Измените программы `is_seqnum_sv.c` и `is_seqnum_cl.c` из листингов 55.6 и 55.7 так, чтобы в них использовались функции `inetListen()` и `inetConnect()`, представленные в листинге 55.9 (`inet_sockets.c`).

55.3. Напишите библиотеку для работы с сокетами домена UNIX. Ее программный интерфейс должен быть похож на тот, который применяется для сокетов интернет-домена в разделе 55.12. Перепишите программы `us_xfr_sv.c` (листинг 53.3) и `us_xfr_cl.c` (листинг 53.4) с использованием этой библиотеки.

55.4. Напишите сетевой сервер, хранящий пары вида «имя-значение». Он должен поддерживать добавление, удаление, изменение и извлечение имен со стороны клиентов. Чтобы протестировать этот сервер, напишите одну или несколько клиентских программ. При желании реализуйте некий механизм безопасности, позволяющий удалять имя или изменять связанное с ним значение только тому клиенту, который его создал.

55.5. Предположим, мы создаем в интернет-домене два датаграммных сокета, привязанных к определенным адресам, и соединяем первый со вторым. Что произойдет, если мы создадим третий датаграммный сокет и попытаемся с его помощью послать сообщение (`sendto()`) первому сокету? Напишите программу, которая отвечает на этот вопрос.

56 Сокеты: архитектура сервера

В этой главе мы обсудим основы проектирования итерационных и параллельных серверов, а также рассмотрим специальный демон inetd, который облегчает создание серверных интернет-приложений.

56.1. Итерационные и параллельные серверы

Существуют две распространенные архитектуры сетевых серверов на основе сокетов:

- *итерационная:* сервер обслуживает клиентов по одному, сначала обрабатывая запрос (или несколько запросов) одного клиента и затем переходя к следующему;
- *параллельная:* сервер спроектирован для обслуживания нескольких клиентов одновременно.

В разделе 44.8 уже был представлен пример итерационного сервера на основе очередей FIFO.

Итерационные серверы обычно подходят только в ситуациях, когда клиентские запросы можно обработать достаточно быстро, так как каждый клиент вынужден ждать, пока не обслужат любых других клиентов, находящихся перед ним. Обычным сценарием использования этого подхода является обмен единичными запросами и ответами между клиентом и сервером.

Параллельные серверы подходят в случаях, когда на обработку каждого запроса уходит значительное количество времени или клиент и сервер выполняют длительный обмен сообщениями. В данной главе мы в основном сосредоточимся на традиционном (и наиболее простом) способе проектирования параллельных серверов, который состоит в создании отдельного дочернего процесса для каждого нового клиента. Такой процесс выполняет всю работу по обслуживанию клиента, после чего завершается. Поскольку каждый из этих процессов функционирует независимо, можно обслуживать несколько клиентов одновременно. Основная задача главного серверного процесса (родителя) заключается в создании отдельного потомка для каждого нового клиента (как вариант, вместо процессов можно создавать потоки выполнения).

В следующих разделах мы рассмотрим примеры итерационного и параллельного серверов на основе сокетов интернет-домена. Эти два сервера реализуют упрощенный вариант службы `echo` (RFC 862), которая возвращает копию любого сообщения, посланного ей клиентом.

56.2. Итерационный UDP-сервер echo

В этом и следующем разделе мы представим серверы для службы `echo`. Она доступна на порте с номером 7 и работает как по UDP, так и по TCP (данный порт зарезервирован, в связи с чем сервер `echo` необходимо запускать с привилегиями администратора).

UDP-сервер `echo` постоянно считывает датаграммы и возвращает отправителю их копии. Поскольку серверу нужно обрабатывать только одно сообщение за раз, здесь будет достаточно итерационной архитектуры. Заголовочный файл для серверов показан в листинге 56.1.

Листинг 56.1. Заголовочный файл для программ id_echo_sv.c и id_echo_cl.c

sockets/id_echo.h

```
#include "inet_sockets.h"   /* Объявляет функции нашего сокета */
#include "tlpi_hdr.h"

#define SERVICE "echo"      /* Имя UDP-службы */

#define BUF_SIZE 500        /* Максимальный размер датаграмм, которые
                               могут быть прочитаны клиентом и сервером */
```

sockets/id_echo.h

В листинге 56.2 представлена реализация сервера. Стоит отметить следующие моменты:

- для перевода сервера в режим демона мы задействуем функцию `becomeDaemon()` из раздела 37.2;
- чтобы сделать программу более компактной, мы используем библиотеку для работы с сокетами интернет-домена, разработанную в разделе 55.12;
- если сервер не может вернуть ответ клиенту, то записывает сообщение в журнал, применяя вызов `syslog()`.

  В реальном приложении мы бы, скорее всего, ввели определенное ограничение на частоту записи сообщений с помощью syslog(). Это исключило бы возможность переполнения системного журнала злоумышленником. К тому же не стоит забывать, что каждый вызов syslog() довольно затратный, так как по умолчанию использует fsync().

Листинг 56.2. Итерационный сервер, который реализует UDP-службу echo

sockets/id_echo_sv.c

```
#include <syslog.h>
#include "id_echo.h"
#include "become_daemon.h"

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    socklen_t len;
    struct sockaddr_storage claddr;
    char buf[BUF_SIZE];
    char addrStr[IS_ADDR_STR_LEN];

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");

    sfd = inetBind(SERVICE, SOCK_DGRAM, NULL);
    if (sfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)",
            strerror(errno));
        exit(EXIT_FAILURE);
    }
```

```
    /* Получаем датаграммы и возвращаем отправителям их копии */
    for (;;) {
        len = sizeof(struct sockaddr_storage);
        numRead = recvfrom(sfd, buf, BUF_SIZE, 0, (struct sockaddr *) &claddr, &len);

        if (numRead == -1)
            errExit("recvfrom");
        if (sendto(sfd, buf, numRead, 0, (struct sockaddr *) &claddr, len)
                != numRead)
            syslog(LOG_WARNING, "Error echoing response to %s (%s)",
                    inetAddressStr((struct sockaddr *) &claddr, len,
                                   addrStr, IS_ADDR_STR_LEN),
                    strerror(errno));
    }
}
```

sockets/id_echo_sv.c

Для проверки работы сервера мы используем программу из листинга 56.3. В ней тоже применяется библиотека для работы с сокетами интернет-домена, разработанная в разделе 55.12. В качестве первого аргумента командной строки клиентская программа принимает имя сетевого узла, на котором находится сервер. Клиент входит в цикл, где отправляет серверу каждый из оставшихся аргументов в виде отдельных датаграмм, а затем считывает и выводит датаграммы, полученные от сервера в ответ.

Листинг 56.3. Клиент для UDP-службы echo

sockets/id_echo_cl.c

```
#include "id_echo.h"

int
main(int argc, char *argv[])
{
    int sfd, j;
    size_t len;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host msg...\n", argv[0]);

    /* Формируем адрес сервера на основе первого аргумента командной строки */
    sfd = inetConnect(argv[1], SERVICE, SOCK_DGRAM);
    if (sfd == -1)
        fatal("Could not connect to server socket");

    /* Посылаем серверу остальные аргументы в виде отдельных датаграмм */
    for (j = 2; j < argc; j++) {
        len = strlen(argv[j]);
        if (write(sfd, argv[j], len) != len)
            fatal("partial/failed write");

        numRead = read(sfd, buf, BUF_SIZE);
        if (numRead == -1)
            errExit("read");
        printf("[%ld bytes] %.*s\n", (long) numRead, (int) numRead, buf);
    }
    exit(EXIT_SUCCESS);
}
```

sockets/id_echo_cl.c

Ниже показан пример того, что мы увидим при запуске сервера и двух экземпляров клиента:

```
$ su                                    Для привязки к зарезервированному
                                        порту нужны привилегии
Password:
# ./id_echo_sv                          Сервер переходит в фоновый режим
# exit                                  Отказываемся от прав администратора
$ ./id_echo_cl localhost hello world    Этот клиент отправляет две датаграммы
[5 bytes] hello                         Клиент выводит ответ, полученный от сервера
[5 bytes] world
$ ./id_echo_cl localhost goodbye        Этот клиент шлет одну датаграмму
[7 bytes] goodbye
```

56.3. Параллельный TCP-сервер echo

TCP-служба `echo` тоже работает на порте под номером 7. Сервер принимает соединение и входит в бесконечный цикл, считывая все переданные ему данные и отправляя их обратно клиенту с помощью того же сокета. Сервер продолжает выполнять чтение, пока не обнаружит конец файла; после этого он закрывает свой сокет (чтобы клиент тоже получил символ конца файла, если все еще продолжает читать данные из своего сокета).

Клиент может отправить серверу данные неограниченного объема (следовательно, обслуживание клиента может занять неопределенное время), так что в подобном случае подходит архитектура параллельного сервера, которая позволяет работать с несколькими клиентами одновременно. Реализация серверного приложения показана в листинге 56.4 (реализация клиента для этой службы будет представлена в разделе 57.2). Стоит отметить следующие моменты.

- Чтобы стать демоном, сервер использует функцию `becomeDaemon()` из раздела 37.2.
- С целью сделать программу более компактной мы применяем библиотеку для работы с сокетами интернет-домена, разработанную в листинге 55.9.
- Поскольку сервер создает потомков для каждого клиентского соединения, мы должны следить за уничтожением зомби-процессов. Для этого используется обработчик `SIGCHLD`.
- В основе кода сервера лежит цикл `for`, который принимает клиентские соединения и выполняет вызов `fork()` в целях создать новый дочерний процесс, обслуживающий клиента с помощью функции `handleRequest()`. Тем временем родитель продолжает работу цикла `for`, чтобы принять следующее клиентское соединение.

  В реальном приложении мы бы, вероятно, предусмотрели ограничение максимального количества дочерних процессов, которые может создать наш сервер; это помогло бы защититься от удаленной атаки, заключающейся в слишком интенсивном использовании службы и создании количества потомков, приводящего к зависанию системы. Данное ограничение можно ввести путем подсчета имеющихся дочерних процессов (счетчик инкрементируется после каждого успешного вызова fork() и декрементируется при уничтожении потомка с помощью обработчика SIGCHLD). В случае достижения ограничения мы могли бы временно приостановить прием соединений (или, как вариант, принимать соединения и тут же их закрывать).

- После каждого вызова `fork()` в дочернем процессе дублируется файловый дескриптор для слушающего и подключающегося сокетов (см. подраздел 24.2.1). То есть взаимодействовать с клиентским сокетом может как родитель, так и потомок. Однако заниматься этим нужно только потомку, так что сразу же после выполнения `fork()` родитель закрывает свой дескриптор подключенного сокета. (Если не совершить дан-

ное действие, то сокет так и будет оставаться открытым; кроме того, рано или поздно родитель исчерпает допустимое количество открытых файловых дескрипторов.) Так как потомок не принимает новые соединения, он закрывает свою копию файлового дескриптора для слушающего сокета.

- Обслужив клиента, дочерний процесс завершает работу.

Листинг 56.4. Параллельный сервер, реализующий TCP-службу echo

sockets/is_echo_sv.c

```
#include <signal.h>
#include <syslog.h>
#include <sys/wait.h>
#include "become_daemon.h"
#include "inet_sockets.h"    /* Объявление функций сокета вида inet*() */
#include "tlpi_hdr.h"

#define SERVICE "echo"       /* Имя TCP-службы */
#define BUF_SIZE 4096

static void                  /* Обработчик SIGCHLD, уничтожающий дочерние процессы */
grimReaper(int sig)
{
    int savedErrno;          /* Сохраняем значение 'errno' на случай,
                                если оно здесь изменится */
    savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}

/* Обрабатываем клиентский запрос: передаем сокету копию полученного от него ввода */

static void
handleRequest(int cfd)
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
        if (write(cfd, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

int
main(int argc, char *argv[])
{
    int lfd, cfd;            /* Слушающий и подключенный сокеты */
    struct sigaction sa;
```

```
    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = grimReaper;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        syslog(LOG_ERR, "Error from sigaction(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    lfd = inetListen(SERVICE, 10, NULL);
    if (lfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)",
            strerror(errno));
        exit(EXIT_FAILURE);
    }

    for (;;) {
        cfd = accept(lfd, NULL, NULL);    /* Ожидаем соединения */
        if (cfd == -1) {
            syslog(LOG_ERR, "Failure in accept(): %s", strerror(errno));
            exit(EXIT_FAILURE);
        }

        /* Обрабатываем каждый клиентский запрос в новом дочернем процессе */
        switch (fork()) {
        case -1:
            syslog(LOG_ERR, "Can't create child (%s)", strerror(errno));
            close(cfd);           /* Отклоняем запрос данного клиента */
            break;                /* Возможно, это временная проблема;
                                     пробуем следующий запрос */
        case 0:                   /* Потомок */
            close(lfd);           /* Ненужная копия слушающего сокета */
            handleRequest(cfd);
            _exit(EXIT_SUCCESS);

        default:              /* Родитель */
            close(cfd);       /* Ненужная копия подключенного сокета */
            break;            /* Повторяем цикл, чтобы принять следующее соединение */
        }
    }
}
```

sockets/is_echo_sv.c

56.4. Другие разновидности архитектуры параллельного сервера

Традиционная модель параллелизма, описанная в предыдущем разделе, подходит для многих приложений, которым нужно одновременно обрабатывать несколько клиентов, подключенных по TCP. Однако в высоконагруженных серверах (например, в веб-серверах, обрабатывающих тысячи запросов в минуту) создание нового дочернего процесса (или даже потока) для каждого клиента является довольно затратным (см. раздел 28.3). В этом случае требуются альтернативные подходы, и часть из них мы кратко рассмотрим ниже.

Серверы с пулом готовых процессов или потоков

Серверы с пулом готовых процессов или потоков подробно описываются в главе 30 книги [Stevens et al., 2004]. Их ключевая идея состоит в следующем.

- Вместо создания нового потомка (или потока) для каждого клиента сервер заранее, во время запуска (то есть до того, как начать прием клиентских запросов), создает определенное количество дочерних процессов (или потоков). Эти потомки формируют так называемый *серверный пул* (от англ. pool — «резервуар»).
- Каждый потомок в данном пуле обслуживает по одному клиенту за раз. Когда обслуживание окончено, потомок не завершается, а переходит к следующему клиенту и повторяет процедуру.

Применение вышеописанной методики требует тщательного планирования в рамках серверного приложения. Пул должен быть достаточно большим для обеспечения адекватной реакции на клиентские запросы. Это значит следующее: родительскому процессу сервера необходимо следить за количеством свободных потомков и увеличивать размер пула в периоды пиковых нагрузок, чтобы клиенты всегда обслуживались без задержек. Когда нагрузка спадает, размер серверного пула следует уменьшать, так как избыток процессов ухудшает общую производительность системы.

Кроме того, потомки в пуле должны соблюдать некий протокол, позволяющий им полностью брать на себя обслуживание отдельных клиентских соединений. В большинстве UNIX-систем (включая Linux) для этого достаточно сделать так, чтобы потомок блокировался на время вызова `accept()` для слушающего дескриптора. Иными словами, родительский процесс сервера создает слушающий сокет до создания любого потомка; притом в результате вызова `fork()` каждый из участников серверного пула наследует файловый дескриптор сокета. Когда появится новое клиентское соединение, только один из потомков завершит выполнение операции `accept()`. Но, поскольку в некоторых старых реализациях системный вызов `accept()` не является атомарным, к нему, возможно, придется применить механизм взаимного исключения (например, файловую блокировку). Это послужит гарантией того, что данная операция будет выполняться последовательно каждым дочерним процессом ([Stevens et al., 2004]).

> Выполнение всеми участниками серверного пула вызова accept() является не единственным вариантом. Если пул состоит из отдельных процессов, то родитель может сам выполнить accept() и затем передать файловый дескриптор с новым соединением одному из свободных дочерних процессов (методика, которая для этого применяется, кратко описана в подразделе 57.13.3). При нахождении в пуле потоков главная программа может выполнить accept() и затем проинформировать одного из свободных участников пула о том, что для заданного дескриптора доступен новый клиент.

Обслуживание нескольких клиентов с помощью одного процесса

В некоторых случаях для обслуживания нескольких клиентов может хватить и одного серверного процесса. Чтобы реализовать такой подход, необходимо воспользоваться одной из моделей ввода/вывода (epoll, мультиплексированный или сигнальный ввод/вывод), позволяющей одному процессу следить за событиями сразу нескольких файловых дескрипторов.

При использовании однопроцессной архитектуры сервер должен взять на себя роль планировщика. В многопроцессной модели данная роль обычно отводится ядру, которое само распределяет между потомками (и, следовательно, между клиентами) доступ к ресурсам сервера. Но когда процесс один, ему приходится следить за тем, чтобы никто из клиентов не смог получить эксклюзивный доступ к серверу и затруднить тем самым обработку остальных соединений. Мы разберем этот момент чуть более подробно в подразделе 59.4.6.

Использование серверных ферм

Среди других подходов, рассчитанных на высокие нагрузки, можно выделить применение нескольких серверных систем — *серверной фермы*.

Один из самых простых способов построения такой фермы (применяется в некоторых веб-серверах) заключается в *циклическом распределении нагрузки с помощью DNS* — когда полномочный сервер доменных имен, отвечающий за определенную зону, привязывает одно и то же доменное имя к нескольким IP-адресам (то есть несколько серверов имеют одно и то же имя). Последовательные запросы разрешения этого доменного имени обрабатываются в соответствии с циклическим алгоритмом, благодаря чему IP-адреса возвращаются в другом порядке. Дальнейшие подробности о циклическом распределении нагрузки путем DNS можно найти в книге [Albitz & Liu, 2006].

Циклическое распределение нагрузки является малозатратным и простым в настройке вариантом. Но у этого способа есть некоторые проблемы. Одной из них является кэширование, выполняемое удаленными DNS-серверами, из-за чего повторные запросы клиентов, находящихся на определенных компьютерах, не балансируются и всегда обрабатываются одним и тем же сервером. Кроме того, циклический алгоритм не предусматривает механизма обеспечения качественной балансировки (разные клиенты могут оказывать разную нагрузку на сервер) или высокой доступности (представьте, что один из серверов перестает работать или в его серверном приложении происходит сбой). Существует еще одна потенциальная проблема, характерная для многих архитектур, основанных на применении нескольких серверов — *привязка к серверу* (англ. server affinity). Речь идет о ситуации, когда последовательные запросы, выполняемые одним клиентом, должны быть направлены к одному и тому же серверу; это делается для того, чтобы сохранить актуальность информации о состоянии клиента, которая хранится на сервере.

Более гибким, но в то же время сложным решением является использование *балансировщика нагрузки*. Это подразумевает наличие балансирующего сервера, направляющего входящие клиентские запросы к одному из участников серверной фермы (для обеспечения высокой доступности может потребоваться запасной сервер, начинающий работать в случае сбоя в основном балансировщике). Так можно устранить проблемы, связанные с кэшированием в службе DNS, поскольку с точки зрения клиентов серверная ферма имеет единый IP-адрес (тот, который принадлежит балансировщику нагрузки). Балансировщик применяет специальные алгоритмы для измерения или оценки серверной нагрузки (возможно, исходя из показателей, предоставляемых участниками серверной фермы) и грамотно распределяет запросы между серверами. Он также автоматически обнаруживает сбои, происходящие с участниками фермы (и добавляет новые серверы, если нужно). Наконец, балансировщик также может поддерживать привязку к серверу. Подробности об этой методике можно найти в книге [Коррагари, 2002].

56.5. Демон inetd

Если взглянуть на содержимое файла `/etc/services`, можно увидеть буквально сотни разных служб. Это говорит о том, что система теоретически способна выполнять большое количество серверных процессов. Однако большинство данных серверов обычно простаивает в ожидании редких запросов на подключение или датаграмм. Тем не менее все они занимают место в таблице процессов ядра и некоторую память и пространство подкачки, вследствие чего создается нагрузка на систему.

Демон `inetd` разработан для того, чтобы устранить необходимость выполнять большое количество малоиспользуемых серверов. Его два основных преимущества заключаются в следующем.

- Вместо запуска отдельного демона для каждой службы `inetd` отслеживает заданный набор портов сокетов и при необходимости запускает соответствующие процессы. Таким образом количество выполняющихся процессов в системе снижается.
- Написание серверных программ, запускаемых с помощью `inetd`, упрощается, поскольку этот демон выполняет определенные действия, которые требуются при запуске любых сетевых серверов.

Поскольку демон `inetd` следит за диапазоном служб и при необходимости запускает другие серверы, его иногда называют *интернет-суперсервером*.

> В дистрибутивах Linux предоставляется расширенная версия inetd под названием xinetd, которая, помимо всего прочего, обладает целым рядом улучшений, связанных с безопасностью. Информацию об xinetd можно найти на www.xinetd.org.

Принцип работы демона inetd

Обычно демон `inetd` запускается вместе с системой. Перейдя в фоновый режим (см. раздел 37.2), он выполняет следующие шаги.

1. Для каждой службы, заданной в конфигурационном файле `/etc/inetd.conf`, демон создает сокет подходящего типа (потоковый или датаграммный) и привязывает его к указанному порту. Для каждого TCP-сокета дополнительно включается поддержка входящих соединений с помощью вызова `listen()`.
2. Используя системный вызов `select()` (см. подраздел 59.2.1), `inetd` отслеживает входящие датаграммы или соединения для всех сокетов, созданных в предыдущем шаге.
3. До появления доступной для чтения датаграммы в UDP-сокете или запроса о подключении в TCP-сокете вызов `select()` блокируется. В случае с TCP-соединением `inetd` сначала выполняет для соединения вызов `accept()` и только потом переходит к следующему шагу.
4. Чтобы запустить сервер, указанный для текущего сокета, `inetd` вызывает `fork()`, создавая новый процесс, запускающий серверную программу с помощью функции `exec()`. Но, прежде чем выполнять `exec()`, дочерний процесс проделывает следующие действия.
   - Закрывает все файловые дескрипторы, унаследованные от родителя, за исключением относящегося к сокету, для которого доступна UDP-датаграмма или установлено TCP-соединение.
   - Использует методики, описанные в разделе 5.5, чтобы создать три копии файлового дескриптора сокета с номерами 0, 1 и 2, после чего закрывает оригинал (так как он больше не нужен). В результате запущенный сервер может взаимодействовать через сокет, задействуя три стандартных файловых дескриптора.
   - При необходимости устанавливает запущенному серверу идентификаторы пользователя и группы, соответствующие значениям, указанным в файле `/etc/inetd.conf`.
5. Если в шаге 3 было принято TCP-соединение, `inetd` закрывает подключенный сокет (так как он нужен только на серверной стороне).
6. Сервер `inetd` возвращается к шагу 2.

Файл /etc/inetd.conf

Работа демона `inetd` управляется с помощью конфигурационного файла; обычно им является `/etc/inetd.conf`. Каждая строчка в этом файле описывает одну службу, запуском которой занимается `inetd`. В листинге 56.5 показан пример записей в файле `/etc/inetd.conf`, поставляющемся вместе с одним из дистрибутивов Linux.

Листинг 56.5. Пример записей в файле /etc/inetd.conf

```
# echo     stream     tcp      nowait     root     internal
# echo     dgram      udp      wait       root     internal
ftp        stream     tcp      nowait     root     /usr/sbin/tcpd     in.ftpd
telnet     stream     tcp      nowait     root     /usr/sbin/tcpd     in.telnetd
login      stream     tcp      nowait     root     /usr/sbin/tcpd     in.rlogind
```

Первые две строки в листинге 56.5 закомментированы с помощью символа # в начале; мы приводим их здесь, так как они связаны со службой `echo`, которую рассмотрим чуть ниже.

Каждая запись в файле `/etc/inetd.conf` состоит из следующих полей, разделенных пробельными символами.

- *Имя службы.* В этом поле содержится имя службы, в сочетании с *протоколом* применяемое для поиска по файлу `/etc/services` с целью определить номер порта, который должен отслеживаться демоном `inetd`.
- *Тип сокета.* В данном поле определяется тип сокета, используемый службой, например `stream` или `dgram`.
- *Протокол.* В этом поле содержится протокол, который будет применен для данного сокета. Здесь можно указывать любые интернет-протоколы, перечисленные в файле `/etc/protocols` (и задокументированные на странице `protocols(5)` руководства), но в подавляющем большинстве случаев используется одно из двух значений: `tcp` (для TCP) или `udp` (для UDP).
- *Флаги.* Данное поле может содержать либо `wait`, либо `nowait`. Оно указывает, будет ли сокет службы управляться (временно) самим сервером, запущенным с помощью `inetd`. Если ответ положительный, то значение должно быть равно `wait`. Это заставит `inetd` удалить свой файловый дескриптор для данного сокета из набора, отслеживаемого путем вызова `select()`. Демон `inetd` не будет иметь доступа к сокету, пока сервер не завершит свою работу (это определяется благодаря обработчику для сигнала `SIGCHLD`). Мы еще вернемся к данному полю чуть ниже.
- *Имя пользователя.* Это поле содержит имя пользователя из файла `/etc/passwd`; за ним через точку (`.`) может быть указано название группы, заданное в файле `/etc/group`. Так определяются групповые и пользовательские идентификаторы, которые будут применены к запускаемому серверу (демон `inetd` выполняется с помощью действующего идентификатора администратора, поэтому его потомки тоже наделены повышенными привилегиями; так что, если им нужно изменить учетные данные своих процессов, они должны задействовать вызовы `setuid()` и `setgid()`).
- *Серверная программа.* В данном поле указывается путь к серверной программе, которую нужно запустить.
- *Аргументы серверной программы.* В этом поле содержится один или несколько аргументов, разделенных пробельными символами, которые будут указаны при запуске серверной программы. Первый из них соответствует `argv[0]` и обычно обозначает имя исполняемого файла. Следующий аргумент соответствует `argv[1]` и т. д.

В строчках для служб ftp, telnet и login, приведенных в листинге 56.5, можно видеть, что серверные программы и их аргументы указаны не совсем так, как это описано выше. Во всех трех случаях изначально вызывается одна и та же программа, tcpd(8) (обертка для TCP-демона). Прежде чем запустить соответствующий сервер, она выполняет учет данных и проверку прав доступа. Исполняемый файл определяется на основе значения, заданного в качестве первого аргумента (который доступен для tcpd в виде элемента argv[0]). Дальнейшие подробности о tcpd можно найти на странице tcpd(8) руководства и в книге [Mann & Mitchell, 2003].

Серверы на основе потоковых сокетов (TCP), запущенные с помощью `inetd`, обычно обрабатывают единственное соединение, после чего завершаются, давая `inetd` возможность наблюдать за дальнейшими подключениями. В поле с флагом для таких серверов должно быть указано значение `nowait` (если запускаемый сервер продолжает принимать соединения, то нужно указать `wait`; в этом случае `inetd` не станет отслеживать запросы и передаст серверу *слушающий* сокет в виде дескриптора под номером 0).

Для большинства UDP-серверов поле с флагом должно быть равно `wait`. UDP-сервер, запущенный с помощью `inetd`, обычно считывает и обрабатывает все доступные для сокета датаграммы, после чего завершается. (Как правило, это требует применения времени ожидания при чтении из сокета, чтобы сервер мог завершиться, если за отведенный отрезок не поступило ни одной новой датаграммы.) Значение `wait` не дает демону `inetd` выполнить для сокета вызов `select()`, пока сервер работает, так как это может привести к нежелательным последствиям, например к соперничеству демона `inetd` и UDP-сервера за право проверить наличие датаграммы (и если первый выиграет, то ему придется запустить еще один экземпляр сервера).

> Демон inetd и формат его конфигурационного файла не входят в стандарт SUSv3, поэтому набор допустимых значений для /etc/inetd.conf может варьироваться (обычно незначительно). Большинство версий inetd поддерживают как минимум тот синтаксис, который мы описали выше. Больше подробностей см. на странице inetd.conf(8) руководства.

Для лучшей эффективности `inetd` сам реализует несколько служб, чтобы не запускать отдельные серверы для выполнения ряда задач. Как пример можно привести службу `echo`, работающую по UDP и TCP. В таких случаях в качестве серверной программы в файле `/etc/inetd.conf` указывается значение `internal`, а соответствующие аргументы опускаются (в листинге 56.5 мы видели: строчки для службы `echo` закомментированы; для включения службы `echo` нужно убрать в начале этих строк символ `#`).

Чтобы демон `inetd` смог заново прочитать обновленную конфигурацию, при каждом изменении файла `/etc/inetd.conf` ему нужно отправлять сигнал `SIGHUP`:

```
# killall -HUP inetd
```

Пример: вызов TCP-службы echo с помощью inetd

Ранее мы отмечали, что `inetd` упрощает написание серверных программ, особенно работающих в параллельном режиме (обычно по TCP). Это достигается за счет выполнения следующих шагов во время запуска сервера.

1. С использованием вызовов `socket()`, `bind()` и `listen()` (для TCP-серверов) инициализируются все структуры, связанные с сокетом.
2. Выполняется вызов `accept()` для новых TCP-соединений.
3. Создается новый процесс, который обрабатывает входящие UDP-датаграммы или TCP-соединения. Этот процесс автоматически переходит в фоновый режим. Демон `inetd` берет на себя всю работу по созданию потомков с помощью вызова `fork()` и уничтожению завершившихся процессов (задействуя обработчик `SIGCHLD`).
4. Файловым дескрипторам с номерами 0, 1 и 2 присваиваются копии дескриптора для UDP- или TCP-сокета; все остальные файловые дескрипторы закрываются (так как не будут использоваться запущенным сервером).
5. Запускается серверная программа.

В описании вышеприведенных шагов мы исходим из того, что поле с флагом в файле `/etc/inetd.conf` содержит значение `nowait` для TCP-служб и `wait` для UDP-служб (так обычно и происходит).

Пример того, как inetd упрощает написание TCP-служб, приводится в листинге 56.6, где показан эквивалент сервера echo из листинга 56.4, который запускается с помощью inetd. Поскольку этот демон берет на себя все действия, перечисленные выше, нам остается лишь запрограммировать поведение дочерних процессов, обрабатывающих клиентские запросы, доступные через файловый дескриптор 0 (STDIN_FILENO).

Допустим, сервер находится в каталоге /bin. Тогда, чтобы демон inetd смог его запустить, мы должны создать в файле /etc/inetd.conf следующую запись:

```
echo stream tcp nowait root /bin/is_echo_inetd_sv is_echo_inetd_sv
```

Листинг 56.6. TCP-сервер echo, рассчитанный на запуск с помощью inetd

sockets/is_echo_inetd_sv.c

```
#include <syslog.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 4096

int
main(int argc, char *argv[])
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0) {
        if (write(STDOUT_FILENO, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/is_echo_inetd_sv.c

56.6. Резюме

Итерационный сервер обслуживает по одному клиенту за раз: сначала обрабатывает запрос(-ы) одного клиента и только потом переходит к следующему. Параллельные серверы выполняют одновременную обработку нескольких клиентских запросов. Традиционная архитектура параллельного сервера, которая подразумевает создание нового дочернего процесса (или потока) для каждого клиента, может оказаться не самым оптимальным решением при высокой нагрузке; в связи с этим мы выделили несколько альтернативных подходов для параллельной обработки большого количества запросов.

Демон inetd следит за несколькими сокетами и в ответ на входящие UDP-датаграммы или TCP-соединения запускает подходящие серверы. Использование демона позволяет снизить нагрузку на систему путем минимизации количества запущенных серверных

процессов, а также упрощает написание серверного кода, так как `inetd` берет на себя большую часть шагов, связанных с инициализацией.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 55.14.

56.7. Упражнения

56.1. Добавьте в программу из листинга 56.4 (`is_echo_sv.c`) код, ограничивающий максимальное количество дочерних процессов, выполняемых одновременно.

56.2. Иногда возникает необходимость в написании сервера, который можно запускать как напрямую, из командной строки, так и через `inetd`. Для различения этих двух случаев используются параметры командной строки. Измените программу из листинга 56.4 так, чтобы в случае обнаружения параметра `-i` она считала, что ее запустили с помощью `inetd`. При этом программа должна обслуживать только одного клиента, подключенного к ее сокету путем файлового дескриптора `STDIN_FILENO`, предоставленного демоном `inetd`. Если параметр `-i` не указан, то программа считает, что ее запустили из командной строки, и выполняется в традиционной манере (для внесения данного изменения требуется всего лишь несколько дополнительных строчек кода). Отредактируйте файл `/etc/inetd.conf`, чтобы эта программа запускалась в ответ на вызов службы `echo`.

57 Сокеты: углубленный материал

В этой главе рассмотрен ряд продвинутых тем, связанных с написанием программ на основе сокетов, включая следующие:

- обстоятельства, в которых работа с потоковыми сокетами приводит к частичному чтению или записи;
- использование функции `shutdown()` для закрытия одной из ветвей двунаправленного канала между сокетами;
- системные вызовы `recv()` и `send()`, предоставляющие специфические возможности, связанные с сокетами и недоступные для операций `read()` и `write()`;
- системный вызов `sendfile()`, в некоторых ситуациях позволяющий эффективно выводить данные в сокет;
- детали реализации протокола TCP, призванные развенчать ряд распространенных заблуждений, способных привести к ошибкам при написании программ на основе потоковых сокетов;
- использование команд `netstat` и `tcpdump` для мониторинга и отладки приложений, задействующих сокеты;
- применение системных вызовов `getsockopt()` и `setsockopt()` для извлечения и изменения параметров, влияющих на работу сокета.

Мы также рассмотрим ряд других, менее важных тем, а в конце сделаем краткий обзор отдельных продвинутых возможностей, которыми обладают сокеты.

57.1. Частичное чтение и запись в контексте потоковых сокетов

При первом знакомстве с системными вызовами `read()` и `write()` в главе 4 мы отметили, что в ряде ситуаций они могут передать не все запрошенные данные. Такая неполная передача может возникнуть при выполнении операций ввода/вывода для потоковых сокетов. В данном разделе мы рассмотрим причины, которые могут за этим стоять, а также продемонстрируем две функции, автоматически справляющиеся с частичным чтением и записью.

Частичное чтение может возникнуть в ситуации, когда количество байтов, запрошенных вызовом `read()`, превышает то, которое доступно в сокете. В таком случае операция `read()` просто возвращает все доступные байты (такое же поведение мы видели в разделе 44.10, когда рассматривали каналы и очереди FIFO).

Частичная запись может возникнуть, если в буфере не хватает места, чтобы вместить все запрашиваемые байты. При этом должно выполняться одно из следующих условий:

- после передачи части запрашиваемых данных вызов `write()` был прерван обработчиком сигнала (см. раздел 21.5);
- сокет работал в неблокирующем режиме (`O_NONBLOCK`), позволяя передавать только некоторые из запрашиваемых байтов;

- во время передачи данных возникла *асинхронная ошибка* — то есть ошибка, не синхронизированная с работой программного интерфейса сокета. Это может случиться, например, при возникновении проблем с TCP-соединением (возможно, в результате сбоя в удаленном приложении).

Во всех этих случаях операция `write()` завершается успешно и возвращает количество байтов, переданных в исходящий буфер (при условии, что там было место хотя бы для одного байта).

При возникновении неполной операции ввода/вывода (например, если `read()` возвращает меньше байтов, чем было запрошено, или если заблокированный вызов `write()` прерывается обработчиком сигнала, не успев передать все данные) иногда имеет смысл сделать повторный системный вызов, чтобы завершить передачу. В листинге 57.1 представлены две функции, умеющие это делать: `readn()` и `writen()` (идея их создания была заимствована из книги [Stevens et al., 2004], в которой описаны аналогичные функции с такими же именами).

```
#include "rdwrn.h"

ssize_t readn(int fd, void *buffer, size_t count);
```

Возвращает количество прочитанных байтов, `0`, если обнаружен конец файла, или `-1` при ошибке

```
ssize_t writen(int fd, void *buffer, size_t count);
```

Возвращает количество записанных байтов или `-1` при ошибке

Функции `readn()` и `writen()` принимают те же аргументы, что и `read()` и `write()`. Однако они используют цикл для перезапуска системных вызовов, гарантируя передачу запрашиваемого количества данных (если только при чтении не произойдет ошибка или не будет обнаружен конец файла).

Листинг 57.1. Реализация функций readn() и writen()

sockets/rdwrn.c

```
#include <unistd.h>
#include <errno.h>
#include "rdwrn.h"        /* Объявляет readn() и writen() */

ssize_t
readn(int fd, void *buffer, size_t n)
{
    ssize_t numRead;    /* Количество байтов, полученных предыдущей операцией read() */
    size_t totRead;     /* Общее количество байтов, прочитанных на данный момент */
    char *buf;

    buf = buffer;       /* Избегаем арифметики с указателями для "void *" */
    for (totRead = 0; totRead < n; ) {
        numRead = read(fd, buf, n - totRead);

        if (numRead == 0)     /* Конец файла */
            return totRead;   /* Может быть равно 0, если это первый вызов read() */
        if (numRead == -1) {
            if (errno == EINTR)
                continue;     /* Прервано --> перезапускаем read() */
            else
```

```
                return -1;    /* Какая-то другая ошибка */
        }
        totRead += numRead;
        buf += numRead;
    }
    return totRead;   /* Должно равняться 'n' байтам, если сюда добрались */
}

ssize_t
writen(int fd, const void *buffer, size_t n)
{
    ssize_t numWritten;    /* Количество байтов, записанных предыдущей
                              операцией write() */
    size_t totWritten;     /* Общее количество байтов, записанных на данный момент */
    const char *buf;
    buf = buffer;          /* Избегаем арифметики с указателями для "void *" */
    for (totWritten = 0; totWritten < n; ) {
        numWritten = write(fd, buf, n - totWritten);

        if (numWritten <= 0) {
            if (numWritten == -1 && errno == EINTR)
                continue;     /* Прервано --> перезапускаем write() */
            else
                return -1;    /* Какая-то другая ошибка */
        }
        totWritten += numWritten;
        buf += numWritten;
    }
    return totWritten;       /* Должно равняться 'n' байтам, если сюда добрались */
}
```

sockets/rdwrn.c

57.2. Системный вызов shutdown()

Вызов `close()` закрывает обе ветви двунаправленного канала взаимодействия, основанного на сокетах. Но иногда нужно закрыть только одну часть соединения, чтобы данные через сокет могли передаваться в каком-то определенном направлении. Для этого предусмотрен системный вызов `shutdown()`.

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Данный системный вызов закрывает одно или оба направления сокета `sockfd` в зависимости от значения аргумента `how`, который может иметь одно из следующих значений.

- `SHUT_RD` — запрещает чтение из сокета. Последующие операции `read()` возвращают конец файла (0). При этом запись данных может продолжаться. После применения флага `SHUT_RD` к потоковому сокету домена UNIX приложение на другом конце соединения, пытающееся передать новые данные, получит сигнал `SIGPIPE` и ошибку `EPIPE`. Как отмечается в подразделе 57.6.6, использование данного флага для TCP-сокетов не имеет смысла.

- `SHUT_WR` — запрещает запись в сокет. Когда удаленное приложение прочитает все оставшиеся данные, оно обнаружит конец файла. Последующая запись в локальный сокет приведет к сигналу `SIGPIPE` и ошибке `EPIPE`. Данные, записываемые на другом конце соединения, по-прежнему можно будет прочитать. Иными словами, эта операция позволяет сообщить удаленному приложению о завершении файла, но притом иметь возможность продолжать чтение данных, которое посылает приложение. Флаг `SHUT_WR` используется в таких программах, как `ssh` и `rsh` (см. раздел 18.5 книги [Stevens, 1994]). Это наиболее распространенный способ применения вызова `shutdown()`, а соответствующий сокет иногда называют *полузакрытым*.
- `SHUT_RDWR` — полностью закрывает соединение: как для чтения, так и для записи. То же, что последовательное выполнение операций `SHUT_RD` и `SHUT_WR`.

Помимо семантики аргумента `how` вызов `shutdown()` имеет еще одно существенное отличие от `close()`: он закрывает канал(-ы) сокета вне зависимости от того, ссылаются ли на этот сокет какие-нибудь другие файловые дескрипторы. (Иными словами, `shutdown()` оперирует описанием открытого файла, а не его дескриптором; см. рис. 5.1.) Представьте, к примеру, что `sockfd` указывает на подключенный потоковый сокет. Если сделать следующие вызовы, то соединение останется открытым и по-прежнему можно будет выполнять для него ввод/вывод, используя файловый дескриптор `fd2`:

```
fd2 = dup(sockfd);
close(sockfd);
```

Но если выполнить следующую последовательность вызовов, то оба канала соединения будут закрыты, а дескриптор `fd2` станет непригодным для ввода/вывода:

```
fd2 = dup(sockfd);
shutdown(sockfd, SHUT_RDWR);
```

Нечто похожее происходит, когда во время вызова `fork()` файловый дескриптор сокета дублируется. Если после этого один из процессов выполнит для своей копии дескриптора операцию `SHUT_RDWR`, то другой процесс больше не сможет пользоваться своей копией для ввода/вывода.

Обратите внимание: `shutdown()` не закрывает файловый дескриптор, даже если аргумент `how` равен `SHUT_RDWR`. Чтобы добиться закрытия, нужно дополнительно вызвать `close()`.

Пример программы

Использование вызова `shutdown()` с флагом `SHUT_WR` демонстрируется в листинге 57.2. Эта программа представляет собой TCP-клиент для службы `echo` (TCP-сервер для данной службы был показан в разделе 56.3). Чтобы сделать реализацию более компактной, мы задействуем функции из библиотеки для работы с сокетами интернет-домена из раздела 55.12.

> В некоторых дистрибутивах Linux служба echo не включена по умолчанию. Следовательно, прежде чем запускать программу из листинга 57.2, ее следует включить. Обычно эта служба реализована внутри демона inetd(8) (см. раздел 56.5), и для ее активации нужно удалить комментарии из двух строчек в файле /etc/inetd.conf, относящихся к протоколам UDP и TCP (см. листинг 56.5), а затем послать сигнал SIGHUP демону inetd.
>
> Многие дистрибутивы предоставляют вместо inetd(8) более современный демон xinetd(8). Чтобы узнать, как выполнить для него аналогичные изменения, обратитесь к его документации.

В качестве единственного аргумента командной строки программа принимает имя сетевого узла, на котором запущена служба `echo`. Клиент выполняет вызов `fork()`, разделяясь на родительский и дочерний процессы.

Родитель записывает содержимое стандартного ввода в сокет, чтобы сервер `echo` мог его прочитать. При обнаружении конца файла в этом вводе родитель использует вызов `shutdown()` для закрытия записывающего канала своего сокета. В результате сервер `echo` тоже сталкивается с завершением файла и закрывает свой сокет.

Дочерний процесс клиента считывает ответ сервера `echo` из сокета и направляет его в стандартный вывод. Обнаружив конец файла, он завершает работу.

Ниже показан пример вывода, который можно увидеть при выполнении этой программы:

```
$ cat > tell-tale-heart.txt                    Создаем проверочный файл
It is impossible to say how the idea entered my brain;
but once conceived, it haunted me day and night.
Нажимаем Ctrl+D
$ ./is_echo_cl tekapo < tell-tale-heart.txt
It is impossible to say how the idea entered my brain;
but once conceived, it haunted me day and night.
```

Листинг 57.2. Клиент службы echo

sockets/is_echo_cl.c

```
#include "inet_sockets.h"
#include "tlpi_hdr.h"

#define BUF_SIZE 100

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host\n", argv[0]);

    sfd = inetConnect(argv[1], "echo", SOCK_STREAM);
    if (sfd == -1)
        errExit("inetConnect");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:
    /* Потомок считывает ответ сервера и направляет его в стандартный вывод */
        for (;;) {
            numRead = read(sfd, buf, BUF_SIZE);
            if (numRead <= 0)
            /* Выходим при завершении файла или ошибке */
                break;
            printf("%.*s", (int) numRead, buf);
        }
        exit(EXIT_SUCCESS);

    default:           /* Родитель записывает в сокет стандартный ввод */
        for (;;) {
            numRead = read(STDIN_FILENO, buf, BUF_SIZE);
            if (numRead <= 0)
```

```
            /* Выходим из цикла при завершении файла или ошибке */
                break;
            if (write(sfd, buf, numRead) != numRead)
                fatal("write() failed");
        }

        /* Закрываем записывающий канал, чтобы сервер увидел конец файла */
        if (shutdown(sfd, SHUT_WR) == -1)
            errExit("shutdown");
        exit(EXIT_SUCCESS);
    }
}
```

sockets/is_echo_cl.c

57.3. Специальные системные вызовы для работы с сокетами: recv() и send()

Системные вызовы `recv()` и `send()` выполняют операции ввода/вывода с подключенными сокетами. Они предоставляют специальные возможности, недоступные в традиционных вызовах `read()` и `write()`.

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buffer, size_t length, int flags);
```

Возвращает количество полученных байтов, `0`, если обнаружился конец файла, или `-1` при ошибке

```
ssize_t send(int sockfd, const void *buffer, size_t length, int flags);
```

Возвращает количество отправленных байтов или `-1` при ошибке

Возвращаемые значения и три первых аргумента вызовов `recv()` и `send()` такие же, как у `read()` и `write()`. Последний аргумент, `flags`, представляет собой битовую маску, влияющую на выполнение ввода/вывода. Вызов `recv()` поддерживает следующие флаги, к которым можно применять побитовое ИЛИ.

- `MSG_DONTWAIT` — выполняет неблокирующее чтение. При отсутствии доступных данных немедленно возвращает ошибку `EAGAIN`. Того же можно добиться с помощью вызова `fcntl()`, если перевести сокет в неблокирующий режим (`O_NONBLOCK`); разница лишь в том, что флаг `MSG_DONTWAIT` позволяет делать неблокирующими отдельные вызовы.
- `MSG_OOB` — принимает из сокета внеканальные данные. Эта возможность кратко описывается в подразделе 57.13.1.
- `MSG_PEEK` — получает из буфера сокета копию запрашиваемых байтов, не удаляя их оттуда. Позже данные можно опять прочитать, используя вызов `recv()` или `read()`.
- `MSG_WAITALL` — обычно вызов `recv()` возвращает меньше байтов, чем было запрошено (`length`) и чем доступно в сокете. При указании флага `MSG_WAITALL` вызов заблокируется, пока не будут получены `length` байтов. Но даже в этом случае можно получить меньше байтов, чем указано в запросе, если:
  - был перехвачен сигнал;
  - удаленное приложение на другом конце сокета разорвало соединение;
  - были обнаружены внеканальные данные (см. подраздел 57.13.1);

- сообщение, полученное из датаграммного сокета, занимает меньше `length` байтов;
- в сокете произошла ошибка.

Флаг `MSG_WAITALL` может заменить собой функцию `readn()`, представленную в листинге 57.1, с той лишь разницей, что не перезагружается в случае прерывания обработчиком сигнала.

Все флаги, перечисленные выше, входят в стандарт SUSv3. Исключение составляет флаг `MSG_DONTWAIT`, который, тем не менее, доступен в ряде других реализаций UNIX. Флаг `MSG_WAITALL` появился в программном интерфейсе сокетов позже остальных, поэтому в отдельных старых системах не представлен.

Вызов `send()` поддерживает следующие флаги, к которым можно применять побитовое ИЛИ.

- `MSG_DONTWAIT` — выполняет неблокирующую запись. Если данные нельзя передать немедленно (поскольку исходящий буфер сокета заполнен), то операция не блокируется, а сразу же возвращает ошибку `EAGAIN`. Как и в случае с `recv()`, того же результата можно достичь, установив сокету флаг `O_NONBLOCK`.
- `MSG_MORE` (начиная с Linux 2.4.4) — этот флаг используется в TCP-соединениях и является эквивалентом параметра `TCP_CORK`, передаваемого сокету (см. раздел 57.4); разница лишь в том, что он обеспечивает закупоривание данных для каждого отдельного вызова. Начиная с версии ядра 2.6 `MSG_MORE` можно применять и для датаграммных сокетов, но в таком случае он будет иметь другое значение. Если указать его в последовательных вызовах `send()` или `sendto()`, то данные будут упакованы в единую датаграмму; она будет передана только при выполнении следующего вызова, в котором этот флаг не указан. (Linux также поддерживает аналогичный параметр для сокета — `UDP_CORK`; он позволяет группировать данные из вызовов `send()` или `sendto()` в единую датаграмму и отсылает их, только если его отключить.) Флаг `MSG_MORE` никак не влияет на работу сокетов домена UNIX.
- `MSG_NOSIGNAL` — делает так, что передача данных через подключенный сокет не генерирует сигнал `SIGPIPE`, если другой конец соединения закрыт. При этом вызов `send()` завершается ошибкой `EPIPE`. Такого же поведения можно добиться, проигнорировав сигнал `SIGPIPE`; разница лишь в том, что флаг `MSG_NOSIGNAL` влияет на отдельные вызовы.
- `MSG_OOB` — отправляет сокету внеканальные данные (см. подраздел 57.13.1).

Из всех флагов, перечисленных выше, только `MSG_OOB` входит в стандарт SUSv3. Флаг `MSG_DONTWAIT` встречается в ряде других систем, а `MSG_NOSIGNAL` и `MSG_MORE` поддерживаются только в Linux.

Дополнительные флаги, которые здесь не упоминаются, описаны на страницах `send(2)` и `recv(2)` руководства.

57.4. Системный вызов sendfile()

Таким приложениям, как файловые и веб-серверы, часто приходится передавать через (подключенный) сокет точную копию содержимого файла, хранящегося на диске. Один из способов, с помощью которого это можно сделать, выглядит следующим образом:

```
while ((n = read(diskfilefd, buf, BUZ_SIZE)) > 0)
    write(sockfd, buf, n);
```

Во многих случаях такой цикл является вполне приемлемым. Но если часто передавать через сокет большие файлы, то данный подход перестает быть эффективным. Для передачи файла следует задействовать два системных вызова (возможно, по нескольку

раз, поместив их в цикл): один — для копирования содержимого файла из буферного кэша ядра в пользовательское пространство, а другой — для обратного копирования, чтобы данные можно было передать через сокет. Этот сценарий показан на рис. 57.1, *а*. Такой двухэтапный процесс получается слишком расточительным, если приложение никак не обрабатывает данные перед отправкой. Для оптимизации процесса предусмотрен системный вызов `sendfile()`. При его использовании содержимое файла направляется непосредственно в сокет, минуя пространство пользователя (см. рис. 57.1, *б*). Эта методика называется *передачей с нулевым копированием*.

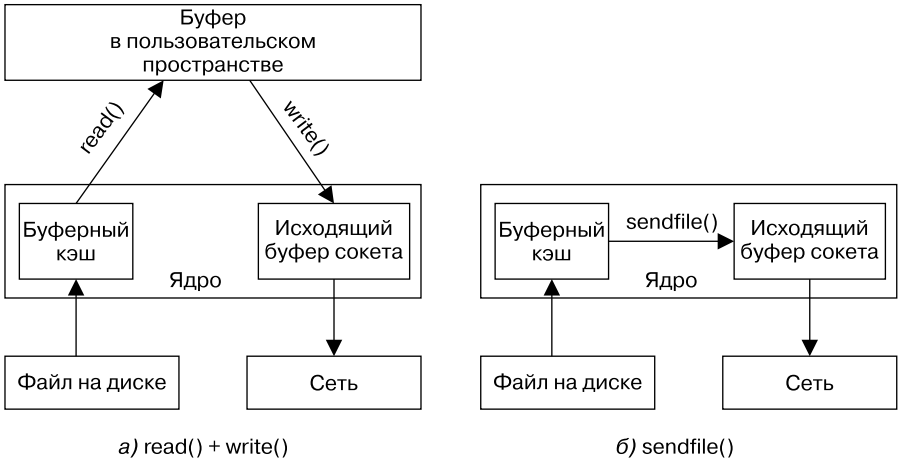


Рис. 57.1. Передача содержимого файла через сокет

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Возвращает количество переданных байтов или `-1` при ошибке

Системный вызов `sendfile()` передает данные из файла, на который ссылается дескриптор `in_fd`, дескриптору `out_fd`, ссылающемуся на сокет. Аргумент `in_fd` должен указывать на файл, к которому можно применить вызов `mmap()`; в реальных приложениях для этого чаще всего берется обычный файл. Такая манипуляция в определенной мере ограничивает применение вызова `sendfile()`. Можно использовать его для передачи данных из файла в сокет, но не наоборот, и нельзя с помощью этого вызова передать данные непосредственно из одного сокета в другой.

Если аргумент `offset` не равен `NULL`, то должен указывать на значение типа `off_t`, определяющее начальный сдвиг в файле `in_fd`, с которого необходимо начинать передачу данных. Этот аргумент применяется для возвращения результата. После завершения вызова он содержит сдвиг, начинающийся вслед за последним байтом, переданным из `in_fd`. В таком случае вызов `sendfile()` не изменяет файловый сдвиг.

Если аргумент `offset` равен `NULL`, то данные файла `in_fd` начинают передаваться с его текущего сдвига, а сам сдвиг обновляется с каждым переданным байтом.

Аргумент `count` обозначает количество байтов, которые нужно передать. Если конец файла обнаруживается раньше достижения этого значения, процесс передачи данных завершается. В случае успеха `sendfile()` возвращает количество байтов, дошедших по назначению.

Вызов sendfile() не предусмотрен стандартом SUSv3. Его разновидности доступны в некоторых UNIX-системах, но их аргументы обычно отличаются от версии, реализованной в Linux.

Параметр сокета TCP_CORK

Для дальнейшего повышения эффективности TCP-приложений, использующих вызов sendfile(), иногда имеет смысл применить параметр сокета TCP_CORK (доступный только в Linux). В качестве примера рассмотрим веб-сервер, который возвращает страницу в ответ на запрос веб-браузера. Ответ сервера состоит из двух частей: HTTP-заголовков (возможно, полученных с помощью вызова write()) и тела страницы (скажем, сформированного с применением вызова sendfile()). Обычно в таком случае передается *два* TCP-сегмента: сначала отсылаются заголовки (небольшого объема), а затем следуют данные самой страницы. Это приводит к неэффективному расходу ресурсов сети. К тому же может потребоваться дополнительная работа обеих сторон TCP-соединения, так как во многих случаях HTTP-заголовки и данные страницы можно вместить в единственный TCP-сегмент. Параметр TCP_CORK призван оптимизировать описанную процедуру.

Если в потоковом сокете включить параметр TCP_CORK, то весь его вывод будет буферизироваться в одном TCP-сегменте, пока не произойдет одно из следующих событий: достижение максимального размера сегмента, выключение параметра TCP_CORK, закрытие сокета или завершение интервала в 200 миллисекунд с момента записи первого «закупоренного» байта. (Использование времени ожидания гарантирует, что данные будут переданы, даже если приложение забудет отключить TCP_CORK.)

Для включения и отключения параметра TCP_CORK служит системный вызов setsockopt() (см. раздел 57.9). Применение этого режима продемонстрировано на примере следующего кода, в котором реализован наш гипотетический HTTP-сервер (мы намеренно опустили проверку ошибок):

```
int optval;

/* Включаем TCP_CORK для 'sockfd' – последующий TCP-вывод закупоривается,
   пока этот параметр не будет выключен. */
optval = 1;
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, sizeof(optval));

write(sockfd, ...);                /* Записываем HTTP-заголовки */
sendfile(sockfd, ...);             /* Отправляем содержимое страницы */

/* Отключаем TCP_CORK для 'sockfd' – закупоренный вывод начинает
   передаваться в виде единого TCP-сегмента. */
optval = 0
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, sizeof(optval));
```

Мы могли бы избежать потенциальной передачи двух сегментов, создав в нашем приложении единый буфер данных, который можно было бы передавать с помощью одного вызова write(). (Как вариант, можно было бы воспользоваться вызовом writev(), чтобы объединить два разных буфера в одну групповую операцию.) Но если мы хотим выполнять передачу с нулевым копированием, которую обеспечивает вызов sendfile(), в сочетании с включением заголовка в первый сегмент передаваемого файла, то для этого нужно использовать параметр TCP_CORK.

В разделе 57.3 мы отмечали: флаг MSG_MORE обеспечивает похожее на TCP_CORK поведение, но для отдельных системных вызовов. Это не всегда является преимуществом. Мы можем включить для сокета параметр TCP_CORK и затем запустить программу, которая на-

правляет вывод в унаследованный файловый дескриптор, не имея никакого представления о том, что использует данный параметр. Для сравнения, флаг MSG_MORE требует непосредственного изменения исходного кода программы.

В FreeBSD существует параметр, похожий на TCP_CORK; он называется TCP_NOPUSH.

57.5. Получение адреса сокета

Системные вызовы `getsockname()` и `getpeername()` возвращают локальный адрес, к которому привязан сокет, и, соответственно, адрес удаленного сокета на другом конце соединения.

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Оба вызова возвращают `0` при успешном завершении или `-1` при ошибке

В обоих вызовах аргумент `sockfd` обозначает файловый дескриптор, ссылающийся на сокет, а `addr` — указатель на буфер подходящего размера, применяемый для возвращения структуры с адресом сокета. Размер и тип этой структуры зависят от домена сокета. С помощью аргумента `addrlen` возвращается итоговый результат. Перед вызовом он должен быть равен размеру буфера, на который указывает `addr`; при возвращении он содержит количество байтов, записанных в этот буфер.

Функция `getsockname()` возвращает семейство адресов сокета и адрес, к которому привязан данный сокет. Это может пригодиться, если он был привязан другой программой (например, `inetd(8)`), а его файловый дескриптор был сохранен на протяжении работы вызова `exec()`.

Функция `getsockname()` тоже может оказаться полезной, если мы хотим определить номер динамического порта, который был назначен ядром сокету интернет-домена во время его автоматической привязки. Ядро выполняет эту привязку в следующих случаях:

- после вызова `connect()` или `listen()` для TCP-сокета, который ранее не был привязан к адресу с помощью операции `bind()`;
- при первом вызове `sendto()` для UDP-сокета, ранее не привязанного к адресу;
- после вызова `bind()`, если в качестве номера порта (`sin_port`) был указан 0. В этом случае данный вызов определяет IP-адрес сокета, а ядро выбирает для него номер динамического порта.

Системный вызов `getpeername()` возвращает адрес удаленного сокета на другом конце соединения. Это может пригодиться в ситуации, когда серверу нужно получить адрес клиента, инициировавшего соединение. Данную информацию можно также получить при выполнении вызова `accept()`; но если сервер был запущен программой, которая вызвала `accept()` (например, `inetd`), то не сможет получить сведения об адресе, несмотря на наличие файлового дескриптора сокета.

Использование функций `getsockname()` и `getpeername()` продемонстрировано в листинге 57.3. Эта программа задействует функции, которые мы определили в листинге 55.9, для выполнения следующих шагов.

1. Применение нашей функции `inetListen()` для создания слушающего сокета, `listenFd`, привязанного к универсальному IP-адресу и порту, чей номер указан в виде единственного аргумента командной строки (его можно задать в виде числа или имени службы).

Аргумент `len` возвращает размер структуры с адресом для сокета текущего домена. Позже это значение передается в вызов `malloc()`, где на его основе выделяется буфер для хранения адреса сокета, полученного из вызовов `getsockname()` и `getpeername()`.

2. Использование функции `inetConnect()` для создания второго сокета, `connFd`, с помощью которого сокету, созданному в предыдущем шаге, отправляется запрос на соединение.
3. Вызов `accept()` для слушающего сокета. В результате создается третий сокет, `acceptFd`, соединенный с сокетом, созданным на шаге 2.
4. Применение вызовов `getsockname()` и `getpeername()` для получения локального и удаленного адресов двух соединенных сокетов, `connFd` и `acceptFd`. После каждого из этих вызовов программа обращается к функции `inetAddressStr()` для преобразования адреса в формат, пригодный для печати.
5. Приостановка работы на несколько секунд, чтобы запустить утилиту `netstat` (описана в разделе 57.7) и подтвердить информацию об адресе сокета.

Пример выполнения этой программы показан в следующей сессии командной строки:

```
$ ./socknames 55555 &
getsockname(connFd):      (localhost, 32835)
getsockname(acceptFd):    (localhost, 55555)
getpeername(connFd):      (localhost, 55555)
getpeername(acceptFd):    (localhost, 32835)
[1] 8171
$ netstat -a | egrep '(Address|55555)'
Proto Recv-Q Send-Q Local Address    Foreign Address    State
tcp        0      0 *:55555          *:*                LISTEN
tcp        0      0 localhost:32835  localhost:55555    ESTABLISHED
tcp        0      0 localhost:55555  localhost:32835    ESTABLISHED
```

Как видите, подключенный сокет (`connFd`) был привязан к динамическому порту под номером 32835. Команда `netstat` выводит сведения обо всех трех сокетах, созданных нашей программой, и позволяет проверить информацию о номерах портов двух соединенных сокетов, которые находятся в состоянии `ESTABLISHED` (описанном в подразделе 57.6.3).

Листинг 57.3. Использование вызовов getsockname() и getpeername()

sockets/socknames.c

```
#include "inet_sockets.h"    /* Определяет наши функции для работы с сокетами */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int listenFd, acceptFd, connFd;
    socklen_t len;                  /* Размер буфера с адресом сокета */
    void *addr;                     /* Буфер для адреса сокета */
    char addrStr[IS_ADDR_STR_LEN];

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s service\n", argv[0]);

    listenFd = inetListen(argv[1], 5, &len);
    if (listenFd == -1)
        errExit("inetListen");
    connFd = inetConnect(NULL, argv[1], SOCK_STREAM);
    if (connFd == -1)
        errExit("inetConnect");
```

```
    acceptFd = accept(listenFd, NULL, NULL);
    if (acceptFd == -1)
        errExit("accept");
    addr = malloc(len);
    if (addr == NULL)
        errExit("malloc");

    if (getsockname(connFd, addr, &len) == -1)
        errExit("getsockname");
    printf("getsockname(connFd):    %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));
    if (getsockname(acceptFd, addr, &len) == -1)
        errExit("getsockname");
    printf("getsockname(acceptFd): %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));

    if (getpeername(connFd, addr, &len) == -1)
        errExit("getpeername");
    printf("getpeername(connFd):    %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));
    if (getpeername(acceptFd, addr, &len) == -1)
        errExit("getpeername");
    printf("getpeername(acceptFd): %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));

    sleep(30);            /* Дает нам время для запуска netstat(8) */
    exit(EXIT_SUCCESS);
}
```

sockets/socknames.c

57.6. Подробности реализации протокола TCP

Представление о некоторых особенностях работы протокола TCP поможет нам отлаживать приложения, основанные на потоковых сокетах, и в ряде случаев делать их более эффективными. В следующих разделах мы рассмотрим:

- формат TCP-сегментов;
- модель подтверждения в TCP;
- машину состояний протокола TCP;
- установку и разрыв TCP-соединений;
- состояние `TIME_WAIT`.

57.6.1. Формат TCP-сегментов

На рис. 57.2 показан формат TCP-сегмента, который передается между разными концами TCP-соединения. Данные поля означают следующее.

- *Номер исходного порта* — номер порта сокета-отправителя.
- *Номер конечного порта* — номер порта сокета-получателя.
- *Порядковый номер* — порядковый номер сегмента. Представляет собой сдвиг относительно первого байта в этом сегменте в рамках потока данных, передаваемого по соединению в заданном направлении (см. подраздел 54.6.3).
- *Номер подтверждения* — если установлен бит `ACK` (см. ниже), то данное поле содержит порядковый номер следующего байта, который получатель ожидает принять от получателя.

- *Длина заголовка* — длина заголовка, измеряемая 32-разрядными словами. Поскольку данное поле равно 4 битам, итоговая длина заголовка может достигать 60 байт (15 слов). Благодаря этому полю принимающая сторона соединения способна определить размер поля с *параметрами* и начальный сдвиг для *данных*.
- *Зарезервировано* — состоит из 4 неиспользуемых битов и должно быть равно 0.
- *Управляющие биты* — это поле состоит из 8 битов, которые определяют дополнительные свойства сегмента:

Рис. 57.2. Формат TCP-сегмента

- `CWR`: от англ. congestion window reduced — «сужение окна перегрузки».
- `ECE`: от англ. explicit congestion notification echo — «отклик явного уведомления о перегрузке». Флаги CWR и ECE являются частью алгоритма `ECN` (англ. explicit congestion notification — «явное уведомление о перегрузке»), который применяется в TCP/IP. Это относительно новая технология; ее описание можно найти в документе RFC 3168 и книге [Floyd, 1994]. Алгоритм `ECN` реализован в Linux начиная с версии 2.4; включить его можно, задав ненулевое значение в файле `/proc/sys/net/ipv4/tcp_ecn` (доступном только в Linux).
- `URG`: от англ. urgent pointer — «указатель важности». Если его установить, то он будет содержать корректную информацию.
- `ACK`: от англ. acknowledgement number — «номер подтверждения». Если его установить, то поле с номером подтверждения будет содержать корректную информацию (то есть этот сегмент сможет подтверждать получение данных, посланных ранее удаленным сокетом).
- `PSH`: от англ. push — «толкать». «Проталкивает» все полученные данные к принимающему процессу. Этот флаг описан в RFC 993 и [Stevens, 1994].

- `RST`: от англ. reset — «сбрасывать». Сбрасывает соединение. Используется для обработки различных ошибок.
- `SYN`: от англ. synchronize — «синхронизировать». Синхронизирует порядковые номера. Сегменты с этим флагом передаются во время установки соединения, чтобы оба сокета могли указать начальные порядковые номера, которые будут применяться для передачи данных в обоих направлениях.
- `FIN`: от англ. finish — «завершать». Используется отправителем для уведомления о завершении передачи данных.

Сегменту можно установить сразу несколько контролирующих битов (или ни одного), благодаря чему он может иметь разные назначения. Например, позже мы увидим, что во время установки TCP-соединения сокеты обмениваются сегментами с битами `SYN` и `ACK`.

- *Размер окна* — когда получатель отправляет сегмент с битом `ACK`, данное поле сигнализирует о том, сколько свободного места есть у получателя для приема новых данных (это имеет отношение к алгоритму скользящего окна, который мы затронули в подразделе 54.6.3).
- *Контрольная сумма* — 16-разрядная контрольная сумма, охватывающая как заголовок, так и данные сегмента.

  В протоколе TCP контрольная сумма охватывает не только заголовок и данные, но и 12 дополнительных байтов, которые обычно называют псевдозаголовком. Он состоит из следующих элементов: исходного и конечного IP-адресов (по четыре байта каждый); двух байтов, обозначающих размер TCP-сегмента (это вычисляемое значение, но оно не входит ни в IP-, ни в TCP-заголовок); одного байта со значением 6, которое является уникальным номером протокола TCP в рамках стека TCP/IP; одного сдвигающего байта, содержащего 0 (чтобы длина псевдозаголовка была кратна 16 битам). Применение псевдозаголовка при вычислении контрольной суммы позволяет получателю перепроверять тот факт, что входящий сегмент дошел туда, куда нужно (то есть IP-клиент не принял по ошибке датаграмму, направленную другому сетевому узлу, или TCP-пакет, который должен был перейти на более высокий уровень). Вычисление контрольных сумм в пакетных заголовках протокола UDP выполняется похожим образом и по аналогичным причинам. Больше подробностей о псевдозаголовках можно найти в книге [Stevens, 1994].

- *Указатель важности* — если установлен управляющий бит `URG`, то это поле указывает на положение так называемых важных данных, которые в рамках потока передаются от отправителя к получателю. Мы еще коснемся настоящей темы в подразделе 57.13.1.
- *Параметры* — поле переменной длины, содержащее параметры, влияющие на работу TCP-соединения.
- *Данные* — поле содержит пользовательские данные, передаваемые в текущем сегменте. Оно может иметь нулевую длину, если не содержит никакой информации (например, если это всего лишь сегмент `ACK`).

57.6.2. Порядковые номера и подтверждения в протоколе TCP

Каждому байту, переданному по TCP-соединению, протокол TCP назначает логический порядковый номер (каждый из двух потоков в соединении имеет отдельную нумерацию). После передачи сегмента его полю с *порядковым номером* присваивается сдвиг относительно первого байта данных, которые он передал в рамках одного из потоков соединения. Это позволяет принимающей стороне собрать полученные сегменты в правильном порядке и послать отправителю подтверждение.

Для реализации надежного обмена данными протокол TCP использует положительные подтверждения. То есть, когда сегмент успешно принят, получатель отсылает отправителю сообщение с подтверждением (сегмент с установленным битом ACK), как показано на рис. 57.3. Поле с *номером подтверждения* в этом сообщении содержит логический порядковый номер байта, следующего за данными, которые принимающая сторона надеется получить (иными словами, номер подтверждения ровно на 1 больше порядкового номера последнего байта в сегменте, передающем подтверждение).

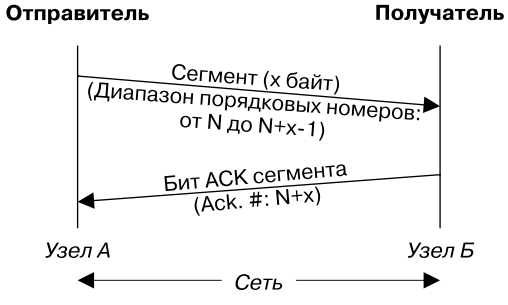


Рис. 57.3. Подтверждения в протоколе TCP

Когда отправляющая сторона передает сегмент, она устанавливает таймер. Если по истечении времени ожидания подтверждение все еще не получено, сегмент отправляется заново.

На рис. 57.3 и аналогичных диаграммах, представленных далее, иллюстрируется обмен TCP-сегментами между двумя конечными точками. Подразумевается, что время связано с осью ординат и течет сверху вниз.

57.6.3. Машина состояний и диаграмма перехода состояний в протоколе TCP

Поддержание TCP-соединения требует координации между обоими его концами. Для упрощения данной задачи конечные точки в TCP имеют вид *машины состояний* (или конечного автомата). Это значит, что в любой момент каждый конец соединения может находиться только в одном из строго определенных *состояний* и переход от одного состояния к другому происходит в ответ на *события*: например, системные вызовы, выполняемые приложением, работающим поверх TCP, или поступление нового TCP-сегмента, отправленного удаленным узлом. Протокол TCP поддерживает следующие состояния.

- `LISTEN` — сокет ожидает запрос на соединение со стороны удаленного клиента.
- `SYN_SENT` — сокет отправил от имени приложения, выполняющего активное открытие, сегмент с байтом `SYN` и теперь ждет ответа от удаленной стороны, чтобы завершить установку соединения.
- `SYN_RECV` — сокет, ранее пребывавший в состоянии `LISTEN`, принял байт `SYN`, послал в ответ `SYN/ACK` (то есть сегмент с установленными байтами `SYN` и `ACK`) и теперь ждет от удаленного сокета подтверждения, чтобы завершить установку соединения.
- `ESTABLISHED` — установка соединения с удаленным сокетом завершилась. Теперь можно передавать сегменты с данными в любом направлении.
- `FIN_WAIT1` — приложение закрыло соединение. Сокет послал удаленной стороне байт `FIN`, чтобы прекратить взаимодействие, и теперь ждет подтверждения (бит `ACK`). Это

и следующие три состояния относятся к приложению, выполняющему активное закрытие, — тому, которое первым разрывает соединение на своем конце.

- `FIN_WAIT2` — сокет, ранее пребывавший в состоянии `FIN_WAIT1`, получил подтверждение от удаленной стороны.
- `CLOSING` — сокет, ранее ожидавший подтверждения в состоянии `FIN_WAIT1`, получил вместо этого бит `FIN`, сигнализирующий о том, что удаленная сторона тоже в данный момент пыталась выполнить активное закрытие (иными словами, оба конца TCP-соединения почти одновременно послали друг другу сегменты с битом `FIN`; такое бывает редко).
- `TIME_WAIT` — выполнив активное закрытие, сокет получил бит `FIN`, сигнализирующий о том, что удаленная сторона уже завершила пассивное закрытие. Теперь данный сокет должен провести определенное время в состоянии `TIME_WAIT`, чтобы гарантировать разрыв TCP-соединения и обеспечить истечение срока годности старых дублирующих сегментов в сети перед созданием нового экземпляра того же соединения (более подробно состояние `TIME_WAIT` рассматривается в подразделе 57.6.7). По завершении этого времени соединение закрывается, а связанные с ним ресурсы ядра освобождаются.
- `CLOSE_WAIT` — сокет получил от удаленной стороны байт `FIN`. Это и следующее состояния относятся к приложению, выполняющему пассивное закрытие, то есть ко второму приложению, закрывающему соединение.
- `LAST_ACK` — приложение выполнило пассивное закрытие, а сокет, ранее пребывавший в состоянии `CLOSE_WAIT`, послал удаленной стороне бит `FIN` и теперь ждет, когда та передаст подтверждение. Получив бит `ACK`, сокет закрывает соединение и освобождает связанные с собой ресурсы ядра.

Помимо описанных выше состояний документ RFC 793 предусматривает еще одно, фиктивное. Оно называется `CLOSED` и описывает ситуацию, когда соединения нет (то есть для поддержания TCP-соединения ядро не выделило никаких ресурсов).

> В списке, приведенном выше, используются названия констант, определенных в исходном коде Linux. Они немного отличаются от названий, описанных в RFC 793.

На рис. 57.4 показана *диаграмма перехода состояний* для протокола TCP (за основу взяты диаграммы из RFC 793 и книги [Stevens et al., 2004]). Она описывает то, как одна из сторон TCP-соединения переходит от одного состояния к другому в зависимости от различных событий. Каждая стрелка символизирует потенциальный переход и помечена событием, которое инициирует данный переход. Это может быть действие приложения (выделяется жирным шрифтом) или строка `recv`, сигнализирующая о получении сегмента от удаленного сокета. По мере перехода от одного состояния к другому сокет может передавать сегменты удаленной стороне; в таком случае указывается метка `send`. Например, стрелка для перехода из `ESTABLISHED` в `FIN_WAIT1` показывает, что инициатором является вызов `close()`, выполненный локальным приложением, и при этом удаленному сокету передается сегмент с битом `FIN`.

Обычное для TCP-клиента направление перехода показано на рис. 57.4 в виде сплошной стрелки, а пунктирной обозначено направление, которое, как правило, выбирает TCP-сервер (другие стрелки изображают направления, используемые не так часто). Глядя на номера, указанные в скобках рядом с этими стрелками, можно сделать вывод: отправляемые и принимаемые сегменты на обоих концах соединения являются зеркальным отражением друг друга. (После прохождения состояния `ESTABLISHED` маршруты, выбираемые сервером и клиентом, могут оказаться обратными тем, что изображены на диаграмме; так происходит, если активное закрытие выполняет серверная сторона.)

> На рис. 57.4 показаны не все возможные переходы машины состояний протокола TCP, а только представляющие особый интерес. Больше подробностей о диаграмме перехода состояний для протокола TCP можно найти на www.cl.cam.ac.uk/~pes20/Netsem/poster.pdf.

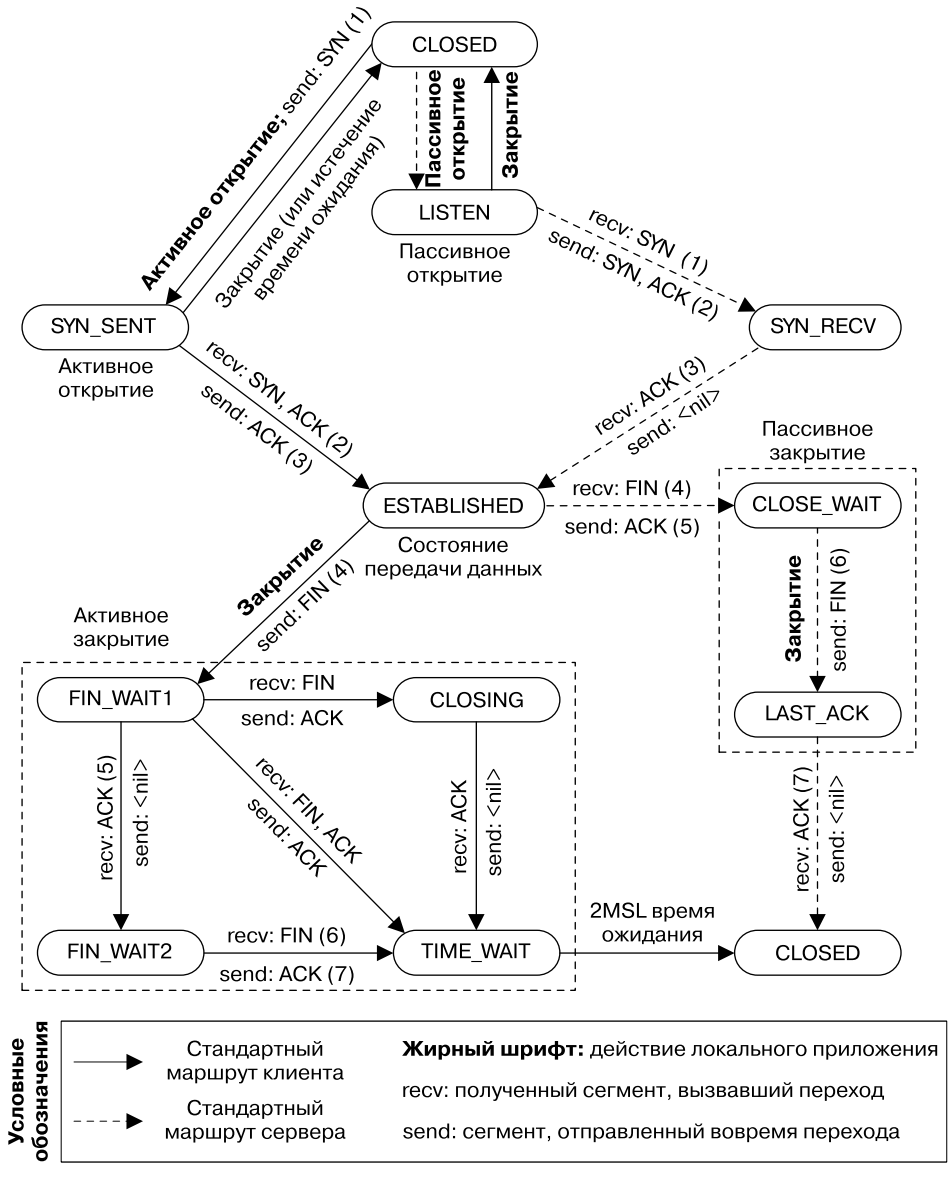


Рис. 57.4. Диаграмма перехода состояний для протокола TCP

57.6.4. Установка TCP-соединения

На уровне своих программных интерфейсов два потоковых сокета соединяются, выполняя два следующих шага (см. рис. 52.1).

1. Сервер делает вызов `listen()`, чтобы выполнить пассивное открытие сокета, и затем вызывает операцию `accept()`, которая блокируется до тех пор, пока соединение не будет установлено.

2. Клиент делает вызов `connect()` с целью выполнить активное открытие сокета и установить соединение с пассивным сокетом на стороне сервера.

Этапы установки TCP-соединения изображены на рис. 57.5. Данную процедуру часто называют *трехэтапным согласованием* (англ. three-way handshake), так как между двумя концами соединения проходят три сегмента.

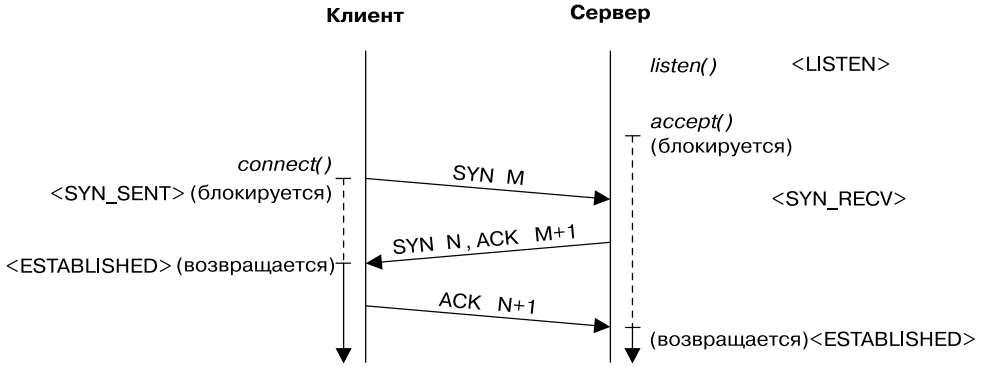


Рис. 57.5. Трехэтапное согласование при установке TCP-соединения

В процессе выполняются следующие шаги.

1. Вызов `connect()` заставляет TCP-клиент послать сегмент `SYN`, который информирует сервер о начальном порядковом номере клиента (на диаграмме обозначен как *M*). Эта информация является необходимой, поскольку, как было отмечено в подразделе 54.6.3, порядковые номера не начинаются с 0.
2. TCP-сервер должен подтвердить получение сегмента `SYN` и проинформировать клиента о своем собственном начальном порядковом номере (на диаграмме обозначен как *N*). Оба номера необходимы, так как потоковый сокет является двунаправленным. Для выполнения этих двух операций сервер может вернуть единственный сегмент с установленными битами `SYN` и `ACK` (мы как бы *цепляем* бит `ACK` к `SYN`).
3. TCP-клиент отправляет сегмент `ACK`, чтобы подтвердить получение сегмента `SYN`, посланного сервером.

Сегменты SYN, чей обмен происходит на первых двух шагах трехэтапного согласования, могут содержать в TCP-заголовке параметры, на основе которых определяются свойства соединения. Подробности можно найти в [Stevens et al., 2004], [Stevens, 1994] и [Wright & Stevens, 1995].

Метки в угловых скобках (например, `<LISTEN>`) на рис. 57.5 обозначают состояния обоих концов соединения.

Флаг `SYN` занимает один байт от места, выделенного для порядкового номера соединения. Это позволяет однозначно подтвердить получение `SYN`, так как сегмент, вместе с которым он передан, может также содержать какие-то данные. Вот почему подтверждение получения сегмента `SYN M` на рис. 57.5 изображено как `ACK M+1`.

57.6.5. Разрыв TCP-соединения

Закрытие TCP-соединения обычно происходит следующим образом.

1. Приложение на одном из концов соединения выполняет вызов `close()` (часто это делает клиент). Считается, что таким образом производится *активное закрытие*.

2. Позже приложение на другом конце соединения (сервер) тоже выполняет вызов `close()`. Данное действие называется *пассивным закрытием*.

На рис. 57.6 показано, какие действия выполняют соответствующие TCP-сокеты (подразумевается, что активное закрытие производит клиент).

1. Клиент выполняет активное закрытие, это заставляет его сокет отправить серверу сегмент `FIN`.
2. Получив сегмент `FIN`, сервер шлет в ответ подтверждение `ACK`. Любые последующие попытки сервера прочитать данные из сокета будут приводить к получению символа конца файла (то есть значения `0`).
3. Позже, когда сервер закроет свой конец соединения, он пошлет клиенту сегмент `FIN`.
4. В ответ на полученный сегмент `FIN` клиент отправляет подтверждение `ACK`.

Флаг `FIN` (по аналогии с флагом `SYN` и по той же причине) занимает один байт от места, выделенного для порядкового номера соединения. Именно поэтому на рис. 57.6 подтверждение получения сегмента `FIN M` показано как `ACK M+1`.

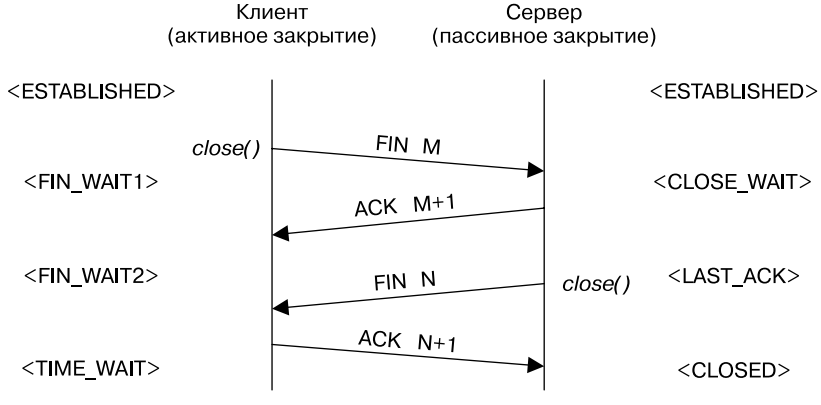


Рис. 57.6. Разрыв TCP-соединения

57.6.6. Вызов shutdown() для TCP-сокета

В предыдущих разделах подразумевалось, что мы выполняем полное закрытие — то есть закрываем оба канала потокового сокета с помощью вызова `close()`. Но, как отмечалось в разделе 57.2, можно воспользоваться вызовом `shutdown()` и закрыть только один канал соединения (выполнив тем самым частичное закрытие). В этом разделе мы рассмотрим некоторые особенности поведения вызова `shutdown()` в контексте TCP-сокетов.

Передавая аргументу `how` значение `SHUT_WR` или `SHUT_RDWR`, мы инициируем процедуру разрыва соединения (то есть активного закрытия), описанную в подразделе 57.6.5; при этом неважно, ссылается ли на данный сокет какой-нибудь другой файловый дескриптор. Далее локальный сокет переходит сначала в состояние `FIN_WAIT1`, а затем в `FIN_WAIT2`, тогда как удаленный сокет переходит в состояние `CLOSE_WAIT` (см. рис. 57.6). Если аргументу `how` передать значение `SHUT_WR`, то удаленный сокет сможет продолжать передавать данные, так как его файловый дескриптор остается актуальным, и считывающий канал соединения по-прежнему открыт.

Операция `SHUT_RD` не имеет смысла в контексте TCP-сокетов. Дело в том, что большинство реализаций протокола TCP не обеспечивают предсказуемое поведение при использовании этой константы, а итоговый результат может варьироваться. В Linux и не-

скольких других системах в результате операции SHUT_RD (и после прочтения оставшихся данных) вызов read() возвращает символ конца файла, чего и следует ожидать согласно описанию данной константы в разделе 57.2. Но если после этого удаленное приложение запишет какую-либо информацию в свой сокет, то ее по-прежнему можно будет прочитать на другом конце соединения.

В некоторых других системах (таких как BSD) операция SHUT_RD в самом деле приводит к тому, что последующие вызовы read() всегда возвращают 0. Но если в этом случае удаленное приложение продолжит записывать данные в сокет, то соответствующий канал соединения в какой-то момент заполнится, после чего любые (блокирующие) вызовы write() на удаленной стороне будут блокироваться. (Будь потоковый сокет в домене UNIX, при продолжении записи после выполнения операции SHUT_RD удаленное приложение получило бы сигнал SIGPIPE и ошибку EPIPE.)

В целом в портируемых приложениях, работающих с протоколом TCP, следует избегать использования значения SHUT_RD.

57.6.7. Состояние TIME_WAIT

Состояние TIME_WAIT часто вызывает недопонимание у программистов, пишущих сетевые приложения. Если взглянуть на рис. 57.4, то можно увидеть, что через это состояние проходит сокет, выполняющий активное закрытие. Оно служит двум целям:

- реализация надежного разрыва соединения;
- возможность автоматического удаления старых дублирующих сегментов в сети, так предотвращается их прием новыми экземплярами предыдущих соединений.

Состояние TIME_WAIT является особенным в том смысле, что событие, которое позволяет из него выйти (и перейти в CLOSED), заключается в истечении времени ожидания. Оно равно двум *максимальным жизненным циклам сегмента* (англ. maximum segment lifetime, MSL) — то есть максимальному времени существования сегмента в сети.

> Восьмиразрядное поле TTL (от англ. time-to-live — «время жизни») в IP-заголовке гарантирует, что все IP-пакеты рано или поздно перестанут быть действительными, если не достигнут адресата за фиксированное количество переходов по маршруту между начальным и конечным узлами. MSL является оценкой максимального времени, которое может уйти на исчерпание ограничения TTL. Так как поле TTL занимает 8 бит, оно делает возможным максимум 255 переходов. Это намного больше, чем обычно требуется для прохождения полного маршрута. Исчерпание данного ограничения может быть связано с определенными аномалиями в работе маршрутизатора (например, если он неправильно сконфигурирован), из-за которых пакет попадает в замкнутый круг и не может оттуда выбраться, пока не превысит значение TTL.

В системах семейства BSD значение MSL равно 30 секундам, и Linux следует этой норме. Таким образом, в Linux состояние TIME_WAIT может длиться на протяжении 60 секунд. Хотя документ RFC 1122 рекомендует для MSL значение в 2 минуты. Многие системы следуют данной рекомендации, в связи с чем продолжительность состояния TIME_WAIT может достигать 4 минут.

Первый пункт, касающийся назначения состояния TIME_WAIT (обеспечение надежного разрыва соединения), проиллюстрирован на рис. 57.6. На этой диаграмме видно: во время завершения TCP-соединения происходит обмен четырьмя сегментами. Последним из них является подтверждение ACK, которое отправляется стороной, выполняющей активное закрытие, приложению, закрывающему свой сокет в пассивном режиме. Представьте, что данный сегмент затерялся в сети. В таком случае сторона, выполняющая пассивное

закрытие, рано или поздно повторит передачу сегмента FIN. Поскольку противоположная сторона остается в состоянии TIME_WAIT на протяжении фиксированного отрезка времени, она все еще может повторно отправить подтверждение ACK. Но если сторона, выполнившая активное соединение, больше недоступна, то не будет иметь никакой информации о состоянии соединения. В данном случае в ответ на повторно отправленный сегмент FIN протокол TCP ответит сегментом RST (сброс), который будет интерпретирован как ошибка. (Это объясняет, почему продолжительность состояния TIME_WAIT вдвое больше MSL: один MSL уходит на отправку завершающего подтверждения ACK удаленному концу и еще один нужен на случай, если сегмент FIN нужно будет отправлять повторно.)

> Стороне, выполняющей пассивное закрытие, не нужно переходить в состояние TIME_WAIT, поскольку именно она инициировала завершающий обмен сегментами при разрыве соединения. После отправки сегмента FIN она будет ждать подтверждения ACK от удаленного сокета, и если срок годности этого сегмента завершится до получения ACK, то она отправит его снова.

Чтобы понять второе назначение состояния TIME_WAIT (удаление просроченных дублирующих сегментов), нужно вспомнить такой факт: алгоритм повторной передачи, который применяется в протоколе TCP, может привести к дублированию сегментов, и в зависимости от выбранных маршрутов эти дубликаты могут дойти до адресата уже после разрыва соединения. Допустим, у нас имеется соединение между двумя сокетами: 204.152.189.116 порт 21 (служба FTP) и 200.0.0.1 порт 50000. Теперь представим, что мы его разорвали и позже установили аналогичное соединение с теми же IP-адресами и портами. Таким образом получен новый экземпляр старого соединения. В данном случае протокол TCP должен убедиться: никакие старые дублирующие сегменты из предыдущего экземпляра не будут приняты в качестве корректных данных. Чтобы этого добиться, новый экземпляр нельзя установить, пока хотя бы одна из сторон существующего TCP-соединения находится в состоянии TIME_WAIT.

На интернет-форумах часто спрашивают, как отключить состояние TIME_WAIT, ведь оно может привести к ошибке EADDRINUSE («этот адрес уже занят»), если перезапущенный сервер попытается привязать сокет к адресу, используемому на одном из концов соединения, находящихся в данном состоянии. Теоретически это возможно (см. [Stevens et al., 2004]), как и полностью удалять любые сокеты с состоянием TIME_WAIT (см. [Snader, 2000]). Но такие решения являются нежелательными, поскольку их результат отменяет все гарантии надежности, обеспечиваемые этим состоянием. В разделе 57.10 мы рассмотрим параметр SO_REUSEADDR, с помощью которого можно устранить причины, обычно вызывающие ошибку EADDRINUSE, и притом иметь возможность надежно разрывать соединение, задействуя TIME_WAIT.

57.7. Мониторинг сокетов: утилита netstat

Программа netstat отображает состояние сокетов в UNIX- и интернет-доменах. Этот инструмент может пригодиться при отладке приложений, основанных на сокетах. Он доступен в большинстве UNIX-систем, хотя синтаксис его аргументов командной строки может варьироваться в зависимости от реализации.

Если запустить утилиту netstat без параметров, то она выведет информацию о подключенных сокетах в обоих доменах. Чтобы изменить данный вывод, можно использовать целый ряд аргументов командной строки. Некоторые из них перечислены в табл. 57.1.

Таблица 57.1. Параметры команды netstat

Параметр	Описание
-a	Выводит информацию о всех сокетах, включая слушающие
-e	Выводит дополнительную информацию (включая идентификатор владельца сокета)
-c	Постоянно обновляет информацию о сокетах (ежесекундно)
-l	Выводит информацию только о слушающих сокетах
-n	Выводит IP-адреса, номера портов и имена пользователей в числовом формате
-p	Выводит идентификатор процесса и название программы, которой принадлежит сокет
--inet	Выводит информацию о сокетах в интернет-домене
--tcp	Выводит информацию о TCP-сокетах (потоковых) в интернет-домене
--udp	Выводит информацию о UDP-сокетах (датаграммных) в интернет-домене
--unix	Выводит информацию о сокетах в домене UNIX

Ниже показан урезанный вывод, который можно получить, если запросить с помощью `netstat` сведения обо всех сокетах интернет-домена в системе:

```
$ netstat -a --inet
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address    Foreign Address    State
tcp        0      0 *:50000          *:*                LISTEN
tcp        0      0 *:55000          *:*                LISTEN
tcp        0      0 localhost:smtp   *:*                LISTEN
tcp        0      0 localhost:32776  localhost:58000    TIME_WAIT
tcp    34767      0 localhost:55000  localhost:32773    ESTABLISHED
tcp        0 115680 localhost:32773  localhost:55000    ESTABLISHED
udp        0      0 localhost:61000  localhost:60000    ESTABLISHED
udp      684      0 *:60000          *:*
```

Для каждого сокета в интернет-домене доступна следующая информация.

- `Proto` — протокол сокета, например `tcp` или `udp`.
- `Recv-Q` — количество непрочитанных байтов во входящем буфере локального сокета. В случае с UDP в этом поле помимо самих данных учитываются заголовки датаграммы и другая метаинформация.
- `Send-Q` — количество байтов, ожидающих отправки в исходящем буфере сокета. По аналогии с предыдущим полем здесь учитываются заголовки датаграммы и другая метаинформация, если используется протокол UDP.
- `Local Address` — адрес в формате `IP-адрес:порт`, к которому привязан сокет. По умолчанию обе составляющие адреса отображаются в виде имен (если они были найдены для соответствующих IP-адреса и порта). Звездочка (`*`) в имени сетевого узла обозначает универсальный адрес.
- `Foreign Address` — адрес удаленного сокета, к которому подключено локальное приложение. Строка `*:*` указывает на отсутствие удаленного адреса.
- `State` — текущее состояние сокета. Все допустимые состояния для протокола TCP перечислены в подразделе 57.6.3.

Более подробную информацию см. на странице `netstat(8)` руководства.

Различные файлы в каталоге `/proc/net` (доступные только в Linux) позволяют получить большинство из тех данных, которые выводит утилита netstat. По очевидным причинам для этих файлов выбраны следующие имена: `tcp`, `udp`, `tcp6`, `udp6` и `unix`. Дополнительную информацию см. на странице `proc(5)` руководства.

57.8. Мониторинг данных, проходящих по протоколу TCP, с помощью утилиты tcpdump

Программа tcpdump является полезным отладочным инструментом, который позволяет администратору наблюдать за данными, передаваемыми по сети, в режиме реального времени, генерируя текстовые аналоги диаграмм, приведенных на рис. 57.3. Несмотря на свое название, эта программа может использоваться для вывода любых пакетов, применяемых в TCP/IP (например, TCP-сегментов, UDP-датаграмм или ICMP-пакетов). Для каждой сети выводятся такие сведения, как временные отметки, исходный и конечный IP-адреса и дополнительные подробности, относящиеся к тому или иному протоколу. Отслеживаемые пакеты можно фильтровать по типу протокола, исходному и конечному IP-адресам, номеру порта и т. д. Полное описание утилиты tcpdump содержится на одноименной странице руководства.

Программа wireshark (ранее известная как ethereal; www.wireshark.org) по своим функциям похожа на tcpdump, но выводит информацию с помощью графического интерфейса.

Для каждого TCP-сегмента программа tcpdump выводит отдельную строку следующего вида:

```
src > dst: flags data-seqno ack window urg <options>
```

Ниже представлены значения этих полей.

- `src` — исходные IP-адрес и порт.
- `dst` — конечные IP-адрес и порт.
- `flags` — поле может быть пустым или содержать следующие буквы, каждая из которых относится к соответствующему управляющему биту (см. подраздел 57.5.1): S (`SYN`), F (`FIN`), P (`PSH`), R (`RST`), E (`ECE`) и C (`CWR`).
- `data-seqno` — диапазон порядковых номеров, охватываемый байтами в данном пакете.

  Диапазон порядковых номеров по умолчанию выводится относительно первого байта, отслеженного в текущем канале потока данных. Параметр -S программы tcpdump позволяет выводить порядковые номера в виде абсолютных значений.

- `ack` — строка вида `ack num`, обозначающая порядковый номер следующего байта, который должен прийти с противоположного конца соединения.
- `window` — строка вида `win num`, обозначающая количество байтов, доступных во входящем буфере для передачи данных в противоположном направлении текущего соединения.
- `urg` — строка вида `urg num`, сигнализирующая о том, что в текущем сегменте находится важная информация с заданным сдвигом.
- `options` — строка описывает все параметры TCP-соединения, содержащиеся в сегменте.

Поля `src`, `dst` и `flags` выводятся всегда, а остальные — по необходимости.

В сессии командной строки, представленной ниже, показан пример использования утилиты tcpdump для мониторинга данных, которыми обмениваются клиент (запущенный на компьютере `pukaki`) и сервер (запущенный на компьютере `tekapo`). Мы задействуем два параметра, делающие вывод более компактным: `-t` предотвращает отображение временных отметок, а `-N` позволяет опустить полные доменные имена компьютеров. К тому же, поскольку мы не стали углубляться в параметры TCP-соединения, из вывода утилиты tcpdump было убрано поле `options`.

Команда `tcpdump` фильтрует данные по порту 55555, так как на нем работает наш сервер. В вывод попали три сегмента, обмен которыми происходит во время установки соединения:

```
$ tcpdump -t -N 'port 55555'
IP pukaki.60391 > tekapo.55555: S 3412991013:3412991013(0) win 5840
IP tekapo.55555 > pukaki.60391: S 1149562427:1149562427(0) ack 3412991014 win 5792
IP pukaki.60391 > tekapo.55555: . ack 1 win 5840
```

Указанными тремя сегментами являются `SYN`, `SYN/ACK` и `ACK`, применяемые для трехэтапного согласования (см. рис. 57.5).

В следующей сессии клиент отправляет серверу два сообщения размером 16 и 32 бита, на каждое из которых сервер отвечает 4-байтным сообщением:

```
IP pukaki.60391 > tekapo.55555: P 1:17(16) ack 1 win 5840
IP tekapo.55555 > pukaki.60391: . ack 17 win 1448
IP tekapo.55555 > pukaki.60391: P 1:5(4) ack 17 win 1448
IP pukaki.60391 > tekapo.55555: . ack 5 win 5840
IP pukaki.60391 > tekapo.55555: P 17:49(32) ack 5 win 5840
IP tekapo.55555 > pukaki.60391: . ack 49 win 1448
IP tekapo.55555 > pukaki.60391: P 5:9(4) ack 49 win 1448
IP pukaki.60391 > tekapo.55555: . ack 9 win 5840
```

Для каждого сегмента с данными в противоположном направлении передается подтверждение `ACK`.

В конце мы выводим сегменты, обмен которыми происходит во время разрыва соединения (сначала закрывается клиентский сокет, а затем серверный):

```
IP pukaki.60391 > tekapo.55555: F 49:49(0) ack 9 win 5840
IP tekapo.55555 > pukaki.60391: . ack 50 win 1448
IP tekapo.55555 > pukaki.60391: F 9:9(0) ack 50 win 1448
IP pukaki.60391 > tekapo.55555: . ack 10 win 5840
```

В представленном выше выводе показано, что во время разрыва соединения было передано четыре сегмента (см. рис. 57.6).

57.9. Параметры сокета

Параметры сокета влияют на различные аспекты его работы. В этой книге мы опишем всего лишь несколько из них. В издании [Stevens et al., 2004] содержится углубленное описание большинства стандартных параметров сокета. Дополнительные подробности, относящиеся к Linux, см. на страницах `tcp(7)`, `udp(7)`, `ip(7)`, `socket(7)` и `unix(7)` руководства.

Чтобы задать и извлечь параметры сокета, можно использовать системные вызовы `setsockopt()` и `getsockopt()`.

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Оба вызова возвращают `0` при успешном завершении или `-1` при ошибке

В обоих вызовах, приведенных выше, аргумент `sockfd` является файловым дескриптором, ссылающимся на сокет.

Аргумент `level` определяет протокол, к которому применяется параметр сокета, например IP или TCP. Для большинства параметров, описанных в нашей книге, данный аргумент должен быть равен `SOL_SOCKET`; это значит, что параметр действует на уровне программного интерфейса сокета.

Аргумент `optname` обозначает имя параметра, чье значение мы хотим установить или получить. Аргумент `optval` является указателем на буфер, в котором возвращается значение параметра; это могут быть целое число или структура в зависимости от того, какой параметр указан.

Аргумент `optlen` хранит размер буфера (в байтах), на который указывает `optval`. В вызов `setsockopt()` данный аргумент передается по значению. В вызове `getsockopt()` он используется для возвращения результата. Его следует предварительно инициализировать с помощью размера вышеупомянутого буфера; во время возвращения ему присваивается количество байтов, записанных в этот буфер.

Как отмечается в разделе 57.11, файловый дескриптор сокета, возвращаемый вызовом `accept()`, наследует от слушающего сокета значения его установленных параметров.

Параметры сокета привязываются к дескриптору открытого файла (см. рис. 5.2). Это значит, что дескрипторы, продублированные в результате вызова `dup()` (или ему подобного) либо `fork()`, будут иметь одинаковые параметры сокета.

В качестве примера воспользуемся параметром `SO_TYPE`, который позволяет узнать тип сокета:

```
int optval;
socklen_t optlen;

optlen = sizeof(optval);
if (getsockopt(sfd, SOL_SOCKET, SO_TYPE, &optval, &optlen) == -1)
    errExit("getsockopt");
```

После данного вызова аргумент `optval` будет содержать тип сокета, допустим, `SOCK_STREAM` или `SOCK_DGRAM`. Это может пригодиться в программах, которые наследуют файловый дескриптор на время выполнения `exec()` (например, если они запущены с помощью `inetd`) и не знают, с сокетом какого типа имеют дело.

Параметр `SO_TYPE` является одним из тех, что доступны только для чтения. Их значение нельзя изменить, задействуя вызов `setsockopt()`.

57.10. Параметр сокета SO_REUSEADDR

Параметр сокета `SO_REUSEADDR` имеет сразу несколько назначений (см. главу 7 книги [Stevens et al., 2004]). Но нас интересует только одно из них, довольно распространенное: предотвращение ошибки `EADDRINUSE` («этот адрес уже занят»), когда после перезапуска

TCP-сервер привязывает сокет к порту, уже используемому другим TCP-соединением. Существует два сценария, в которых это обычно происходит.

- Предыдущий экземпляр сервера, подключенный к клиенту, выполнил активное закрытие — либо с помощью вызова `close()`, либо в результате сбоя (например, если был завершен по сигналу). В этом случае один из концов соединения остается в состоянии `TIME_WAIT` до истечения времени ожидания длиной 2MSL.
- Предыдущий экземпляр сервера создал дочерний процесс для обработки клиентского запроса. Позже сервер завершил работу, тогда как его потомок продолжил обслуживать клиента. В итоге его конец соединения остался привязанным к общеизвестному порту сервера.

В обоих случаях оставшийся конец соединения не может принимать новые запросы. Хотя большинство реализаций протокола TCP не дали бы новому слушающему сокету привязаться к уже занятому серверному порту.

> Ошибка EADDRINUSE обычно не встречается на клиентской стороне, где, как правило, используются динамические порты, которые никогда не совпадают с теми, что уже заняты соединением в состоянии TIME_WAIT. Однако клиенты, привязанные к порту с определенным номером, не защищены от этой ошибки.

Чтобы понять принцип работы параметра `SO_REUSEADDR`, вернемся к нашей аналогии с телефонами, которую мы приводили ранее в разделе 52.5 при знакомстве с потоковыми сокетами. Как и любой телефонный вызов (кроме, разве что, телеконференций), TCP-соединение идентифицируется с помощью сочетания из двух конечных точек, подключенных друг к другу. Вызов `accept()` аналогичен процедуре, выполняемой внутренним коммутатором («сервером»). Обнаружив внешний вызов, коммутатор направляет его к какому-то телефону внутри организации («новому сокету»). Внешний наблюдатель не имеет возможности определить внутренний телефон. Единственный способ различить несколько звонков, выполненных извне, — использовать комбинацию внешнего номера вызывающего абонента и номера коммутатора (последний необходим, поскольку таких коммутаторов в телефонной сети может быть сколько угодно). Аналогично мы создаем новый сокет каждый раз, когда принимаем соединение с помощью слушающего сокета. Все эти сокеты (включая слушающий) связаны с одним и тем же локальным адресом. Различить их можно только по тому, с какими удаленными сокетами они соединены.

Иными словами, подключенный TCP-сокет идентифицируется путем комбинации из четырех значений следующего вида:

```
{ локальный-IP-адрес, локальный-порт, удаленный-IP-адрес, удаленный-порт }
```

Спецификация протокола TCP требует, чтобы каждый такой набор был уникальным; то есть у соответствующего соединения может быть только один экземпляр («телефонный звонок»). Проблема вот в чем: в большинстве реализаций (включая Linux) действует более строгое ограничение: локальный порт нельзя использовать повторно (передавая его вызову `bind()`), если в системе существует экземпляр TCP-соединения с тем же портом. Это правило действует даже в том случае, если сокет не принимает новые соединения (как в сценарии, описанном в начале данного раздела).

Применение параметра `SO_REUSEADDR` смягчает указанное ограничение, делая его более близким к требованиям протокола TCP. По умолчанию этот параметр равен 0 (то есть выключен). Чтобы его включить, перед привязкой сокета ему нужно установить ненулевое значение, как показано в листинге 57.4.

Установка параметра `SO_REUSEADDR` позволяет привязывать сокет к локальному порту, даже если он занят другим TCP-соединением (в любом из двух сценариев, описанных

в начале этого раздела). Данный параметр следует включать в большинстве TCP-серверов. Мы уже сталкивались с его применением в листингах 55.6 и 55.9.

Листинг 57.4. Установка параметра сокета SO_REUSEADDR

```
int sockfd, optval;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1)
    errExit("socket");

optval = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval,
        sizeof(optval)) == -1)
    errExit("socket");

if (bind(sockfd, &addr, addrlen) == -1)
    errExit("bind");
if (listen(sockfd, backlog) == -1)
    errExit("listen");
```

57.11. Наследование флагов и параметров сокета при выполнении вызова accept()

С дескрипторами и описаниями открытого файла могут быть связаны различные флаги и настройки (см. раздел 5.4). Кроме того, как было показано в разделе 57.9, сокету можно устанавливать разные параметры. Если речь идет о слушающем сокете, то наследуются ли все эти метаданные новым сокетом, который возвращается вызовом `accept()`? В данном разделе мы дадим подробный ответ.

В Linux новый файловый дескриптор, полученный из вызова `accept()`, не наследует следующие атрибуты.

- Флаги состояния, связанные с описанием открытого файла, например `O_NONBLOCK` или `O_ASYNC`. Для их включения или выключения можно использовать операцию `F_SETFL` вызова `fcntl()` (см. раздел 5.3).
- Флаги файлового дескриптора. В эту категорию входит лишь один флаг, `FD_CLOEXEC` (описанный в разделе 27.4), и для его изменения тоже можно применять операцию `F_SETFL` вызова `fcntl()`.
- Атрибуты файлового дескриптора `F_SETOWN` (идентификатор процесса-владельца) и `F_SETSIG` (сгенерированный сигнал), связанные с вводом/выводом на основе сигналов (см. раздел 59.3).

С другой стороны, новый дескриптор, возвращаемый вызовом `accept()`, наследует копии большинства параметров сокета, которые можно установить с помощью `setsockopt()` (см. раздел 57.9).

В стандарте SUSv3 нет никакой информации относительно подробностей, озвученных здесь, так что правила наследования для нового сокета, возвращаемого вызовом `accept()`, варьируются в зависимости от реализации. В частности, отдельные UNIX-системы обеспечивают наследование флагов состояния открытого файла, таких как `O_NONBLOCK` или `O_ASYNC`. При написании портируемых приложений может иметь смысл сбросить эти атрибуты для нового сокета.

57.12. Выбор между TCP и UDP

Учитывая, что протокол TCP, в отличие от UDP, обеспечивает надежную доставку данных, встает вопрос: «Для чего вообще может понадобиться UDP?» Ответ на него частично дается в главе 22 книги [Stevens et al., 2004]. Здесь же мы кратко рассмотрим некоторые факторы, способные склонить выбор в пользу UDP.

- UDP-сервер может принимать датаграммы от множества клиентов (и отвечать на них), не требуя при этом создания и разрыва соединений для каждого из них (то есть передача одного сообщения по UDP имеет меньшие накладные расходы, чем если бы использовался TCP).
- Протокол UDP может демонстрировать более высокую производительность при обмене простыми запросами и ответами, так как не требует установки и разрыва соединений. В приложении A книги [Stevens, 1996] отмечается, что в идеальной ситуации время обмена пакетами по протоколу TCP равно

  ```
  2 * RTT + SPT
  ```

  где `RTT` — время прохождения пакета туда-обратно (уходит на отправку запроса и получение ответа), а `SPT` — время, за которое сервер обрабатывает запрос (в глобальных сетях значение `SPT` может быть значительно меньше, чем `RTT`). В случае с протоколом UDP идеальный сценарий обмена пакетами выглядит как

  ```
  RTT + SPT
  ```

  Это на одно `RTT` меньше, чем при использовании TCP. Величина `RTT` между узлами, находящимися на большом расстоянии друг от друга (то есть на разных континентах) или разделенными большим количеством промежуточных маршрутизаторов, может достигать десятков секунд. Данное обстоятельство может сделать протокол UDP довольно привлекательным для некоторых видов сетевого взаимодействия. Хорошим примером может служить система DNS, позволяющая находить IP-адрес путем передачи между серверами всего лишь по одному пакету в каждом направлении.
- UDP-сокеты поддерживают широковещательную и многоадресную передачу данных. *Широковещательная* передача позволяет отправить датаграмму всем узлам, подключенным к сети, используя один и тот же порт. *Многоадресная* работает похожим образом, но позволяет передать датаграмму определенному набору узлов. Больше подробностей можно найти в главах 21 и 22 книги [Stevens et al., 2004].
- Определенные виды приложений (например, серверы потокового вещания видео и аудио) могут функционировать на приемлемом уровне и без гарантий, предоставляемых протоколом TCP. К тому же задержки, связанные с восстановлением потерянных сегментов, могут привести к неприемлемому запаздыванию данных (для потокового мультимедийного вещания задержка может быть хуже потери небольшой части потока). В связи с этим для подобных приложений часто предпочитают задействовать протокол UDP в сочетании с нестандартными стратегиями восстановления пакетов, которые периодически могут теряться.

Приложение, использующее UDP, но притом требующее определенного уровня надежности, должно само следить за доставкой данных. Обычно для этого применяются как минимум порядковые номера, подтверждение, повторная передача потерянных пакетов и обнаружение дублирования. Пример реализации такого подхода продемонстрирован в книге [Stevens et al., 2004]. Но если в вашем случае требуются более продвинутые возможности, такие как управление потоком и перегрузкой, то вам, вероятно, лучше воспользоваться протоколом TCP. Реализовать все эти функции поверх UDP будет непросто, и даже если у вас получится, вряд ли результат превзойдет протокол TCP.

57.13. Продвинутые возможности

Сокеты UNIX- и интернет-доменов поддерживают множество других возможностей, на которых мы не станем останавливаться подробно в этой книге. Часть из них будет кратко рассмотрена в данном разделе. Больше подробностей см. в книге [Stevens et al., 2004].

57.13.1. Внеканальные данные

Внеканальные данные — функция потоковых сокетов, которая позволяет отправителю назначать передаваемым пакетам высокий приоритет; то есть принимающая сторона может получить уведомление о наличии таких пакетов без необходимости читать все промежуточные данные в потоке. Эта возможность используется в таких программах, как `elnet`, `rlogin` и `ftp`, позволяя им отменять ранее переданные команды. Для отправки и получения внеканальных данных вызовам `send()` и `recv()` нужно указать флаг `MSG_OOB`. Когда сокет получает уведомление о наличии таких данных, ядро генерирует для его владельца (обычно это процесс, применяющий сокет) сигнал `SIGURG`, как при операции `F_SETOWN` для вызова `fcntl()`.

Протокол TCP позволяет делать внеканальными данные объемом не больше одного байта за раз. Если отправитель передает новый внеканальный байт до того, как получатель обработал предыдущий, уведомление о ранее посланном байте теряется.

> Ограничение размера внеканальных данных одним байтом является свидетельством несоответствия между универсальной внеканальной моделью программного интерфейса сокета и конкретной реализацией режима важности. Последний был затронут в подразделе 57.6.1, когда мы рассматривали формат TCP-сегментов. Чтобы уведомить о наличии важных (внеканальных) данных, протокол TCP устанавливает бит URG в TCP-заголовке и присваивает соответствующему полю указатель на эти данные. Однако TCP не может сообщить о длине байтовой последовательности, вследствие чего объем важных данных считается равным одному байту.
>
> Информацию о важных данных в протоколе TCP можно найти в документе RFC 793.

В ряде систем (в их число не входит Linux) внеканальные данные поддерживаются потоковыми сокетами домена UNIX.

Применение внеканальных данных в наши дни нежелательно и в некоторых обстоятельствах может оказаться ненадежным (см. [Gont & Yourtchenko, 2009]). Альтернативой является использование двух потоковых сокетов. Один из них занимается обычным взаимодействием, а второй отвечает за обмен высокоприоритетной информацией. Для мониторинга обоих каналов приложение может применять одну из методик, описанных в главе 59. Такой подход позволяет устанавливать приоритет для данных, объем которых превышает один байт. Кроме того, его можно задействовать для потоковых сокетов в любом домене (в том числе UNIX).

57.13.2. Системные вызовы sendmsg() и recvmsg()

Наиболее универсальными операциями ввода/вывода для сокетов являются системные вызовы `sendmsg()` и `recvmsg()`. Первый вобрал в себя все возможности вызовов `write()`, `send()` и `sendto()`; второй способен заменить вызовы `read()`, `recv()` и `recvfrom()`. Кроме того, `sendmsg()` и `recvmsg()` позволяют делать следующее.

- Выполнять векторный ввод/вывод по примеру `readv()` и `writev()` (см. раздел 5.7). При использовании вызова `sendmsg()` для векторного вывода через датаграммный сокет (или вызова `writev()` в сочетании с подключенным датаграммным сокетом)

генерируется единственная датаграмма. Аналогично вызов `recvmsg()` (и `readv()`) позволяет выполнить векторный ввод, разбивая единую датаграмму на несколько буферов в пользовательском пространстве.

- Передавать сообщения со вспомогательными (или управляющими) данными, относящимися к определенному домену. Вспомогательные данные могут быть переданы как через потоковые, так и через датаграммные сокеты. Некоторые примеры их использования представлены ниже.

57.13.3. Передача файловых дескрипторов

С помощью вызовов `sendmsg()` и `recvmsg()` и сокета в домене UNIX между двумя локальными процессами можно передавать вспомогательные данные, содержащие файловые дескрипторы. Это могут быть дескрипторы любого типа, например те, что возвращаются вызовам `open()` или `pipe()`. В качестве примера, имеющего более прямое отношение к этой главе, можно привести следующий сценарий. Главный сервер принимает клиентское соединение, используя слушающий TCP-сокет, и передает полученный дескриптор одному из своих дочерних процессов, входящих в состав серверного пула (см. раздел 56.4), который и ответит на клиентский запрос.

И хотя данную процедуру обычно называют передачей дескрипторов, на самом деле между процессами передается ссылка на один и тот же дескриптор (см. рис. 5.2). Номер файлового дескриптора на принимающей стороне обычно отличается от номера, используемого отправителем.

Пример передачи файловых дескрипторов приводится в файлах scm_rights_send.c и scm_rights_recv.c внутри подкаталога sockets, предоставленных с исходным кодом к данной книге.

57.13.4. Получение учетных данных отправителя

Еще одним примером использования вспомогательных данных является получение учетной информации через сокет домена UNIX. Такая информация состоит из идентификаторов пользователя, группы и процесса-отправителя. Отправляющая сторона может указать реальные, действующие или сохраненные идентификаторы. Это позволяет принимающему процессу аутентифицировать отправителя, находящегося в той же системе. Дополнительные подробности см. на страницах `socket(7)` и `unix(7)` руководства.

Передача учетных данных отправителя (в отличие от информации о файле) не предусмотрена стандартом SUSv3. Помимо Linux эту возможность поддерживают некоторые современные реализации BSD (передающие более подробную информацию, чем Linux) и несколько других систем семейства UNIX. Подробности о передаче учетных данных в FreeBSD описаны в книге [Stevens et al., 2004].

В Linux привилегированный процесс может подменять идентификаторы пользователя, группы и процесса, которые передаются в качестве учетных данных; для этого он должен поддерживать возможности `CAP_SETUID`, `CAP_SETGID` и, соответственно, `CAP_SYS_ADMIN`.

Пример передачи учетных данных приводится в файлах scm_cred_send.c и scm_cred_recv.c внутри подкаталога sockets, которые предоставлены с исходным кодом к данной книге.

57.13.5. Последовательный обмен пакетами

Сокеты с поддержкой последовательного обмена пакетами объединяют в себе возможности потоковых и датаграммных сокетов.

- Как и потоковые сокеты, они ориентированы на работу с соединениями, которые устанавливаются тем же способом — с помощью вызовов `bind()`, `listen()`, `accept()` и `connect()`.
- Как и датаграммные сокеты, они способны различать границы между сообщениями. Операция чтения из такого сокета возвращает ровно одно сообщение (записанное удаленной стороной). Если длина сообщения превышает размер буфера, предоставленного вызывающим процессом, лишние байты теряются.
- По аналогии с потоковыми сокетами (но в отличие от датаграммных) взаимодействие является надежным. Сообщения передаются адресату без ошибок, в том же порядке, в котором были отправлены, без дубликатов и с гарантией доставки (при условии отсутствия перегрузки сети или системных/программных сбоев).

Сокеты с поддержкой последовательного обмена пакетами создаются путем передачи аргументу `type` вызова `socket()` значения `SOCK_SEQPACKET`.

Изначально в Linux, как и в большинстве других UNIX-систем, не поддерживались сокеты такого типа, вне зависимости от домена. Однако в ядре 2.6.4 появилась поддержка сокетов `SOCK_SEQPACKET` в домене UNIX.

В интернет-домене эти сокеты поддерживаются исключительно протоколом SCTP (который описан в следующем разделе). То есть в UDP и TCP их поддержка отсутствует.

Последовательный обмен пакетами очень похож на работу потоковых сокетов (за исключением соблюдения границ сообщений), так что мы не станем приводить пример его использования.

57.13.6. Протоколы транспортного уровня SCTP и DCCP

SCTP и DCCP — относительно новые протоколы транспортного уровня, которые в будущем имеют шанс стать довольно распространенными.

SCTP (от англ. Stream Control Transmission Protocol — «протокол передачи с управлением потоком»; www.sctp.org) является протоколом общего назначения, хотя разрабатывался с упором на обмен телефонными сигналами. В отличие от TCP, он соблюдает границы отдельных сообщений. Одна из его отличительных черт — поддержка многопоточной передачи, что позволяет использовать несколько логических потоков данных в рамках одного соединения.

Протокол SCTP описан в книгах [Stewart & Xie, 2001], [Stevens et al., 2004], а также в документах RFC 4960, 3257 и 3286. Он доступен в Linux, начиная с версии ядра 2.6. Подробные сведения о его реализации можно найти на lksctp.sourceforge.net.

В предыдущих главах, посвященных программному интерфейсу сокетов, мы исходили из того, что потоковые сокеты в интернет-домене работают по протоколу TCP. Но на самом деле для потоковых сокетов существует альтернатива — протокол SCTP, который можно указать следующим образом:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
```

Начиная с версии 2.6.14 ядро Linux поддерживает новый датаграммный протокол DCCP (Datagram Congestion Control Protocol). Как и TCP, он предоставляет средства контроля перегрузок (устраняя необходимость в реализации подобных механизмов на уровне приложения), защищающие сеть от перегрузки слишком активной передающей стороной (принцип их работы был рассмотрен в подразделе 54.6.3 при описании протокола TCP). Но, в отличие от TCP (и по аналогии с UDP), протокол DCCP не обеспечивает надежную доставку и упорядоченность пакетов; благодаря этому при-

ложения, которым не нужны такие возможности, могут избежать связанных с ними задержек. Информацию о протоколе DCCP можно найти на www.read.cs.ucla.edu/dccp/ и в документах RFC 4336 и 4340.

57.14. Резюме

В различных ситуациях при выполнении операций ввода/вывода с потоковыми сокетами можно столкнуться с частичным чтением или записью. Мы продемонстрировали реализацию двух функций, `readn()` и `writen()`, которые гарантируют чтение или запись всех данных в буфере.

Системный вызов `shutdown()` предоставляет более тонкий контроль над процедурой разрыва соединения. С его помощью можно принудительно закрыть один или оба канала двунаправленного потока взаимодействия; при этом не важно, существуют ли другие файловые дескрипторы, ссылающиеся на наш сокет.

Вызовы `recv()` и `send()` — аналоги операций `read()` и `write()` и могут быть использованы для выполнения ввода/вывода через сокет. Их особенностью является дополнительный аргумент `flags`, влияющий на различные аспекты ввода/вывода, относящиеся к сокетам.

Системный вызов `sendfile()` обеспечивает эффективное копирование содержимого файла в сокет. Эта эффективность достигается за счет предотвращения копирования данных в пользовательскую память и обратно, выполняемого вызовами `read()` и `write()`.

Системные вызовы `getsockname()` и `getpeername()` извлекают локальный адрес, к которому привязан сокет, и, соответственно, адрес удаленной стороны, к которой подключен этот сокет.

Мы рассмотрели ряд подробностей работы протокола TCP, включая его состояния и диаграмму их переходов, а также процедуры установки и разрыва соединений. Одновременно было показано, почему состояние `TIME_WAIT` играет важную роль в обеспечении надежности протокола TCP. И хотя в случае перезапуска сервера `TIME_WAIT` может привести к ошибке «этот адрес уже занят», мы объяснили, как избежать такой ситуации с помощью параметра сокета `SO_REUSEADDR` и в то же время позволить данному состоянию выполнять свою функцию.

Команды `netstat` и `tcpdump` могут пригодиться для мониторинга и отладки приложений, которые используют сокеты.

Системные вызовы `getsockopt()` и `setsockopt()` позволяют извлекать и изменять параметры, влияющие на работу сокетов.

В Linux новый сокет, создаваемый с помощью вызова `accept()`, не наследует флаги состояния открытого файла слушающего сокета, а также флаги и атрибуты файлового дескриптора, связанные с вводом/выводом, основанным на сигналах. Хотя параметры сокета все же наследуются. Мы отметили, что стандарт SUSv3 никак не оговаривает это поведение, вследствие чего оно может варьироваться в зависимости от реализации.

В отличие от TCP, протокол UDP не обладает механизмами обеспечения надежности, но, как мы могли убедиться, имеет ряд достоинств, которые делают его более подходящим для отдельных приложений.

В завершение мы кратко рассмотрели несколько продвинутых возможностей сокетов.

Дополнительная информация

Ознакомьтесь с источниками, приведенными в разделе 55.14.

57.15. Упражнения

57.1. Представьте, что программа из листинга 57.2 (`is_echo_cl.c`) была модифицирована и теперь вместо применения вызова `fork()` для создания потомков, работающих параллельно, используется всего один процесс, который сначала копирует свой стандартный ввод в сокет, а затем считывает ответ сервера. С какой проблемой можно столкнуться при работе с этим клиентом (см. рис. 54.8)?

57.2. Реализуйте вызов `pipe()` по примеру `socketpair()`. Задействуйте вызов `shutdown()`, чтобы сделать итоговый канал однонаправленным.

57.3. Реализуйте замену `sendfile()` с применением вызовов `read()`, `write()` и `lseek()`.

57.4. Напишите программу на основе вызова `getsockname()`, демонстрирующую, что сокет, для которого операция `listen()` выполняется без предварительного вызова `bind()`, привязывается к динамическому порту.

57.5. Напишите клиентскую и серверную программы, позволяющие выполнять произвольные консольные команды на удаленном компьютере. (Если вы не собираетесь реализовывать в этом приложении никаких механизмов безопасности, то следует сделать так, чтобы сервер работал от имени обычной учетной записи и не мог причинить существенного вреда в случае использования злоумышленниками.) Клиент должен запускаться с помощью двух аргументов командной строки:

```
$ ./is_shell_cl server-host 'some-shell-command'
```

Подключившись к серверу, клиент отправляет ему заданную команду, после чего закрывает свой записывающий канал сокета путем вызова `shutdown()`, чтобы на другом конце можно было увидеть конец файла. Серверу следует обрабатывать каждое входящее соединение с помощью отдельного дочернего процесса (то есть параллельно). Каждый потомок должен прочитать данные из своего сокета (пока не столкнется с завершением файла) и затем запустить командную оболочку для выполнения соответствующей команды. Несколько подсказок:

- за пример запуска консольных команд возьмите реализацию вызова `system()` из раздела 27.7;
- используйте вызов `dup2()`, чтобы продублировать дескриптор сокета для стандартных потоков `stdout` и `stderr` — благодаря этому запущенная команда автоматически будет записывать свой вывод в сокет.

57.6. В подразделе 57.13.1 отмечалось: в качестве альтернативы внеканальным данным между клиентом и сервером можно было бы создать два TCP-соединения: одно — для обмена обычной информацией, а другое — для высокоприоритетных данных. Напишите клиентскую и серверную программы, которые реализуют этот принцип. Вот несколько подсказок.

- Сервер должен каким-то образом знать, какие два сокета принадлежат клиенту. Этого можно добиться, создав на клиентской стороне слушающий сокет и привязав его к динамическому порту (с номером 0). Получив номер своего динамического порта (с помощью вызова `getsockname()`), клиент соединяет другой свой сокет со слушающим сокетом сервера и отправляет сообщение, содержащее номер динамического порта клиента. Затем клиент ждет, чтобы сервер мог подключиться к его слушающему сокету и установить соединение для «приоритетных» данных, направленное в обратную сторону (сервер может получить IP-адреса клиента во время выполнения вызова `accept()` для обычного соединения).
- Реализуйте некий механизм безопасности, который не дает несанкционированному процессу подключиться к слушающему порту клиента. Для этого клиентская сторона могла бы отправлять серверу определенный код (то есть какое-то уникальное

сообщение наподобие cookie), используя обычный сокет. Затем сервер мог бы возвращать данный код через приоритетный сокет с целью предоставить клиенту возможность проверить его подлинность.

- Чтобы иметь возможность экспериментировать с передачей обычных и высокоприоритетных данных от клиента к серверу, вы должны предусмотреть в своем сервере мультиплексирование ввода из двух сокетов с помощью вызова `select()` или `poll()` (которые будут описаны в разделе 59.2).

58 Терминалы

Исторически сложилось так, что для получения доступа к системе UNIX использовался терминал, подключенный через последовательный порт (соединение RS-232). В те времена терминалы представляли собой мониторы на основе электронно-лучевой трубки (ЭЛТ), способные отображать символы и, в некоторых случаях, примитивную графику. Обычно такие мониторы имели черно-белый экран с 24 строчками и 80 столбцами. С точки зрения современных стандартов они были маленькими и дорогими. А еще раньше в качестве терминалов служили телетайпы. Последовательные порты также применялись для подключения других устройств, таких как принтеры и модемы, и для соединения компьютеров между собой.

> В первых системах UNIX терминал, подключенный к компьютеру через последовательный порт, был представлен в виде символьного устройства с именем вида /dev/ttyn (в Linux такие устройства являются виртуальными консолями). Аббревиатуру tty (от англ. teletype — «телетайп») часто задействуют для сокращенного обозначения терминала.

Изначально, особенно в первые годы существования UNIX, терминальные устройства не были стандартизованы; это значит, что для выполнения таких операций, как перемещение курсора в начало строки или в верхнюю часть экрана применялись разные последовательности символов. (Со временем некоторые фирменные реализации таких управляющих последовательностей, как, например, VT-100 компании Digital, стали стандартами ANSI — сначала де-факто, а затем и формально; тем не менее параллельно с ними продолжало существовать множество других разновидностей терминалов.) Нехватка стандартизации усложняла написание портируемых программ, полагавшихся на возможности терминала. Одним из первых примеров подобных программ был редактор `vi`. В ответ на такую ситуацию были разработаны базы данных `termcap` и `terminfo` (описаны в [Strang et al., 1988]), которые инкапсулировали выполнение различных экранных операций для широкого множества терминалов. Можно также вспомнить библиотеку curses с аналогичными функциями [Strang, 1986].

В наши дни традиционный терминал вышел из широкого обихода. Стандартным интерфейсом к современным UNIX-системам является оконный диспетчер X Window System, выводящий информацию на высокопроизводительном графическом мониторе. (Все возможности традиционного терминала в X Window System можно заменить одним оконным приложением — `xterm` или подобным; именно тот факт, что пользователи таких терминалов имели всего лишь одно «окно» в систему, стал движущей силой для разработки механизмов управления заданиями, описанных в разделе 34.7.) Аналогично многие устройства, которые ранее подключались к компьютеру напрямую (такие как принтеры), стали более «умными» и теперь доступны по сети.

Все вышесказанное является преамбулой к следующему факту: потребность в программировании терминальных устройств возникает не так часто, как когда-то. Поэтому в данной главе мы сосредоточимся на аспектах программирования, связанных с программными эмуляторами терминалов (такими как `xterm` и ему подобными). Последовательные порты будут затронуты лишь вскользь; источники дополнительной информации о них указаны в конце главы.

58.1. Краткий обзор

И традиционные терминалы, и их эмуляторы полагаются на специальный драйвер, выполняющий операции ввода/вывода с соответствующим устройством (в случае с эмулятором таким устройством является псевдотерминал, который будет описан в главе 60). Влиять на различные аспекты работы этого драйвера можно с помощью функций, описанных в настоящей главе.

Драйвер терминала поддерживает два режима ввода.

- *Канонический*. В данном режиме ввод терминала обрабатывается построчно, при этом включена возможность редактирования строк. В конце каждой строчки находится символ перехода на новую строку, генерируемый при нажатии клавиши Enter. Операция `read()` возвращает по одной строке за раз и только когда ввод содержит целую строку. (Если длина строки превышает объем данных, запрошенных вызовом `read()`, то оставшиеся байты будут доступны при следующей операции чтения.) Этот режим ввода используется по умолчанию.
- *Неканонический*. Ввод терминала не делится на строки. Такие программы, как `vi`, `more` и `less`, переключаются в данный режим, чтобы иметь возможность считывать отдельные символы, не требуя от пользователя нажатия клавиши Enter.

Драйвер терминала также интерпретирует ряд специальных символов, таких как прерывание (обычно Ctrl+C) и конец файла (обычно Ctrl+D). Это может приводить к генерированию сигнала для фоновой группы процессов или выполнению некоего условия, которого ожидает программа, считывающая данные из терминала. Программы, переключающие терминал в неканонический режим, обычно также отключают часть специальных символов (или все).

Драйвер терминала управляет двумя очередями (см. рис. 58.1): одна — для ввода символов, передающихся из устройства в считывающий процесс (их может быть несколько), а другая — для вывода символов, которые процесс передает терминалу. Если включена функция эхо-контроля, драйвер автоматически добавляет копию любого введенного символа в конец исходящей очереди; это позволяет отображать в терминале вводимый текст.

Стандарт SUSv3 оговаривает ограничение MAX_INPUT, позволяющее системе устанавливать максимальную длину входящей очереди терминала. Еще одно ограничение, MAX_CANON, определяет максимальное количество байтов, которое может содержаться в одной строке в каноническом режиме. В Linux вызовы sysconf(_SC_MAX_INPUT) и sysconf(_SC_MAX_CANON) возвращают значение 255. Однако ни одно из этих ограничений не используется самим ядром, имеющим собственное ограничение для размера входящей очереди — 4096 байт. Аналогичное ограничение существует и для исходящей очереди, но оно не влияет на работу приложений, поскольку в случае если процесс генерирует вывод быстрее, чем драйвер может его обработать, то ядро приостанавливает выполнение данного процесса, пока в исходящей очереди опять не появится свободное место.

В Linux доступен вызов ioctl(fd, FIONREAD, &cnt), позволяющий получить количество непрочитанных байтов во входящей очереди терминала, на который указывает файловый дескриптор fd. Эта возможность не предусмотрена стандартом SUSv3.

58.2. Извлечение и изменение атрибутов терминала

Для извлечения и изменения атрибутов терминала предусмотрены функции `tcgetattr()` и `tcsetattr()`.

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

Обе функции возвращают `0` при успешном завершении или `-1` при ошибке

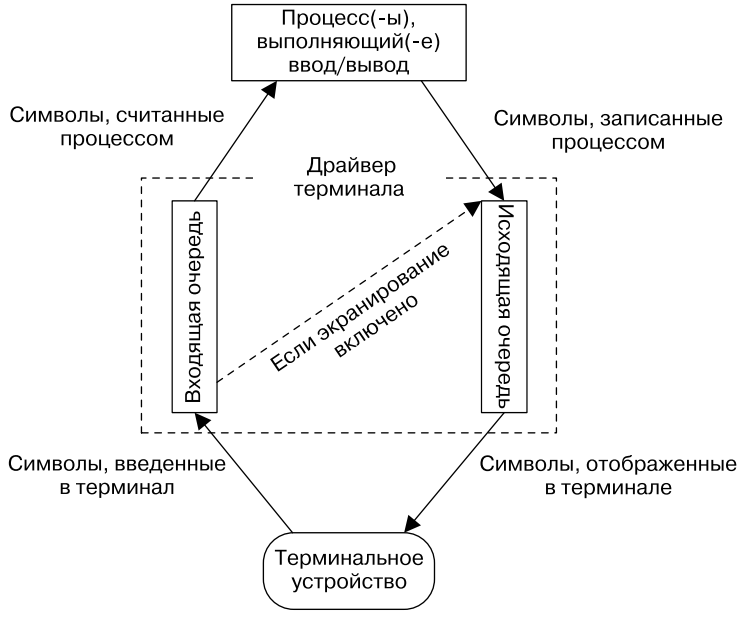


Рис. 58.1. Входящая и исходящая очереди терминального устройства

Аргумент `fd` представляет собой файловый дескриптор, который должен ссылаться на терминал (в противном случае эти функции завершаются ошибкой `ENOTTY`).

Аргумент `termios_p` является указателем на структуру `termios`, хранящую атрибуты терминала:

```
struct termios {
    tcflag_t c_iflag;         /* Флаги ввода */
    tcflag_t c_oflag;         /* Флаги вывода */
    tcflag_t c_cflag;         /* Управляющие флаги */
    tcflag_t c_lflag;         /* Локальные режимы */
    cc_t     c_line;          /* Порядок следования строк (нестандартное) */
    cc_t     c_cc[NCCS];      /* Специальные символы терминала */
    speed_t  c_ispeed;        /* Скорость ввода (нестандартное; не используется) */
    speed_t  c_ospeed;        /* Скорость вывода (нестандартное; не используется) */
};
```

Первые четыре поля структуры `termios` — битовые маски (тип данных `tcflag_t` представлен целым числом подходящего размера), содержащие флаги, которые влияют на различные аспекты работы драйвера терминала:

- `c_iflag` содержит флаги, управляющие вводом терминала;
- `c_oflag` содержит флаги, управляющие выводом терминала;

- `c_cflag` содержит флаги, связанные с аппаратным управлением последовательного порта;
- `c_lflag` содержит флаги, управляющие пользовательским интерфейсом для терминального ввода.

Все флаги, используемые в приведенных выше полях, будут перечислены в табл. 58.2 далее.

Поле `c_line` определяет порядок следования строк для текущего терминала. С целью поддержки эмуляторов терминала оно всегда равно `N_TTY`; это новый режим, который является частью кода ядра, занимающегося работой с терминалом и реализующего обработку ввода/вывода в каноническом режиме. Изменение данного поля имеет смысл при программировании последовательных портов.

Массив `c_cc` содержит специальные символы терминала (*прерывания*, *приостановку* и т. д.), а также поле, отвечающее за работу неканонического режима ввода. Тип данных `cc_t` представляет собой беззнаковое целое число, подходящее для хранения этих значений, а константа `NCCS` определяет количество элементов в данном массиве. Специальные символы терминала описаны в разделе 58.4.

Поля `c_ispeed` и `c_ospeed` не используются в Linux (и не предусмотрены стандартом SUSv3). О том, как эта ОС хранит данные о скорости последовательного порта, мы расскажем в разделе 58.7.

При изменении атрибутов терминала с помощью вызова `tcsetattr()` аргумент `optional_actions` определяет, когда именно эти изменения вступят в силу. Он может принимать одно из следующих значений.

- `TCSANOW` — изменение применяется немедленно.
- `TCSADRAIN` — изменение вступает в силу после передачи терминалу текущего содержимого исходящей очереди. Обычно этот флаг указывают при изменениях, влияющих на вывод терминала, чтобы не затронуть данные, которые уже попали в очередь, но еще не были выведены на экран.
- `TCSAFLUSH` — действуют те же правила, что и для `TCSADRAIN`, но с одним уточнением: любой ввод, находящийся в очереди на момент применения изменений, отклоняется. Это может пригодиться, например, при чтении пароля, когда мы хотим отключить эхо-контроль терминала и предотвратить отображение вводимых символов.

Общепринятым (и рекомендованным) способом изменения атрибутов терминала является извлечение структуры `termios` с копией текущих параметров с функции `tcgetattr()`, изменение интересующих атрибутов и затем запись обновленной информации обратно в драйвер с помощью функции `tcsetattr()` (этот подход гарантирует передачу данной функции полностью инициализированной структуры). Например, чтобы отключить эхо-контроль, можно воспользоваться следующим кодом:

```
struct termios tp;

if (tcgetattr(STDIN_FILENO, &tp) == -1)
    errExit("tcgetattr");
tp.c_lflag &= ~ECHO;
if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
    errExit("tcsetattr");
```

Функция `tcsetattr()` завершается успешно, если удалось изменить хотя бы *один* из указанных атрибутов; ошибка возникает, только если изменения были отклонены *полностью*. Это значит, что после изменения нескольких атрибутов имеет смысл выполнить еще один вызов `tcgetattr()` с целью извлечь обновленные параметры терминала и сравнить их со своим запросом.

В подразделе 34.7.2 отмечалось: если вызов tcsetattr() выполняется в фоновой группе процессов, драйвер терминала приостанавливает эту группу, посылая ей сигнал SIGTTOU; следовательно, вызов tcsetattr() завершается ошибкой EIO, если его сделать из осиротевшей группы процессов. То же самое касается и некоторых других функций, описанных в данной главе, включая tcflush(), tcflow(), tcsendbreak() и tcdrain().

В старых версиях UNIX доступ к атрибутам терминала осуществлялся с помощью вызова ioctl(). Как и часть других функций, описанных в этой главе, tcgetattr() и tcsetattr() впервые появились в стандарте POSIX и были призваны решить проблему с третьим аргументом вызова ioctl(), тип которого невозможно проверить. В Linux, как и во многих других реализациях UNIX, функции tcgetattr() и tcsetattr() являются обертками поверх ioctl().

58.3. Команда stty

Команда `stty` является консольным аналогом функций `tcgetattr()` и `tcsetattr()`, позволяя просматривать и изменять атрибуты терминала из командной строки. Она может пригодиться при мониторинге, отладке или отмене изменений, вносимых в атрибуты терминала какой-либо программой.

Текущее состояние атрибутов терминала можно просмотреть с помощью следующей команды (в данном случае мы имеем дело с виртуальной консолью):

```
$ stty -a
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

Первая строка в вышеприведенном выводе содержит скорость последовательного порта (в битах в секунду), размер окна терминала и порядок следования строк в числовом виде (0 соответствует режиму перехода на новую строку `N_TTY`).

В следующих трех строчках находятся параметры различных специальных символов терминала. Так, `^C` означает Ctrl+C и т. д. Строка `<undef>` говорит о том, что соответствующий специальный символ еще не задан. Значения `min` и `time` относятся к неканоническому режиму ввода; мы вернемся к ним в подразделе 58.6.2.

Остальные строки содержат параметры разных флагов из полей `c_cflag`, `c_iflag`, `c_oflag` и `c_lflag` структуры `termios` (именно в таком порядке). Флаги, имена которых начинаются с дефиса (`-`), отключены.

Если запустить команду `stty` без аргументов командной строки, она выведет только скорость последовательного порта, порядок следования строк и любые другие параметры с необычными значениями.

Чтобы изменить настройки специальных символов терминала, можно воспользоваться командами наподобие следующей:

```
$ stty intr ^L        Назначаем Ctrl+L в качестве символа прерывания
```

Управляющий символ в конце можно указывать в разных форматах:

- сочетание двух символов, первым из которых является карет (^), а вторым — соответствующая буква (как показано выше);

- восьмеричное или шестнадцатеричное число (например, `014` или `0xC`);
- сам символ, введенный как есть.

Если выбрать последний вариант, то перед вводом специального символа, который затем будет интерпретирован драйвером терминала, следует ввести *маркер литерала* (обычно это Ctrl+V):

```
$ stty intr                 Ctrl+V Ctrl+L
```

Для большей наглядности примера мы вставили пробел между Ctrl+V и Ctrl+L, хотя на самом деле после ввода Ctrl+V не должно быть никаких пробельных символов.

Специальные символы терминала не обязательно должны быть управляющими (хотя это довольно редкий случай):

```
$ stty intr q               Делаем q символом прерывания
```

Естественно, в этом случае мы теряем возможность использовать клавишу q для ввода соответствующей буквы.

Чтобы изменить флаг терминала, такой как `TOSTOP`, можно воспользоваться следующими командами:

```
$ stty tostop               Включаем флаг TOSTOP
$ stty -tostop              Выключаем флаг TOSTOP
```

Иногда при написании кода, который изменяет атрибуты терминала, с вашей программой может произойти сбой, в результате чего терминал продолжит выводить информацию, но перестанет быть доступным для ввода. Если речь идет об эмуляторе, то можно просто закрыть соответствующее окно и открыть новое. Но следующая последовательность символов также может привести флаги и специальные символы терминала к нормальному состоянию:

```
Ctrl+J stty sane Ctrl+J
```

Нажатие сочетания клавиш Ctrl+J генерирует символ перехода на новую строку (код ASCII 10). Его можно использовать в случаях, когда драйвер терминала больше не выполняет переход на новую строку при нажатии клавиши Enter (код ASCII 13). После первого нажатия Ctrl+J мы оказываемся в свежей строке, не содержащей никаких других символов. Иногда это бывает не совсем очевидно, если, например, в терминале был отключен эхо-контроль.

Команда `stty` работает с терминалом, указанным в стандартном вводе. Параметр `-F` позволяет получать и устанавливать атрибуты для внешнего терминала, а не для того, в котором запущена команда `stty` (при условии наличия подходящих прав доступа):

```
$ su                        Для доступа к терминалу другого пользователя нужны
                            особые привилегии
Password:
# stty -a -F /dev/tty3      Извлекаем атрибуты терминала /dev/tty3
Для краткости опускаем вывод
```

Параметр `-F` команды `stty` не является стандартным и доступен только в Linux. Во многих других реализациях UNIX команда `stty` всегда работает с терминалом, на который указывает стандартный ввод; чтобы это изменить, приходится использовать альтернативный подход (работающий и в Linux):

```
# stty -a < /dev/tty3
```

58.4. Специальные символы терминала

В табл. 58.1 перечислены специальные символы, распознаваемые драйвером терминала в Linux. В первых двух столбцах указаны названия символов и соответствующие константы, которые могут применяться в качестве подстрочных значений в массиве `c_cc` (как видите, эти константы отличаются от названий символов только префиксом `V`). Символы `CR` и `NL` не имеют соответствующих подстрочных аналогов в массиве `c_cc`, так как их значения нельзя менять.

В четвертом столбце указаны значения специальных символов, которые обычно используются по умолчанию. Можно не только менять значения специальных символов, но и полностью их отключать; для этого им достаточно назначить результат выполнения вызова `fpathconf(fd, _PC_VDISABLE)`, где `fd` — файловый дескриптор, ссылающийся на терминал (в большинстве UNIX-систем данный вызов возвращает `0`).

На поведение каждого специального символа влияют различные флаги, которые указаны в полях структуры `termios` в виде битовых масок (см. раздел 58.5). Эти флаги перечислены во втором столбце с конца.

В последнем столбце отмечается, входят ли данные символы в стандарт SUSv3, хотя большинство из них поддерживается всеми реализациями UNIX.

Таблица 58.1. Специальные символы терминала

Символ	Константа c_cc	Описание	Значение по умолчанию	Связанные битовые флаги	SUSv3
CR	(нет)	Разрыв строки (возврат каретки)	^M	ICANON, IGNCR, ICRNL, OPOST, OCRNL, ONOCR	*
DISCARD	VDISCARD	Отмена вывода	^O	Не реализованы	
EOF	VEOF	Конец файла	^D	ICANON	*
EOL	VEOL	Конец строки		ICANON	*
EOL2	VEOL2	Альтернативный конец строки		ICANON, IEXTEN	
ERASE	VERASE	Символ удаления	^?	ICANON	*
INTR	VINTR	Прерывание (SIGINT)	^C	ISIG	*
KILL	VKILL	Удаление строки	^U	ICANON	*
LNEXT	VLNEXT	Маркер литерала	^V	ICANON, IEXTEN	
NL	(нет)	Новая строка	^J	ICANON, INLCR, ECHONL, OPOST, ONLCR, ONLRET	*
QUIT	VQUIT	Выход (SIGQUIT)	^\	ISIG	
REPRINT	VREPRINT	Повторный вывод введенной строки	^R	ICANON, IEXTEN, ECHO	*
START	VSTART	Начало вывода	^Q	IXON, IXOFF	*
STOP	VSTOP	Остановка вывода	^S	IXON, IXOFF	*

Символ	Константа c_cc	Описание	Значение по умолчанию	Связанные битовые флаги	SUSv3
SUSP	VSUSP	Временная остановка (SIGTSTP)	^Z	ISIG	*
WERASE	VWERASE	Удаление слова	^W	ICANON, IEXTEN	

На следующих страницах приводится более подробное описание специальных символов терминала. Стоит отметить, что если драйвер терминала производит особую интерпретацию, то любой из этих символов (за исключением `CR`, `EOL`, `EOL2` и `NL`) отклоняется (то есть не передается считывающему процессу).

CR

Это символ *разрыва строки*. Он передается считывающему процессу. В каноническом режиме (флаг `ICANON`) с установленным флагом `ICRNL` (когда `CR` привязан к `NL`) данный символ перед отправкой в стандартный ввод переводится в символ новой строки (код ASCII 10 или `^]`). Если установлен флаг `IGNCR` (когда `CR` игнорируется), то этот символ не воспринимается при вводе (в таком случае строка должна завершаться настоящим символом новой строки). При выводе символ `CR` заставляет терминал перемещать курсор в начало строки.

DISCARD

Это символ *отмены вывода*. Он определен в массиве `c_cc`, но в Linux не имеет никакого эффекта. В некоторых других UNIX-системах его ввод приводит к отмене программного вывода. Символ работает как переключатель — его повторный ввод снова включит отображение вывода. Такая возможность полезна в ситуациях, когда программа генерирует большой объем данных и какую-то часть из них нужно пропустить (функция была намного более востребованной в традиционных терминалах с относительно низкой скоростью последовательного порта, где нельзя было использовать отдельные окна). Этот символ не передается считывающему процессу.

EOF

Это *конец файла* в каноническом режиме (обычно Ctrl+D). Ввод данного символа в начале строки приводит к обнаружению конца файла процессом, выполняющим чтение из терминала (то есть вызов `read()` возвращает `0`). Если ввести его в каком-то другом месте, то операция чтения немедленно завершится и вернет символы, которые были введены в текущей строке до этого момента. В обоих случаях сам символ `EOF` не передается считывающему процессу.

EOL и EOL2

Эти символы являются *дополнительными разделителями строк*, которые ведут себя аналогично стандартному символу новой строки (`NL`) в каноническом режиме, завершая строку и делая ее доступной для считывающего процесса. По умолчанию данные символы не определены. Но если их определить, то они будут передаваться в считывающий процесс. Символ `EOL2` действует только при установленном флаге `IEXTEN` (*расширенная обработка ввода*).

Эти символы используются довольно редко. Одним из приложений, которое их применяет, является `telnet`. Устанавливая значения `EOL` или `EOL2` в качестве символа выхода (обычно Ctrl+] или, как вариант, знак тильда (~) во включенном режиме `rlogin`), `telnet`

получает возможность перехватывать отдельные символы, даже если чтение происходит в каноническом режиме.

ERASE

Ввод данного символа в каноническом режиме приводит к удалению предыдущего символа, введенного в той же строке. Ни ERASE, ни удаленный символ не передаются считывающему процессу.

INTR

Это символ *прерывания*. При включенном флаге ISIG (который позволяет передавать сигналы; включен по умолчанию) ввод INTR приводит к отправке сигнала *прерывания* (SIGINT) активной группе процессов (см. раздел 34.2). Сам символ не передается.

KILL

Это символ *удаления строки*. В каноническом режиме его ввод приводит к отмене текущей строки (то есть любые символы, введенные до сего момента, а также сам символ KILL не передаются считывающему процессу).

LNEXT

Это *маркер литерала*. В некоторых ситуациях специальные символы терминала нужно интерпретировать как обычные и передавать их считывающему процессу. Ввод LNEXT (обычно с помощью Ctrl+V) приводит к тому, что следующий символ воспринимается буквально, без какой-либо интерпретации со стороны драйвера терминала. Таким образом, для передачи считывающему процессу настоящего символа Ctrl+C (код ASCII 3) достаточно ввести по очереди Ctrl+V и Ctrl+C. Сам символ LNEXT при этом не передается. Он интерпретируется только в каноническом режиме с установленным флагом IEXTEN (*расширенная обработка ввода*; действует по умолчанию).

NL

Это символ *новой строки*. В каноническом режиме он завершает ввод строки, передающейся считывающему процессу, и добавляется в ее конец (в каноническом режиме символ CR обычно конвертируется в NL). Исходящий символ NL заставляет терминал переместить курсор на одну строку ниже. Если включить флаги OPOST и ONLCR (привязав NL к CR-NL), что происходит по умолчанию, то при выводе символ новой строки переводится в последовательность из CR плюс NL. (Сочетание флагов ICRNL и ONLCR означает: входящий символ CR преобразуется в NL, а затем выводится на экран в виде CR плюс NL.)

QUIT

Если установлен флаг ISIG (происходит по умолчанию), то ввод символа QUIT приводит к отправке активной группе процессов (см. раздел 34.2) сигнала *выхода* (SIGQUIT). Сам по себе этот символ не передается считывающему процессу.

REPRINT

Это символ *повторного вывода введенной строки*. В каноническом режиме с включенным флагом IEXTEN (происходит по умолчанию) его ввод приводит к повторному отображению в терминале текущей (и еще не полной) строчки ввода. Такая опция может оказаться полезной, если какая-то другая программа (например, wall(1) или write(1)) сгенерировала вывод, который нарушил отображение содержимого терминала. Сам символ REPRINT не передается считывающему процессу.

START и STOP

Это символы *начала* и *остановки* вывода, действующие при включенном флаге `IXON` (происходит по умолчанию; в некоторых терминалах данные символы не поддерживаются).

Ввод символа `STOP` останавливает вывод терминала. Сам он не передается считывающему процессу. Если флаг `IXOFF` включен, а входящая очередь терминала заполнена, то драйвер терминала автоматически отправляет символ `STOP`, чтобы приостановить ввод.

Ввод символа `START` заставляет терминал возобновить вывод в случае его предварительной остановки символом `STOP`. Символ `START` не передается считывающему процессу. Если флаг `IXOFF` включен (по умолчанию он выключен), а драйвер терминала ранее послал символ `STOP` из-за заполненной входящей очереди, то при появлении свободного места в этой очереди драйвер автоматически отправляет символ `START`.

При установленном флаге `IXANY` для возобновления вывода достаточно ввести любой символ, а не только `START` (данный символ точно так же не будет передан считывающему процессу).

Символы `START` и `STOP` используются для программного управления потоками в любых направлениях между компьютером и терминальным устройством. Одна из их функций — позволить пользователям останавливать и возобновлять терминальный вывод. Этот механизм включается с помощью флага `IXON`. Однако управление потоком в противоположном направлении (то есть контроль ввода с терминального устройства в компьютер, который включается путем флага `IXOFF`) тоже важен — например, когда соответствующим устройством является модем или другой компьютер. Управление потоком позволяет избежать потери данных в случае, если приложение слишком медленное для обработки ввода, а буферы ядра заполнены до отказа.

Если доступ к терминалу осуществляется на более высокой скорости, как это обычно происходит в современных условиях, то вместо программного управления потоком используется аппаратное (RTS/CTS). Его включение и выключение выполняются путем отправки сигналов по отдельным каналам последовательного порта (RTS означает request to send — «запрос на отправку», а CTS расшифровывается как clear to send — «разрешение отправки»).

SUSP

Это символ *приостановки*. Если установить флаг `ISIG` (происходит по умолчанию), то ввод данного символа вызывает отправку сигнала *приостановки терминала* (`SIGTSTP`) активному процессу (см. раздел 34.2). Сам символ не передается считывающему процессу.

WERASE

Это символ *удаления слова*. В каноническом режиме и с включенным флагом `IEXTEN` (происходит по умолчанию) его ввод удаляет все символы вплоть до начала предыдущего слова. Здесь словом считается последовательность из букв, цифр и знаков подчеркивания (в некоторых реализациях UNIX слова должны разделяться пробелами).

Другие специальные символы терминала

Другие UNIX-системы предоставляют дополнительные специальные символы, не вошедшие в табл. 58.1.

Система BSD поддерживает символы `DSUSP` и `STATUS`. Первый похож на `SUSP` (и обычно вводится нажатием Ctrl+Y), но приостанавливает группу процессов первого плана, только если те попытаются прочитать символ (то есть после прочтения всего предшествующего ввода). Символ `DSUSP` поддерживается некоторыми другими системами, основанными на BSD.

Символ STATUS (обычно вводится как Ctrl+T) заставляет ядро вывести данные о состоянии терминала (включая состояние активного процесса и количество израсходованного процессорного времени) и отправляет сигнал SIGINFO активной группе процессов. При желании процесс может перехватить этот сигнал и вывести дополнительные сведения (Linux предоставляет похожую возможность в виде так называемой *магической клавиши* SysRq; подробности см. в исходном файле ядра Documentation/sysrq.txt).

Системы, основанные на System V, поддерживают символ SWTCH, используемый для переключения между *уровнями командной оболочки* (применялись до появления системы управления заданиями).

Пример программы

В листинге 58.1 показано, как с помощью функций tcgetattr() и tcsetattr() можно изменить символ *прерывания* терминала. Эта программа позволяет задействовать для прерывания символ, числовой код которого указан в качестве аргумента командной строки; если аргумент отсутствует, то символ прерывания отключается.

Использование данной программы продемонстрировано в следующей сессии командной строки. Вначале мы применим для прерывания символ Ctrl+L (код ASCII 12), а затем проверяем изменения с помощью команды stty:

```
$ ./new_intr 12
$ stty
speed 38400 baud; line = 0;
intr = ^L;
```

Затем запускаем процесс, выполняющий команду sleep(1). Мы обнаружим, что нажатие Ctrl+C больше не приводит к завершению процесса и для этого нужно нажать Ctrl+L.

```
$ sleep 10
^C                                  Ctrl+C не работает и просто экранируется
Чтобы завершить sleep, нажмите Ctrl+L
```

Теперь выведем значение переменной среды $?, которое содержит код завершения последней команды:

```
$ echo $?
130
```

Как видите, код завершения процесса был равен 130. Из этого следует, что процесс был завершен с помощью сигнала с номером 130 – 128 = 2, то есть SIGINT.

Воспользуемся нашей командной строкой, чтобы отключить символ прерывания.

```
$ ./new_intr
$ stty                          Подтверждаем изменение
speed 38400 baud; line = 0;
intr = <undef>;
```

В итоге можно обнаружить, что ни Ctrl+C, ни Ctrl+L не генерируют сигнал SIGINT и для завершения программы нужно нажать Ctrl+\:

```
$ sleep 10
^C^L                    Ctrl+C и Ctrl+L просто экранируются
Нажмите Ctrl+\, чтобы сгенерировать SIGQUIT
Quit
$ stty sane             Возвращаем терминал к нормальному состоянию
```

Листинг 58.1. Изменение символа прерывания терминала

tty/new_intr.c

```
#include <termios.h>
#include <ctype.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct termios tp;
    int intrChar;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [intr-char]\n", argv[0]);

    /* Определяем новый параметр INTR из командной строки */
    if (argc == 1) {                                    /* Отключаем */
        intrChar = fpathconf(STDIN_FILENO, _PC_VDISABLE);
        if (intrChar == -1)
            errExit("Couldn't determine VDISABLE");
    } else if (isdigit((unsigned char) argv[1][0])) {
        intrChar = strtoul(argv[1], NULL, 0);
          /* Поддерживает шестнадцатеричные и восьмеричные значения */
    } else {                                            /* Литерал */
        intrChar = argv[1][0];
    }

    /* Получаем текущие параметры терминала, изменяем символ INTR
       и отправляем изменения драйверу терминала */
    if (tcgetattr(STDIN_FILENO, &tp) == -1)
        errExit("tcgetattr");
    tp.c_cc[VINTR] = intrChar;
    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
        errExit("tcsetattr");

    exit(EXIT_SUCCESS);
}
```

tty/new_intr.c

58.5. Флаги терминала

В табл. 58.2 перечислены параметры, управляемые каждым из четырех полей структуры `termios`, предназначенных для хранения флагов. Приведенные константы соответствуют одиночным битам. Исключение составляют битовые маски, способные вмещать несколько битов; они поддерживают сочетания значений, перечисленные в скобках. В столбце с названием SUSv3 указаны флаги, входящие в одноименный стандарт. Столбец «Включен» показывает стандартные параметры при входе в систему с помощью виртуальной консоли.

Многие командные оболочки с функцией редактирования текста предоставляют собственные механизмы изменения флагов, перечисленных в табл. 58.2. Это значит следующее: изменение данных параметров путем утилиты stty(1) может не повлиять на ввод консольных

команд. Чтобы этого избежать, следует отключить функцию редактирования в командной оболочке. Например, чтобы сделать это в оболочке bash, нужно запустить ее с параметром командной строки --noediting.

Таблица 58.2. Флаги терминала

Поле/флаг	Описание	Включен	SUSv3
c_iflag			
BRKINT	Сигнал прерывания (SIGINT) при условии BREAK	Да	*
ICRNL	Привязка CR к NL во время ввода	Да	*
IGNBRK	Игнорирование условия BREAK	Нет	*
IGNCR	Игнорирование CR при вводе	Нет	*
IGNPAR	Игнорирование символов с ошибками соответствия	Нет	*
IMAXBEL	Звуковой сигнал, когда входящая очередь терминала заполнена (не используется)	(Да)	
INLCR	Привязка NL к CR	Нет	*
INPCK	Проверка на соответствие ввода	Нет	*
ISTRIP	Удаление из вводимых символов старшего бита (с номером 8)	Нет	*
IUTF8	Ввод в кодировке UTF-8 (начиная с Linux 2.6.4)	Нет	
IUCLC	Привязка ввода нижнего регистра к верхнему (если включен флаг IEXTEN)	Нет	
IXANY	Возможность возобновить вывод с помощью любого символа	Нет	*
IXOFF	Возможность включить/выключить управление входящим потоком	Нет	*
IXON	Возможность включить/выключить управление исходящим потоком	Да	*
PARMRK	Маркировка ошибок соответствия (с помощью двух начальных байтов: 0377+0)	Нет	*
c_oflag			
BSDLY	Маска задержки клавиши возврата (BS0, BS1)	BS0	*
CRDLY	Маска задержки CR (CR0, CR1, CR2, CR3)	CR0	*
FFDLY	Маска задержки разрыва страницы (FF0, FF1)	FF0	*
NLDLY	Маска задержки новой строки (NL0, NL1)	NL0	*
OCRNL	Привязка CR к NL при выводе (см. также ONOCR)	Нет	*
OFDEL	Задействование DEL (0177) в качестве символа заполнения; в противном случае применяется NULL (0)	Нет	*
OFILL	Использование символов заполнения для задержки	Нет	*
OLCUC	Привязка вывода нижнего регистра к верхнему	Нет	
ONLCR	Привязка NL к CR-NL при выводе	Да	*
ONLRET	Исходит из того, что NL выполняет функцию символа CR (переход в начало строки)	Нет	*

Поле/флаг	Описание	Включен	SUSv3
ONOCR	Не выводит CR, если курсор находится в столбце 0 (в начале строки)	Нет	*
OPOST	Постобработка вывода	Да	*
TABDLY	Маска задержки горизонтальной табуляции (TAB0, TAB1, TAB2, TAB3)	TAB0	*
VTDLY	Маска задержки вертикальной табуляции (VT0, VT1)	VT0	*
c_cflag			
CBAUD	Маска скорости передачи (B0, B2400, B9600 и т. д.)	B38400	
CBAUDEX	Расширенная маска скорости передачи (если скорость превышает 38 400 бит/с)	Нет	
CIBAUD	Скорость ввода, если отличается от скорости вывода (не используется)	(Нет)	
CLOCAL	Игнорирование каналов состояния модема (не проверяет сигнал оператора)	Нет	*
CMSPAR	Применение «закрепленного» соответствия (MARK/SPACE)	Нет	
CREAD	Возможность принимать ввод	Да	*
CRTSCTS	Включение аппаратного управления потоком (RTS/CTS)	Нет	
CSIZE	Маска размером в символ (от 5 до 8 бит: CS5, CS6, CS7, CS8)	CS8	*
CSTOPB	Использование для каждого символа по два бита остановки; в противном случае только один	Нет	*
HUPCL	Разрыв (модемного) соединения при последнем закрытии	Да	*
PARENB	Включение соответствия	Нет	*
PARODD	Применение нечетного соответствия; в противном случае четного	Нет	*
c_lflag			
ECHO	Вывод вводимых символов	Да	*
ECHOCTL	Визуальное отображение контрольных символов (например, ^L)	Да	
ECHOE	Визуальное выполнение операции ERASE	Да	*
ECHOK	Визуальное выполнение операции KILL	Да	*
ECHOKE	Не выводит символ новой строки после операции KILL	Да	
ECHONL	Вывод NL (в каноническом режиме), даже если выключить эхо-контроль	Нет	*
ECHOPRT	Вывод удаляемых символов в обратном порядке (между \ и /)	Нет	
FLUSHO	Сброс вывода (не используется)	—	
ICANON	Ввод в каноническом режиме (построчно)	Да	*

Продолжение ⇙

Таблица 58.2 *(продолжение)*

Поле/флаг	Описание	Включен	SUSv3
IEXTEN	Включение расширенной обработки вводимых символов	Да	*
ISIG	Возможность вводить символы, генерирующие сигналы (INTR, QUIT, SUSP)	Да	*
NOFLSH	Отключение сброса при вводе INTR, QUIT и SUSP	Нет	*
PENDIN	Новый вывод отложенного ввода при следующем чтении (не реализовано)	(Нет)	
TOSTOP	Генерация SIGTTOU для фонового вывода (см. раздел 34.7.1)	Нет	*
XCASE	Каноническое представление верхнего/нижнего регистра	(Нет)	

Отдельные флаги, перечисленные в табл. 58.2, были доступны в традиционных терминалах с ограниченными возможностями; они редко используются в современных системах. Например, флаги `IUCLC`, `OLCUC` и `XCASE` применялись в терминалах, которые были способны отображать только прописные буквы. Когда пользователь при входе вводил свое имя в верхнем регистре, во многих старых UNIX-системах программа `login` исходила из того, что работа выполняется именно на таком терминале. В результате устанавливались вышеперечисленные флаги, а строка приглашения для ввода пароля имела следующий вид:

```
\PASSWORD:
```

С этого момента все строчные буквы будут выводиться в верхнем регистре, а перед настоящими прописными буквами будет вставляться символ обратного слэша (\). Аналогично ввод настоящих прописных букв необходимо начинать с данного символа. Флаг `ECHOPRT` тоже был разработан для терминалов с ограниченными возможностями.

Различные маски задержки тоже являются наследием былых времен; они позволяют выводить символы разрыва строки и разрыва страницы таким устройствам, как медленные терминалы и принтеры. Флаги `OFILL` и `OFDEL` давали возможность определить способ выполнения задержки. Большинство из таких флагов не используется в Linux. Исключение составляет параметр `TAB3` для маски `TABDLY`, позволяющий выводить символ табуляции в виде совокупности пробелов (не больше восьми).

В следующих подразделах приводятся подробности о некоторых флагах структуры `termios`.

BRKINT

Если установить этот флаг (и выключить при этом флаг `IGNBRK`), то при выполнении условия `BREAK` активной группе процессов будет передаваться сигнал `SIGINT`.

ECHO

Установка данного флага включает эхо-контроль вводимых символов. При вводе паролей его лучше сбрасывать. Эхо-контроль также отключается в командном режиме редактора `vi`, когда вводимые символы интерпретируются как команды редактирования, а не как текст. Флаг `ECHO` работает как в каноническом, так и в неканоническом режиме.

ECHOCTL

В сочетании с `ECHO` флаг `ECHOCTL` приводит к эхо-контролю управляющих символов (например, `^A` для Ctrl+A); исключение составляют табуляция, символ новой строки, `START` и `STOP`. Если флаг `ECHOCTL` не установлен, то управляющие символы не экранируются.

Управляющими считаются символы, у которых коды ASCII меньше 32 (плюс символ DEL, чей код равен 127). Управляющий символ (назовем его x) выводится в виде знака ^, за которым идет результат выражения x^64. Для всех символов, кроме DEL, оператор XOR (^) в этом выражении добавляет к их кодам значение 64. Таким образом, Ctrl+A (код ASCII 1) выводится на экран как ^A (A имеет код 65). В случае с DEL происходит вычитание 64 из 127, в результате чего получается 63 — код ASCII символа ?, поэтому DEL выводится как ^?.

ECHOE

В каноническом режиме установка данного флага приводит к визуальному представлению операции `ERASE`; это выглядит как вывод последовательности из трех символов: возврат-пробел-возврат. Если флаг `ECHOE` выключить, то операция `ERASE` все так же будет удалять символы, но ее выполнение будет экранироваться (например, как `^?`).

ECHOK и ECHOKE

Эти флаги управляют визуальным представлением операции `KILL` (например, удаление строки) в каноническом режиме. По умолчанию (когда оба флага включены) строка удаляется визуально (см. `ECHOE`). Если сбросить хотя бы один из этих флагов, то визуальное удаление перестанет выполняться (однако входящая строка по-прежнему будет отклоняться), а операция `KILL` будет экранироваться (например, как `^U`). Если флаг `ECHOK` установлен, а `ECHOKE` сброшен, то дополнительно выводится символ новой строки.

ICANON

Установка данного флага включает канонический режим ввода. Ввод группируется по строкам, а символы `EOF`, `EOL`, `EOL2`, `ERASE`, `LNEXT`, `KILL`, `REPRINT` и `WERASE` интерпретируются особым образом (обратите внимание на влияние флага `IEXTEN`).

IEXTEN

Установка данного флага включает расширенную обработку вводимых символов. Он (равно как и `ICANON`) требуется для интерпретации следующих символов: `EOL2`, `LNEXT`, `REPRINT` и `WERASE`. Кроме того, без него не работает флаг `IUCLC`. В стандарте SUSv3 всего лишь указано, что флаг `IEXTEN` включает расширенные функции; детали могут варьироваться в зависимости от реализации.

IMAXBEL

В Linux установка данного флага игнорируется. В консоли при заполнении входящей очереди всегда срабатывает звуковой сигнал.

IUTF8

Установка этого флага включает режим с обработкой (англ. cooked mode; см. подраздел 58.6.3), позволяющий корректно интерпретировать ввод символов UTF-8 при редактировании строки.

NOFLSH

По умолчанию, когда при вводе символов `INTR`, `QUIT` или `SUSP` генерируется сигнал, любые дополнительные данные во входящей и исходящей очередях терминала сбрасываются (отклоняются). Во избежание этого устанавливается флаг `NOFLSH`.

OPOST

Данный флаг включает постобработку вывода. Его наличие требуется для работы любых флагов в поле `c_oflag` структуры `termios` (и наоборот — сброс флага `OPOST` предотвращает любую постобработку вывода).

PARENB, IGNPAR, INPCK, PARMRK и PARODD

Эти флаги относятся к генерированию и проверке соответствия.

Флаг PARENB включает генерирование битов, которые используются для проверки соответствия выводимых и вводимых символов. Если интересует только соответствие выводимых символов, то можно отключить проверку ввода, сбросив флаг INPCK. Сброс и установка флага PARODD делают проверку соответствия четной и нечетной соответственно.

Остальные флаги определяют то, как именно будут обрабатываться вводимые символы с ошибками соответствия. При установленном флаге IGNPAR символ отклоняется (не передается считывающему процессу). В противном случае, если установлен флаг PARMRK, то символ доходит до считывающего процесса, но содержит в начале двухбайтовую последовательность 0377 + 0 (когда флаг PARMRK установлен, а ISTRIP сброшен, настоящий символ с кодом 0377 дублируется и превращается в 0377 + 0377). Если флаг PARMRK сброшен, а INPCK установлен, то символ отклоняется, а считывающему процессу передается нулевой байт. Когда сброшены все три флага, IGNPAR, PARMRK и INPCK, символ передается процессу в исходном виде.

Пример программы

В листинге 58.2 демонстрируется применение функций tcgetattr() и tcsetattr() для выключения флага ECHO, чтобы вводимые символы не отображались на экране. Ниже показан пример того, что можно увидеть при запуске данной программы:

```
$ ./no_echo
Enter text:                 Мы вводим текст, который не экранируется,
                            но при этом считывается
Read: Knock, knock, Neo.
```

Листинг 58.2. Отключение эхо-контроля в терминале

tty/no_echo.c

```
#include <termios.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 100

int
main(int argc, char *argv[])
{
    struct termios tp, save;
    char buf[BUF_SIZE];

    /* Получаем текущие параметры терминала, выключаем эхо-контроль */
    if (tcgetattr(STDIN_FILENO, &tp) == -1)
        errExit("tcgetattr");
    save = tp;           /* Позже это позволит восстановить параметры */
    tp.c_lflag &= ~ECHO; /* Сбрасываем флаг ECHO, остальные биты не трогаем */
    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
        errExit("tcsetattr");

    /* Считываем ввод и отображаем его на экране */
    printf("Enter text: ");
    fflush(stdout);
    if (fgets(buf, BUF_SIZE, stdin) == NULL)
        printf("Got end-of-file/error on fgets()\n");
    else
        printf("\nRead: %s", buf);
```

```
    /* Восстанавливаем исходные параметры терминала */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &save) == -1)
        errExit("tcsetattr");

    exit(EXIT_SUCCESS);
}
```

tty/no_echo.c

58.6. Режимы ввода/вывода терминала

Мы уже отмечали, что драйвер терминала способен обрабатывать ввод в каноническом и неканоническом режимах в зависимости от того, установлен ли флаг `ICANON`. Теперь остановимся на этом более подробно, после чего познакомимся с тремя дополнительными режимами — с обработкой, без обработки и cbreak, которые стали доступными в седьмой редакции системы UNIX. Мы покажем, как указанные режимы эмулируются в современных системах путем установки соответствующих значений в структуре `termios`.

58.6.1. Канонический режим

Данный режим ввода включается путем установки флага `ICANON`. Терминальный ввод в каноническом режиме имеет следующие отличительные черты.

- Ввод группируется в строки, в конце каждой из которых находится символ-разделитель: `NL`, `EOL`, `EOL2` (если установлен флаг `IEXTEN`), `EOF` (если только курсор находится не в начале строки) или `CR` (при установленном флаге `ICRNL`). Любые разделители, кроме `EOF`, передаются считывающему процессу (в качестве последнего символа строки).
- Включено редактирование строк, что позволяет изменять содержимое текущей строки. Следовательно, поддерживаются следующие символы: `ERASE`, `KILL` и (если установлен флаг `IEXTEN`) `WERASE`.
- Символы `REPRINT` и `LNEXT` тоже разрешены, если установлен флаг `IEXTEN`.

В каноническом режиме операция `read()` в терминале возвращается, когда доступна полная строка (сам вызов `read()` может вернуть только часть данной строки, если было запрошено недостаточно байтов; оставшиеся символы будут извлечены при последующем чтении). Вызов `read()` может также преждевременно завершиться, если прерван обработчиком сигнала, а перезапуск системных вызовов для этого сигнала отключен (см. раздел 21.5).

> При описании флага NOFLSH в разделе 58.5 мы отмечали, что символы, генерирующие сигналы, тоже заставляют драйвер терминала сбросить входящую очередь. Так происходит вне зависимости от того, был сигнал перехвачен или проигнорирован. Установка флага NOFLSH позволяет предотвратить это сбрасывание.

58.6.2. Неканонический режим

Некоторым приложениям (таким как `vi` и `less`) нужно считывать ввод посимвольно, не дожидаясь перехода на новую строку. Для таких случаев предусмотрен неканонический режим. Когда он включен (то есть сброшен флаг `ICANON`), ввод не проходит никакой специальной обработки. В частности, вводимые символы больше не группируются в строки, а сразу становятся доступными.

В каких обстоятельствах неканоническая операция `read()` считается завершенной? Можно сделать так, чтобы результат чтения возвращался спустя определенный промежуток времени или после прочтения определенного количества байтов (или в зависимости от сочетания этих условий). Данное поведение регулируется двумя элементами массива `c_cc` в структуре `termios`: `TIME` и `MIN`. Первый (индексируется с помощью константы `VTIME`) определяет время ожидания значения в десятых долях секунды, второй (индексируется путем `VMIN`) — минимальное количество байтов для чтения (ни один из этих элементов не влияет на ввод/вывод терминала в каноническом режиме).

То, как элементы `MIN` и `TIME` ведут себя и взаимодействуют друг с другом, зависит от того, содержит ли каждый из них ненулевое значение. Ниже перечислены четыре возможных сценария. Обратите внимание: в каждом из этих случаев (если на момент чтения уже доступен достаточный объем данных, удовлетворяющий условию `MIN`) операция `read()` немедленно возвращает количество байтов, меньшее, чем было доступно и указано в запросе.

MIN == 0, TIME == 0 (активное чтение)

Если на момент вызова данные доступны, то `read()` немедленно возвращает те из них, объем которых меньше доступных и указанных в запросе. Если доступных байтов не обнаружено, то `read()` сразу же возвращает `0`.

Приведенный сценарий удовлетворяет стандартным требованиям, предъявляемым к активному чтению, позволяя программе проверять наличие ввода и не блокировать выполнение, если данные недоступны. Этот режим в чем-то похож на эффект от установки флага терминала `O_NONBLOCK` (см. раздел 5.9). Однако при отсутствии данных операция `read()` с флагом `O_NONBLOCK` возвращает `-1` вместе с ошибкой `EAGAIN`.

MIN > 0, TIME == 0 (блокирующее чтение)

Вызов `read()` блокируется (возможно, перманентно), пока объем доступных данных не будет равен меньшему из запрошенных значений или параметру `MIN`; в результате возвращается меньшее из двух значений.

Такие программы, как `less`, обычно присваивают `MIN` значение `1`, а `TIME` — `0`. Это позволяет ждать нажатия одиночных клавиш, не затрачивая процессорное время на постоянные запросы, выполняемые в цикле.

Если терминал находится в неканоническом режиме, а значения `MIN` и `TIME` равны соответственно `1` и `0`, то методики, описанные в главе 59, можно использовать для проверки ввода одиночного символа (а не целой строки).

MIN == 0, TIME > 0 (чтение с временным интервалом)

При вызове `read()` запускается таймер. Результат возвращается, как только станет доступным хотя бы один байт или по истечении десятых долей секунды, указанных с помощью `TIME`. Во втором случае `read()` возвращает `0`.

Этот сценарий подходит для программ, взаимодействующих с устройствами, которые подключены к последовательному порту (такими как модем). Программа может послать устройству данные и ждать ответа; время ожидания поможет избежать бесконечного зависания, если устройство не отвечает.

MIN > 0, TIME > 0 (чтение с побайтовым временным интервалом)

Таймер перезапускается с появлением каждого нового байта, начиная с первого. Операция `read()` возвращается после прочтения количества байтов, которое либо меньше `MIN`,

либо равно значению, указанному при запросе, или же когда истекает время ожидания (`TIME`, заданное в десятых долях секунды) между чтением двух соседних байтов. Поскольку таймер запускается только с появлением начального байта, `read()` возвращает как минимум один байт (в такой ситуации чтение может заблокироваться навсегда).

Приведенный сценарий подходит для обработки в терминале комбинаций клавиш, генерирующих управляющие последовательности. Например, во многих терминалах клавиша ← генерирует последовательность из трех символов: `Escape`, `O` и `D`. Эти символы передаются с небольшими промежутками времени. Приложения, их обрабатывающие, должны отличать такой случай от ситуации, когда пользователь медленно вводит каждый из указанных символов отдельно. Для этого можно выполнить операцию `read()` с небольшим побайтовым временным интервалом, например 0,2 секунды. Такой подход применяется в командном режиме некоторых версий редактора `vi` (в зависимости от продолжительности времени ожидания нажатие клавиши ← можно эмулировать с помощью быстрого ввода вышеупомянутой последовательности символов).

Изменение и восстановление значений MIN и TIME с учетом портируемости

Для совместимости с отдельными старыми реализациями UNIX стандарт SUSv3 допускает использование одних и тех же значений для констант `VMIN`/`VTIME` и `VEOF`/`VEOL`, из-за чего могут совпадать соответствующие элементы массива `c_cc` в структуре `termios` (в Linux значения указанных констант отличаются). Это становится возможным благодаря тому, что `VEOF` и `VEOL` не применяются в неканоническом режиме. Значения констант `VMIN` и `VEOF` могут совпадать, в связи с чем нужно соблюдать осторожность при написании программ, которые входят в неканонический режим, инициализируют `MIN` (обычно с помощью значения `1`) и позже возвращаются в канонический режим. На последнем этапе код символа `EOF` (ASCII 4 или Ctrl+D) больше не будет стандартным. Для сохранения портируемости программы перед входом в неканонический режим параметры структуры `termios` следует сохранить и затем использовать их, чтобы переключиться обратно.

58.6.3. Режимы с обработкой, без обработки и cbreak

В седьмой редакции системы UNIX (а также в ранних версиях BSD) драйвер терминала мог принимать ввод в трех режимах: *с обработкой*, *без обработки* и *cbreak*. Особенности этих режимов перечислены в табл. 58.3.

Режим с обработкой фактически представлял собой канонический режим, в котором по умолчанию включена интерпретация всех специальных символов (то есть интерпретация `CR`, `NL` и `EOF`; редактирование текущей строки; обработка символов, генерирующих сигналы; `ICRNL`, `OCRNL` и т. д.).

Режим без обработки, напротив, являлся аналогом неканонического; в нем была выключена любая обработка ввода и вывода (он использовался приложениями, которым нужно было гарантировать, что драйвер терминала не вносит никаких изменений в данные, передающиеся через последовательный порт).

Режим cbreak служил чем-то средним между двумя предыдущими. Ввод был неканоническим, однако символы, генерирующие сигналы, интерпретировались; к тому же входящие и исходящие данные могли подвергаться различным трансформациям (в зависимости от установки отдельных флагов). Режим cbreak не отключал эхо-контроль по умолчанию, однако это обычно делали приложения, которые его использовали. Он был востребован в программах, работающих с экраном (таких как `less`), позволявших вводить

текст посимвольно, но все же требовавших обработки ряда специальных последовательностей, таких как INTR, QUIT и SUSP.

Таблица 58.3. Различия между тремя режимами терминала: с обработкой, без обработки и cbreak

Возможность	**Режим**		
	С обработкой	**cbreak**	**Без обработки**
Ввод доступен	Построчно	Посимвольно	Посимвольно
Окончание строки	Да	Нет	Нет
Интерпретация символов, генерирующих сигналы	Да	Да	Нет
Интерпретация символов START/STOP	Да	Да	Нет
Интерпретация других специальных символов	Да	Нет	Нет
Выполнение дополнительной обработки ввода	Да	Да	Нет
Выполнение дополнительной обработки вывода	Да	Да	Нет
Эхо-контроль ввода	Да	Возможно	Нет

Пример: переход в режим без обработки и cbreak

В седьмой редакции системы UNIX и в оригинальной версии BSD драйвер терминала позволял переключаться в режим без обработки или cbreak, изменяя всего лишь один бит (RAW или CBREAK) в соответствующей структуре данных. С переходом к POSIX-интерфейсу termios (который поддерживается всеми современными реализациями UNIX) такой способ переключения больше недоступен, а приложения, эмулирующие указанные режимы, должны явно изменять необходимые поля. В листинге 58.3 представлены две функции, ttySetCbreak() и ttySetRaw(), реализующие аналоги этих двух режимов терминала.

> Приложения, использующие библиотеку ncurses, могут выполнять аналогичные действия с помощью функций cbreak() и raw().

Листинг 58.3. Переключение терминала между режимом без обработки и cbreak

tty/tty_functions.c

```
#include <termios.h>
#include <unistd.h>
#include "tty_functions.h"        /* Объявляет определяемые здесь функции */

/* Переводим терминал, на который ссылается 'fd', в режим cbreak
   (неканонический, с выключенным эхо-контролем). Мы исходим из того,
   что терминал пребывает в режиме с обработкой (то есть мы не должны
   вызывать эту функцию, если терминал находится в режиме без обработки,
   так как она не сбрасывает все изменения, вносимые функцией ttySetRaw(),
   приведенной ниже). Возвращает 0 при успешном завершении или -1 в случае
   ошибки. Если аргумент 'prevTermios' не равен NULL, то должен указывать
   на буфер с предыдущими параметрами терминала. */

int
ttySetCbreak(int fd, struct termios *prevTermios)
{
```

```
    struct termios t;

    if (tcgetattr(fd, &t) == -1)
        return -1;

    if (prevTermios != NULL)
        *prevTermios = t;

    t.c_lflag &= ~(ICANON | ECHO);
    t.c_lflag |= ISIG;
    t.c_iflag &= ~ICRNL;
    t.c_cc[VMIN] = 1;                /* Посимвольный ввод */
    t.c_cc[VTIME] = 0;               /* с блокировкой */
    if (tcsetattr(fd, TCSAFLUSH, &t) == -1)
        return -1;

    return 0;
}

/* Переводим терминал, на который ссылается 'fd', в режим без обработки
   (неканонический режим с отключением любой обработки ввода и вывода).
   Возвращает 0 при успешном завершении или -1 в случае ошибки. Если
   аргумент 'prevTermios' не равен NULL, он должен указывать на буфер
   с предыдущими параметрами терминала. */

int
ttySetRaw(int fd, struct termios *prevTermios)
{
    struct termios t;

    if (tcgetattr(fd, &t) == -1)
        return -1;

    if (prevTermios != NULL)
        *prevTermios = t;

    t.c_lflag &= ~(ICANON | ISIG | IEXTEN | ECHO);
                  /* Неканонический режим, отключаем сигналы,
                     расширенную обработку ввода и эхо-контроль */

    t.c_iflag &= ~(BRKINT | ICRNL | IGNBRK | IGNCR | INLCR |
                       INPCK | ISTRIP | IXON | PARMRK);
           /* Отключаем интерпретацию символов CR, NL и BREAK. Урезание до 8 бит
              и проверка на ошибки соответствия отсутствуют. Отключаем управление
              потоком с помощью символов START/STOP. */

    t.c_oflag &= ~OPOST;          /* Полностью отключаем обработку вывода */
    t.c_cc[VMIN] = 1;             /* Посимвольный ввод */
    t.c_cc[VTIME] = 0;            /* с блокировкой */

    if (tcsetattr(fd, TCSAFLUSH, &t) == -1)
        return -1;
    return 0;
}
```

tty/tty_functions.c

Программа, переводящая терминал в режим без обработки или cbreak, должна позаботиться о возвращении в нормальный режим после своего завершения. Помимо прочего,

это подразумевает обработку всех сигналов, которые ей могут отправить, чтобы завершение работы не оказалось преждевременным (сигналы управления заданиями можно генерировать с помощью клавиатуры и в режиме cbreak).

Пример того, как это делается, показан в листинге 58.4. Данная программа выполняет следующие шаги.

- Переключает терминал либо в режим cbreak (9), либо в режим без обработки (12) в зависимости от наличия аргумента командной строки (которым может быть любой набор символов) (8). Предыдущие параметры терминала сохраняются в глобальной переменной `userTermios` (1).
- Если терминал был помещен в режим cbreak, то из него могут генерироваться сигналы. Их нужно обрабатывать, чтобы в случае приостановки или завершения программы терминал был возвращен в нормальный режим, привычный для пользователя. Программа устанавливает один и тот же обработчик для сигналов `SIGQUIT` и `SIGINT` (10). Сигнал `SIGTSTP` требует особого обращения, поэтому для него предусмотрен отдельный обработчик (11).
- Устанавливает обработчик для сигнала `SIGTERM`, который по умолчанию генерируется командой `kill` (13).
- Входит в цикл, посимвольно считывающий стандартный ввод, и направляет его в стандартный вывод (14). Прежде чем выводить различные входящие символы, программа их интерпретирует (15):
  - перед выводом все буквы переводятся в нижний регистр;
  - символы новой строки (`\n`) и разрыва строки (`\r`) экранируются без изменений;
  - управляющие символы, кроме `\n` и `\r`, экранируются в виде последовательностей из двух символов: знака `^` и соответствующей буквы в верхнем регистре (например, Ctrl+A выводится как `^A`);
  - остальные символы экранируются в виде звездочек (`*`);
  - буква `q` приводит к завершению цикла (16).
- При выходе из цикла программа восстанавливает состояние терминала, установленное пользователем ранее, и завершает свою работу (17).

Программа устанавливает для сигналов `SIGQUIT`, `SIGINT` и `SIGTERM` один и тот же обработчик, который возвращает терминал в предыдущее состояние, и завершает работу (2).

Обработчик сигнала `SIGTSTP` (3) ведет себя так, как было описано в подразделе 34.7.3. Ниже приведена часть особенностей его работы.

- Во время ввода он сохраняет текущие параметры терминала (в переменной `ourTermios`) (4), после чего возвращает терминал к состоянию, актуальному на момент запуска программы (сохраненному в переменной `userTermios`) (5), и генерирует еще один сигнал `SIGTSTP`, чтобы на самом деле остановить процесс.
- При возобновлении работы, инициированном сигналом `SIGCONT`, обработчик повторно сохраняет текущие параметры терминала в переменной `userTermios` (6), так как за это время пользователь мог их поменять (например, с помощью команды `stty`). Затем обработчик возвращает терминал к состоянию, которое требуется для работы программы (`ourTermios`) (7).

Листинг 58.4. Демонстрация режима без обработки и cbreak

`tty/test_tty_functions.c`

```
#include <termios.h>
#include <signal.h>
#include <ctype.h>
#include "tty_functions.h"      /* Объявление ttySetCbreak() и ttySetRaw() */
```

```
#include "tlpi_hdr.h"

❶ static struct termios userTermios;
                /* Параметры терминала, определенные пользователем */

static void      /* Общий обработчик: восстанавливает параметры tty и завершается */
handler(int sig)
{
❷ if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
        errExit("tcsetattr");
    _exit(EXIT_SUCCESS);
}

static void                         /* Обработчик для SIGTSTP */
❸ tstpHandler(int sig)
{
    struct termios ourTermios;      /* Для сохранения параметров нашего терминала */
    sigset_t tstpMask, prevMask;
    struct sigaction sa;
    int savedErrno;

    savedErrno = errno;             /* Здесь можно было бы изменить 'errno' */

    /* Сохраняем текущие параметры терминала, возвращаем терминал
       к состоянию, в котором он был на момент запуска программы */
❹ if (tcgetattr(STDIN_FILENO, &ourTermios) == -1)
        errExit("tcgetattr");
❺ if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
        errExit("tcsetattr");

    /* Устанавливаем для SIGTSTP действие по умолчанию, посылаем сигнал
       еще раз и разблокируем его, чтобы программа могла остановиться */
    if (signal(SIGTSTP, SIG_DFL) == SIG_ERR)
        errExit("signal");
    raise(SIGTSTP);
    sigemptyset(&tstpMask);
    sigaddset(&tstpMask, SIGTSTP);
    if (sigprocmask(SIG_UNBLOCK, &tstpMask, &prevMask) == -1)
        errExit("sigprocmask");

    /* Выполнение возобновляется после SIGCONT */
    if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
        errExit("sigprocmask");        /* Повторно блокируем SIGTSTP */
    sigemptyset(&sa.sa_mask);          /* Заново устанавливаем обработчик */
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = tstpHandler;
    if (sigaction(SIGTSTP, &sa, NULL) == -1)
        errExit("sigaction");

    /* С момента остановки программы пользователь мог изменить параметры
       терминала; сохраняем параметры, чтобы позже их восстановить */
❻ if (tcgetattr(STDIN_FILENO, &userTermios) == -1)
        errExit("tcgetattr");

    /* Восстанавливаем наши параметры терминала */
❼ if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &ourTermios) == -1)
        errExit("tcsetattr");
    errno = savedErrno;
}
```

```
int
main(int argc, char *argv[])
{
    char ch;
    struct sigaction sa, prev;
    ssize_t n;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

8   if (argc > 1) {            /* Используем режим cbreak */
9       if (ttySetCbreak(STDIN_FILENO, &userTermios) == -1)
            errExit("ttySetCbreak");

        /* В режиме cbreak специальные символы терминала могут генерировать сигналы.
           Перехватываем их, чтобы откорректировать режим терминала. Устанавливаем
           обработчики только для сигналов, которые не игнорируются. */
10      sa.sa_handler = handler;

        if (sigaction(SIGQUIT, NULL, &prev) == -1)
            errExit("sigaction");
        if (prev.sa_handler != SIG_IGN)
            if (sigaction(SIGQUIT, &sa, NULL) == -1)
                errExit("sigaction");

        if (sigaction(SIGINT, NULL, &prev) == -1)
            errExit("sigaction");
        if (prev.sa_handler != SIG_IGN)
            if (sigaction(SIGINT, &sa, NULL) == -1)
                errExit("sigaction");

11      sa.sa_handler = tstpHandler;
        if (sigaction(SIGTSTP, NULL, &prev) == -1)
            errExit("sigaction");
        if (prev.sa_handler != SIG_IGN)
            if (sigaction(SIGTSTP, &sa, NULL) == -1)
                errExit("sigaction");
    } else {                    /* Используем режим без обработки */
12      if (ttySetRaw(STDIN_FILENO, &userTermios) == -1)
            errExit("ttySetRaw");
    }

13  sa.sa_handler = handler;
    if (sigaction(SIGTERM, &sa, NULL) == -1)
        errExit("sigaction");

    setbuf(stdout, NULL);   /* Отключаем буферизацию стандартного вывода */

14  for (;;) {              /* Считываем и отображаем стандартный ввод */
        n = read(STDIN_FILENO, &ch, 1);
        if (n == -1) {
            errMsg("read");
            break;
        }
        if (n == 0)      /* Может произойти после отключения терминала */
            break;
15      if (isalpha((unsigned char) ch))  /* Буквы --> нижний регистр */
            putchar(tolower((unsigned char) ch));
```

```
        else if (ch == '\n' || ch == '\r')
            putchar(ch);
        else if (iscntrl((unsigned char) ch))
            printf("^%c", ch ^ 64);    /* Выводим Ctrl+A как ^A и т. д. */
        else
            putchar('*');       /* Остальные символы выводятся как '*' */
(16)    if (ch == 'q')          /* Выходим из цикла */
            break;
    }

(17) if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
        errExit("tcsetattr");
    exit(EXIT_SUCCESS);
}
```

tty/test_tty_functions.c

Пример запуска программы из листинга 58.4 в режиме без обработки показан ниже:

```
$ stty              Изначально терминал находится в обычном состоянии (с обработкой)
speed 38400 baud; line = 0;
$ ./test_tty_functions
abc                                 Вводим abc и Ctrl+J
    def                             Вводим DEF, Ctrl+J и Enter
^C^Z                                Вводим Ctrl+C, Ctrl+Z и Ctrl+J
    q$                              Вводим q, чтобы выйти
```

В последней строке сессии, представленной выше, видно: командная оболочка вывела свое приглашение в той же строке, что и символ q, который привел к выходу из программы.

Ниже показан пример использования режима cbreak:

```
$ ./test_tty_functions x
XYZ                         Вводим XYZ и Ctrl+Z
[1]+ Stopped       ./test_tty_functions x
$ stty                      Проверяем, что режим терминала восстановлен
speed 38400 baud; line = 0;
$ fg                        Возобновляем работу в активном режиме
./test_tty_functions x
***                         Вводим 123 и Ctrl+J
    $                       Вводим Ctrl+C, чтобы завершить программу
Нажмите Enter для перехода к следующему приглашению оболочки
$ stty                      Проверяем, что режим терминала восстановлен
speed 38400 baud; line = 0;
```

58.7. Скорость передачи данных в терминале

Разные терминалы (и последовательные порты) имеют разную скорость отправки и приема данных (в битах в секунду). Функции `cfgetispeed()` и `cfsetispeed()` позволяют получить и изменить скорость ввода последовательного порта. Функции `cfgetospeed()` и `cfsetospeed()` отвечают за скорость вывода.

> Термин «бод» (англ. baud) служит в качестве синонима скорости терминала (в битах в секунду), хотя формально он здесь не совсем уместен. В частности, бод измеряет скорость изменения сигнала в соединении, которая может не совпадать с количеством битов, передаваемых за секунду, ведь последняя зависит от способа кодирования битов в сигналы. Тем не менее этот термин продолжает использоваться в качестве единицы измерения скорости в битах в секунду. Чтобы избежать путаницы, будем применять термины «скорость терминала» или «скорость последовательного порта».

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);
```

Обе функции возвращают скорость из заданной структуры `termios`

```
int cfsetospeed(struct termios *termios_p, speed_t speed);
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

Обе функции возвращают `0` при успешном завершении или `-1` при ошибке

Каждая из этих функций работает со структурой `termios`, которая должна быть предварительно инициализирована с помощью вызова `tcgetattr()`.

Например, чтобы получить текущую скорость вывода терминала, нужно сделать следующее:

```
struct termios tp;
speed_t rate;

if (tcgetattr(fd, &tp) == -1)
    errExit("tcgetattr");
rate = cfgetospeed(&tp);
if (rate == -1)
    errExit("cfgetospeed");
```

Если впоследствии понадобится изменить данную скорость, то можно сделать это следующим образом:

```
if (cfsetospeed(&tp, B38400) == -1)
    errExit("cfsetospeed");
if (tcsetattr(fd, TCSAFLUSH, &tp) == -1)
    errExit("tcsetattr");
```

Тип данных используется для хранения скорости последовательного порта. Скорость назначается не напрямую, а с помощью набора символьных констант (определенных в заголовочном файле `<termios.h>`). Они соответствуют нескольким дискретным значениям. Например, `B300`, `B2400`, `B9600` и `B38400` обозначают 300, 2400, 9600 и, соответственно, 38 400 битов в секунду. Применение дискретных значений является следствием того факта, что терминалы обычно предназначены для работы на нескольких стандартных скоростях, которые определяются делением некой базовой скорости (например, для ПК это обычно 115 200) на целые числа (например, 115 200 / 12 = 9600).

Стандарт SUSv3 гласит: скорость терминала хранится в структуре `termios`, но не уточняет, где именно (так делается намеренно). Многие системы, включая Linux, используют для этого поле `c_cflag`, где указываются маска `CBAUD` и флаг `CBAUDEX` (в разделе 58.2 отмечалось, что в Linux нестандартные поля `c_ispeed` и `c_ospeed` структуры `termios` не применяются).

Функции `cfsetispeed()` и `cfsetospeed()` позволяют указывать отдельные скорости для ввода и вывода, однако во многих терминалах данные значения должны совпадать. Кроме того, Linux использует только одно поле для хранения скорости терминала (то есть скорость всегда будет одной и той же). Это значит, что все функции для работы с входящей и исходящей скоростью терминала обращаются к одному и тому же полю структуры `termios`.

Передавая функции cfsetispeed() нулевой параметр speed, мы тем самым устанавливаем входящую скорость на уровне исходящей (когда вызывается tcsetattr()). Это бывает удобно в системах, где скорости ввода и вывода хранятся в виде одного значения.

58.8. Управление последовательным портом

Функции `tcsendbreak()`, `tcdrain()`, `tcflush()` и `tcflow()` выполняют действия, которые можно объединить под общим названием *управление последовательным портом* (они были разработаны для стандарта POSIX, чтобы заменить различные операции с вызовом `ioctl()`).

```
#include <termios.h>

int tcsendbreak(int fd, int duration);
int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcflow(int fd, int action);
```

Все возвращают `0` при успешном завершении или `-1` при ошибке

В каждой из этих функций `fd` является файловым дескриптором, ссылающимся на терминал или другое удаленное устройство, подключенное к последовательному порту.

Функция `tcsendbreak()` генерирует условие `BREAK`, безостановочно передавая поток нулевых битов. Аргумент `duration` обозначает длину передачи. Допустим, он равен 0, тогда нулевые биты будут передаваться на протяжении 0,25 секунды (стандарт SUSv3 ограничивает продолжительность в пределах от 0,25 до 0,5 секунды). Если `duration` больше 0, то нулевые биты станут передаваться на протяжении заданного количества миллисекунд. Этот случай не предусмотрен стандартом SUSv3; обработка ненулевых значений `duration` сильно варьируется в зависимости от реализации (детали, описанные здесь, актуальны для библиотеки glibc).

Функция `tcdrain()` блокируется, пока не будет передан весь вывод (то есть пока не опустеет исходящая очередь терминала).

Функция `tcflush()` сбрасывает (отклоняет) данные во входящей и/или исходящей очереди терминала (см. рис. 58.1). Сброс входящей очереди приводит к потере данных, которые уже дошли до драйвера терминала, но еще не были прочитаны ни одним процессом. Например, приложение может использовать `tcflush()` для отмены всех запоздалых символов, прежде чем предложить ввести пароль. Сброс исходящей очереди отклоняет все данные, которые уже были записаны (переданы драйверу терминала), но еще не переданы устройству. Аргумент `queue_selector` может принимать одно из значений, описанных в табл. 58.4.

> Стоит отметить, что термин «сброс» в контексте функции tcflush() имеет другое значение, чем в случае с файловым вводом/выводом. При работе с файлами «сбросить» означает принудительно записать вывод в пользовательский сегмент памяти или буферный кэш (если речь идет о вызове fflush()) либо же переместить данные из буферного кэша на диск, как при использовании вызовов fsync(), fdatasync() и sync().

Таблица 58.4. Значения аргумента queue_selector для функции tcflush()

Значение	Описание
TCIFLUSH	Сбрасывает входящую очередь
TCOFLUSH	Сбрасывает исходящую очередь
TCIOFLUSH	Сбрасывает входящую и исходящую очереди

Функция tcflow() управляет потоком данных, передающихся между компьютером и терминалом (или другим удаленным устройством) в любом направлении. Аргумент action принимает одно из значений, перечисленных в табл. 58.5. Константы TCIOFF и TCION действуют только в терминалах, способных интерпретировать символы STOP и START; они приводят к приостановке и, соответственно, возобновлению передачи данных компьютеру.

Таблица 58.5. Значения аргумента action для функции tcflush()

Значение	Описание
TCOOFF	Приостанавливает вывод в терминал
TCOON	Возобновляет вывод в терминал
TCIOFF	Передает терминалу символ STOP
TCION	Передает терминалу символ START

58.9. Размер окна терминала

В оконной среде приложения, работающие с экраном, должны уметь отслеживать размер окна терминала, чтобы в случае его изменения иметь возможность перерисовать свой вывод. Для этого ядро предоставляет два инструмента.

- После изменения размера окна терминала активной группе процессов отправляется сигнал SIGWINCH. По умолчанию он игнорируется.
- В любой момент (обычно после получения сигнала SIGWINCH) процесс может использовать вызов ioctl() с флагом TIOCGWINSZ, чтобы получить текущий размер окна терминала.

Вызов ioctl() с флагом TIOCGWINSZ применяется следующим образом:

```
if (ioctl(fd, TIOCGWINSZ, &ws) == -1)
    errExit("ioctl");
```

Аргумент fd является файловым дескриптором, ссылающимся на окно терминала. Последний аргумент вызова ioctl() представляет собой указатель на структуру winsize (определена в заголовочном файле <sys/ioctl.h>), которая используется для возвращения размера окна терминала:

```
struct winsize {
    unsigned short ws_row;        /* Количество строк (в символах) */
    unsigned short ws_col;        /* Количество столбцов (в символах) */
    unsigned short ws_xpixel;     /* Ширина (в пикселах) */
    unsigned short ws_ypixel;     /* Высота (в пикселах) */
};
```

Как и многие другие системы, Linux не задействует поля структуры winsize с пиксельными размерами.

Использование сигнала SIGWINCH в сочетании с операцией ioctl() TIOCGWINSZ демонстрируется в листинге 58.5. Ниже показан пример того, что можно увидеть при запуске этой программы в среде с оконным менеджером и изменении размера окна терминала три раза подряд:

```
$ ./demo_SIGWINCH
Caught SIGWINCH, new window size: 35 rows * 80 columns
Caught SIGWINCH, new window size: 35 rows * 73 columns
Caught SIGWINCH, new window size: 22 rows * 73 columns
Нажмите Ctrl+C, чтобы завершить программу
```

Листинг 58.5. Мониторинг изменений размера окна терминала

tty/demo_SIGWINCH.c

```
#include <signal.h>
#include <termios.h>
#include <sys/ioctl.h>
#include "tlpi_hdr.h"

static void
sigwinchHandler(int sig)
{
}

int
main(int argc, char *argv[])
{
    struct winsize ws;
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigwinchHandler;
    if (sigaction(SIGWINCH, &sa, NULL) == -1)
        errExit("sigaction");

    for (;;) {
        pause();                        /* Ждем сигнала SIGWINCH */

        if (ioctl(STDIN_FILENO, TIOCGWINSZ, &ws) == -1)
            errExit("ioctl");

        printf("Caught SIGWINCH, new window size: "
                "%d rows * %d columns\n", ws.ws_row, ws.ws_col);
    }
}
```

tty/demo_SIGWINCH.c

Можно также изменить представление драйвера терминала о размере окна, передав вызову `ioctl()` с флагом `TIOCSWINSZ` инициализированную структуру `winsize`:

```
ws.ws_row = 40;
ws.ws_col = 100;
if (ioctl(fd, TIOCSWINSZ, &ws) == -1)
    errExit("ioctl");
```

Если новые значения в структуре `winsize` отличаются от текущего представления драйвера терминала о размере окна, происходят две вещи:

- структуры данных драйвера терминала обновляются с помощью новых значений, указанных в аргументе `ws`;
- активной группе процессов терминала передается сигнал `SIGWINCH`.

Стоит отметить, что эти события сами по себе не могут изменить реальный размер отображаемого окна, который контролируется кодом за пределами ядра (например, оконным менеджером или эмулятором терминала).

Большинство UNIX-систем предоставляют доступ к размеру окна терминала, используя операции `ioctl()`, описанные в данном разделе, хотя это не предусмотрено стандартом SUSv3.

58.10. Идентификация терминала

В разделе 34.4 мы познакомились с функцией `ctermid()`, возвращающей имя управляющего терминала, в котором запущен процесс (в системах UNIX это обычно `/dev/tty`). Функции, описанные в данном разделе, тоже могут пригодиться для идентификации терминала.

Функция `isatty()` позволяет определить, связан ли файловый дескриптор `fd` с каким-либо терминалом (а не с файлом другого типа).

```
#include <unistd.h>

int isatty(int fd);
```

Возвращает `true` (`1`), если дескриптор `fd` связан с терминалом, или `false` (`0`), если нет

Функция `isatty()` может пригодиться в редакторах и других программах, работающих с экраном, которым нужно знать, связаны ли их ввод и вывод с терминалом.

На основе файлового дескриптора функция `ttyname()` возвращает имя соответствующего терминального устройства.

```
#include <unistd.h>

char *ttyname(int fd);
```

Возвращает указатель на (статически выделенную) строку с именем терминала или `NULL` при ошибке

Для нахождения имени терминала `ttyname()` используются функции `opendir()` и `readdir()`, описанные в разделе 18.8. Они перебирают каталоги в поисках идентификатора (поля `st_rdev` в структуре `stat`), принадлежащего устройству, на которое ссылается файловый дескриптор `fd`. Записи о терминальных устройствах обычно находятся в двух каталогах: `/dev` и `/dev/pts`. В первом перечислены виртуальные консоли (например, `/dev/tty1`) и псевдотерминалы систем BSD, а во втором — псевдотерминалы вторичных устройств (в стиле System V). Мы вернемся к псевдотерминалам в главе 60.

У функции ttyname() есть реентерабельный аналог под названием ttyname_r().

Команда tty(1), отображающая имя терминала, с которым связан ее стандартный ввод, является консольной альтернативой функции ttyname().

58.11. Резюме

Когда-то терминалы в UNIX-системах представляли собой настоящие аппаратные устройства, подключаемые к компьютерам через последовательный порт. Ранние разновидности терминалов не были стандартизованы — то есть управляющие последовательности, требуемые для работы с терминалом, зависели от его производителя. В современных рабочих станциях вместо терминалов используются графические мониторы под управлением X Window System. Однако возможность управления терминалами все

еще требуется при работе с виртуальными устройствами, такими как виртуальные консоли и эмуляторы терминалов (основанных на псевдотерминалах), а также с реальными устройствами, подключенными через последовательный порт.

Параметры терминала (за исключением размеров его окна) хранятся в структуре типа `termios`, состоящей из четырех битовых масок с различными атрибутами терминала и массива, который определяет разные управляющие последовательности, интерпретируемые драйвером терминала. Для получения и изменения этих параметров предусмотрены функции `tcgetattr()` и `tcsetattr()`.

Драйвер терминала поддерживает два режима ввода. В каноническом режиме ввод группируется в строки (с одним из символов-разделителей в конце), позволяя редактировать текущую строку. Неканонический режим дает возможность приложению считывать ввод посимвольно, не дожидаясь ввода символа-разделителя; редактирование текущей строки при этом отключено. Завершение ввода в неканоническом режиме определяется полями `MIN` и `TIME` структуры `termios`, обозначающими минимальное количество символов, которое нужно прочитать, и, соответственно, время ожидания, относящееся к операции чтения. Мы описали четыре разных сценария чтения в неканоническом режиме.

Так сложилось, что драйверы терминала в седьмой редакции UNIX и в системе BSD предоставляли три режима ввода: с обработкой, без обработки и cbreak. Каждый из них обеспечивал определенную степень интерпретации ввода и вывода. Cbreak и режим без обработки можно эмулировать путем изменения различных полей в структуре `termios`.

Для выполнения других операций с терминалом предусмотрен ряд функций. Это касается изменения скорости терминала и управления строками (генерирование разрывов строки, ожидание передачи вывода, сброс входящей и исходящей очередей, приостановка и возобновление передачи данных между компьютером и терминалом). Другие функции позволяют получить имя терминала и проверить, ссылается ли на него заданный файловый дескриптор. С помощью системного вызова `ioctl()` можно выполнить ряд операций, связанных с терминалом, включая извлечение и изменение информации о размере его окна.

Дополнительная информация

В книге [Stevens, 1992] тоже описывается программирование терминалов и раскрывается гораздо больше подробностей о работе с последовательными портами. Этой теме посвящено также несколько сетевых ресурсов. В частности, на веб-сайте проекта LDP (www.tldp.org) находятся методические пособия Дэвида Лойера по *текстовым терминалам* и *последовательным портам*. Еще одним полезным источником информации является *руководство по программированию последовательных портов для операционных систем POSIX* Майкла Суита, доступное на www.easysw.com/~mike/serial/.

58.12. Упражнения

58.1. Реализуйте функцию `isatty()` (для этого вам может пригодиться описание функции `tcgetattr()` в разделе 58.2).

58.2. Реализуйте функцию `ttyname()`.

58.3. Реализуйте функцию `getpass()`, описанную в разделе 8.5 (для получения файлового дескриптора управляющего терминала можно открыть файл `/dev/tty`).

58.4. Напишите программу, которая выводит информацию о том, в каком режиме работает терминал, связанный со стандартным вводом, — каноническом или неканоническом, и затем отображает значения `TIME` и `MIN`.

59 Альтернативные модели ввода/вывода

В данной главе будут рассмотрены три альтернативы традиционной модели файлового ввода/вывода, которую мы применяли в большинстве программ в этой книге:

- мультиплексированный ввод/вывод (системные вызовы `select()` и `poll()`);
- ввод/вывод, основанный на сигналах;
- программный интерфейс `epoll`, доступный только в Linux.

59.1. Краткий обзор

Большинство программ, представленных на данный момент в этой книге, использует модель ввода/вывода, согласно которой процесс работает одновременно только с одним файловым дескриптором, и каждый системный вызов блокируется в ожидании передачи данных. Например, при чтении из канала вызов `read()` обычно останавливается, если в этом канале не обнаружено никаких данных; то же самое происходит и с вызовом `write()` при нехватке в канале места для записи. Аналогичное поведение проявляется при работе с файлами других типов, включая очереди FIFO и сокеты.

> Дисковые файлы представляют собой особый случай. Как уже говорилось в главе 13, ядро использует буферный кэш, чтобы ускорить запросы ввода/вывода к диску. Следовательно, вызов write() возвращается сразу после передачи данных в буферный кэш ядра, не дожидаясь фактической их записи на диск (если только при открытии файла не был указан флаг O_SYNC). Соответственно, вызов read() передает данные из буферного кэша в пользовательский буфер, и если данных в кэше нет, то ядро приостанавливает процесс, считывая тем временем данные с диска.

Традиционной блокирующей модели ввода/вывода достаточно для большинства приложений, но не для всех. В частности, иногда может возникнуть необходимость в выполнении одной из следующих операций (или сразу обеих):

- проверка доступности ввода/вывода для файлового дескриптора; при этом, если ответ отрицательный, то операция не должна заблокироваться;
- мониторинг нескольких файловых дескрипторов для определения доступности ввода/вывода для любого из них.

Мы уже познакомились с двумя методиками, которые позволяют частично удовлетворить эти потребности: неблокирующий ввод/вывод и применение нескольких процессов или потоков.

Неблокирующий ввод/вывод был описан в разделах 5.9 и 44.9. Если поместить файловый дескриптор в неблокирующий режим, указав флаг состояния открытого файла `O_NONBLOCK`, то системные вызовы, которые не могут завершиться немедленно, не блокируются, а возвращают ошибку. Этот подход можно применять к именованным каналам, очередям FIFO, сокетам, терминалам, псевдотерминалам и ряду других видов устройств.

Неблокирующий ввод/вывод позволяет периодически проверять («опрашивать») возможность чтения или записи в файловый дескриптор. Например, можно сделать вхо-

дящий файловый дескриптор неблокирующим и затем периодически выполнять неблокирующее чтение. При необходимости отслеживать несколько файловых дескрипторов можно пометить их все как неблокирующие и выполнять аналогичную проверку для каждого из них. Однако в данном случае активное чтение является нежелательным. Если выполнять его нечасто, то время реакции приложения на ввод/вывод может оказаться неприемлемо большим; с другой стороны, слишком частое циклическое чтение тратит впустую ресурсы процессора.

> Термин «опрашивать» (англ. poll), который мы используем в этой главе, относится как к системному вызову poll(), предназначенному для мультиплексированного ввода/вывода, так и к процедуре «неблокирующей проверки состояния файлового дескриптора».

Если мы не хотим, чтобы процесс блокировался при выполнении ввода/вывода, то можем выделить специально для этого отдельный процесс. Пока родитель выполняет какие-то другие задачи, потомок блокируется, дожидаясь завершения ввода/вывода. При необходимости работать с несколькими файловыми дескрипторами можно создать по одному дочернему процессу для каждого из них. Проблема данного подхода заключается в ресурсоемкости и сложности. Создание и обслуживание процессов нагружает систему, а потомкам обычно приходится использовать некий механизм межпроцессного взаимодействия с целью информирования родителя о состоянии операций ввода/вывода.

Применение нескольких потоков вместо процессов является менее ресурсоемким, но все равно придется передавать между этими потоками информацию о состоянии операций ввода/вывода, что усложняет код, особенно если использовать пулы потоков для минимизации ресурсов, затрачиваемых на параллельное обслуживание большого количества клиентов. (Потоки могут быть особенно полезными в приложениях, которые вызывают сторонние библиотеки для выполнения блокирующего ввода/вывода; можно избежать блокировки главной программы, если обращаться к библиотеке в отдельном потоке.)

В связи с ограничениями, связанными с неблокирующим вводом/выводом и применением множественных потоков или процессов, зачастую имеет смысл использовать следующие альтернативные подходы.

- *Мультиплексированный ввод/вывод* позволяет процессу отслеживать сразу несколько файловых дескрипторов и определять, возможно ли выполнять операции чтения или записи с каким-либо из них. Для этого предусмотрены системные вызовы `select()` и `poll()`.
- *Ввод/вывод на основе сигналов* — методика, согласно которой процесс просит ядро отправить ему сигнал, когда в заданном дескрипторе станет доступным ввод или в него можно будет записать какие-нибудь данные. Далее процесс может перейти к выполнению каких-то других задач; о возможности ввода/вывода он будет уведомлен с помощью сигнала. При мониторинге большого количества файловых дескрипторов данный подход демонстрирует намного лучшую производительность, чем вызовы `select()` и `poll()`.
- Программный интерфейс `epoll` поддерживается только в Linux версии 2.6 и выше. По аналогии с мультиплексированным вводом/выводом он позволяет следить за множеством файловых дескрипторов и проверять, допускают ли они чтение или запись. Как и ввод/вывод на основе сигналов, интерфейс `epoll` имеет значительно лучшую производительность при работе с большим количеством файловых дескрипторов.

> В оставшейся части этой главы при обсуждении данных методик мы в основном будем применять отдельные процессы, хотя тот же подход уместен и в многопоточных приложениях.

Все три методики, описанные выше, направлены на достижение одного и того же результата: одновременного мониторинга одного или (чаще всего) нескольких файловых дескрипторов, чтобы проверить, *готовы* ли они к выполнению ввода/вывода (или, если быть точным, можно ли к ним применить системные вызовы чтения или записи без блокировки). Файловый дескриптор становится *готовым* в результате какого-то *события*, например появления ввода, завершения соединения с сокетом или освобождения места в ранее заполненном исходящем буфере сокета после передачи отложенных данных удаленной стороне. Наблюдение за множеством файловых дескрипторов может пригодиться в таких приложениях, как сетевые серверы, которые занимаются мониторингом сразу нескольких клиентских сокетов или вынуждены следить за вводом одновременно с терминала и именованного канала (или сокета).

Обратите внимание: ни одна из этих методик не занимается непосредственным вводом/выводом. Они всего лишь сообщают нам о готовности файлового дескриптора. Для выполнения чтения или записи нужно использовать дополнительные системные вызовы.

> Одной из моделей, не описанной в этой главе, является POSIX AIO (асинхронный ввод/вывод), который позволяет процессу помещать операции чтения или записи файла в очередь и затем оповещает его об их завершении. Преимущество POSIX AIO заключается вот в чем: исходный вызов возвращается без промедления, так что процессу не нужно ждать, когда данные будут переданы ядру или операция будет завершена. Это позволяет процессу выполнять другие задачи параллельно с вводом/выводом (возможно даже размещение в очереди дополнительных запросов на чтение или запись). POSIX AIO обеспечивает заметный прирост производительности для определенной категории приложений. В современных версиях Linux этот интерфейс реализован на основе потоков в рамках библиотеки glibc. На момент написания книги ведется работа над реализацией POSIX AIO на уровне ядра, что должно обеспечить лучшее масштабирование. Описание данного интерфейса можно найти в книгах [Gallmeister, 1995] и [Robbins & Robbins, 2003].

Выбор подходящей методики

Далее в этой главе мы постараемся взвешивать причины, которые могут подтолкнуть нас к использованию той или иной методики. А пока что перечислим несколько основных моментов.

- `select()` и `poll()` — устоявшиеся системные вызовы; они присутствуют в UNIX-системах на протяжении многих лет. Их основным преимуществом перед остальными методиками выступает портируемость на другие платформы. Главный же недостаток заключается в проблемах с масштабированием при мониторинге множества (сотен или тысяч) файловых дескрипторов.
- Ключевым преимуществом программного интерфейса `epoll` является то, что он позволяет приложению эффективно следить за большим количеством дескрипторов. Главный его недостаток — он поддерживается только в Linux.

  > Ряд реализаций UNIX предоставляют (нестандартные) механизмы, похожие на epoll. Например, в Solaris существует специальный файл /dev/poll (страница poll(7d) руководства), а в некоторых разновидностях BSD доступен программный интерфейс kqueue (представляющий собой более универсальную систему мониторинга, чем epoll). Эти два механизма кратко описаны в книге [Stevens et al., 2004]; подробную информацию об интерфейсе kqueue можно найти в книге [Lemon, 2001].

- По аналогии с интерфейсом `epoll` ввод/вывод на основе сигналов позволяет приложениям более эффективно отслеживать большое количество файловых дескрипторов. Вместе с тем `epoll` имеет несколько заметных преимуществ:

- позволяет избежать трудностей, связанных с сигналами;
- позволяет выбирать нужный способ мониторинга (например, отслеживать готовность для чтения или для записи);
- позволяет выбирать, когда будут генерироваться уведомления — при изменении уровня или при достижении граничного значения (см. подраздел 59.1.1).

Кроме того, чтобы в полной мере воспользоваться преимуществами ввода/вывода на основе сигналов, необходимо применять функции, которые доступны только в Linux и отсутствуют в других системах; то есть данная методика ничем не лучше `epoll` в плане портируемости.

Ввиду того что вызовы `select()` и `poll()` имеют лучшую портируемость, а ввод/вывод на основе сигналов и интерфейс `epoll` обладают более высокой производительностью, в ряде случаев для мониторинга событий, связанных с файловыми дескрипторами, имеет смысл написать отдельный уровень абстракции. Это позволит использовать интерфейс `epoll` (или аналогичный) в системах, которые его поддерживают, а в остальных случаях прибегать к вызовам `select()` или `poll()`.

> Библиотека libevent представляет собой слой абстракции для мониторинга событий, относящихся к файловым дескрипторам. Она была перенесена на множество UNIX-систем. Позволяет прозрачно задействовать любые методики, описанные в этой главе: вызовы select() и poll(), ввод/вывод на основе сигналов, интерфейс epoll, а также файл /dev/poll в Solaris и интерфейс kqueue в системах семейства BSD (таким образом, libevent служит примером использования всех этих механизмов). Библиотека libevent написана Найлсом Провосом и доступна на http://libevent.org/.

59.1.1. Уведомления, срабатывающие по уровню или фронту

Прежде чем переходить к подробному обсуждению различных альтернативных механизмов ввода/вывода, необходимо выделить два способа определения готовности файлового дескриптора:

- *срабатывание по уровню* — файловый дескриптор считается готовым, если позволяет выполнить системный вызов ввода/вывода без блокировки;
- *срабатывание по фронту* — уведомление предоставляется в случае, если с момента последней проверки в файловом дескрипторе обнаружена новая активность (например, новый ввод).

В табл. 59.1 собраны модели уведомлений, которые используются при мультиплексировании ввода/вывода, в вводе/выводе на основе сигналов и в интерфейсе `epoll`. Последний отличается от первых двух возможностью применения обеих моделей: по уровню и по фронту.

Таблица 59.1. Использование уведомлений, срабатывающих по уровню и по фронту

Модель ввода/вывода	**По уровню?**	**По фронту?**
select() и poll()	*	
Ввод/вывод на основе сигналов		*
Интерфейс epoll	*	*

Отличия между этими двумя моделями уведомлений будут проясняться по мере чтения настоящей главы. А пока что обсудим то, как выбор той или иной модели влияет на архитектуру нашей программы.

Используя уведомления, срабатывающие по фронту, можно проверять готовность файлового дескриптора в любое время. То есть, когда файловый дескриптор готов (например, при появлении в нем входящих данных), можно выполнить для него операцию ввода/вывода и затем повторно проверить, готов ли он для дальнейшего применения (например, остались ли в нем данные для чтения); в случае положительного ответа можно продолжить выполнение ввода/вывода. Иными словами, благодаря возможности постоянного наблюдения в любой момент времени можно не загружать приложение по максимуму (например, считывая как можно больше данных) при каждом уведомлении о готовности файлового дескриптора (на самом деле можно вообще не выполнять никакого ввода/вывода).

Для сравнения: если выбрать модель срабатывания по фронту, то уведомления будут приходить только в ответ на события ввода/вывода. Кроме того, при возникновении такого события обычно не известно, сколько именно данных можно прочитать или записать. Следовательно, программы, которые используют эту модель, как правило, проектируются в соответствии со следующими правилами.

- Получив уведомление о событии ввода/вывода, программа в какой-то момент должна прочитать или записать в заданный дескриптор как можно больше данных. Если ей не удастся это сделать, то она может потерять возможность выполнить определенный ввод/вывод, поскольку до получения следующего события не будет знать о необходимости дополнительных действий с файловым дескриптором. Так может возникнуть ложная потеря данных или блокировка программы. Словосочетание «в какой-то момент» выбрано неслучайно, так как в некоторых случаях ввод/вывод не стоит выполнять сразу же после получения уведомления о готовности дескриптора. Дело в том, что при чтении или записи большого объема данных можно оставить другие файловые дескрипторы без работы. Мы рассмотрим этот случай более подробно в подразделе 59.4.6, когда будем обсуждать применение модели уведомлений, срабатывающих по фронту, для интерфейса `epoll`.
- Если программа использует цикл, чтобы прочитать или записать как можно больше данных, а дескриптор помечен как блокирующий, то системный вызов ввода/вывода рано или поздно заблокируется, когда передача данных станет невозможной. В связи с этим каждый отслеживаемый файловый дескриптор обычно переводится в неблокирующий режим, а после события ввода/вывода операции чтения или записи продолжают выполняться до тех пор, пока соответствующий системный вызов (например, `read()` или `write()`) не завершится ошибкой `EAGAIN` или `EWOULDBLOCK`.

59.1.2. Применение неблокирующего режима в сочетании с альтернативными моделями ввода/вывода

Неблокирующий режим (флаг `O_NONBLOCK`) часто используется в сочетании с альтернативными моделями ввода/вывода, описанными в данной главе. Ниже приводятся некоторые примеры того, почему это может быть полезным.

- Как уже объяснялось в предыдущем разделе, неблокирующий режим обычно применяется в связке с моделями ввода/вывода, обеспечивающими срабатывание уведомлений о событиях чтения или записи по фронту.
- Если несколько процессов (или потоков) выполняют ввод/вывод в контексте одних и тех же описаний открытых файлов, то с практической точки зрения готовность де-

скриптора может измениться между моментом принятия уведомления и последующей операцией ввода/вывода. Таким образом, блокирующий вызов может заблокироваться и не дать процессу отслеживать другие файловые дескрипторы (это может случиться при использовании любых моделей ввода/вывода, описанных в настоящей главе, независимо от способа срабатывания — по уровню или по фронту).

- Даже после срабатывания уведомлений по уровню (интерфейсы `select()` или `poll()`), сообщающих о готовности файлового дескриптора для потокового сокета к чтению или записи, вызов может заблокироваться при попытке записать достаточно большой объем данных за одну операцию `write()` или `send()`.
- В редких случаях программные интерфейсы уведомлений, срабатывающих по уровню, такие как `select()` или `poll()`, могут возвращать ложную информацию о готовности файлового дескриптора. Это может быть вызвано ошибкой в ядре или стать результатом нормальной работы в нестандартном сценарии.

  В разделе 16.6 книги [Stevens et al., 2004] приводится пример ложных уведомлений о готовности слушающего сокета в системах BSD. Если клиент подключается к серверу, а затем разрывает соединение, то вызов select(), выполняемый сервером между этими двумя событиями, сообщит о том, что слушающий сокет готов к чтению; однако последующий вызов accept(), выполненный после разрыва, заблокируется.

59.2. Мультиплексирование ввода/вывода

Эта процедура позволяет отслеживать сразу несколько файловых дескрипторов, проверяя доступность операций чтения или записи для любого из них. Для ее выполнения можно использовать один из двух системных вызовов, которые фактически идентичны по своим возможностям. Первый, `select()`, появился вместе с программным интерфейсом сокетов в системе BSD. Так сложилось, что он стал более распространенным. Второй системный вызов, `poll()`, был заимствован из System V. Оба они входят в современную редакцию стандарта SUSv3.

Вызовы `select()` и `poll()` можно применять для мониторинга дескрипторов обычных файлов, терминалов, псевдотерминалов, именованных каналов, очередей FIFO, сокетов и некоторых видов символьных устройств. Они позволяют либо перманентно заблокировать процесс в ожидании готовности дескриптора, либо указать время ожидания вызова.

59.2.1. Системный вызов select()

Системный вызов `select()` блокируется, пока один или несколько дескрипторов из заданного набора не станут доступными.

```
#include <sys/time.h>            /* Для портируемости */
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Возвращает количество готовых файловых дескрипторов, `0` в случае истечения времени ожидания или `-1` при ошибке

Аргументы nfds, readfds, writefds и exceptfds обозначают файловые дескрипторы, которые вызов select() должен отслеживать. С помощью аргумента timeout можно задать максимальное время блокировки вызова. Подробное описание всех этих параметров приводится ниже.

В прототипе вызова select() выше одключается заголовочный файл <sys/time.h>. Таково требование стандарта SUSv2, и в части реализаций UNIX оно является обязательным (заголовок <sys/time.h> присутствует в Linux, и его подключение никак не повредит).

Наборы файловых дескрипторов

Аргументы readfds, writefds и exceptfds являются указателями на *наборы файловых дескрипторов*, представленных в виде типа данных fd_set. Эти аргументы имеют следующее применение:

- readfds является набором файловых дескрипторов, которые нужно проверить на возможность ввода;
- writefds является набором файловых дескрипторов, которые нужно проверить на возможность вывода;
- exceptfds является набором файловых дескрипторов, которые нужно проверить на наличие исключительного условия.

Термин *исключительное условие* часто принимают за некую ошибку, произошедшую с файловым дескриптором. На самом деле это не так. Исключительное условие в Linux возникает в двух ситуациях (похоже ведут себя и другие UNIX-системы):

- когда меняет свое состояние вторичный псевдотерминал, подключенный к первичному, находящемуся в пакетном режиме (см. раздел 60.5);
- когда потоковый сокет принимает внеканальные данные (см. раздел 57.13.1).

Обычно тип данных fd_set реализуется в виде битовой маски. Но нам необязательно вникать в эти подробности, так как вся работа с наборами файловых дескрипторов выполняется с помощью четырех макросов: FD_ZERO(), FD_SET(), FD_CLR() и FD_ISSET().

```
#include <sys/select.h>

void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

Возвращают true (1), если fd входит в fdset; в противном случае возвращается false (0)

Эти макросы работают следующим образом:

- FD_ZERO() инициализирует набор, на который указывает fdset, делая его пустым;
- FD_SET() добавляет дескриптор fd в набор, на который указывает fdset;
- FD_CLR() удаляет дескриптор fd из набора, на который указывает fdset;
- FD_ISSET() возвращает true, если файловый дескриптор fd входит в набор, на который указывает fdset.

Максимальный размер набора файловых дескрипторов определяется константой FD_SETSIZE, которая в Linux равна 1024 (другие UNIX-системы имеют похожие значения для этого ограничения).

Макросы вида FD_* оперируют структурами данных в пользовательском пространстве, а реализация вызова select() в ядре способна обслуживать наборы дескрипторов больших размеров. Тем не менее библиотека glibc не предусматривает простого способа изменения константы FD_SETSIZE. При необходимости изменить это ограничение придется отредактировать определение в соответствующих заголовочных файлах .glibc. Но если нужно отслеживать большое количество дескрипторов, то интерфейс epoll, вероятно, будет более предпочтительным по сравнению с вызовом select(). Причины этого будут описаны позже в данной главе.

Аргументы `readfds`, `writefds` и `exceptfds` возвращают результат выполнения. Структуры `fd_set`, на которые они указывают, должны содержать нужные нам наборы дескрипторов до вызова `select()`; здесь применяются макросы `FD_ZERO()` и `FD_SET()`. Вызов `select()` изменяет все эти структуры таким образом, что на момент возвращения они содержат наборы с готовыми дескрипторами. (Поскольку структуры изменяются во время вызова, их нужно заново инициализировать, если они используются многократно внутри цикла.) Содержимое структур можно изучить с помощью макроса `FD_ISSET()`.

Если не интересует какой-то определенный вид событий, то соответствующему аргументу `fd_set` можно присвоить `NULL`. Подробности о каждом из трех видов событий будут описаны в подразделе 59.2.3.

Значение аргумента `nfds` должно быть на единицу больше, чем максимальный номер файлового дескриптора, содержащегося в любом из трех наборов. Этот аргумент делает вызов `select()` более эффективным, поскольку благодаря ему ядро знает, что файловые дескрипторы, чьи номера превышают данное значение, можно не проверять, ведь они точно не входят ни в один набор.

Аргумент timeout

Аргумент `timeout` влияет на поведение вызова `select()`, связанное с блокировкой. Ему можно присвоить либо `NULL` (в этом случае `select()` перманентно блокируется), либо указатель на структуру `timeval`:

```
struct timeval {
    time_t      tv_sec;                 /* Секунды */
    suseconds_t tv_usec;                /* Миллисекунды (long int) */
};
```

Если оба поля аргумент `timeout` равны 0, то вызов `select()` не блокируется; он просто проверяет заданные файловые дескрипторы на готовность и сразу же возвращается. В противном случае `timeout` определяет максимальное время ожидания вызова `select()`.

Структура `timeval` позволяет указывать время с точностью до микросекунд, однако точность самого вызова ограничена системными часами (см. раздел 10.6). Согласно стандарту SUSv3 время ожидания округляется в большую сторону, если оно не делится без остатка.

Стандарт SUSv3 требует, чтобы максимально допустимое время ожидания не превышало 31 дня. Большинство реализаций UNIX допускают куда большие значения. На платформе Linux/x86-32 тип time_t представляет собой 32-разрядное целое число, поэтому максимальное значение измеряется многими годами.

Если аргумент `timeout` равен `NULL` или указывает на структуру, содержащую ненулевые поля, то вызов `select()` блокируется, пока не возникнет одно из следующих событий:

- хотя бы один из файловых дескрипторов, указанных в наборах `readfds`, `writefds` или `exceptfds`, становится готовым;

- вызов прерывается обработчиком сигнала;
- истекает время, указанное в аргументе `timeout`.

> В старых версиях UNIX вызов sleep() поддерживал значения лишь с точностью до секунды (в отличие от, скажем, nanosleep()). Для эмуляции этой возможности аргументу nfds вызова select() передавался 0, аргументам readfds, writefds и exceptfds присваивался NULL, а в timeout указывался желаемый интервал.

Если в Linux вызов `select()` возвращается, сигнализируя о готовности одного или нескольких файловых дескрипторов, и при этом время ожидания не равно `NULL`, то структура, на которую указывает аргумент `timeout`, в итоге обновляется и сообщает о том, сколько времени оставалось до истечения заданного интервала. Но стоит отметить, что данное поведение зависит от реализации. Стандарт SUSv3 позволяет оставлять структуру, на которую указывает `timeout`, без изменений — именно *так* и поступает большинство UNIX-систем. Портируемые приложения, использующие вызов `select()` внутри цикла, должны всегда передавать уже инициализированную структуру со временем ожидания и игнорировать информацию, возвращаемую в ней в результате этого вызова.

Стандарт SUSv3 гласит, что структура, на которую указывает аргумент `timeout`, может быть обновлена только в случае успешного завершения вызова `select()`. Однако в Linux обновление происходит, даже если вызов прерывается обработчиком сигнала (и, как следствие, завершается ошибкой `EINTR`). Это делается для того, чтобы сообщить о том, сколько осталось до истечения времени ожидания (если бы вызов завершился успешно).

> Если предварительно воспользоваться системным вызовом personality() (доступным только в Linux), чтобы установить двоичный программный интерфейс (ABI), содержащий бит STICKY_TIMEOUTS, то вызов select() не станет изменять структуру, на которую указывает timeout.

Значение, возвращаемое вызовом select()

В качестве результата функция `select()` возвращает одно из следующих значений.

- `-1` свидетельствует об ошибке: `EBADF` или `EINTR`. Первая говорит о том, что один из файловых дескрипторов в группе `readfds`, `writefds` или `exceptfds` является недействительным (например, неоткрытым). Вторая сигнализирует о прерывании вызова обработчиком сигнала (как отмечалось в разделе 21.5, вызов `select()` в этой ситуации никогда не перезапускается автоматически).
- `0` означает, что время ожидания вызова истекло до того, как один из файловых дескрипторов стал готовым. В таком случае все возвращаемые наборы будут пустыми.
- Положительное значение говорит о готовности одного или нескольких файловых дескрипторов и соответствует их количеству. В данной ситуации следует проверить каждый возвращаемый набор (используя макрос `FD_ISSET()`), чтобы узнать, какие события ввода/вывода имели место. Если один и тот же дескриптор входит сразу в несколько наборов, то учитывается в каждом из них и считается готовым к более чем одному событию. Иными словами, вызов `select()` возвращает общее количество файловых дескрипторов, помеченных как готовые во всех трех наборах.

Пример программы

Применение вызова `select()` демонстрируется на примере листинга 59.1. С помощью аргументов командной строки можно указать время ожидания и файловые дескрипторы, которые мы хотим отслеживать. Первый параметр соответствует аргументу `timeout`

вызова `select()` (в секундах). Если указать знак минус (`-`), то аргументу `timeout` будет передано значение `NULL`, что приведет к перманентной блокировке. Каждый следующий аргумент командной строки обозначает номер файлового дескриптора, за которым нужно наблюдать; за ним идут буквы, описывающие проверяемые операции. Букв может быть только две: `r` (готовность к чтению) и `w` (готовность к записи).

Листинг 59.1. Мониторинг нескольких файловых дескрипторов с помощью вызова select()
altio/t_select.c

```
#include <sys/time.h>
#include <sys/select.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s {timeout|-} fd-num[rw]...\n", progName);
    fprintf(stderr, "    - means infinite timeout; \n");
    fprintf(stderr, "    r = monitor for read\n");
    fprintf(stderr, "    w = monitor for write\n\n");
    fprintf(stderr, "    e.g.: %s - 0rw 1w\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    fd_set readfds, writefds;
    int ready, nfds, fd, numRead, j;
    struct timeval timeout;
    struct timeval *pto;
    char buf[10];            /* Достаточно большой для хранения "rw\0" */

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageError(argv[0]);

    /* Время ожидания для select() указывается в argv[1] */
    if (strcmp(argv[1], "-") == 0) {
        pto = NULL;          /* Бесконечное время ожидания */
    } else {
        pto = &timeout;
        timeout.tv_sec = getLong(argv[1], 0, "timeout");
        timeout.tv_usec = 0;   /* Без микросекунд */
    }

    /* Обрабатываем остальные аргументы, чтобы сформировать
       наборы файловых дескрипторов */
    nfds = 0;
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);

    for (j = 2; j < argc; j++) {
        numRead = sscanf(argv[j], "%d%2[rw]", &fd, buf);
        if (numRead != 2)
            usageError(argv[0]);
        if (fd >= FD_SETSIZE)
            cmdLineErr("file descriptor exceeds limit (%d)\n", FD_SETSIZE);
```

```
        if (fd >= nfds)
            nfds = fd + 1;            /* Записываем максимум fd +1 */
        if (strchr(buf, 'r') != NULL)
            FD_SET(fd, &readfds);
        if (strchr(buf, 'w') != NULL)
            FD_SET(fd, &writefds);
    }

    /* Все аргументы сформированы; теперь вызываем select() */
    ready = select(nfds, &readfds, &writefds, NULL, pto);
                                /* Игнорируем исключительные события */

    if (ready == -1)
        errExit("select");

    /* Выводим результат выполнения select() */
    printf("ready = %d\n", ready);
    for (fd = 0; fd < nfds; fd++)
        printf("%d: %s%s\n", fd, FD_ISSET(fd, &readfds) ? "r" : "",
                FD_ISSET(fd, &writefds) ? "w" : "");

    if (pto != NULL)
        printf("timeout after select(): %ld.%03ld\n",
                (long) timeout.tv_sec, (long) timeout.tv_usec / 1000);
    exit(EXIT_SUCCESS);
}
```

altio/t_select.c

Использование программы из листинга 59.1 демонстрируется в следующей сессии командной строки. В первом примере мы делаем запрос на мониторинг ввода в файловом дескрипторе 0 со временем ожидания 10 секунд:

```
$ ./t_select 10 0r
Нажимаем Enter, чтобы введенная строка была доступна в файловом дескрипторе 0
ready = 1
0: r
timeout after select(): 8.003
$                     Отображается следующее приглашение командной строки
```

Как видите, вызов `select()` определил, что готовым к чтению был один файловый дескриптор с номером 0. Мы также видим обновление значения аргумента `timeout`. Последняя строчка вывода содержит только приглашение командной строки; оно выводится, так как программа `t_select` не прочитала символ новой строки, сделавший файловый дескриптор 0 готовым. Следовательно, он был прочитан командной оболочкой, которая в ответ вывела еще одно приглашение.

В следующем примере мы опять следим за вводом в файловом дескрипторе 0, но на этот раз время ожидания равно 0:

```
$ ./t_select 0 0r
ready = 0
timeout after select(): 0.000
```

Вызов `select()` немедленно возвращается, не найдя ни одного готового дескриптора.

В следующем примере мы отслеживаем два файловых дескриптора с номерами 0 (для ввода) и 1 (для вывода). Аргументу `timeout` присваивается значение `NULL` (первый аргумент командной строки равен знаку минус), что говорит о бесконечном времени ожидания:

```
$ ./t_select - 0r 1w
ready = 1
0:
1: w
```

Вызов `select()` немедленно возвращается, информируя нас о возможности вывода для дескриптора 1.

59.2.2. Системный вызов poll()

Системный вызов `poll()` может выполнять действие, сравнимое с `select()`. Главное отличие между этими двумя операциями состоит в том, каким способом задаются дескрипторы для мониторинга. Вызов `select()` предоставляет три набора; каждый из них должен сигнализировать о готовности тех или иных дескрипторов. Вызов `poll()` предоставляет один список; каждый дескриптор в нем имеет набор событий, которые нас интересуют.

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

Возвращает количество готовых файловых дескрипторов, `0`, если истекло время ожидания, или `-1` при ошибке

Аргумент `fds` и массив `pollfd` (`nfds`) обозначают файловые дескрипторы, которые нужно отслеживать. Аргумент `timeout` можно использовать, чтобы задать максимальный отрезок времени, на протяжении которого будет блокироваться вызов `poll()`. Каждый из представленных аргументов подробно описывается ниже.

Массив pollfd

Аргумент `fds` содержит список файловых дескрипторов, за которыми вызов `poll()` должен наблюдать. Это массив структур типа `pollfd`, определяемых следующим образом:

```
struct pollfd {
    int   fd;               /* Файловый дескриптор */
    short events;           /* Битовая маска с заданными событиями */
    short revents;          /* Битовая маска с возвращаемыми событиями */
};
```

Аргумент `nfds` обозначает количество элементов в массиве `fds`. Тип данных `nfds_t`, применяемый в аргументе `nfds`, представляет собой беззнаковое целое число.

Поля `events` и `revents` структуры `pollfd` являются битовыми масками. Вызывающий процесс инициализирует `events`, указывая события, которые нужно отслеживать для файлового дескриптора `fd`. Во время возвращения вызова `poll()` полю `revents` присваивается значение, сигнализирующее о событиях, на самом деле произошедших в этом дескрипторе.

В табл. 59.2 перечислены биты, которые могут содержаться в полях `events` и `revents`. Первая группа битов (`POLLIN`, `POLLRDNORM`, `POLLRDBAND`, `POLLPRI` и `POLLRDHUP`) относится к событиям ввода, а вторая (`POLLOUT`, `POLLWRNORM` и `POLLWRBAND`) — к событиям вывода. Третья группа (`POLLERR`, `POLLHUP` и `POLLNVAL`) содержит биты, устанавливаемые в поле `revents` для получения дополнительных сведений о файловом дескрипторе. Любой из этих трех битов будет проигнорирован, если указать его в поле `events`. Последний бит, `POLLMSG`, не используется в вызове `poll()`, реализованном в Linux.

В UNIX-системах, предоставляющих механизм STREAMS для символьных устройств, бит POLLMSG уведомляет о том, что сообщение с сигналом SIGPOLL достигло начала потока. Linux не использует бит POLLMSG, так как не поддерживает механизм STREAMS.

Таблица 59.2. Значения битовых масок для полей events и revents структуры pollfd

Бит	Передается в events?	Возвращается из revents?	Описание
POLLIN	*	*	Данные (не считая высокоприоритетных) могут быть прочитаны
POLLRDNORM	*	*	Эквивалент бита POLLIN
POLLRDBAND	*	*	Приоритетные данные могут быть прочитаны (не используется в Linux)
POLLPRI	*	*	Высокоприоритетные данные могут быть прочитаны
POLLRDHUP	*	*	Удаленный сокет был закрыт
POLLOUT	*	*	Обычные данные могут быть записаны
POLLWRNORM	*	*	Эквивалент бита POLLOUT
POLLWRBAND	*	*	Приоритетные данные могут быть записаны
POLLERR		*	Произошла ошибка
POLLHUP		*	Произошел разрыв соединения
POLLNVAL		*	Файловый дескриптор не открыт
POLLMSG			Не используется в Linux (и не входит в стандарт SUSv3)

При отсутствии надобности в событиях для конкретного файлового дескриптора полю `events` можно присвоить `0`. Кроме того, если указать отрицательное значение для `fd` (например, путем умножения номера дескриптора на –1, при условии, что тот не равен нулю), соответствующее поле `events` будет проигнорировано, а `events` в любом случае вернет `0`. Любой из вариантов можно использовать, чтобы отключить мониторинг определенного файлового дескриптора (возможно, временно) без необходимости заново создавать весь список `fds`.

Стоит отметить следующие аспекты реализации вызова `poll()` в Linux.

- Биты `POLLIN` и `POLLRDNORM` определены отдельно, но являются синонимами.
- Биты `POLLOUT` и `POLLWRNORM` определены отдельно, но являются синонимами.
- Бит `POLLRDBAND` обычно не применяется, то есть игнорируется в поле `events` и не указывается в `revents`.

  Бит POLLRDBAND указывается только в коде, реализующем (устаревший) сетевой протокол DECnet.

- Бит `POLLWRBAND` устанавливается в некоторых случаях, но не несет в себе никакой полезной информации (кроме того, его нельзя установить без использования битов `POLLOUT` и `POLLWRNORM`).
- Для получения определений констант `POLLRDNORM`, `POLLRDBAND`, `POLLWRNORM` и `POLLWRBAND` из заголовочного файла `<poll.h>` необходимо определить макрос проверки возможностей `_XOPEN_SOURCE`.

- Флаг `POLLRDHUP` применяется только в Linux и только с версией ядра 2.6.17. Чтобы получить его определение из файла `<poll.h>`, нужно определить макрос проверки возможностей `_GNU_SOURCE`.
- Бит `POLLNVAL` возвращается, если на момент вызова `poll()` заданный файловый дескриптор был закрыт.

Подводя итог всему вышесказанному, наиболее важными флагами вызова `poll()` являются `POLLIN`, `POLLOUT`, `POLLPRI`, `POLLRDHUP`, `POLLHUP` и `POLLERR`. Их назначение будет подробно описано в подразделе 59.2.3.

Аргумент timeout

Аргумент `timeout` влияет на блокировку вызова `poll()`:

- если он равен `-1`, то блокировка происходит до тех пор, пока не станет готовым хотя бы один файловый дескриптор из массива `fds` (в соответствии с полем `events`) или не будет перехвачен сигнал;
- если он равен `0`, то вызов проверяет готовность дескрипторов, не блокируясь;
- если он больше нуля, то блокировка происходит на протяжении `timeout` миллисекунд, пока не станет готовым один из файловых дескрипторов в массиве `fds` или не будет перехвачен сигнал.

Как и в случае с вызовом `select()`, точность аргумента `timeout` ограничена системными часами (см. раздел 10.6). А в стандарте SUSv3 сказано, что время ожидания округляется в большую сторону, если оно не делится без остатка.

Значение, возвращаемое вызовом poll()

В качестве результата вызов `poll()` возвращает одно из следующих значений:

- `-1` в случае ошибки. Одной из возможных ошибок является `EINTR`, которая говорит о том, что вызов был прерван обработчиком сигнала (как отмечалось в разделе 21.5, в этой ситуации вызов `poll()` никогда не перезапускается автоматически);
- `0`, если время ожидания истекло до того, как любой из файловых дескрипторов стал готовым;
- положительное значение, если один или несколько дескрипторов оказались готовыми. Это значение соответствует количеству структур `pollfd` в массиве `fds`, которые имеют ненулевое поле `revents`.

  Стоит отметить, что положительные значения, возвращаемые вызовами select() и poll(), имеют немного разный смысл. Системный вызов select() может учесть один и тот же файловый дескриптор несколько раз, если он входит сразу в несколько итоговых наборов. Системный вызов poll() возвращает количество готовых файловых дескрипторов, любой из которых может быть учтен лишь один раз, даже если в соответствующем поле revents установлено несколько битов.

Пример программы

Простой пример использования вызова `poll()` демонстрируется в листинге 59.2. Программа создает несколько именованных каналов (каждый из которых задействует по два файловых дескриптора с соседними номерами), записывает байты во входящий конец одного из них, выбранного случайным образом, и затем выполняет вызов `poll()`, чтобы узнать, в каком канале можно прочитать данные.

Пример выполнения показан в следующей сессии командной строки. Согласно предоставленным аргументам программа должна создать десять именованных каналов и выполнить запись в три из них (выбранных произвольно).

```
$ ./poll_pipes 10 3
Writing to fd:  4 (read fd:  3)
Writing to fd: 14 (read fd: 13)
Writing to fd: 14 (read fd: 13)
poll() returned: 2
Readable:  3
Readable: 13
```

Как видите, вызов `poll()` нашел два канала с данными, доступными для чтения.

Листинг 59.2. Использование вызова poll() для мониторинга нескольких файловых дескрипторов
altio/poll_pipes.c

```
#include <time.h>
#include <poll.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    int numPipes, j, ready, randPipe, numWrites;
    int (*pfds)[2];        /* Файловые дескрипторы для всех каналов */
    struct pollfd *pollFd;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-pipes [num-writes]\n", argv[0]);

    /* Выделяем место для наших массивов. Их размеры соответствуют
       количеству каналов, заданному в командной строке */
    numPipes = getInt(argv[1], GN_GT_0, "num-pipes");
    pfds = calloc(numPipes, sizeof(int [2]));
    if (pfds == NULL)
        errExit("calloc");
    pollFd = calloc(numPipes, sizeof(struct pollfd));
    if (pollFd == NULL)
        errExit("calloc");

    /* Создаем столько каналов, сколько указано в командной строке */
    for (j = 0; j < numPipes; j++)
        if (pipe(pfds[j]) == -1)
            errExit("pipe %d", j);

    /* Выполняем заданное число операций записи в произвольные каналы */
    numWrites = (argc > 2) ? getInt(argv[2], GN_GT_0, "num-writes") : 1;
    srandom((int) time(NULL));
    for (j = 0; j < numWrites; j++) {
        randPipe = random() % numPipes;
        printf("Writing to fd: %3d (read fd: %3d)\n",
                pfds[randPipe][1], pfds[randPipe][0]);
        if (write(pfds[randPipe][1], "a", 1) == -1)
            errExit("write %d", pfds[randPipe][1]);
    }
/* Формируем список файловых дескрипторов, которые передаются в poll().
   Он должен содержать дескрипторы считывающих концов каждого из каналов. */
    for (j = 0; j < numPipes; j++) {
        pollFd[j].fd = pfds[j][0];
```

```
        pollFd[j].events = POLLIN;
    }

    ready = poll(pollFd, numPipes, 0);          /* Неблокирующий */
    if (ready == -1)
        errExit("poll");
    printf("poll() returned: %d\n", ready);

    /* Проверяем, в каких каналах есть данные, доступные для чтения */
    for (j = 0; j < numPipes; j++)
        if (pollFd[j].revents & POLLIN)
            printf("Readable: %3d\n", pollFd[j].fd);

    exit(EXIT_SUCCESS);
}
```

altio/poll_pipes.c

59.2.3. Условия готовности файлового дескриптора

Для правильного использования вызовов `select()` и `poll()` необходимо понимать условия, при которых файловый дескриптор считается готовым. Согласно стандарту SUSv3 это происходит, когда вызов функции ввода/вывода не блокируется *вне зависимости от того, передает ли указанная функция какие-либо данные* (притом в дескрипторе не должен быть установлен флаг `O_NONBLOCK`). Ключевой момент выделен курсивом: вызовы `select()` и `poll()` говорят о том, будет ли блокироваться операция ввода/вывода, а не об успешности потенциальной передачи данных. Учитывая это, посмотрим, как приведенные системные вызовы ведут себя с разными видами файловых дескрипторов. Мы разделим полученную информацию на разные таблицы, содержащие по два столбца:

- столбец `select()` сигнализирует о том, помечен ли файловый дескриптор как доступный для чтения (`r`), записи (`w`) или имеющий исключительное условие (`x`);
- столбец `poll()` содержит бит(-ы), возвращаемые в поле `revents`. В этих таблицах не упоминаются флаги `POLLRDNORM`, `POLLWRNORM`, `POLLRDBAND` и `POLLWRBAND`. И хотя часть из них в определенных условиях может вернуться вместе с `revents` (если их указать в `events`), они не несут в себе никакой полезной информации, которую нельзя было бы получить с помощью флагов `POLLIN`, `POLLOUT`, `POLLHUP` и `POLLERR`.

Обычные файлы

Когда речь идет о дескрипторах, ссылающихся на обычные файлы, вызов `select()` помечает их доступными для чтения и записи, а вызов `poll()` устанавливает флаги `POLLIN` и `POLLOUT` в поле `revents`. Это объясняется следующими причинами:

- вызов `read()` в любой ситуации немедленно возвращает данные, конец файла или ошибку (например, если файл не был открыт для чтения);
- вызов `write()` всегда делает одно из двух: немедленно передает данные или сразу же завершается ошибкой.

Стандарт SUSv3 гласит, что, помимо прочего, вызов select() должен помечать дескриптор обычного файла как имеющий исключительное условие (хотя для обычных файлов значение этого факта совсем не очевидно). Так поступают не все системы; Linux в их число не входит.

Терминалы и псевдотерминалы

В табл. 59.3 описывается поведение вызовов `select()` и `poll()` для терминалов и псевдотерминалов (см. главу 60).

При закрытии одного из двух соединенных псевдотерминалов параметры `revents`, возвращаемые вызовом `poll()` для оставшегося конца соединения, зависят от реализации. В Linux как минимум устанавливается флаг `POLLHUP`. Хотя другие системы для оповещения об этом событии могут возвращать другие параметры, например `POLLERR` или `POLLIN`. Кроме того, в некоторых реализациях устанавливаемые флаги зависят от того, какое устройство отслеживалось — первичное или вторичное.

Таблица 59.3. Возвращаемые параметры вызовов select() и poll() для терминалов и псевдотерминалов

Условие или событие	select()	poll()
Доступен ввод	r	POLLIN
Возможен вывод	w	POLLOUT
После закрытия удаленного псевдотерминала	rw	См. текст
Первичный псевдотерминал в пакетном режиме обнаруживает изменение состояния вторичного конца соединения	x	POLLPRI

Именованные каналы и очереди FIFO

В табл. 59.4 собраны подробности о считывающем конце именованного канала и очереди FIFO. Столбец «Данные в канале?» указывает на то, можно ли прочитать из канала хотя бы один байт данных. Мы исходим из того, что поле `events` вызова `poll()` содержит флаг `POLLIN`.

Таблица 59.4. Параметры, возвращаемые вызовами select() и poll() для считывающего конца именованного канала и очереди FIFO

Условие или событие		select()	poll()
Данные в канале?	Записывающий конец открыт?		
Нет	Нет	r	POLLHUP
Да	Да	r	POLLIN
Да	Нет	r	POLLIN | POLLHUP

В некоторых других реализациях UNIX, если записывающий конец канала закрыт, то вызов `poll()` возвращается с установленным флагом `POLLIN` (так как вызов `read()` сразу же возвращает конец файла), а не с `POLLHUP`. Проверяя возможность блокировки операции чтения, портируемые приложения должны учитывать оба эти бита.

В табл. 59.5 собраны подробности о записывающем конце канала. Мы исходим из того, что поле `events` вызова `poll()` содержит флаг `POLLOUT`. Столбец «Место для PIPE_BUF байт?» указывает на то, достаточно ли в канале места для автоматической записи `PIPE_BUF` байтов без блокировки. Это критерий, по которому Linux определяет готовность канала к записи. Ряд других реализаций UNIX поступает аналогичным образом. Но есть системы, считающие канал доступным для записи, если в него можно записать хотя бы один байт. (В Linux 2.6.10 и ниже вместимость именованного канала была равна `PIPE_BUF`; это значит, что канал считался недоступным для записи при наличии в нем хотя бы одного байта данных.)

Таблица 59.5. Параметры, возвращаемые вызовами select() и poll() для записывающего конца именованного канала и очереди FIFO

Условие или событие		**select()**	**poll()**
Данные в канале?	**Записывающий конец открыт?**		
Нет	Нет	w	POLLERR
Да	Да	w	POLLOUT
Да	Нет	w	POLLOUT | POLLERR

В некоторых других UNIX-системах, если считывающий конец канала закрыт, то вызов `poll()` возвращается с установленным битом `POLLERR`, а не `POLLHUP`. Проверяя возможность блокировки операции записи, портируемые приложения должны учитывать оба эти бита.

Сокеты

В табл. 59.6 описывается поведение вызовов `select()` и `poll()` для сокетов. В столбце `poll()` предполагается, что поле `events` равно `(POLLIN | POLLOUT | POLLPRI)`. В столбце `select()` мы исходим из такого условия: файловый дескриптор проверяется на возможность ввода, вывода или наличия исключительного условия (то есть дескриптор указан во всех трех наборах, которые передаются в `select()`). Эта таблица охватывает только распространенные сценарии.

В ОС Linux поведение вызова poll() для сокетов домена UNIX отличается от показанного в табл. 59.6, если проверка выполняется после закрытия удаленного конца соединения: помимо прочих флагов в поле revents возвращается бит POLLHUP.

Таблица 59.6. Параметры, возвращаемые вызовами select() и poll() для сокетов

Условие или событие	**select()**	**poll()**
Доступен ввод	r	POLLIN
Возможен вывод	w	POLLOUT
Слушающий сокет установил входящее соединение	r	POLLIN
Получены внеканальные данные (только для TCP)	x	POLLPRI
Удаленный потоковый сокет разорвал соединение или выполнил shutdown(SHUT_WR)	rw	POLLIN | POLLOUT | POLLRDHUP

Флаг `POLLRDHUP` (доступный только в Linux 2.6.17 и выше) требует дополнительного разъяснения. На самом деле он имеет вид `EPOLLRDHUP` и предназначен в основном для использования в режиме срабатывания по фронту программного интерфейса `epoll` (см. раздел 59.4). Он возвращается, когда потоковый сокет на другом конце соединения закрывает свой записывающий канал. Этот флаг позволяет приложению задействовать интерфейс `epoll`, срабатывающий по фронту, чтобы упростить распознавание удаленного закрытия (альтернативой было бы определить наличие флага `POLLIN` и выполнить операцию `read()`, которая в случае удаленного закрытия возвращает `0`).

59.2.4. Сравнение вызовов select() и poll()

В этом подразделе мы сосредоточим внимание на сходствах и отличиях, присущих вызовам `select()` и `poll()`.

Отличия в программных интерфейсах

Ниже перечислены отдельные различия между программными интерфейсами `select()` и `poll()`.

- Использование типа данных `fd_set` ограничивает максимальное количество файловых дескрипторов, которое можно отследить с помощью вызова `select()` (`FD_SETSIZE`). В Linux это ограничение по умолчанию равно 1024, а его изменение требует повторной компиляции приложения. Для сравнения: вызов `poll()` не имеет никаких внутренних ограничений на диапазон отслеживаемых файловых дескрипторов.
- Аргументы типа `fd_set` возвращают результат, поэтому, если вызов `select()` выполняется в цикле, их нужно заново инициализировать на каждой итерации. Вызов `poll()` применяет отдельные поля `events` (для ввода) и `revents` (для вывода), так что на него данное требование не распространяется.
- Вызов `select()` обеспечивает более высокую точность времени ожидания по сравнению с `poll()` (микросекунды вместо миллисекунд). Хотя в обоих случаях точность ограничивается системными часами.
- Если один из наблюдаемых файловых дескрипторов был закрыт, то вызов `poll()` установит бит `POLLNVAL` в поле `revents` соответствующего дескриптора. Для сравнения: вызов `select()` просто возвращает `-1` с ошибкой `EBADF`; чтобы определить, какой именно дескриптор закрылся, придется проверять ошибки соответствующих системных вызовов ввода/вывода. Однако в большинстве случаев эта разница несущественна, так как приложение обычно способно само отслеживать закрывающиеся дескрипторы.

Портируемость

Вызов `select()` традиционно является более популярным по сравнению с `poll()`. В наши дни оба интерфейса входят в стандарт SUSv3 и широко применяются в современных системах. Тем не менее, как отмечалось в подразделе 59.2.3, поведение вызова `poll()` может варьироваться в зависимости от реализации.

Производительность

Производительность вызовов `poll()` и `select()` одинакова, если выполняется любое из следующих условий:

- диапазон файловых дескрипторов, за которыми нужно наблюдать, достаточно узок (выбран небольшой максимальный номер дескриптора);
- отслеживается большое количество файловых дескрипторов, но все они достаточно плотно упакованы (это значит, что контролируются все или большинство дескрипторов в заданном диапазоне).

Однако производительность вызовов `select()` и `poll()` может заметно отличаться в ситуации, когда набор отслеживаемых файловых дескрипторов разрежен, то есть если максимальный номер дескриптора, N, является большим, но в диапазоне от 0 до N находится один или несколько элементов. В данном случае `poll()` может работать быстрее, чем `select()`. Чтобы понять, почему так происходит, рассмотрим передаваемые

этим двум системным вызовам аргументы. Вызов `select()` принимает один или несколько наборов с файловыми дескрипторами и целое число, `nfds`, которое на единицу больше, чем максимальный номер отслеживаемых дескрипторов в каждом из наборов. Значение `nfds` не зависит от того, наблюдаем ли мы за всеми дескрипторами в диапазоне от `0` до `(nfds - 1)` или только за одним из них, `(nfds - 1)`. В обоих случаях ядро должно проверить элементы `nfds` в каждом из наборов, чтобы определить, какие именно дескрипторы нужно контролировать. Для сравнения: вызову `poll()` передаются лишь те дескрипторы, которые нас интересуют, и ядро проверяет только их.

Мы еще вернемся к производительности вызовов `select()` и `poll()` в подразделе 59.4.5, где они будут сравниваться с интерфейсом `epoll`.

59.2.5. Проблемы, присущие вызовам select() и poll()

Системные вызовы `select()` и `poll()` являются портируемыми, устоявшимися и широко распространенными инструментами для мониторинга готовности множества файловых дескрипторов. Однако при мониторинге большого количества дескрипторов эти программные интерфейсы испытывают определенные трудности.

- При каждом вызове `select()` или `poll()` ядро вынуждено проверять все указанные файловые дескрипторы, чтобы определить их готовность. При большом количестве элементов, плотно размещенных в некоем диапазоне, для этого требуется намного больше времени, чем для следующих двух операций.
- При каждом вызове `select()` или `poll()` программа должна передавать ядру структуру данных, описывающую все интересующие нас файловые дескрипторы, а после их проверки ядро должно вернуть обратно измененную версию этой структуры (кроме того, перед каждым вызовом `select()` нужно инициализировать структуру данных). В случае с `poll()` размер структуры прямо пропорционален количеству подконтрольных дескрипторов, поэтому, если она достаточно большая, на ее копирование в пространство ядра и обратно может уходить довольно много процессорного времени. Применительно к `select()` размер структуры данных является фиксированным и равен `FD_SETSIZE`; он не зависит от того, сколько файловых дескрипторов отслеживаются.
- После вызова `select()` или `poll()` программа должна проверить каждый элемент полученной структуры данных, чтобы узнать, какие файловые дескрипторы являются готовыми.

Следствием приведенных выше замечаний является тот факт, что процессорное время, которое уходит на выполнение вызовов `select()` и `poll()`, прямо пропорционально количеству отслеживаемых файловых дескрипторов (больше подробностей см. в подразделе 59.4.5). И если это количество велико, то у нас могут возникнуть проблемы.

Плохое масштабирование вызовов `select()` и `poll()` вызвано простым ограничением их программных интерфейсов: обычно для проверки одного и того же набора файловых дескрипторов выполняются циклические вызовы; но ядро не запоминает содержимое данного набора, поэтому его нужно передавать снова и снова.

Ввод/вывод на основе сигналов и интерфейс `epoll` (будут рассмотрены в следующих разделах) позволяют ядру записывать набор файловых дескрипторов, в которых заинтересован процесс. Это устраняет проблемы с масштабированием, присущие вызовам `select()` и `poll()`, и обеспечивает производительность, зависящую не от количества отслеживаемых дескрипторов, а от частоты событий ввода/вывода. Следовательно, ввод/вывод на основе сигналов и интерфейс `epoll` являются более предпочтительными в ситуациях, когда дескрипторов слишком много.

59.3. Ввод/вывод на основе сигналов

Если ввод/вывод мультиплексируется, то системный вызов (`select()` или `poll()`) делается для проверки того, возможно ли выполнить чтение или запись для заданного дескриптора. При вводе/выводе, основанном на сигналах, процесс просит ядро послать ему сигнал о том, что файловый дескриптор поддерживает операции чтения или записи. После этого процесс может выполнять любые другие действия, а о доступности ввода/вывода он будет уведомлен с помощью сигнала. Для использования данной методики программа должна выполнить следующие шаги.

1. Установить обработчик для сигнала, передаваемого механизмом ввода/вывода. По умолчанию это сигнал `SIGIO`.
2. Установить *владельца* файлового дескриптора — то есть процесс или группу процессов, которые будут получать сигнал о возможности чтения или записи. Обычно владельцем является вызывающий процесс. Чтобы его установить, выполняется операция `fcntl() F_SETOWN` следующего вида:

   ```
   fcntl(fd, F_SETOWN, pid);
   ```

3. Включить неблокирующий ввод/вывод путем установки флага состояния открытого файла `O_NONBLOCK`.
4. Включить ввод/вывод на основе сигналов путем установки флага состояния открытого файла `O_ASYNC`. Это можно сделать одновременно с предыдущим шагом, так как в обоих случаях нужно выполнять операцию `fcntl() F_SETFL` (см. раздел 5.3). Например:

   ```
   flags = fcntl(fd, F_GETFL);         /* Текущие флаги */
   fcntl(fd, F_SETFL, flags | O_ASYNC | O_NONBLOCK);
   ```

5. После этого вызывающий процесс может заняться другими задачами. Когда ввод/вывод станет возможным, ядро передаст ему сигнал и вызовет соответствующий обработчик, установленный в шаге 1.
6. Ввод/вывод на основе сигналов предоставляет уведомления, срабатывающие по фронту (см. подраздел 59.1.1). То есть процесс, получив уведомление, должен прочитать или записать как можно больше данных. Исходя из того, что файловый дескриптор не блокируется, это значит циклическое выполнение системных вызовов ввода/вывода, пока один из них не завершится ошибкой `EAGAIN` или `EWOULDBLOCK`.

На следующих страницах мы сначала рассмотрим пример использования ввода/вывода на основе сигналов, а затем более подробно объясним некоторые из шагов, приведенных выше.

> Когда-то ввод/вывод на основе сигналов называли асинхронным, что отразилось на имени соответствующего флага состояния открытого файла (O_ASYNC). Но на сегодняшний день асинхронным вводом/выводом считается механизм, предоставляемый стандартом POSIX AIO. Задействуя этот механизм, процесс просит ядро выполнить операцию ввода/вывода, а то, в свою очередь, инициирует соответствующую операцию, но сразу же передает управление обратно вызывающему процессу; позже, когда операция завершается (успешно или с ошибкой), процесс получает уведомление.
>
> Флаг O_ASYNC появился в спецификации POSIX.1g, но так и не вошел в стандарт SUSv3, поскольку описание его поведения было признано недостаточно подробным.
>
> В некоторых UNIX-системах, особенно более старых, вызов fcntl() не поддерживает константу O_ASYNC. Вместо этого применяется константа FASYNC; в библиотеке glibc она является синонимом O_ASYNC.

Пример программы

Простой пример использования ввода/вывода на основе сигналов показан в листинге 59.3. Данная программа производит вышеприведенные шаги с целью позволить выполнять стандартный ввод с помощью сигналов, а затем переключает терминал в режим cbreak (см. подраздел 58.6.3), чтобы данные можно было вводить посимвольно. После этого она входит в бесконечный цикл, занимаясь инкрементацией переменной `cnt`, и ждет возможность записи. Когда ввод становится доступным, обработчик `SIGIO` устанавливает флаг `gotSigio`, который отслеживается главной программой. Обнаружив данный флаг, она считывает все введенные символы и выводит их вместе с текущим значением переменной `cnt`. При считывании знака `#` программа завершается.

Ниже показано, что произойдет, если запустить нашу программу и ввести несколько символов `x`, завершив ввод знаком `#`:

```
$ ./demo_sigio
cnt=37; read x
cnt=100; read x
cnt=159; read x
cnt=223; read x
cnt=288; read x
cnt=333; read #
```

Листинг 59.3. Использование ввода/вывода на основе сигналов в терминале

altio/demo_sigio.c

```
#include <signal.h>
#include <ctype.h>
#include <fcntl.h>
#include <termios.h>
#include "tty_functions.h"        /* Объявление ttySetCbreak() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigio = 0;
        /* Устанавливаем ненулевое значение при получении SIGIO */

static void
sigioHandler(int sig)
{
    gotSigio = 1;
}

int
main(int argc, char *argv[])
{
    int flags, j, cnt;
    struct termios origTermios;
    char ch;
    struct sigaction sa;
    Boolean done;

    /* Устанавливаем обработчик сигнала, оповещающего о возможности ввода/вывода */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = sigioHandler;
    if (sigaction(SIGIO, &sa, NULL) == -1)
        errExit("sigaction");
```

```
    /* Определяем процесс-владелец, который получит сигнал */

    if (fcntl(STDIN_FILENO, F_SETOWN, getpid()) == -1)
        errExit("fcntl(F_SETOWN)");

    /* Разрешаем передачу сигнала и делаем ввод/вывод
       неблокирующим для заданного дескриптора */

    flags = fcntl(STDIN_FILENO, F_GETFL);
    if (fcntl(STDIN_FILENO, F_SETFL, flags | O_ASYNC | O_NONBLOCK) == -1)
        errExit("fcntl(F_SETFL)");

    /* Переключаем терминал в режим cbreak */
    if (ttySetCbreak(STDIN_FILENO, &origTermios) == -1)
        errExit("ttySetCbreak");

    for (done = FALSE, cnt = 0; !done ; cnt++) {
        for (j = 0; j < 100000000; j++)
            continue;               /* Немного замедляем главный цикл */

        if (gotSigio) {             /* Доступен ли ввод? */
            gotSigio = 0;

    /* Считываем весь доступный ввод, пока не случится ошибка (вероятно,
       EAGAIN), не обнаружится конец файла (что невозможно в режиме cbreak)
       или не будет прочитан символ # */

            while (read(STDIN_FILENO, &ch, 1) > 0 && !done) {
                printf("cnt=%d; read %c\n", cnt, ch);
                done = ch == '#';
            }
        }
    }

    /* Восстанавливаем исходные параметры терминала */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &origTermios) == -1)
        errExit("tcsetattr");
    exit(EXIT_SUCCESS);
}
```

altio/demo_sigio.c

Устанавливайте обработчик сигнала до включения ввода/вывода на основе сигналов

По умолчанию действием сигнала `SIGIO` является завершение процесса, так что его обработчик следует устанавливать до включения ввода/вывода на основе сигналов. Если поменять местами эти две операции, то образует своеобразный «зазор», на протяжении которого появление сигнала `SIGIO` будет завершать процесс.

В ряде реализаций UNIX сигнал SIGIO по умолчанию игнорируется.

59.3.1. Установка владельца файлового дескриптора

Владелец файлового дескриптора устанавливается с помощью операции `fcntl()` следующего вида:

```
fcntl(fd, F_SETOWN, pid);
```

Для получения сигнала о возможности ввода/вывода можно выбрать как один процесс, так и все процессы в группе. Если значение `pid` больше нуля, то оно интерпретируется как идентификатор процесса. Если же является отрицательным, то его модуль берется в качестве идентификатора группы процессов.

В старых реализациях UNIX флаги FIOSETOWN или SIOCSPGRP в вызове ioctl() имели тот же эффект, что и F_SETOWN. В Linux они поддерживаются с целью совместимости.

Обычно аргумент `pid` равен идентификатору вызывающего процесса (это позволяет передавать сигнал процессу, который удерживает открытым файловый дескриптор). Но можно указать и другой процесс или целую группу (например, группу вызывающего процесса); таким образом сигнал будет доставляться всем адресатам, проходя проверку прав доступа, описанную в разделе 20.5. При этом отправителем считается процесс, выполняющий операцию `F_SETOWN`.

Операция `fcntl() F_GETOWN` возвращает идентификатор процесса или группы процессов, которые получат сигнал о возможности ввода/вывода для заданного дескриптора:

```
id = fcntl(fd, F_GETOWN);
if (id == -1)
    errExit("fcntl");
```

Идентификатор группы процессов возвращается в виде отрицательного числа.

В старых реализациях UNIX аналогами флага F_SETOWN для вызова ioctl() были флаги FIOGETOWN и SIOCGPGRP. Оба они поддерживаются в Linux.

Ограничения, традиционно применяемые в Linux к системным вызовам (в некоторых архитектурах, таких как x86), имеют следующие последствия: если идентификатор группы процессов, владеющей файловым дескриптором, меньше `4096`, то вместо возвращения его отрицательного значения операция `fcntl() F_GETOWN` завершается ошибкой. Таким образом, функция-обертка `fcntl()` возвращает `-1`, а глобальная переменная `errno` содержит (положительный) идентификатор группы процессов. Это является следствием того факта, что интерфейс системных вызовов ядра сигнализирует об ошибках, присваивая `errno` отрицательные значения. Существует несколько ситуаций, когда такие результаты следует отличать от успешного вызова, возвращающего корректное отрицательное значение. Для этого библиотека glibc интерпретирует отрицательные итоговые результаты в диапазоне от `-1` до `-4095` в качестве ошибки, копирует их значение (по модулю) в переменную `errno` и делает так, что функция возвращает вызывающему процессу `-1`. Обычно данных действий достаточно для работы с теми немногими системными вызовами, которые способны возвращать отрицательный результат.

Операция `fcntl() F_GETOWN` является единственным реальным случаем, когда описанный подход не работает. Это значит следующее: приложение, выбирающее для получения сигналов о «готовности ввода/вывода» группу процессов (что довольно необычно), не может достоверно определить группу, владеющую дескриптором, задействуя флаг `F_GETOWN`.

Начиная с версии 2.11 функция-обертка fcntl() в библиотеке glibc больше не имеет проблемы с использованием флага F_GETOWN для групп процессов, чей идентификатор меньше 4096. Для этого данный вызов реализован в пользовательском пространстве на основе операции F_GETOWN_EX (см. подраздел 59.3.3), которая поддерживается в Linux 2.6.32 и выше.

59.3.2. Когда генерируется сигнал о возможности ввода/вывода?

В этом подразделе вы узнаете, когда именно генерируется сигнал о возможности ввода/вывода для разных типов файлов.

Терминалы и псевдотерминалы

В случае с терминалами и псевдотерминалами сигнал генерируется в момент, когда становится доступным новый ввод, даже если предыдущий ввод еще не был прочитан. Сигнал о возможности ввода также передается при обнаружении конца файла (только для терминала).

Терминалы не получают сигналы о возможности вывода и разрыве соединения.

С версии ядра 2.4.19 Linux передает сигнал о возможности вывода вторичному псевдотерминалу. Он генерируется всякий раз, когда ввод принимается на первичной стороне.

Именованные каналы и очереди FIFO

Для считывающего конца именованного канала или очереди FIFO сигнал генерируется в следующих ситуациях:

- данные записаны в канал (даже если там уже находится непрочитанный ввод);
- записывающий конец канала закрыт.

Для записывающего конца именованного канала или очереди FIFO сигнал генерируется в следующих ситуациях:

- чтение из канала увеличивает объем свободного пространства в нем, благодаря чему становится возможным записать без блокировки `PIPE_BUF` байтов;
- считывающий конец канала закрыт.

Сокеты

Ввод/вывод на основе сигналов поддерживает датаграммные сокеты в UNIX- и интернет-доменах. Сигнал генерируется в следующих ситуациях:

- в сокет поступает входящая датаграмма (даже если в очереди находятся другие непрочитанные датаграммы);
- в сокете происходит асинхронная ошибка.

Ввод/вывод на основе сигналов работает также и для потоковых сокетов в обоих доменах. Сигнал генерируется в следующих ситуациях:

- слушающий сокет принял новое соединение;
- завершается TCP-запрос `connect()`; то есть активный конец TCP-соединения входит в состояние `ESTABLISHED` (см. рис. 57.5). Условие не срабатывает для сокетов в домене UNIX;
- сокет принял новый ввод (даже если в нем уже доступны непрочитанные данные);
- удаленная сторона закрывает свою записывающую часть соединения с помощью вызова `shutdown()` или все соединение целиком, используя вызов `close()`;
- становится возможным вывод из сокета (например, когда освобождается место в его исходящем буфере);
- в сокете происходит асинхронная ошибка.

Файловые дескрипторы inotify

Сигнал генерируется, когда становится доступным файловый дескриптор `inotify`, то есть если с любым из файлов, которые он отслеживает, происходит какое-то событие.

59.3.3. Эффективное использование ввода/вывода на основе сигналов

В приложениях, которым нужно наблюдать за большим количеством (то есть тысячами) файловых дескрипторов (например, отдельными видами сетевых серверов), ввод/вывод на основе сигналов может обеспечить значительное увеличение производительности по сравнению с вызовами `select()` и `poll()`. Дело в том, что ядро «запоминает» список отслеживаемых дескрипторов и оповещает программу только в момент, когда с ними происходят события ввода/вывода. Таким образом, производительность программы, применяющей ввод/вывод на основе сигналов, масштабируется в зависимости от количества событий, а не подконтрольных файловых дескрипторов.

Чтобы применять весь потенциал этой методики, нужно выполнить два шага:

- воспользоваться вызовом `fcntl()` с флагом `F_SETSIG` (поддерживается только в Linux) с целью указать сигнал реального времени, который должен быть доставлен вместо `SIGIO`, когда станет возможным ввод/вывод для заданного файлового дескриптора;
- указать флаг при использовании вызова `sigaction()`, чтобы установить обработчик сигнала реального времени, указанного в предыдущем шаге (см. раздел 21.4).

Операция `fcntl() F_SETSIG` позволяет указать альтернативный сигнал, который должен быть доставлен вместо `SIGIO` как оповещение о возможности ввода/вывода для файлового дескриптора:

```
if (fcntl(fd, F_SETSIG, sig) == -1)
    errExit("fcntl");
```

`F_GETSIG` выполняет действие, обратное операции `F_SETSIG`: извлекает сигнал, установленный на данный момент для файлового дескриптора:

```
sig = fcntl(fd, F_GETSIG);
if (sig == -1)
    errExit("fcntl");
```

Для того чтобы получить определения констант F_SETSIG и F_GETSIG из заголовочного файла <fcntl.h>, следует определить макрос проверки возможностей _GNU_SOURCE.

Флаг `F_SETSIG` для изменения сигнала, оповещающего о возможности ввода/вывода, служит двум целям, каждая из которых является обязательной при мониторинге большого количества событий ввода/вывода для множества файловых дескрипторов.

- По умолчанию для оповещения о возможности ввода/вывода применяется стандартный сигнал `SIGIO`, который минует очередь. Если события начнут поступать в момент, когда этот сигнал заблокирован (возможно, его обработчик уже вызван), то все они, кроме первого, будут утеряны. При использовании вместо `SIGIO` сигнала реального времени `F_SETSIG` уведомления будут складываться в очередь.
- Если для установки обработчика сигнала задействован вызов `sigaction()` с флагом `SA_SIGINFO` в поле `sa.sa_flags`, то в качестве второго аргумента обработчику передается структура `siginfo_t` (см. раздел 21.4). Поля этой структуры определяют файловый дескриптор и тип события, которое с ним произошло.

Стоит отметить, что для передачи обработчику сигнала корректной структуры `siginfo_t` необходимо использовать *оба* флага: `F_SETSIG` и `SA_SIGINFO`.

Если при выполнении операции `F_SETSIG` присвоить полю `sig` значение `0`, то мы вернемся к поведению по умолчанию: будет сгенерирован сигнал `SIGIO`, а его обработчик не получит дополнительный аргумент `siginfo_t`.

В контексте событий, связанных с возможностью ввода/вывода, нас могут заинтересовать следующие поля структуры `siginfo_t`, передаваемой обработчику сигнала:

- `si_signo` — номер сигнала, на который среагировал обработчик. Это значение совпадает с первым аргументом, передаваемым обработчику сигнала.
- `si_fd` — файловый дескриптор, где произошло событие ввода/вывода.
- `si_code` — код, указывающий на тип произошедшего события. Значения, которые могут храниться в этом поле, а также их описания приводятся в табл. 59.7.
- `si_band` — битовая маска, содержащая те же биты, возвращаемые вызовом `poll()` в аналогичном поле структуры `revents`. Значение поля `si_code` соотносится с параметрами битовой маски `si_band` один к одному (см. табл. 59.7).

Таблица 59.7. Значения полей si_code и si_band в структуре siginfo_t для оповещений о возможности ввода/вывода

si_code	Значение маски si_band	Описание
POLL_IN	POLLIN | POLLRDNORM	Доступен ввод; конец файла
POLL_OUT	POLLOUT | POLLWRNORM | POLLWRBAND	Доступен вывод
POLL_MSG	POLLIN | POLLRDNORM | POLLMSG	Доступно входящее сообщение (не используется)
POLL_ERR	POLLERR	Ошибка ввода/вывода
POLL_PRI	POLLPRI | POLLRDNORM	Доступен высокоприоритетный ввод
POLL_HUP	POLLHUP | POLLERR	Произошел разрыв соединения

Применение операции `F_SETSIG` можно и дальше оптимизировать, если речь идет о программе, которая занимается исключительно обработкой ввода. Вместо того чтобы отслеживать события ввода/вывода, задействуя обработчик, можно заблокировать назначенный сигнал и затем принимать сигналы из очереди, используя вызов `sigwaitinfo()` или `sigtimedwait()` (см. раздел 22.10). Обе эти операции возвращают структуру `siginfo_t`, содержащую ту же информацию, что передается обработчику сигнала, установленного с помощью флага `SA_SIGINFO`. Такой способ приема сигналов возвращает нас к асинхронной модели обработки событий, но обеспечивает куда более эффективное получение уведомлений о файловых дескрипторах, в которых произошли события ввода/вывода, если сравнивать с вызовами `select()` и `poll()`.

Переполнение очереди сигналов

В разделе 22.8 было указано: максимальное количество сигналов реального времени, которые могут попасть в очередь, ограничено. При достижении данного ограничения ядро возвращается к уведомлению о возможности ввода/вывода с помощью стандартного сигнала `SIGIO`. Так процесс может узнать, что очередь с сигналами переполнена. Вместе с этим теряется информация о том, в каких файловых дескрипторах происходят события ввода/вывода, поскольку сигналы `SIGIO` не попадают в очередь (кроме того, обработчик `SIGIO` не получает аргумент `siginfo_t`; это значит, что он не может определить, какой именно дескриптор сгенерировал сигнал).

Можно снизить вероятность переполнения очереди сигналов, увеличив ее максимальную вместимость, как описывается в разделе 22.8. Однако это не избавляет от необходимо-

сти следить за потенциальным переполнением. Хорошо спроектированные приложения, использующие операцию F_SETSIG для получения уведомлений реального времени о возможности ввода/вывода, должны также устанавливать обработчик SIGIO. При получении данного сигнала приложение может очистить очередь с помощью вызова sigwaitinfo() и временно переключиться на применение интерфейсов select() или poll(), которые позволят запрашивать списки файловых дескрипторов с новыми событиями ввода/вывода.

Использование ввода/вывода на основе сигналов в многопоточных приложениях

Начиная с версии 2.6.32 ядро Linux предоставляет два новых, нестандартных флага для вызова fcntl(), позволяющих указать адресата сигналов, оповещающих о «возможности ввода/вывода»: F_SETOWN_EX и F_GETOWN_EX.

Операция F_SETOWN_EX похожа на F_SETOWN, но помимо процесса и группы процессов с ее помощью можно также указать поток выполнения. При ее использовании третьим аргументом вызова fcntl() является указатель на структуру следующего вида:

```
struct f_owner_ex {
    int   type;
    pid_t pid;
};
```

Поле type определяет то, как будет интерпретироваться поле pid, и может принимать следующие значения.

- F_OWNER_PGRP — поле pid содержит идентификатор группы процессов, которая будет получать сигналы о возможности ввода/вывода. В отличие от F_SETOWN, идентификатор представляет собой положительное число.
- F_OWNER_PID — поле pid содержит идентификатор процесса, который будет получать сигналы о возможности ввода/вывода.
- F_OWNER_TID — поле pid содержит идентификатор потока, который будет получать сигналы о возможности ввода/вывода. Идентификатор, хранящийся в поле pid, представляет собой значение, возвращаемое вызовом clone() или gettid().

Операция F_GETOWN_EX выполняет действие, обратное F_GETOWN_EX. Она использует структуру f_owner_ex, на которую указывает третий аргумент вызова fcntl(), чтобы вернуть параметры, установленные предыдущей операцией F_SETOWN_EX.

> Операции F_SETOWN_EX и F_GETOWN_EX представляют идентификаторы группы процессов в виде положительных чисел, поэтому F_GETOWN_EX не имеет тех проблем, которые присущи ранее описанному флагу F_GETOWN (когда идентификаторы группы процессов меньше 4096).

59.4. Программный интерфейс epoll

По аналогии с вводом/выводом на основе мультиплексирующих системных вызовов и сигналов программный интерфейс epoll (от англ. event poll — «наблюдение за событиями») используется для контроля готовности множества файловых дескрипторов. Главные преимущества epoll перечислены ниже.

- При мониторинге большого количества файловых дескрипторов интерфейс epoll масштабируется гораздо лучше, чем вызовы select() и poll().

- Интерфейс `epoll` поддерживает уведомления по уровню и по фронту. Для сравнения: вызовы `select()` и `poll()` способны уведомлять лишь по уровню, а ввод/вывод на основе сигналов — только по фронту.

Интерфейс `epoll` и ввод/вывод на основе сигналов имеют аналогичную производительность. Хотя первый обладает некоторыми преимуществами:

- позволяет избежать сложностей, связанных с обработкой сигналов (например, переполнение очереди сигналов);
- обеспечивает лучшую гибкость при описании интересующих событий (например, можно проверять, готов ли файловый дескриптор сокета к чтению, записи или к обеим этим операциям).

Программный интерфейс `epoll` поддерживается только в Linux версии 2.6 и выше.

Главной структурой данных в этом интерфейсе является *экземпляр epoll*, доступ к которому происходит через дескриптор открытого файла. Упомянутый дескриптор не используется для ввода/вывода, а является ссылкой на структуру данных ядра, служащей двум целям:

- запись списка файловых дескрипторов, в отслеживании которых заинтересован текущий процесс *(список интереса)*;
- предоставление списка файловых дескрипторов, готовых к вводу/выводу *(список готовности)*.

Второй список дескрипторов является подмножеством первого.

Для каждого файлового дескриптора, наблюдаемого с помощью интерфейса `epoll`, можно указать битовую маску, определяющую события, о которых мы хотим узнавать. Эта битовая маска аналогична той, что используется для вызова `poll()`.

Программный интерфейс `epoll` состоит из трех системных вызовов.

- Вызов `epoll_create()` создает экземпляр `epoll` и возвращает файловый дескриптор, который на него ссылается.
- Вызов `epoll_ctl()` изменяет список отслеживаемых дескрипторов, связанных с экземпляром `epoll`. Он позволяет добавлять новые и удалять существующие дескрипторы, а также редактировать маску, описывающую интересующие нас события.
- Вызов `epoll_wait()` возвращает элементы списка готовых дескрипторов, связанных с экземпляром `epoll`.

59.4.1. Создание экземпляра epoll: вызов epoll_create()

Системный вызов `epoll_create()` создает новый экземпляр `epoll` с изначально пустым списком интереса.

```
#include <sys/epoll.h>

int epoll_create(int size);
```

Возвращает файловый дескриптор при успешном завершении или `-1` при ошибке

Аргумент `size` обозначает количество дескрипторов, которые мы хотим отслеживать с помощью экземпляра `epoll`. Это не максимальное значение, а некий ориентир; на его основе ядро будет подбирать исходный размер своих внутренних структур данных (начиная с версии Linux 2.6.8 аргумент `size` игнорируется, поскольку в новой реализации указанные сведения больше не нужны).

В качестве результата функция `epoll_create()` возвращает файловый дескриптор, ссылающийся на новый экземпляр `epoll` и позволяющий работать с ним в других системных вызовах данного интерфейса. Когда необходимость в этом дескрипторе пропадает, его следует закрыть стандартным способом — с помощью вызова `close()`. Экземпляр `epoll` уничтожается с закрытием последнего файлового дескриптора, который на него ссылается; притом освобождаются все связанные с ним ресурсы (в результате множественных вызовов `fork()` или операций дублирования, таких как `dup()`, на один и тот же экземпляр `epoll` может ссылаться сразу несколько дескрипторов).

Начиная с версии 2.6.27 ядро Linux поддерживает новый системный вызов epoll_create1(). Он выполняет те же действия, что и epoll_create(), но вместо устаревшего аргумента size принимает аргумент flags, с помощью которого можно влиять на поведение данного системного вызова. На сегодняшний день поддерживается всего одно значение, EPOLL_CLOEXEC, заставляющее ядро включить для нового файлового дескриптора флаг FD_CLOEXEC. Он может пригодиться по тем же причинам, что и флаг O_CLOEXEC для вызова open(), описанный в подразделе 4.3.1.

59.4.2. Редактирование списка интереса epoll: вызов epoll_ctl()

Системный вызов `epoll_ctl()` изменяет список интереса экземпляра `epoll`, на который ссылается файловый дескриптор `epfd`.

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Аргумент `fd` обозначает файловый дескриптор из списка интереса, параметры которого нужно изменить. Он может ссылаться на именованный канал, очередь FIFO, сокет, очередь сообщений POSIX, экземпляр `inotify`, терминал, устройство или даже другой дескриптор `epoll` (то есть можно сформировать некую иерархию отслеживаемых дескрипторов). Однако в качестве `fd` нельзя передать дескриптор обычного файла или каталога (в этом случае вызов завершается ошибкой `EPERM`).

Аргумент `op` обозначает операцию, которую нужно выполнить. Он может принимать одно из следующих значений.

- `EPOLL_CTL_ADD` — добавляет файловый дескриптор `fd` в список интереса экземпляра `epfd`. Набор событий, которые мы хотим отслеживать для этого дескриптора, задается в виде буфера, на какой указывает аргумент `ev` (см. ниже). При попытке добавления дескриптора, уже находящегося в списке интереса, вызов `epoll_ctl()` завершается ошибкой `EEXIST`.
- `EPOLL_CTL_MOD` — изменяет параметры событий для дескриптора `fd`, используя информацию в буфере, на который указывает аргумент `ev`. Если попытаться изменить параметры дескриптора, не входящего в список интереса, то вызов `epoll_ctl()` завершится ошибкой `ENOENT`.
- `EPOLL_CTL_DEL` — удаляет файловый дескриптор `fd` из списка интереса для экземпляра `epfd`. Аргумент `ev` при этом игнорируется. При попытке удалить файловый дескриптор, не входящий в список интереса, вызов `epoll_ctl()` завершится ошибкой `ENOENT`. Закрытие файлового дескриптора автоматически удаляет его из всех списков интереса `epoll`, в которые он входит.

Аргумент ev является указателем на структуру типа epoll_event следующего вида:

```
struct epoll_event {
    uint32_t     events;          /* События epoll (битовая маска) */
    epoll_data_t data;            /* Пользовательские данные */
};
```

Поле data структуры epoll_event выглядит так:

```
typedef union epoll_data {
    void        *ptr;             /* Указатель на пользовательские данные */
    int          fd;              /* Файловый дескриптор */
    uint32_t     u32;             /* 32-разрядное целое число */
    uint64_t     u64;             /* 64-разрядное целое число */
} epoll_data_t;
```

Аргумент ev определяет параметры для файлового дескриптора fd.

- Вложенное поле events представляет собой битовую маску, определяющую набор интересующих нас событий для дескриптора fd. Более подробно о ее битовых значениях будет рассказано в следующем разделе.
- Вложенное поле data является объединением, где один из элементов можно задействовать для описания информации, которая должна возвращаться обратно вызывающему процессу (через вызов epoll_wait()) в случае готовности дескриптора fd.

Пример использования вызовов epoll_create() и epoll_ctl() продемонстрирован в листинге 59.4.

Листинг 59.4. Применение вызовов epoll_create() и epoll_ctl()

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);
if (epfd == -1)
    errExit("epoll_create");

ev.data.fd = fd;
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

Ограничение max_user_watches

Для каждого файлового дескриптора, регистрируемого в списке интереса epoll, выделяется небольшой объем памяти ядра, которую нельзя сбрасывать на диск. Поэтому ядро предоставляет интерфейс, определяющий максимальное количество дескрипторов, доступных для регистрации каждому пользователю. Значение данного ограничения можно просматривать и изменять с помощью файла max_user_watches в каталоге /proc/sys/fs/epoll (доступен только в Linux). По умолчанию данное ограничение вычисляется на основе доступной в системе памяти (см. страницу epoll(7) руководства).

59.4.3. Ожидание событий: вызов epoll_wait()

Системный вызов epoll_wait() возвращает информацию о готовых файловых дескрипторах из экземпляра epoll, на который указывает дескриптор epfd. За одно выполнение этот вызов может вернуть сведения о множестве готовых дескрипторов.

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents,
               int timeout);
```

Возвращает количество готовых файловых дескрипторов, 0, если истекло время ожидания, или -1 при ошибке

Информация о готовых файловых дескрипторах возвращается в виде массива структур `epoll_event`, на который указывает аргумент `evlist` (структура `epoll_event` была описана в предыдущем разделе). Массив `evlist` выделяется вызывающим процессом, а количество содержащихся в нем элементов определяется с помощью аргумента `maxevents`.

Каждый элемент массива `evlist` возвращает информацию об отдельном готовом файловом дескрипторе. Вложенное поле `events` содержит маску событий, произошедших с дескриптором. Во вложенном поле `data` хранится значение, которое было указано внутри `ev.data` при регистрации нашей заинтересованности в этом конкретном дескрипторе с помощью вызова `epoll_ctl()`. Стоит отметить, что поле `data` всего лишь позволяет найти номер файлового дескриптора, связанного с текущим событием. Следовательно, во время вызова `epoll_ctl()`, помещающего файловый дескриптор в список интереса, мы должны либо назначить его номер полю `ev.data.fd` (как показано в листинге 59.3), либо присвоить `ev.data.ptr` указатель на структуру с данным номером.

Аргумент `timeout` определяет поведение вызова `epoll_wait()`, связанное с блокировкой:

- значение `-1` приводит к блокировке до тех пор, пока в одном из дескрипторов в списке интереса для экземпляра `epfd` не возникнет событие или не будет перехвачен сигнал;
- значение `0` инициирует неблокирующую проверку наличия событий в файловых дескрипторах списка интереса для экземпляра `epfd`;
- если значение больше нуля, то вызов блокируется на `timeout` миллисекунд, пока в одном из дескрипторов в списке интереса для экземпляра `epfd` не возникнет событие или не будет перехвачен сигнал.

В случае успеха вызов `epoll_wait()` возвращает количество элементов, помещенных в массив `evlist`, или `0`, если за указанный промежуток времени (`timeout`) не нашлось ни одного готового дескриптора. В случае ошибки `epoll_wait()` возвращает `-1`, меняя соответствующим образом значение глобальной переменной `errno`.

В многопоточных программах возможна ситуация, когда файловые дескрипторы, добавляемые в список интереса для экземпляра `epoll` одним потоком (с помощью `epoll_ctl()`), уже отслеживаются другим (благодаря `epoll_wait()`). Изменения, вносимые в список интереса, немедленно учитываются, а вызов `epoll_wait()` возвращает сведения о готовности файловых дескрипторов, добавленных только что.

События интерфейса epoll

Битовые значения, которые можно указать в поле `ev.events` при вызове `epoll_ctl()` и которые затем помещаются в поля `evlist[].events`, возвращаемые вызовом `epoll_wait()`, перечислены в табл. 59.8. Если не брать во внимание префикс `E`, большинство этих битов имеют те же имена, что и соответствующие биты событий, используемые в вызове `poll()`. (Исключения составляют константы `EPOLLET` и `EPOLLONESHOT`, которые чуть ниже будут описаны более подробно.) Причина такого соответствия — эти биты (неважно, передаются ли они в `epoll_ctl()` или возвращаются из `epoll_wait()`) означают то же самое, что и аналогичные биты событий вызова `poll()`.

Таблица 59.8. Значения битовой маски поля events интерфейса epoll

Бит	Передается в epoll_ctl()?	Возвращается из epoll_wait()?	Описание
EPOLLIN	*	*	Могут быть прочитаны любые данные, кроме высокоприоритетных
EPOLLPRI	*	*	Могут быть прочитаны высокоприоритетные данные
EPOLLRDHUP	*	*	Закрытие удаленного сокета (начиная с Linux 2.6.17)
EPOLLOUT	*	*	Могут быть записаны обычные данные
EPOLLET	*		Применяются уведомления, срабатывающие по фронту
EPOLLONESHOT	*		После уведомления о событии отключается мониторинг
EPOLLERR		*	Произошла ошибка
EPOLLHUP		*	Произошел разрыв соединения

Флаг EPOLLONESHOT

По умолчанию, когда файловый дескриптор добавляется в список интереса `epoll` с помощью `epoll_ctl() EPOLL_CTL_ADD`, он остается активным (то есть последующие вызовы `epoll_wait()` будут информировать нас о его готовности) до тех пор, пока мы вручную не удалим его из этого списка, используя операцию `epoll_ctl() EPOLL_CTL_DEL`. Если мы хотим получить только одно уведомление об определенном дескрипторе, то можем указать в поле `ev.events` вызова `epoll_ctl()` флаг `EPOLLONESHOT` (доступный начиная с Linux 2.6.2). В таком случае следующий вызов `epoll_wait()`, который проинформирует о готовности заданного дескриптора, будет последним (сам дескриптор в списке интереса будет помечен как неактивный). Позже при желании можно снова включить мониторинг этого файлового дескриптора, задействуя операцию `epoll_ctl() EPOLL_CTL_MOD` (операция `EPOLL_CTL_ADD` здесь не подходит, так как неактивный дескриптор по-прежнему входит в список интереса `epoll`).

Пример программы

Использование программного интерфейса `epoll` демонстрируется в листинге 59.5. В качестве аргументов командной строки эта программа принимает путь к одному или нескольким терминалам или очередям FIFO. Программа выполняет следующие действия.

- Создает экземпляр `epoll` ❶.
- Открывает для ввода каждый файл, указанный в командной строке ❷ и добавляет соответствующие файловые дескрипторы в список интереса для экземпляра `epoll` ❸. В качестве значения, определяющего набор отслеживаемых событий, используется `EPOLLIN`.
- Входит в цикл ❹, который вызывает `epoll_wait()` ❺ для мониторинга списка интереса для экземпляра `epoll` и обрабатывает события, возвращаемые каждым вызовом. Следует сделать несколько замечаний относительно этого цикла.

- После вызова `epoll_wait()` программа проверяет наличие ошибки `EINTR` ❻, которая может произойти, если выполнение было остановлено по сигналу прямо во время работы `epoll_wait()` и возобновлено с помощью сигнала `SIGCONT` (см. раздел 21.5). В этом случае программа перезапускает вызов `epoll_wait()`.
- Если вызов `epoll_wait()` завершается успешно, программа входит в еще один цикл, чтобы проверить готовность каждого элемента в списке `evlist` ❼. Для этого в поле `events` каждого элемента ищется не только значение `EPOLLIN`, но также `EPOLLHUP` и `EPOLLERR` ❾. Оба последних события возникают в ситуации, когда закрывается другой конец очереди FIFO или разрывается соединение с терминалом. Если возвращается `EPOLLIN`, то программа считывает ввод из соответствующего файлового дескриптора и направляет его в стандартный вывод. В противном случае дескриптор закрывается ❿, а счетчик открытых файлов (`numOpenFds`) уменьшается на 1.
- Цикл завершается при закрытии всех дескрипторов открытых файлов (то есть когда переменная `numOpenFds` равна `0`).

Пример использования программы из листинга 59.5 приводится в следующих сессиях командной строки. Мы применяем два терминала. В первом наша программа отслеживает ввод в двух очередях FIFO (каждая из них открывается для чтения и будет закрыта только тогда, когда другой ее конец будет открыт для записи другим процессом; см. раздел 44.7). Во втором терминале запускаем несколько экземпляров команды `cat(1)`, которые записывают данные в ранее упомянутые очереди FIFO.

```
Terminal window 1                    Terminal window 2
$ mkfifo p q
$ ./epoll_input p q
                                     $ cat > p
Opened "p" on fd 4
                                     Нажимаем Ctrl+Z, чтобы приостановить cat
                                     [1]+  Stopped        cat >p
                                     $ cat > q
Opened "q" on fd 5
About to epoll_wait()
Нажимаем Ctrl+Z, чтобы приостановить программу epoll_input
[1]+ Stopped        ./epoll_input p q
```

Выше мы приостановили нашу программу мониторинга для получения возможности сгенерировать ввод для очередей FIFO и закрыть записывающий конец одной из них:

```
qqq
Нажимаем Ctrl+D, чтобы завершить команду "cat > q"
$ fg %1
cat >p
ppp
```

Теперь мы возобновляем нашу программу мониторинга, переводя ее в активный режим. После этого вызов `epoll_wait()` возвращает два события:

```
$ fg
./epoll_input p q
About to epoll_wait()
Ready: 2
  fd=4; events: EPOLLIN
    read 4 bytes: ppp

  fd=5; events: EPOLLIN EPOLLHUP
    read 4 bytes: qqq
```

```
    closing fd 5
About to epoll_wait()
```

Две пустые строки в вышеприведенном выводе являются символами новой строки, которые были прочитаны экземплярами `cat`, записаны в очереди FIFO, а затем прочитаны и выведены нашей программой.

Далее мы нажимаем Ctrl+D во втором терминале, чтобы завершить оставшийся экземпляр `cat`; это опять приводит к завершению вызова `epoll_wait()`, который теперь возвращает всего одно событие:

```
                    Нажимаем Ctrl+D, чтобы завершить команду "cat > p"
Ready: 1
  fd=4; events: EPOLLHUP
    closing fd 4
All file descriptors closed; bye
```

Листинг 59.5. Использование программного интерфейса epoll

altio/epoll_input.c

```
#include <sys/epoll.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_BUF     1000    /* Максимальный объем данных, получаемых за одно чтение */
#define MAX_EVENTS     5    /* Максимальное количество событий,
                               возвращаемых одним вызовом epoll_wait() */

int
main(int argc, char *argv[])
{
    int epfd, ready, fd, s, j, numOpenFds;
    struct epoll_event ev;
    struct epoll_event evlist[MAX_EVENTS];
    char buf[MAX_BUF];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file...\n", argv[0]);

(1) epfd = epoll_create(argc - 1);
    if (epfd == -1)
        errExit("epoll_create");

    /* Открываем каждый файл в командной строке и добавляем его
       в «список интереса» экземпляра epoll */
(2) for (j = 1; j < argc; j++) {
        fd = open(argv[j], O_RDONLY);
        if (fd == -1)
            errExit("open");
        printf("Opened \"%s\" on fd %d\n", argv[j], fd);

        ev.events = EPOLLIN;    /* Нас интересуют только события ввода */
        ev.data.fd = fd;
(3)     if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev) == -1)
            errExit("epoll_ctl");
    }

    numOpenFds = argc - 1;
```

```
❹ while (numOpenFds > 0) {
      /* Извлекаем элемент из списка готовности (максимум MAX_EVENTS) */
      printf("About to epoll_wait()\n");
❺     ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
      if (ready == -1) {
❻         if (errno == EINTR)
              continue;
              /* Перезапускаем, если операция была прервана сигналом */
          else
              errExit("epoll_wait");
      }
      printf("Ready: %d\n", ready);

      /* Обрабатываем полученный список событий */
❼     for (j = 0; j < ready; j++) {
          printf("    fd=%d; events: %s%s%s\n", evlist[j].data.fd,
                  (evlist[j].events & EPOLLIN)    ? "EPOLLIN "    : "",
                  (evlist[j].events & EPOLLHUP)   ? "EPOLLHUP "   : "",
                  (evlist[j].events & EPOLLERR)   ? "EPOLLERR "   : "");

❽         if (evlist[j].events & EPOLLIN) {
              s = read(evlist[j].data.fd, buf, MAX_BUF);
              if (s == -1)
                  errExit("read");
              printf("    read %d bytes: %.*s\n", s, s, buf);
❾         } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {

    /* Если установлены флаги EPOLLIN и EPOLLHUP, то количество байтов,
       доступных для чтения, может превышать MAX_BUF. Следовательно,
       мы закрываем файловый дескриптор, только если не был установлен
       флаг EPOLLIN. Остальные байты будут прочитаны во время следующих
       вызовов epoll_wait(). */

              printf("    closing fd %d\n", evlist[j].data.fd);
❿             if (close(evlist[j].data.fd) == -1)
                  errExit("close");
              numOpenFds--;
          }
      }
    }

    printf("All file descriptors closed; bye\n");
    exit(EXIT_SUCCESS);
}
```

altio/epoll_input.c

59.4.4. Подробности семантики интерфейса epoll

В этом подразделе мы рассмотрим некоторые нюансы взаимодействия открытых файлов, их дескрипторов и интерфейса `epoll`. Еще раз взгляните на рис. 5.2, демонстрирующий связь между файловыми дескрипторами, описаниями открытых файлов и общесистемной таблицей индексных дескрипторов (i-node).

Вместо экземпляра `epoll` (вызов `epoll_create()`) ядро создает в памяти новый индексный дескриптор и описание открытого файла, а затем выделяет в вызывающем процессе новый файловый дескриптор, который ссылается на данное описание. Экземпляр

интерфейса epoll связан с описанием открытого файла, а не с файловым дескриптором epoll. Это имеет следующие последствия.

- Если продублировать файловый дескриптор epoll с помощью вызова dup() (или аналогичного), то новая копия будет ссылаться на те же списки интереса и готовности, что и исходный дескриптор. Для изменения списка интереса в качестве аргумента epfd вызова epoll_ctl() можно передавать любую копию. Аналогично любой из дескрипторов можно использовать как аргумент epfd вызова epoll_wait() для извлечения элементов списка готовности.
- Предыдущий пункт также актуален после вызова fork(). Потомок наследует копию родительского файлового дескриптора epoll, которая ссылается на те же структуры данных.

При выполнении операции epoll_ctl() EPOLL_CTL_ADD ядро добавляет в список интереса epoll элемент, хранящий данные о номере отслеживаемого файлового дескриптора, и ссылку на соответствующее описание открытого файла. Чтобы обеспечить корректную работу вызова epoll_wait(), ядро следит за данным описанием. Это значит следующее: мы должны пересмотреть ранее озвученное утверждение о том, что при закрытии файловый дескриптор автоматически удаляется из списка интереса epoll, в который он входит. Нужно добавить уточнение: описание открытого файла удаляется из списка интереса epoll вместе с закрытием последнего файлового дескриптора, ссылающегося на него. Это значит, что если мы продублируем файловый дескриптор, ссылающийся на открытый файл, используя вызов dup() (или аналогичный) или fork(), то данный файл будет удален только после закрытия всех его исходных дескрипторов и их копий.

Такая семантика может вызывать поведение, которое на первый взгляд кажется странным. Представьте, что мы выполняем код из листинга 59.6. Вызов epoll_wait() сообщит о готовности дескриптора fd1 (иными словами, поле evlist[0].data.fd будет равно fd1), даже если он закрыт. Дело вот в чем: у нас по-прежнему остается один открытый дескриптор, fd2; он указывает на описание открытого файла, содержащееся в списке интереса epoll. С похожей ситуацией можно столкнуться, когда один из двух процессов, содержащих дублирующиеся дескрипторы для одного и того же описания открытого файла (обычно это происходит в результате вызова fork()), выполняет epoll_wait() и закрывает свой дескриптор, тогда как второй все еще содержит открытую копию.

Листинг 59.6. Семантика интерфейса epoll при дублировании файловых дескрипторов

```
int epfd, fd1, fd2;
struct epoll_event ev;
struct epoll_event evlist[MAX_EVENTS];

/* Пропущено: код для открытия 'fd1' и создания файлового дескриптора epoll 'epfd' */
ev.data.fd = fd1
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd1, ev) == -1)
    errExit("epoll_ctl");

/* Предположим, что в этот момент 'fd1' становится доступным для ввода */

fd2 = dup(fd1);
close(fd1);
ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
if (ready == -1)
    errExit("epoll_wait");
```

59.4.5. Производительность интерфейса epoll по сравнению с мультиплексированным вводом/выводом

В табл. 59.9 представлены результаты мониторинга набора из *N* файловых дескрипторов в диапазоне от 0 до *N* – 1 с помощью вызовов `poll()`, `select()` и интерфейса `epoll`. (Тестирование проводилось в Linux 2.6.25 и было организовано таким образом, что во время каждой операции наблюдения готовым оказывался ровно один файловый дескриптор.) Как видите, с ростом номеров дескрипторов производительность вызовов `poll()` и `select()` постепенно снижается. Для сравнения: производительность интерфейса `epoll` при больших N почти не меняется (небольшое замедление, вероятно, является результатом переполнения кэша процессора в тестовой системе).

> Перед тестированием мы изменили значение константы FD_SETSIZE в заголовочных файлах glibc на 16 384, чтобы позволить программе отслеживать большое количество файловых дескрипторов, задействуя вызов select().

Таблица 59.9. Время, которое уходит на 100 000 операций мониторинга с помощью poll(), select() и интерфейса epoll

Количество отслеживаемых дескрипторов (N)	Процессорное время poll() (секунды)	Процессорное время select() (секунды)	Процессорное время epoll (секунды)
10	0,61	0,73	0,41
100	2,90	3,00	0,42
1000	35,00	35,00	0,53
10 000	990,00	930,00	0,66

В подразделе 59.2.5 мы узнали, что вызовы `select()` и `poll()` демонстрируют низкую производительность при наблюдении за большим количеством файловых дескрипторов. Теперь посмотрим, почему интерфейс `epoll` не имеет такой проблемы.

- При каждом вызове `select()` или `poll()` ядро вынуждено проверять все указанные файловые дескрипторы. Для сравнения: интерфейс `epoll` позволяет пометить интересующий нас дескриптор с помощью вызова `epoll_ctl()`. Ядро записывает эту информацию в список, связанный с исходным описанием открытого файла, и затем при каждой операции ввода/вывода, которая делает файловый дескриптор готовым, добавляет элемент в соответствующий список готовности `epoll`. (Событие ввода/вывода, связанное с одним описанием открытого файла, может сделать готовыми сразу несколько файловых дескрипторов.) Последующий вызов `epoll_wait()` просто извлекает элементы из списка готовности.
- При каждом вызове `select()` или `poll()` передает ядру структуру данных со сведениями обо всех файловых дескрипторах, за которыми нужно наблюдать; затем ядро возвращает эту структуру обратно, предварительно записав в нее информацию обо всех готовых дескрипторах. Для сравнения: в интерфейсе `epoll` предусмотрен вызов `epoll_ctl()`, создающий структуру данных с перечнем отслеживаемых файловых дескрипторов прямо в *пространстве ядра*. После его выполнения последующим вызовам `epoll_wait()` не нужно передавать ядру информацию о дескрипторах, а в качестве результата мы будем получать список дескрипторов, являющихся готовыми.

Помимо фактов, приведенных выше, стоит отметить еще несколько моментов. Каждый вызов select() требует предварительной инициализации входящей структуры данных, а каждую структуру, возвращаемую вызовами select() и poll() и состоящую из N элементов, следует проверять на наличие готовых файловых дескрипторов. Однако, если верить результатам тестирования, время, которое уходит на эти операции, является несущественным по сравнению с тем, сколько времени тратится на выполнение системных вызовов для мониторинга N дескрипторов. Операции проверки не учитываются в табл. 59.9.

Грубо говоря, производительность вызовов `select()` и `poll()` падает линейно по мере увеличивается значения N (количества отслеживаемых файловых дескрипторов). Эта закономерность начинает проявляться при $N = 100$ и $N = 1000$ (см. табл. 59.9). К моменту достижения $N = 10\,000$ падение производительности становится даже хуже, чем линейное.

Интерфейс `epoll` линейно масштабируется в зависимости от количества происходящих событий ввода/вывода, что делает его особенно эффективным в сценариях, характерных для серверов, которые одновременно обслуживают множество клиентов: большинство отслеживаемых файловых дескрипторов не проявляют никакой активности, и лишь немногие из них являются готовыми.

59.4.6. Уведомления, срабатывающие по фронту

Уведомления, предоставляемые интерфейсом `epoll`, изначально срабатывают по уровню. Это значит, что `epoll` сообщает нам, будет ли заблокирована операция ввода/вывода для заданного файлового дескриптора. Извещения такого вида предоставляются также вызовами `poll()` и `select()`.

Программный интерфейс `epoll` также позволяет получать уведомления, срабатывающие по фронту, — это когда мы можем узнать, производился ли ввод/вывод для заданного дескриптора между двумя последовательными вызовами `epoll_wait()` (или между первым вызовом и моментом открытия дескриптора). Использование данного вида уведомлений семантически похоже на ввод/вывод на основе сигналов; разница лишь в том, что последний генерирует по одному сигналу для каждого события, а вызов `epoll_wait()` возвращает все события сразу.

Для включения уведомлений, срабатывающих по фронту, при вызове `epoll_ctl()` в поле `ev.events` нужно указать флаг `EPOLLET`:

```
struct epoll_event ev;

ev.data.fd = fd
ev.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

Проиллюстрируем разницу между уведомлениями, срабатывающими по уровню и по фронту, на следующем примере. Представьте, что мы задействуем интерфейс `epoll` для мониторинга ввода сокета (`EPOLLIN`), и при этом происходят следующие события.

1. В сокет поступает ввод.
2. Мы выполняем вызов `epoll_wait()`, который возвращает нам сведения о готовности сокета, вне зависимости от типа применяемых уведомлений.
3. Мы выполняем второй вызов `epoll_wait()`.

При использовании уведомлений, срабатывающих по уровню, второй вызов `epoll_wait()` проинформирует нас о готовности сокета. Если же прибегнуть к уведомлениям, срабатывающим по фронту, второе обращение к `epoll_wait()` заблокируется, поскольку с момента предыдущего вызова не появилось никакого нового ввода.

Как отмечалось в подразделе 59.1.1, уведомления, срабатывающие по фронту, обычно применяются в сочетании с неблокирующими файловыми дескрипторами. Таким образом, общий алгоритм применения этих уведомлений в интерфейсе `epoll` выглядит так.

1. Делаем все дескрипторы, которые нужно отслеживать, неблокирующими.
2. Формируем список интереса `epoll` с помощью вызова `epoll_ctl()`.
3. Обрабатываем события ввода/вывода в следующем цикле:
   - извлекаем список готовых дескрипторов, используя `epoll_wait()`;
   - выполняем ввод/вывод для каждого готового дескриптора, пока соответствующий системный вызов (например, `read()`, `write()`, `recv()`, `send()` или `accept()`) не вернет ошибку `EAGAIN` или `EWOULDBLOCK`.

Предотвращение нехватки данных в файловых дескрипторах с помощью уведомлений, срабатывающих по фронту

Представьте, что мы используем уведомления, срабатывающие по фронту, для мониторинга большого количества файловых дескрипторов, и в один из них поступает большое количество входящих данных (возможно, бесконечный поток). Если после определения готовности этого дескриптора попытаться прочитать весь его ввод в неблокирующем режиме, то есть вероятность оставить без внимания остальные дескрипторы (то есть может пройти много времени, прежде чем опять будет проверена их готовность). Одним из решений данной проблемы является хранение списка файловых дескрипторов, о чьей готовности мы проинформированы, и выполнение постоянного цикла, который делает следующее.

1. Отслеживает файловые дескрипторы с помощью вызова `epoll_wait()` и добавляет в список приложения те из них, что являются готовыми. При наличии в списке каких-либо элементов данный шаг должен иметь очень маленькое или нулевое время ожидания; в случае если список не был пополнен, это позволит приложению быстро перейти к следующему шагу и начать обслуживать дескрипторы, которые точно являются готовыми.
2. Выполняет ограниченный объем ввода/вывода для готовых файловых дескрипторов (возможно, перебирая их по кругу, а не возвращаясь к началу списка после каждого вызова `epoll_wait()`). Если соответствующий неблокирующий системный вызов завершается ошибкой `EAGAIN` или `EWOULDBLOCK`, файловый дескриптор можно убрать из списка.

Представленный подход требует дополнительной работы со стороны программиста, однако предоставляемые им преимущества не ограничиваются защитой от нехватки данных в файловых дескрипторах. Например, в вышеупомянутом цикле можно выполнять и другие шаги, такие как управление таймерами или прием сигналов с помощью вызова `sigwaitinfo()` (или аналогичного).

Нехватку данных следует учитывать и при использовании ввода/вывода на основе сигналов, поскольку он также поддерживает механизм уведомлений, срабатывающих по фронту. В то же время уведомления, срабатывающие по уровню, могут не иметь подобных проблем, поскольку позволяют блокировать файловые дескрипторы и проверять их готовность внутри цикла; это дает нам возможность выполнить *некий* объем ввода/вывода, прежде чем вернуться к поиску готовых дескрипторов.

59.5. Ожидание сигналов и готовности файловых дескрипторов

Иногда процессу приходится ждать возникновения сразу двух событий: возможности ввода/вывода в одном или нескольких файловых дескрипторах и доставки сигнала. Эту операцию можно попытаться выполнить с помощью вызова `select()`, как показано в листинге 59.7.

Листинг 59.7. Некорректный способ разблокировки сигналов и вызова select()

```
sig_atomic_t gotSig = 0;
void
handler(int sig)
{
    gotSig = 1;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    ...
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL) == -1)
        errExit("sigaction");

    /* Что если сигнал доставлен сейчас? */
    ready = select(nfds, &readfds, NULL, NULL, NULL);
    if (ready > 0) {
        printf("%d file descriptors ready\n", ready);
    } else if (ready == -1 && errno == EINTR) {
        if (gotSig)
            printf("Got signal\n");
    } else {
        /* Какая-то другая ошибка */
    }
    ...
}
```

У приведенного кода есть проблема: если сигнал (в данном случае `SIGUSR1`) поступает после установки обработчика, но перед вызовом `select()`, то последний все равно заблокируется (это разновидность состояния гонки). Ниже рассматриваются некоторые способы выхода из описанной ситуации.

> С версии 2.6.27 Linux предоставляет дополнительную методику, позволяющую одновременно ожидать поступления сигнала и готовности файлового дескриптора: механизм signalfd, описанный в разделе 22.11. С его помощью можно принимать сигналы через отслеживаемый дескриптор (наряду с другими дескрипторами), используя вызовы select(), poll() или epoll_wait().

59.5.1. Системный вызов pselect()

Системный вызов `pselect()` по своему действию похож на `select()`. Его главным семантическим отличием является дополнительный аргумент, `sigmask`, определяющий сигналы, маски которых будут заменены во время блокировки вызова.

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timespec *timeout, const sigset_t *sigmask);
```

Возвращает количество готовых файловых дескрипторов, `0`, если истекло время ожидания, или `-1` при ошибке

Рассмотрим `pselect()` более подробно. Представьте, что мы делаем следующий вызов:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask);
```

Он тождественен *автоматическому* выполнению следующих шагов:

```
sigset_t origmask;
sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);        /* Восстанавливаем маску сигнала */
```

Можно переписать первую часть тела главной программы из листинга 59.7 с помощью вызова `pselect()`.

Помимо аргумента `sigmask` вызовы `select()` и `pselect()` имеют следующие отличия:

- аргумент `timeout` в вызове `pselect()` представляет собой структуру `timespec` (см. подраздел 23.4.2), которая позволяет указывать время ожидания с точностью до наносекунд (вместо микросекунд);
- в стандарте SUSv3 подчеркивается, что вызов `pselect()` не изменяет аргумент `timeout` при возвращении.

Если передать в качестве аргумента `sigmask` значение `NULL`, то вызов `pselect()` будет вести себя так же, как `select()` (то есть не станет изменять сигнальную маску процесса), за исключением вышеупомянутых моментов.

Интерфейс `pselect()` разрабатывался как часть спецификации POSIX.1g и на сегодняшний день входит в стандарт SUSv3. Он доступен не во всех реализациях UNIX, а в Linux он появился только в версии 2.6.16.

> Раньше функцию pselect() предоставляла библиотека glibc, но она не обеспечивала атомарности, необходимой для ее корректного выполнения. Этого можно добиться только путем реализации pselect() на уровне ядра.

Листинг 59.8. Использование вызова pselect()

```
sigset_t emptyset, blockset;
struct sigaction sa;

sigemptyset(&blockset);
sigaddset(&blockset, SIGUSR1);
if (sigprocmask(SIG_BLOCK, &blockset, NULL) == -1)
    errExit("sigprocmask");

sa.sa_sigaction = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGUSR1, &sa, NULL) == -1)
    errExit("sigaction");

sigemptyset(&emptyset);
ready = pselect(nfds, &readfds, NULL, NULL, NULL, &emptyset);
if (ready == -1)
    errExit("pselect");
```

Системные вызовы ppoll() и epoll_pwait()

В Linux 2.6.16 появился еще один нестандартный вызов, `ppoll()`, который имеет такое же отношение к `poll()`, как `pselect()` к `select()`. В Linux 2.6.19 аналогичное расширение появилось и для вызова `epoll_wait()` — оно имеет имя `epoll_pwait()`. Подробности см. на следующих страницах руководства: `ppoll(2)` и `epoll_pwait(2)`.

59.5.2. Трюк с зацикленным каналом

Поскольку вызов `pselect()` не является широко распространенным, портируемые приложения должны применять какие-то другие стратегии для предотвращения состояния гонки в ситуациях, когда процесс одновременно ждет появления сигнала и вызывает `select()` для набора файловых дескрипторов. Одно из общепринятых решений выглядит так.

1. Мы создаем именованный канал и делаем его считывающий и записывающий концы неблокирующими.
2. Добавляем считывающий конец канала в список `readfds`, передающийся в вызов `select()` (вместе с остальными файловыми дескрипторами, которые нас интересуют).
3. Устанавливаем обработчик для интересующего нас сигнала. При срабатывании он записывает в канал один байт. Следует выделить несколько замечаний относительно данного обработчика.
   - В самом начале мы сделали записывающий конец канала неблокирующим во избежание ситуации, в которой сигнал будет доставляться настолько быстро, что повторные вызовы его обработчика заполнят весь канал, а операция `write()` внутри обработчика (как и весь процесс) заблокируется. (Это неважно в случае, если запись в заполненный канал завершится неудачно, поскольку предыдущие операции уже известили процесс о доставке сигнала.)
   - Обработчик сигнала устанавливается после создания канала, чтобы избежать состояния гонки, связанного с преждевременной доставкой сигнала.
   - Использование операции `write()` внутри обработчика является безопасным, так как это одна из функций, рассчитанных на работу с асинхронными сигналами (см. табл. 21.1).
4. Помещаем вызов `select()` в цикл, чтобы он перезапускался в случае прерывания со стороны обработчика сигнала (подобного рода перезапуск не является обязательным; он просто позволяет узнавать о появлении сигнала путем анализа аргумента `readfds`, не прибегая к проверке ошибки `EINTR` во время возвращения).
5. При успешном завершении вызова `select()` можно определить, дошел ли сигнал. Для этого нужно проверить, входит ли файловый дескриптор считывающего конца канала в набор `readfds`.
6. Приняв сигнал, считываем все данные, находящиеся в канале. Поскольку сигналов может быть несколько, создаем цикл, считывающий данные, пока (неблокирующая) операция `read()` не завершится ошибкой `EAGAIN`. Прочитав все содержимое канала, выполняем действия, которые нужно предпринять в ответ на доставку сигнала.

Эту методику обычно называют *трюком с зацикленным каналом*. Код с ее реализацией представлен в листинге 59.9.

Другие разновидности данной методики можно также использовать в сочетании с вызовами `poll()` и `epoll_wait()`.

Листинг 59.9. Трюк с зацикленным каналом

Из файла **altio/self_pipe.c**

```
#include <sys/time.h>
#include <sys/select.h>
#include <fcntl.h>
#include <signal.h>
#include "tlpi_hdr.h"

static int pfd[2];                /* Файловые дескрипторы для канала */
```

```
static void
handler(int sig)
{
    int savedErrno;                /* В случае, если мы изменим 'errno' */

    savedErrno = errno;
    if (write(pfd[1], "x", 1) == -1 && errno != EAGAIN)
        errExit("write");
    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    fd_set readfds;
    int ready, nfds, flags;
    struct timeval timeout;
    struct timeval *pto;
    struct sigaction sa;
    char ch;

    /* Инициализируем 'timeout', 'readfds' и 'nfds' для вызова select() */
    int fd, j;
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s {timeout|-} fd...\n"
                 "\t\t('-' means infinite timeout)\n", argv[0]);
    if (strcmp(argv[1], "-") == 0) {
        pto = NULL;                     /* Бесконечный таймаут */
    } else {
        pto = &timeout;
        timeout.tv_sec = getLong(argv[1], 0, "timeout");
        timeout.tv_usec = 0;            /* Нет микросекунд */
    }
    nfds = 0;
    FD_ZERO(&readfds);
    for (j = 2; j < argc; j++) {
        fd = getInt(argv[j], 0, "fd");
        if (fd >= FD_SETSIZE)
            cmdLineErr("file descriptor exceeds limit (%d)\n", FD_SETSIZE);
        if (fd >= nfds)
            nfds = fd + 1;
        FD_SET(fd, &readfds);
    }

    if (pipe(pfd) == -1)
        errExit("pipe");
    FD_SET(pfd[0], &readfds);      /* Добавляем в 'readfds' считывающий конец канала */
    nfds = max(nfds, pfd[0] + 1);  /* При необходимости корректируем 'nfds' */
    flags = fcntl(pfd[0], F_GETFL);
    if (flags == -1)
        errExit("fcntl-F_GETFL");
    flags |= O_NONBLOCK;           /* Делаем считывающий конец неблокирующим */
    if (fcntl(pfd[0], F_SETFL, flags) == -1)
        errExit("fcntl-F_SETFL");

    flags = fcntl(pfd[1], F_GETFL);
    if (flags == -1)
        errExit("fcntl-F_GETFL");
```

```
    flags |= O_NONBLOCK;      /* Делаем записывающий конец неблокирующим */
    if (fcntl(pfd[1], F_SETFL, flags) == -1)
        errExit("fcntl-F_SETFL");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;  /* Перезапускаем прерванное чтение */
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    while ((ready = select(nfds, &readfds, NULL, NULL, pto)) == -1 &&
            errno == EINTR)
        continue;             /* Перезапускаем, если прервано по сигналу */
    if (ready == -1)          /* Непредвиденная ошибка */
        errExit("select");

    if (FD_ISSET(pfd[0], &readfds)) {        /* Был вызван обработчик */
        printf("A signal was caught\n");

        for (;;) {                           /* Читаем данные из канала */
            if (read(pfd[0], &ch, 1) == -1) {
                if (errno == EAGAIN)
                    break;                   /* Данных больше не осталось */
                else
                    errExit("read");         /* Какая-то другая ошибка */
            }

            /* Реагируем на сигнал должным образом */
        }
    }

    /* Ищем в массиве, полученном из select(), готовые файловые дескрипторы */
    printf("ready = %d\n", ready);
    for (j = 2; j < argc; j++) {
        fd = getInt(argv[j], 0, "fd");
        printf("%d: %s\n", fd, FD_ISSET(fd, &readfds) ? "r" : "");
    }
    printf("%d: %s     (read end of pipe)\n", pfd[0],
            FD_ISSET(pfd[0], &readfds) ? "r" : "");
    if (pto != NULL)
        printf("timeout after select(): %ld.%03ld\n",
                (long) timeout.tv_sec, (long) timeout.tv_usec / 1000);
    exit(EXIT_SUCCESS);
}
```

Из файла **altio/self_pipe.c**

59.6. Резюме

В этой главе мы познакомились с различными альтернативами стандартной модели ввода/вывода: с мультиплексированным вводом/выводом (`select()` и `poll()`), вводом/выводом на основе сигналов и программным интерфейсом `epoll`, доступным только в Linux. Все перечисленные механизмы позволяют отслеживать множество файловых дескрипторов, проверяя, не готов ли какой-либо из них. При этом ни один из представленных подходов не занимается непосредственно чтением или записью: установив, что файловый дескриптор готов, можно выполнять ввод/вывод с помощью традиционных системных вызовов.

Мультиплексирующие вызовы `select()` и `poll()` отслеживают сразу несколько файловых дескрипторов, позволяя определять, доступен ли в них ввод или вывод. При любом из этих вызовов мы каждый раз передаем ядру весь список дескрипторов, которые нужно проверить, а ядро возвращает нам те из них, что готовы. Ввиду данного факта вызовы `select()` и `poll()` демонстрируют плохую производительность при мониторинге большого количества файловых дескрипторов.

Ввод/вывод на основе сигналов позволяет получать уведомления о возможности чтения или записи для файлового дескриптора. Чтобы включить этот механизм, нужно установить обработчик сигнала `SIGIO`, указать владельца, который будет получать данный сигнал, и позволить генерировать сигналы путем остановки флага состояния открытого файла `O_ASYNC`. При наблюдении за большим количеством файловых дескрипторов описанная методика обеспечивает значительный прирост производительности по сравнению с мультиплексированным вводом/выводом. Linux позволяет использовать для уведомлений нестандартные сигналы. Если они будут работать в режиме реального времени, то появится возможность ставить их в очередь, благодаря чему обработчик сумеет определять на основе аргумента `siginfo_t` файловый дескриптор и тип события, сгенерировавшего сигнал.

Как и ввод/вывод на основе сигналов, интерфейс `epoll` обеспечивает улучшенную производительность при мониторинге большого количества файловых дескрипторов. В обоих случаях данное преимущество следует из того факта, что ядро «запоминает» список дескрипторов, отслеживаемых процессом (в отличие от вызовов `select()` и `poll()`, передающих аналогичные списки при каждом запуске). Программный интерфейс `epoll` имеет ряд заметных преимуществ по сравнению с вводом/выводом на основе сигналов, позволяя избежать сложностей работы с сигналами и выбрать типы событий ввода/вывода, которые нас интересуют (например, запись или чтение).

На протяжении этой главы мы регулярно упоминали об отличиях между уведомлениями о готовности, срабатывающих по уровню и по фронту. Первые сообщают о том, возможен ли на данный момент ввод/вывод для файлового дескриптора. Вторые сигнализируют о вводе/выводе, произошедшем в дескрипторе с момента последней проверки. Системные вызовы для мультиплексирования ввода/вывода предоставляют модель уведомлений, срабатывающих по уровню. Ввод/вывод на основе сигналов сводится к уведомлениям, срабатывающим по фронту; интерфейс `epoll` способен работать с обеими моделями (по умолчанию срабатывание происходит по уровню). Уведомления, срабатывающие по фронту, обычно сочетают с неблокирующим вводом/выводом.

В конце данной главы мы рассмотрели проблему, с которой иногда сталкиваются программы, наблюдающие за множеством файловых дескрипторов: как одновременно с этим ждать доставки сигнала. Обычно для такого случая используется так называемый трюк с зацикленным каналом — когда обработчик сигнала записывает один байт в канал, считывающий конец которого является одним из отслеживаемых файловых дескрипторов. Стандарт SUSv3 предусматривает вызов `pselect()`, являющийся разновидностью операции `select()` и позволяющий решить данную проблему иначе. Однако он доступен не во всех реализациях UNIX. Linux поддерживает аналогичные нестандартные вызовы — `ppoll()` и `epoll_pwait()`.

Дополнительная информация

В книге [Stevens et al., 2004] описываются мультиплексированные операции чтения и записи, а также ввод/вывод на основе сигналов; особый упор сделан на использование этих механизмов в сочетании с сокетами. Документ [Gammo et al, 2004] посвящен сравнению производительности вызовов `select()`, `poll()` и интерфейса `epoll`.

На http://www.kegel.com/c10k.html находится крайне любопытная статья под названием The C10K problem (проблема 10 000 соединений), написанная Дэном Кегелом. В ней рассматриваются трудности, с которыми сталкиваются разработчики веб-серверов, спроектированных для обслуживания десятков тысяч клиентов. Там вы сможете найти множество ссылок на информацию по данной теме.

59.7. Упражнения

59.1. Отредактируйте программу из листинга 59.2 (`poll_pipes.c`), заменив операцию `poll()` вызовом `select()`.

59.2. Напишите сервер `echo` (см. разделы 56.2 и 56.3), который обслуживает TCP- и UDP-клиентов. Для этого сервер должен создать два слушающих сокета, по одному для каждого протокола, и затем отслеживать их с помощью одной из методик, описанных в настоящей главе.

59.3. В последнем пункте описания методики с зацикленным каналом в подразделе 59.5.2 утверждалось: прежде чем как-то реагировать на поступивший сигнал, программа сначала должна очистить канал. Что может случиться, если поменять эти два шага местами?

59.4. Отредактируйте программу из листинга 59.9 (`self_pipe.c`), заменив операцию `select()` вызовом `poll()`.

59.5. Напишите программу, которая использует вызов `epoll_create()` для создания экземпляра `epoll`, после чего сразу же переходит к ожиданию возвращенного файлового дескриптора с помощью вызова `epoll_wait()`. Что произойдет, если вызову `epoll_wait()` передать файловый дескриптор `epoll` с пустым списком интереса (как в данном случае)? Как этим можно воспользоваться?

59.6. Представьте, что у нас есть файловый дескриптор `epoll`, отслеживающий множество файловых дескрипторов, каждый из которых всегда является готовым. Если выполнить несколько вызовов `epoll_wait()` со значением `maxevents`, значительно меньшим, чем количество готовых дескрипторов (например, когда `maxevents` равен 1), и не считывать/записывать все доступные данные между этими вызовами, то какие дескрипторы мы получим в результате в каждом из случаев? Напишите программу, чтобы ответить на данный вопрос (при проведении эксперимента операции ввода/вывода между вызовами `epoll_wait()` можно не выполнять). В каких ситуациях такое поведение может пригодиться?

59.7. Отредактируйте программу из листинга 59.3 (`demo_sigio.c`), используя сигналы реального времени вместо `SIGIO`. Измените обработчик сигнала так, чтобы он принимал в качестве аргумента структуру `siginfo_t` и выводил ее поля `si_fd` и `si_code`.

60 Псевдотерминалы

Псевдотерминал — виртуальное устройство, предоставляющее канал для межпроцессного взаимодействия. На одном конце этого канала находится программа, ожидающая соединения с терминальным устройством. К другому ее концу подключено приложение, которое снабжает первую программу вводом и считывает ее вывод.

Эта глава описывает использование псевдотерминалов, демонстрируя их применение в таких приложениях, как терминальные эмуляторы, программа `script(1)` и сетевые службы входа в систему наподобие ssh.

60.1. Краткий обзор

Одна из проблем, в решении которых помогают псевдотерминалы, проиллюстрирована на рис. 60.1: как позволить пользователю, работающему за одним компьютером, выполнять консольные программы (такие как `vi`) на другом компьютере, подключенном к той же сети?

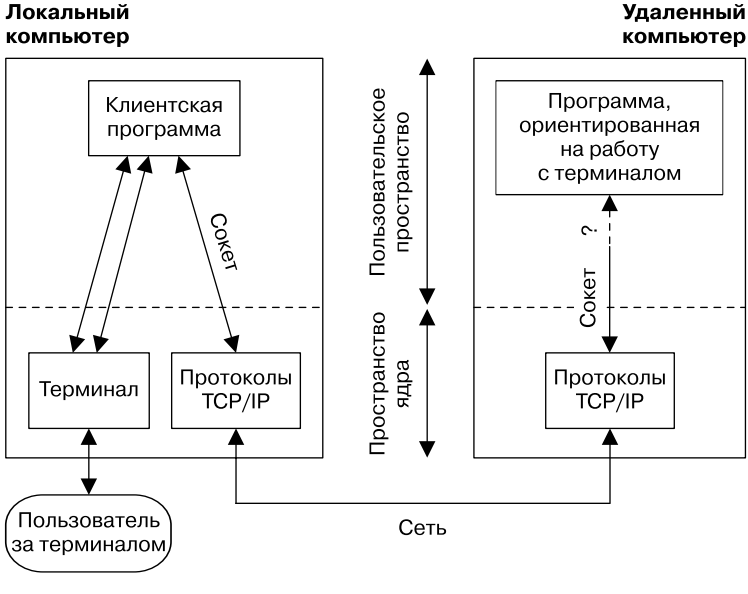


Рис. 60.1. Как управлять по сети программой, ориентированной на работу с терминалом?

Как показано на диаграмме, механизм решения данной проблемы частично основан на сокетах, обеспечивающих сетевое взаимодействие. Однако мы не можем подключить стандартный ввод, вывод или поток stderr программы, ориентированной на работу с терминалом, непосредственно к сокету. Дело в том, что подобные программы выполняют терминальные операции, описанные в главах 34 и 58, поэтому они рассчитаны на подключение

к терминалу. Среди таких операций можно выделить переключение в неканонический режим, включение и выключение эхо-контроля, а также установку активной группы процессов для терминала. При попытке выполнить их в контексте сокета соответствующие системные вызовы завершатся неудачно.

Кроме того, программы, рассчитанные на работу с терминалом, ожидают, что драйвер терминала будет определенным образом обрабатывать их ввод и вывод. Например, если в каноническом режиме драйвер обнаруживает конец файла (обычно это Ctrl+D) в начале строки, то делает так, что следующий вызов `read()` не возвращает данных.

Наконец, программы подобного рода должны иметь управляющий терминал. Это позволяет им получать его файловые дескрипторы, открывая устройство `/dev/tty`, а также делает возможным генерирование сигналов для управления заданиями и самим терминалом (например, `SIGTSTP`, `SIGTTIN` и `SIGINT`).

По приведенному описанию можно понять, что определение программы, ориентированной на работу с терминалом, является довольно расплывчатым. Оно охватывает широкий диапазон приложений, которые обычно запускаются в интерактивном режиме.

Первичные и вторичные устройства псевдотерминалов

Псевдотерминал представляет собой недостающее звено, необходимое для создания сетевого соединения с программой, ориентированной на работу с терминалом. Это пара соединенных между собой виртуальных устройств: *первичный* и *вторичный* псевдотерминалы, которые иногда называют *парой псевдотерминалов*. Такая пара обеспечивает межпроцессное взаимодействие, напоминающее двунаправленный именованный канал: два процесса могут открыть первичный и вторичный концы и затем передавать данные в любом направлении.

Ключевая особенность псевдотерминалов заключается в том, что вторичное устройство выглядит как стандартный терминал. К нему можно применять любые операции, поддерживаемые терминальным устройством. Некоторые из данных операций не имеют смысла в контексте псевдотерминала (например, установка скорости последовательного порта или четности), но это нормально, поскольку вторичное устройство их просто игнорирует.

Как программы используют псевдотерминалы

На рис. 60.2 показан стандартный способ применения псевдотерминала двумя программами (псевдотерминалы часто обозначаются с помощью сокращения *pty*; мы будем применять его в различных диаграммах и именах функций, приводимых в этой главе). Стандартные ввод, вывод и поток `stderr` программы, ориентированной на работу с терминалом, подключаются ко вторичному псевдотерминалу, который в данном случае является еще и управляющим. На другом конце соединения находится драйвер терминала; он выступает в качестве промежуточного звена для пользователя, поставляя программе ввод и считывая ее вывод.

Обычно драйвер терминала одновременно считывает и записывает данные в другой канал ввода/вывода. Он действует как реле, передавая информацию в обоих направлениях между псевдотерминалом и другой программой. Для этого ему нужно следить за вводом одновременно с двух направлений. Здесь обычно используются мультиплексирование ввода/вывода (вызова `select()` или `poll()`) или два отдельных процесса/потока, которые передают данные в разных направлениях.

Приложение, работающее с псевдотерминалом, обычно выполняет следующие шаги.

1. Драйвер открывает первичное устройство псевдотерминала.
2. Драйвер делает вызов `fork()` для создания дочернего процесса, совершающего действия, описанные ниже.
   - Делает вызов `setsid()`, чтобы начать сессию, в которой он является лидером (см. раздел 34.3). Этот шаг также приводит к потере дочерним процессом управляющего терминала.

- Открывает вторичное устройство псевдотерминала, подключенное к первичному. Поскольку дочерний процесс является лидером сессии и не имеет управляющего терминала, данную роль для него начинает выполнять вторичное устройство псевдотерминала.
- Использует вызов `dup()` (или аналогичный), чтобы продублировать дескриптор вторичного устройства в качестве стандартного ввода, вывода и `stderr`.
- Вызывает `exec()` в целях запустить программу, ориентированную на работу с терминалом, которая будет подключена ко вторичному устройству псевдотерминала.

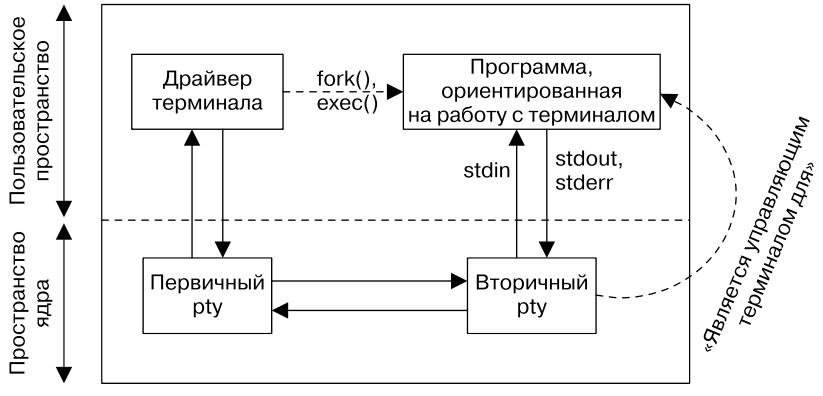


Рис. 60.2. Две программы, общающиеся через псевдотерминал

На данном этапе обе стороны могут взаимодействовать друг с другом с помощью псевдотерминала. Вся информация, записанная драйвером в первичное устройство, появляется в виде ввода во вторичном устройстве, к которому подключена программа, ориентированная на работу с терминалом. А все, что записывает данная программа во вторичное устройство, становится доступным для чтения драйвером из первичного устройства. Дальнейшие подробности о вводе/выводе в псевдотерминале мы рассмотрим в разделе 60.5.

Псевдотерминалы можно также применять для соединения произвольных процессов (которые не обязательно являются родительским и дочерним). Для этого лишь нужно, чтобы процесс, открывающий первичное устройство, передал другому процессу имя соответствующего вторичного устройства. Здесь можно воспользоваться файлом или каким-нибудь механизмом межпроцессного взаимодействия (если вызвать fork() так, как было описано выше, то потомок автоматически унаследует от родителя информацию, достаточную для определения имени вторичного устройства).

До настоящего момента наше обсуждение псевдотерминалов носило абстрактный характер. На рис. 60.3 показан конкретный пример: использование псевдотерминала приложением ssh, которое дает возможность пользователю создавать безопасную сессию на удаленном компьютере, подключенном к сети (эта диаграмма по сути объединяет содержимое рис. 60.1 и 60.2). Роль драйвера в удаленной системе играет SSH-сервер, подключенный к первичному устройству псевдотерминала (sshd), а в качестве программы, ориентированной на работу с терминалом и подключенной ко вторичному устройству, применяется командная оболочка. SSH-сервер задействует сокет, чтобы связать между собой псевдотерминал и SSH-клиент. Выполнив все действия, необходимые для входа в систему, сервер и клиент начинают передавать символы между пользовательским терминалом на локальном компьютере и удаленной командной оболочкой.

Мы опускаем множество подробностей, касающихся SSH-клиента и сервера. Например, эти программы шифруют данные, передающиеся по сети в любом направлении. В нашем примере SSH-сервер на удаленном компьютере состоит лишь из одного процесса, но на самом деле работает в параллельном режиме. Он становится демоном и создает пассивный TCP-сокет, который ждет входящих соединений со стороны SSH-клиентов. Для каждого соединения главный SSH-сервер создает отдельный дочерний процесс, полностью отвечающий за обслуживание конкретной клиентской сессии (именно этот дочерний процесс изображен на рис. 60.3). Помимо настройки псевдотерминала, описанной выше, дочерний процесс SSH-сервера занимается аутентификацией пользователя, обновлением учетных файлов на удаленном компьютере (как описано в главе 40) и впоследствии запуском командной оболочки.

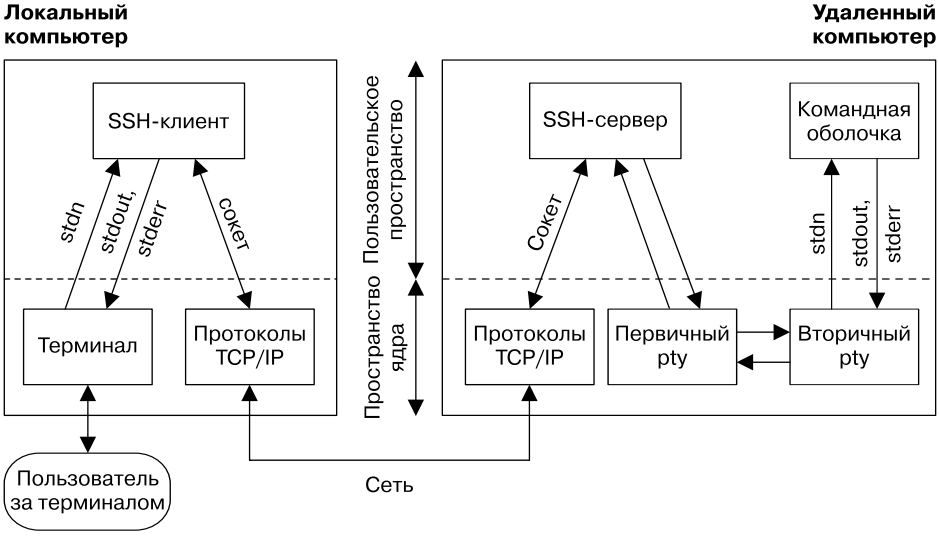


Рис. 60.3. Как ssh использует псевдотерминал

В некоторых ситуациях ко вторичному концу псевдотерминала можно подключать несколько процессов. Такой сценарий проиллюстрирован в нашем примере с ssh. Лидером сессии для вторичного устройства является командная оболочка, создающая группы процессов для выполнения команд, вводимых удаленным пользователем. Для всех этих процессов вторичный конец играет роль управляющего терминала. Как и при работе в обычном терминале, одна из указанных групп может быть активной по отношению ко вторичному концу псевдотерминала, и только ей будет позволено считывать и записывать (если был установлен бит `TOSTOP`) данные в это устройство.

Приложения, использующие псевдотерминалы

Псевдотерминалы применяются не только в сетевых службах, но и во многих других приложениях, среди которых можно выделить следующие.

- Приложение `expect(1)` задействует псевдотерминал, чтобы позволить управлять программой, ориентированной на работу с терминалом, с помощью сценариев.
- Эмуляторы терминалов, такие как `xterm`, используют псевдотерминалы для предоставления терминальных функций в оконном режиме.
- Приложение `screen(1)` применяет псевдотерминалы для мультиплексирования одного физического терминала (или терминального окна) между несколькими процессами (например, между сессиями командной строки).

- Псевдотерминалы используются в программе `script(1)`, которая записывает весь ввод и вывод, возникающий во время сессии командной строки.
- Иногда псевдотерминал может помочь обойти стандартную блочную буферизацию, выполняемую стандартными функциями ввода/вывода при записи данных на диск или в именованный канал вместо линейной буферизации, применяемой для терминального вывода (мы вернемся к этому вопросу в упражнении 60.7).

60.2. Псевдотерминалы стандарта UNIX 98

В данном разделе мы шаг за шагом напишем функцию `ptyFork()`, выполняющую большую часть работы, проиллюстрированной на рис. 60.2. Затем реализуем на ее основе программу `script(1)`. Но сначала рассмотрим различные библиотечные функции, применяемые в сочетании с псевдотерминалами стандарта UNIX 98.

- `posix_openpt()` открывает свободное первичное устройство псевдотерминала и возвращает файловый дескриптор, который будет использоваться для работы с этим устройством в последующих вызовах.
- `grantpt()` изменяет владельца и права доступа ко вторичному устройству, соединенному с первичным концом псевдотерминала.
- `unlockpt()` разблокирует вторичное устройство, соединенное с первичным концом псевдотерминала, чтобы его можно было открыть.
- `ptsname()` возвращает имя вторичного устройства, соединенного с первичным концом псевдотерминала. После этого вторичное устройство может быть открыто с помощью вызова `open()`.

60.2.1. Открытие неиспользуемого первичного устройства: вызов posix_openpt()

Функция `posix_openpt()` находит и открывает неиспользуемое первичное устройство псевдотерминала, после чего возвращает файловый дескриптор для дальнейшего обращения к этому устройству.

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int flags);
```

Возвращает файловый дескриптор при успешном завершении или `-1` при ошибке

Аргумент `flags` может быть пустым или состоять из следующих флагов, к которым применено побитовое ИЛИ.

- `O_RDWR` — открывает устройство одновременно для чтения и записи. Обычно эта константа всегда включается в аргумент `flags`.
- `O_NOCTTY` — не делает данный терминал управляющим для текущего процесса. В Linux первичный конец псевдотерминала не может стать управляющим независимо от того, указан ли при вызове `posix_openpt()` флаг `O_NOCTTY`. (Здесь есть определенный смысл, поскольку первичный конец не является терминалом как таковым; это просто другая сторона терминала, к которой подключено вторичное устройство.) Однако в ряде систем

флаг O_NOCTTY является обязательным, если мы не хотим, чтобы в результате открытия первичного устройства наш процесс обзавелся управляющим терминалом.

По аналогии с вызовом open() функция posix_openpt() использует для открытия первичного конца псевдотерминала файловый дескриптор с наименьшим доступным номером.

Вызов posix_openpt() также приводит к созданию в каталоге /dev/pts файла, связанного с первичным устройством псевдотерминала. Мы подробно остановимся на этом моменте, когда будем рассматривать функцию ptsname().

Функция posix_openpt() появилась в стандарте SUSv3 относительно недавно по инициативе комитета стандартизации POSIX. В исходной реализации псевдотерминала в System V получение доступного первичного устройства достигалось за счет открытия его *копии*, /dev/ptmx. Это, в свою очередь, автоматически приводило к нахождению и открытию следующего неиспользуемого первичного конца, в результате чего возвращался его файловый дескриптор. Данное устройство доступно и в Linux, а реализация функции posix_openpt() выглядит следующим образом:

```
int
posix_openpt(int flags)
{
    return open("/dev/ptmx", flags);
}
```

Ограничение количества псевдотерминалов стандарта UNIX 98

Каждый псевдотерминал использует определенное, пусть и небольшое, количество памяти, которую нельзя сбросить на диск, поэтому ядро вводит ограничение на количество псевдотерминалов стандарта UNIX 98 в системе. В версиях Linux вплоть до 2.6.3 данное ограничение определялось параметром конфигурации ядра (CONFIG_UNIX98_PTYS), по умолчанию равным 256. Но мы могли указать любое значение от 0 до 2048.

Начиная с версии 2.6.4 параметр конфигурации ядра CONFIG_UNIX98_PTYS был заменен более гибким подходом. Теперь максимальное количество псевдотерминалов ограничивается значением, указанным в файле /proc/sys/kernel/pty/max (который доступен лишь в Linux). По умолчанию это значение равно 4096 и не должно превышать 1 048 576. Существует также вспомогательный файл, доступный только для чтения, показывающий, сколько всего псевдотерминалов стандарта UNIX 98 используется на данный момент.

60.2.2. Изменение владельца и прав доступа ко вторичному устройству: вызов grantpt()

В стандарте SUSv3 предусмотрена функция grantpt(), позволяющая изменить владельца и права доступа ко вторичному устройству, связанному с первичным концом псевдотерминала, на который ссылается файловый дескриптор mfd. В Linux вызов grantpt() не является обязательным. Однако он требуется для корректной работы в ряде других систем, поэтому портируемые приложения должны выполнять его после вызова posix_openpt().

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int grantpt(int mfd);
```

Возвращает 0 при успешном завершении или -1 при ошибке

Если вызов `grantpt()` является обязательным, то создает дочерний процесс, который выполняет программу, устанавливающую идентификатор администратора. Эта программа обычно называется `pt_chown` и совершает следующие операции со вторичным устройством псевдотерминала:

- изменяет владельца вторичного устройства в соответствии с действующим пользовательским идентификатором вызывающего процесса;
- переносит вторичное устройство в группу `tty`;
- изменяет права доступа ко вторичному устройству таким образом, чтобы владелец мог выполнять запись и чтение, а группа имела только право на запись.

Причина изменения группы терминала на `tty` и предоставления ей права на запись заключается в том, что программы `wall(1)` и `write(1)` устанавливают идентификаторы группы и сами входят в `tty`.

В Linux вторичное устройство псевдотерминала автоматически конфигурируется вышеописанным образом, поэтому вызова `grantpt()` не требуется (но его все равно нужно сделать).

> Вызов grantpt() может создать дочерний процесс, вследствие чего в стандарте SUSv3 сказано, что его поведение является неопределенным, если в вызывающей программе установлен обработчик для сигнала SIGCHLD.

60.2.3. Разблокировка вторичного устройства: вызов unlockpt()

Функция `unlockpt()` убирает внутреннюю блокировку вторичного устройства, связанного с первичным концом псевдотерминала, на который ссылается файловый дескриптор `mfd`. Такой механизм блокировки позволяет вызывающему процессу выполнять любые действия, необходимые для инициализации вторичного устройства (например, вызов `grantpt()`) до того, как его сможет открыть какой-то другой процесс.

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int unlockpt(int mfd);
```

Возвращает `0` при успешном завершении или `-1` при ошибке

Попытка открыть вторичное устройство псевдотерминала до его разблокировки с помощью вызова `unlockpt()` завершается ошибкой `EIO`.

60.2.4. Получение имени вторичного устройства: вызов ptsname()

Функция `ptsname()` возвращает имя вторичного устройства псевдотерминала, связанного с первичным концом псевдотерминала, на который ссылается файловый дескриптор `mfd`.

В Linux (как и в большинстве других реализаций) вызов `ptsname()` возвращает имя в виде `/dev/pts/`*nn*, где *nn* — уникальный номер вторичного устройства псевдотерминала.

Буфер, с помощью которого возвращается имя, обычно выделяется статическим способом. Это значит, что при следующем вызове `ptsname()` он будет перезаписан.

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

char *ptsname(int mfd);
```

Возвращает указатель на (возможно, статически выделенную) строку при успешном завершении или NULL при ошибке

Библиотека GNU языка C предоставляет реентерабельную версию вызова ptsname() в виде ptsname_r(mfd, strbuf, buflen). Однако данная функция является нестандартной и доступна лишь в нескольких UNIX-системах. Чтобы получить объявление ptsname_r() из заголовочного файла <stdlib.h>, необходимо определить макрос проверки возможностей _GNU_SOURCE.

Разблокировав вторичное устройство с помощью функции unlockpt(), можно его открыть, воспользовавшись традиционным системным вызовом open().

60.3. Открытие первичного устройства: вызов ptyMasterOpen()

Пришло время познакомиться с функцией ptyMasterOpen(), которая использует вызовы, описанные в предыдущих разделах, для открытия первичного устройства псевдотерминала и получения имени связанного с ним вторичного конца. Мы описываем эту функцию по следующим причинам:

- большинство программ выполняют описанные действия точно таким же образом, так что для удобства их можно объединить в единую функцию;
- наша версия ptyMasterOpen() инкапсулирует все подробности и особенности псевдотерминалов стандарта UNIX 98.

```
#include "pty_master_open.h"

int ptyMasterOpen(char *slaveName, size_t snLen);
```

Возвращает файловый дескриптор при успешном завершении или -1 при ошибке

Функция ptyMasterOpen() открывает неиспользуемый первичный конец псевдотерминала, вызывает для него grantpt() и unlockpt() и копирует имя связанного с ним вторичного устройства в буфер, на который указывает аргумент slaveName. Вызывающий процесс должен указать в аргументе snLen количество памяти, доступной этому буферу. Реализация данной функции показана в листинге 60.1.

Листинг 60.1. Реализация функции ptyMasterOpen()

pty/pty_master_open.c

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>
#include "pty_master_open.h"            /* Объявляет ptyMasterOpen() */
#include "tlpi_hdr.h"
```

```
int
ptyMasterOpen(char *slaveName, size_t snLen)
{
    int masterFd, savedErrno;
    char *p;

    masterFd = posix_openpt(O_RDWR | O_NOCTTY);
    /* Открываем первичный pty */
    if (masterFd == -1)
        return -1;

    if (grantpt(masterFd) == -1) {   /* Разрешаем доступ ко вторичному pty */
        savedErrno = errno;
        close(masterFd);             /* Может изменить 'errno' */
        errno = savedErrno;
        return -1;
    }

    if (unlockpt(masterFd) == -1) {  /* Разблокируем вторичный pty */
        savedErrno = errno;
        close(masterFd);             /* Может изменить 'errno' */
        errno = savedErrno;
        return -1;
    }

    p = ptsname(masterFd);           /* Получаем имя вторичного pty */
    if (p == NULL) {
        savedErrno = errno;
        close(masterFd);             /* Может изменить 'errno' */
        errno = savedErrno;
        return -1;
    }

    if (strlen(p) < snLen) {
        strncpy(slaveName, p, snLen);
    } else {                  /* Возвращаем ошибку, если буфер слишком маленький */
        close(masterFd);
        errno = EOVERFLOW;
        return -1;
    }

    return masterFd;
}
```

pty/pty_master_open.c

Точно так же мы могли бы отказаться от применения аргументов slaveName и snLen, давая вызывающему процессу возможность самостоятельно получить имя вторичного устройства псевдотерминала с помощью вызова ptsname(). Мы этого не сделали, поскольку в системе BSD псевдотерминалы не имеют аналога функции ptsname().

60.4. Соединение процессов с помощью псевдотерминала: вызов ptyFork()

Теперь мы готовы реализовать функцию, выполняющую всю работу по установке соединения между двумя процессами, используя два конца псевдотерминала (см. рис. 60.2).

Функция ptyFork() создает дочерний процесс, который подключается к родителю через псевдотерминал.

```
#include "pty_fork.h"

pid_t ptyFork(int *masterFd, char *slaveName, size_t snLen,
              const struct termios *slaveTermios,
              const struct winsize *slaveWS);
```

В родителе: возвращает идентификатор потомка при успешном завершении или -1 при ошибке; в успешно созданном потомке: всегда возвращает 0

Реализация функции ptyFork() показана в листинге 60.2. Рассмотрим работу функции.

1. Открывает первичное устройство псевдотерминала с помощью функции ptyMasterOpen() (листинг 60.1) (1).
2. Если аргумент slaveName равен NULL, то копирует имя вторичного устройства в буфер (2). В противном случае slaveName должен указывать на буфер размером как минимум snLen байт. При необходимости вызывающий процесс может использовать данное имя для обновления учетных файлов (см. главу 40). Это может пригодиться в приложениях, которые обеспечивают вход в систему, например ssh, rlogin и telnet. С другой стороны, такие программы, как script(1) (см. раздел 60.6), не обновляют учетные файлы, поскольку не предоставляют возможность входа в систему.
3. Делает вызов fork() с целью создать дочерний процесс (3).
4. После вызова fork() родителю остается только убедиться в том, что в целом числе, на которое указывает аргумент masterFd, находится файловый дескриптор первичного устройства псевдотерминала (4).
5. После вызова fork() потомок выполняет следующие действия.
   1) Вызывает setsid(), чтобы создать новую сессию (см. раздел 34.3) (5). Становится ее лидером и теряет свой управляющий терминал (если он у него был).
   2) Закрывает файловый дескриптор для первичного устройства, поскольку он ему не нужен (6).
   3) Открывает вторичное устройство псевдотерминала (7), которое становится его управляющим терминалом (так как свой исходный управляющий терминал потомок утратил на предыдущем шаге).
   4) Выполняет операцию для файлового дескриптора вторичного устройства TIOCSCTTY ioctl() (8) (при условии, что определен макрос TIOCSCTTY). Код позволяет функции ptyFork() работать на платформах BSD, где управляющий терминал можно получить только в результате выполнения операции TIOCSCTTY вручную (см. раздел 34.4).
   5) Если аргумент slaveTermios не равен NULL, то делает вызов tcsetattr(), чтобы установить атрибуты вторичного устройства в соответствии со значениями внутри структуры termios, на которую указывает данный аргумент (9). Это может оказаться удобным в ряде интерактивных программ (таких как script(1)), использующих псевдотерминал и требующих, чтобы атрибуты вторичного устройства совпадали с атрибутами терминала, в котором происходит выполнение.
   6) Если аргумент slaveWS не равен NULL, то выполняет операцию ioctl() TIOCSWINSZ с целью установить размер окна вторичного устройства псевдотерминала в соответствии со значениями в структуре winsize, на которую указывает данный аргумент (10). Этот шаг выполняется по той же причине, что и предыдущий.
   7) Использует вызов dup2() для дублирования вторичного файлового дескриптора в качестве стандартного ввода, вывода и потока stderr (11). На данном этапе потомок

уже способен выполнить произвольную программу, которая сможет общаться с псевдотерминалом с помощью стандартных файловых дескрипторов. Программа сможет выполнять любые стандартные операции, доступные в обычном терминале.

По аналогии с вызовом `fork()` функция `ptyFork()` возвращает либо идентификатор потомка (в родительском процессе), либо `0` (в дочернем процессе), либо `-1` в случае ошибки.

В какой-то момент дочерний процесс, созданный с помощью `ptyFork()`, будет завершен. Если после этого родитель продолжает работу, то должен отследить своего потомка, чтобы устранить процесс-зомби. Однако данная процедура часто является необязательной, так как приложения, использующие псевдотерминалы, обычно спроектированы с расчетом на одновременное завершение родителя и потомка.

Системы семейства BSD предоставляют две похожие нестандартные функции для работы с псевдотерминалами. Первая, openpty(), открывает первичное и вторичное устройства, возвращая их файловые дескрипторы. Она также может вернуть имя вторичного устройства и установить размер окна и атрибуты терминала, взятые из аргументов, аналогичных slaveTermios и slaveWS. Другая функция, forkpty(), является аналогом нашей реализации ptyFork(); разница лишь в том, что она не поддерживает аргумент, похожий на snLen. В Linux обе эти функции предоставляются библиотекой glibc и задокументированы на странице openpty(3) руководства.

Листинг 60.2. Реализация функции ptyFork()

pty/pty_fork.c

```
#include <fcntl.h>
#include <termios.h>
#include <sys/ioctl.h>
#include "pty_master_open.h"
#include "pty_fork.h"                /* Объявляет ptyFork() */
#include "tlpi_hdr.h"

#define MAX_SNAME 1000

pid_t
ptyFork(int *masterFd, char *slaveName, size_t snLen,
        const struct termios *slaveTermios, const struct winsize *slaveWS)
{
    int mfd, slaveFd, savedErrno;
    pid_t childPid;
    char slname[MAX_SNAME];

1   mfd = ptyMasterOpen(slname, MAX_SNAME);
    if (mfd == -1)
        return -1;

2   if (slaveName != NULL) {    /* Возвращает имя вторичного устройства */
        if (strlen(slname) < snLen) {
            strncpy(slaveName, slname, snLen);
        } else {                /* Имя 'slaveName' слишком короткое */
            close(mfd);
            errno = EOVERFLOW;
            return -1;
        }
    }

3   childPid = fork();
    if (childPid == -1) {       /* Вызов fork() завершился неудачно */
```

```
        savedErrno = errno;     /* Операция close() может изменить 'errno' */
        close(mfd);             /* Освобождаем ресурс файлового дескриптора */
        errno = savedErrno;
        return -1;
    }

❹ if (childPid != 0) {          /* Родитель */
        *masterFd = mfd;        /* Только родитель получает первичный fd */
        return childPid;        /* Как родитель вызова fork()*/
    }

    /* Потомок переходит в эту точку */
❺ if (setsid() == -1)           /* Начинаем новую сессию */
        err_exit("ptyFork:setsid");

❻ close(mfd);                  /* Является лишним в потомке */

❼ slaveFd = open(slname, O_RDWR);     /* Становится управляющим tty */
    if (slaveFd == -1)
        err_exit("ptyFork:open-slave");
❽ #ifdef TIOCSCTTY             /* Получаем управляющий tty в BSD */
    if (ioctl(slaveFd, TIOCSCTTY, 0) == -1)
        err_exit("ptyFork:ioctl-TIOCSCTTY");
#endif

❾ if (slaveTermios != NULL)  /* Устанавливаем атрибуты вторичного tty */
        if (tcsetattr(slaveFd, TCSANOW, slaveTermios) == -1)
            err_exit("ptyFork:tcsetattr");

❿ if (slaveWS != NULL)         /* Устанавливаем размер окна вторичного tty */
        if (ioctl(slaveFd, TIOCSWINSZ, slaveWS) == -1)
            err_exit("ptyFork:ioctl-TIOCSWINSZ");

    /* Дублируем вторичный pty в качестве stdin, stdout и stderr потомка */
⓫ if (dup2(slaveFd, STDIN_FILENO) != STDIN_FILENO)
        err_exit("ptyFork:dup2-STDIN_FILENO");
    if (dup2(slaveFd, STDOUT_FILENO) != STDOUT_FILENO)
        err_exit("ptyFork:dup2-STDOUT_FILENO");
    if (dup2(slaveFd, STDERR_FILENO) != STDERR_FILENO)
        err_exit("ptyFork:dup2-STDERR_FILENO");
    if (slaveFd > STDERR_FILENO)        /* На всякий случай */
        close(slaveFd);                 /* Дескриптор больше не нужен */

    return 0;                           /* Как потомок вызова fork() */
}
```

pty/pty_fork.c

60.5. Ввод/вывод псевдотерминала

Псевдотерминал похож на двунаправленный именованный канал. Все, что записывается в первичное устройство, появляется в виде ввода для вторичного, а все записанное во вторичное — в виде ввода для первичного.

Главное отличие псевдотерминала от двунаправленного именованного канала заключается в том, что вторичный конец ведет себя подобно терминальному устройству. Он интерпретирует входящие данные таким же образом, как управляющий терминал интерпретирует ввод с клавиатуры. Например, если мы нажмем Ctrl+C (передавая тем

самым стандартный символ *прерывания*), то вторичное устройство сгенерирует сигнал `SIGINT` для своей активной группы процессов. По аналогии с обычным терминалом, когда вторичное устройство работает в каноническом режиме (что происходит по умолчанию), его ввод буферизируется построчно. Иными словами, программа, читающая из вторичного конца псевдотерминала, получит ввод (строку) только в случае записи в первичное устройство символа новой строки.

Как и именованные каналы, псевдотерминалы имеют ограниченную вместимость. Если ее исчерпать, то дальнейшая запись будет блокироваться до тех пор, пока процесс на другом конце псевдотерминала не прочитает часть данных.

В Linux вместимость псевдотерминала в каждом направлении равна около 4 Кбайт.

При закрытии всех файловых дескрипторов, ссылающихся на первичное устройство псевдотерминала, произойдет следующее:

- при наличии у вторичного устройства управляющего процесса последнему будет послан сигнал `SIGHUP` (см. раздел 34.6);
- чтение из вторичного устройства вернет конец файла (0);
- запись во вторичное устройство завершится ошибкой `EIO` (в некоторых других реализациях UNIX операция `write()` в этом случае завершается ошибкой `ENXIO`).

Если закрыть все файловые дескрипторы, ссылающиеся на вторичное устройство псевдотерминала, то:

- чтение из первичного устройства завершится ошибкой `EIO` (в некоторых других реализациях UNIX операция `read()` в этом случае возвращает конец файла);
- запись в первичное устройство завершается успешно при условии, что входящая очередь вторичного конца не была заполнена (в таком случае операция `write()` блокируется). Если после этого открыть вторичное устройство, то данные будут доступны для чтения.

Исход последнего случая может заметно варьироваться в зависимости от реализации. В некоторых UNIX-системах запись завершается ошибкой `EIO`, а в других отрабатывает успешно, но притом исходящие данные отклоняются (то есть не могут быть прочитаны при повторном открытии вторичного устройства). В целом описанные отклонения не составляют проблемы. Обычно процесс на первичном конце обнаруживает закрытие вторичного устройства, поскольку операция чтения либо возвращает конец файла, либо завершается неудачей. В связи с этим процесс перестает записывать данные в первичное устройство.

Пакетный режим

Пакетный режим — механизм, который позволяет процессу, работающему поверх первичного устройства псевдотерминала, быть проинформированным о следующих событиях, происходящих на вторичном конце и связанных с программным управлением потоком:

- сброшена входящая или исходящая очередь;
- остановлен или начат вывод терминала (Ctrl+S/Ctrl+Q);
- включено или выключено управление потоком.

Пакетный режим помогает выполнять программное управление потоком в определенных приложениях, которые работают с псевдотерминалом и позволяют входить в систему (например, `telnet` и `rlogin`).

Для включения пакетного режима нужно применить операцию `ioctl() TIOCPKT` к файловому дескриптору, ссылающемуся на первичный конец псевдотерминала:

```
int arg;

arg = 1;                /* 1 == включить; 0 == выключить; */
if (ioctl(mfd, TIOCPKT, &arg) == -1)
    errExit("ioctl");
```

При включенном пакетном режиме операция чтения из первичного устройства возвращает либо один управляющий байт (представляет собой битовую маску, описывающую изменения в состоянии вторичного устройства), либо нулевой байт, за которым следуют данные, записанные во вторичный конец.

Когда псевдотерминал в пакетном режиме меняет свое состояние, вызов `select()` уведомляет нас о том, что в первичном устройстве произошло исключительное условие (аргумент `exceptfds`), а операция `poll()` возвращает значение `POLLPRI` в поле `revents` (описание вызовов `select()` и `poll()` см. в главе 59).

Пакетный режим не предусмотрен стандартом SUSv3, и некоторые детали его реализации варьируются в зависимости от системы. Больше подробностей о нем, в том числе и о значениях битовой маски, описывающей изменения состояния, можно найти на странице `tty_ioctl(4)` руководства).

60.6. Реализация программы script(1)

Теперь мы готовы к написанию простой версии стандартной программы `script(1)`. Эта программа создает новую сессию командной строки и записывает весь ее ввод и вывод в файл. Большинство сессий командной строки, представленных в настоящей книге, было записано с ее помощью.

В обычной сессии командная оболочка соединяется непосредственно с пользовательским терминалом. После запуска программа `script` становится промежуточным звеном между оболочкой и терминалом пользователя, используя псевдотерминал для общения с оболочкой (см. рис. 60.4). Командная оболочка подключается ко вторичному устройству псевдотерминала, а программа `script` — к первичному. Процесс `script` выполняет для пользователя роль прокси, принимая данные, введенные в терминале, и записывая их в первичный конец псевдотерминала, а также направляя вывод из данного конца обратно в терминал.

Кроме того, `script` создает исходящий файл (который по умолчанию называется `typescript`), содержащий копию всех данных, выведенных на первичное устройство псевдотерминала. Это касается не только вывода, сгенерированного сессией командной оболочки, но и ее ввода. Ввод записывается, потому что, как и в случае с обычным терминалом, ядро экранирует вводимые символы, копируя их в исходящую очередь (см. рис. 58.1). Но если выключить эхо-контроль, как делается в программах, считывающих пароли, ввод вторичного устройства псевдотерминала не будет копироваться в исходящую очередь и, следовательно, не попадет в исходящий файл программы `script`.

Наша реализация программы `script` представлена в листинге 60.3. Она выполняет следующие действия.

1. Получает размер окна и атрибуты терминала, в котором запущена программа ❶. Эти данные будут переданы последующему вызову `ptyFork()`, устанавливающему с их помощью соответствующие значения для вторичного устройства псевдотерминала.
2. Вызывает нашу функцию `ptyFork()` (см. листинг 60.2), чтобы создать дочерний процесс, соединенный с родителем путем псевдотерминала ❷.
3. После вызова `ptyFork()` потомок запускает командную оболочку ❹. Тип оболочки определяется с помощью переменной среды `SHELL` ❸. Если эта переменная не установлена или содержит пустую строку, то потомок запускает `/bin/sh`.

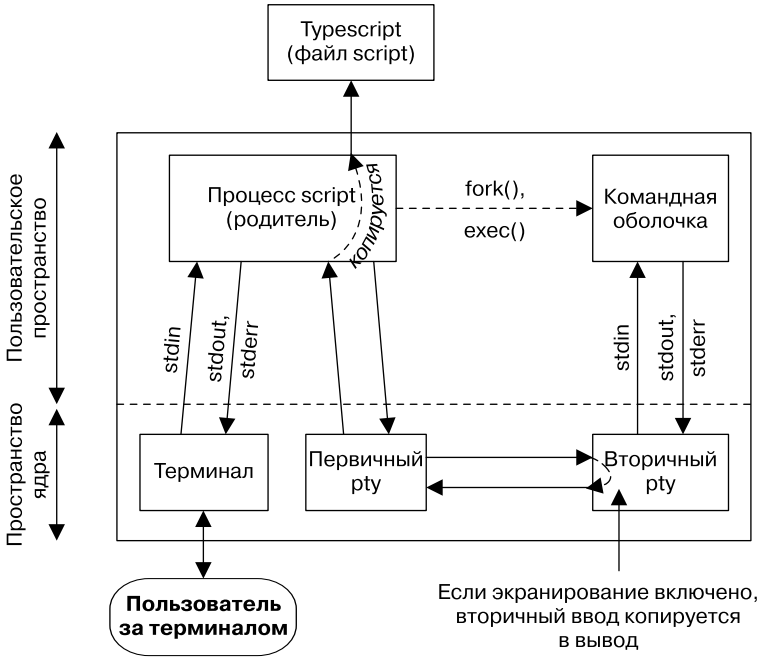


Рис. 60.4. Программа script

4. После вызова `ptyFork()` родитель выполняет следующие шаги.
   1) Открывает файл вывода ❺. Если указан аргумент командной строки, то он служит в качестве имени файла. В противном случае берется стандартное имя (`typescript`).
   2) Переключает терминал в режим без обработки (используя функцию `ttySetRaw()`, представленную в листинге 58.3), чтобы вводимые символы направлялись непосредственно в программу `script` и не изменялись драйвером терминала ❻. Символы, выводимые программой `script`, тоже не подлежат изменению.

Факт нахождения терминала в режиме без обработки вовсе не означает, что командная оболочка (или любая другая группа процессов, которая является активной по отношению ко вторичному устройству псевдотерминала) будет получать управляющие символы в их исходном виде или что эти символы дойдут неизмененными до пользовательского терминала. Интерпретация специальных символов происходит во вторичном устройстве (если только оно тоже не было переключено в режим без обработки). Используя режим без обработки в терминале пользователя, мы отключаем второй уровень интерпретации входящих и исходящих символов.

   3) Вызывает функцию `atexit()` с целью установить обработчик, который при завершении программы возвращает терминал в его исходный режим ❼.
   4) Выполняет цикл, передающий данные от терминала к первичному устройству псевдотерминала и наоборот ❽. В начале каждой итерации делается вызов `select()` (см. подраздел 59.2.1), отслеживающий ввод на обоих концах соединения ❾. Если в терминале доступен ввод, то программа считывает какую-то его часть и записывает ее в первичное устройство псевдотерминала ❿. Аналогично при обнаружении ввода в первичном устройстве программа считывает его часть и записывает ее в терминал

и в файл программы `script` (11). Цикл выполняется, пока в одном из наблюдаемых дескрипторов не будет обнаружен конец файла или не произойдет ошибка.

Листинг 60.3. Простая реализация программы script(1)

pty/script.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <libgen.h>
#include <termios.h>
#include <sys/select.h>
#include "pty_fork.h"                /* Объявление ptyFork() */
#include "tty_functions.h"           /* Объявление ttySetRaw() */
#include "tlpi_hdr.h"

#define BUF_SIZE 256
#define MAX_SNAME 1000

struct termios ttyOrig;
static void             /* Сбрасываем режим терминала при выходе */
ttyReset(void)
{
    if (tcsetattr(STDIN_FILENO, TCSANOW, &ttyOrig) == -1)
        errExit("tcsetattr");
}

int
main(int argc, char *argv[])
{
    char slaveName[MAX_SNAME];
    char *shell;
    int masterFd, scriptFd;
    struct winsize ws;
    fd_set inFds;
    char buf[BUF_SIZE];
    ssize_t numRead;
    pid_t childPid;

(1) if (tcgetattr(STDIN_FILENO, &ttyOrig) == -1)
        errExit("tcgetattr");
    if (ioctl(STDIN_FILENO, TIOCGWINSZ, &ws) < 0)
        errExit("ioctl-TIOCGWINSZ");

(2) childPid = ptyFork(&masterFd, slaveName, MAX_SNAME, &ttyOrig, &ws);
    if (childPid == -1)
        errExit("ptyFork");

    if (childPid == 0) {  /* Потомок запускает командную оболочку во вторичном pty */
(3)     shell = getenv("SHELL");
        if (shell == NULL || *shell == '\0')
            shell = "/bin/sh";

(4)     execlp(shell, shell, (char *) NULL);
        errExit("execlp");
        /* Если мы добрались до этой строчки, значит что-то пошло не так */
    }

    /* Родитель передает данные между терминалом и первичным pty */
```

```
❺ scriptFd = open((argc > 1) ? argv[1] : "typescript",
                  O_WRONLY | O_CREAT | O_TRUNC,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (scriptFd == -1)
        errExit("open typescript");

❻ ttySetRaw(STDIN_FILENO, &ttyOrig);

❼ if (atexit(ttyReset) != 0)
        errExit("atexit");

❽ for (;;) {
        FD_ZERO(&inFds);
        FD_SET(STDIN_FILENO, &inFds);
        FD_SET(masterFd, &inFds);

❾       if (select(masterFd + 1, &inFds, NULL, NULL, NULL) == -1)
            errExit("select");

❿       if (FD_ISSET(STDIN_FILENO, &inFds)) {    /* stdin --> pty */
            numRead = read(STDIN_FILENO, buf, BUF_SIZE);
            if (numRead <= 0)
                exit(EXIT_SUCCESS);
            if (write(masterFd, buf, numRead) != numRead)
                fatal("partial/failed write (masterFd)");
        }

⓫       if (FD_ISSET(masterFd, &inFds)) {    /* pty --> stdout+файл */
            numRead = read(masterFd, buf, BUF_SIZE);
            if (numRead <= 0)
                exit(EXIT_SUCCESS);
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write (STDOUT_FILENO)");
            if (write(scriptFd, buf, numRead) != numRead)
                fatal("partial/failed write (scriptFd)");
        }
    }
}
```

pty/script.c

Применение программы из листинга 60.3 демонстрируется в следующей сессии командной строки. Сначала мы выводим имя псевдотерминала, которое используется в терминале `xterm`, выполняющем нашу сессию, а также идентификатор процесса командной оболочки. Эта информация пригодится в дальнейшем.

```
$ tty
/dev/pts/1
$ echo $$
7979
```

Теперь запустим экземпляр нашей программы `script`, вызывающей еще одну командную оболочку. Опять же, мы выводим имя терминала, в котором выполняется сессия, и идентификатор процесса командной оболочки:

```
$ ./script
$ tty
/dev/pts/24               Вторичное устройство, открытое программой script
$ echo $$
29825                     PID оболочки, запущенной программой script
```

Теперь воспользуемся утилитой ps(1), чтобы вывести сведения о двух командных оболочках и процессе, выполняющем программу script, после чего завершим оболочку, запущенную этой программой:

```
$ ps -p 7979 -p 29825 -C script -o "pid ppid sid tty cmd"
  PID  PPID   SID TT       CMD
 7979  7972  7979 pts/1    /bin/bash
29824  7979  7979 pts/1    ./script
29825 29824 29825 pts/24   /bin/bash
$ exit
```

Вывод утилиты ps(1) показывает, что исходная командная строка, процесс, выполняющий программу script, и вторая командная оболочка, запущенная этой программой, имеют родственные связи.

На данном этапе мы вернулись в исходную командную строку. Если вывести содержимое файла typescript, то можно увидеть запись всех входящих и исходящих данных, которые были сгенерированы во время работы программы script:

```
$ cat typescript
$ tty
/dev/pts/24
$ echo $$
29825
$ ps -p 7979 -p 29825 -C script -o "pid ppid sid tty cmd"
  PID  PPID   SID TT       CMD
 7979  7972  7979 pts/1    /bin/bash
29824  7979  7979 pts/1    ./script
29825 29824 29825 pts/24   /bin/bash
$ exit
```

60.7. Атрибуты терминала и размер окна

Первичное и вторичное устройства используют одни и те же структуры с атрибутами терминала (termios) и размером окна (winsize), которые были описаны в главе 58. Это значит, что программа, работающая в первичном устройстве псевдотерминала, может изменять указанные атрибуты для своего вторичного конца, применяя к первичному файловому дескриптору операции tcsetattr() и ioctl().

Одним из примеров того, как можно применять подобные изменения, является программа script. Представьте, что мы запускаем данную программу в окне эмулятора терминала и изменяем размер данного окна. В таком случае эмулятор оповестит ядро об изменении размера соответствующего терминального устройства, однако это не затронет отдельную запись, выделенную ядром для вторичного конца псевдотерминала (см. рис. 60.4). Следовательно, программы, которые манипулируют экраном (такие как vi) и выполняются во вторичном устройстве псевдотерминала, начнут генерировать некорректный вывод, поскольку их представление о размере окна терминала будет отличаться от его реального размера. Эту проблему можно решить следующим образом.

1. Устанавливаем в родительском процессе программы script обработчик сигнала SIGWINCH с целью получать уведомления об изменениях размера окна терминала.
2. Приняв сигнал SIGWINCH, родительский процесс script использует операцию ioctl() TIOCGWINSZ, чтобы извлечь структуру winsize для окна терминала, связанного с его

стандартным вводом. Затем он передает эту структуру операции `ioctl() TIOCSWINSZ`, устанавливающей размер окна для первичного устройства псевдотерминала.

3. Если новый размер окна псевдотерминала отличается от старого, то ядро генерирует для активной группы процессов вторичного устройства сигнал `SIGWINCH`. Такие программы, как `vi`, манипулирующие экраном, перехватывают этот сигнал и выполняют операцию `ioctl() TIOCGWINSZ`, обновляя сведения о размере окна терминала.

Подробности о размере окна терминала и флагах `TIOCGWINSZ` и `TIOCSWINSZ` для вызова `ioctl()` описаны в разделе 58.9.

60.8. Резюме

Псевдотерминал состоит из двух соединенных между собой устройств: первичного и вторичного. Вместе они обеспечивают двунаправленный канал межпроцессного взаимодействия. Преимущество псевдотерминала состоит в том, что к его вторичному концу можно подключить программу, ориентированную на работу с терминалом, которая управляется другой программой, подключенной к первичному устройству. Вторичный конец ведет себя подобно обычному терминалу. В нем можно выполнять любые операции, доступные в традиционном терминальном устройстве, а данные, направляемые в него из первичного устройства, интерпретируются точно так же, как это происходит в обычном терминале с клавиатурным вводом.

Псевдотерминалы часто применяются в приложениях, которые дают возможность сетевого входа в систему. Но они также используются в других программах, таких как эмуляторы терминала и утилита `script(1)`.

Программные интерфейсы псевдотерминалов, существующие в системах семейства BSD и System V, отличаются друг от друга. И тот, и другой поддерживаются в Linux, однако версия System V является основой для программного интерфейса, входящего в стандарт SUSv3.

60.9. Упражнения

60.1. В каком порядке завершаются родительский процесс программы `script` и ее потомок, который выполняет командную оболочку, когда пользователь вводит символ конца файла (обычно это Ctrl+D) во время работы программы из листинга 60.3? Почему?

60.2. Внесите следующие изменения в программу из листинга 60.3 (`script.c`).

- Стандартная программа `script(1)` добавляет строчки в начало и конец итогового файла, указывая время начала и завершения команды. Добавьте данную возможность.
- Добавьте код, который реагирует на изменение размера окна терминала, как описано в разделе 60.7. Возможно, для проверки этой функции вам пригодится программа `demo_SIGWINCH.c` из листинга 58.5.

60.3. Измените программу из листинга 60.3 (`script.c`), заменив вызов `select()` двумя процессами: один — для передачи данных от терминала к псевдотерминалу, а второй — наоборот.

60.4. Измените программу из листинга 60.3 (`script.c`), добавив возможность указывать временные метки. При записи каждой строчки в файл `typescript` эта программа

должна записывать временную метку во второй файл (например, `typescript.timed`). Содержимое второго файла должно иметь следующий вид:

```
<временная метка> <пробел> <строка> <переход на новую строку>
```

Временная метка должна записываться в виде строки, состоящей из количества миллисекунд, прошедших с момента начала сессии программы. Запись временных меток в текстовом формате упрощает чтение итогового файла. Если внутри текста встречаются символы новой строки, то их следует экранировать. Например, его можно записать как последовательность из двух символов: `\n` и двойной косой черты `\\`. Напишите вторую программу, `script_replay.c`, которая считывает итоговый файл с временными метками и направляет его содержимое в стандартный вывод с той же скоростью, с какой оно изначально записывалось. Вместе эти программы должны обеспечивать простой механизм для записи и воспроизведения сессии командной строки.

60.5. Реализуйте клиент-серверную программу, обеспечивающую удаленный вход в систему в стиле утилиты `telnet`. Сервер должен обслуживать клиентов в параллельном режиме (см. раздел 56.1). Конфигурация, которую нужно установить для входа каждого клиента, представлена на рис. 60.3. На этой диаграмме не показан родительский серверный процесс, обрабатывающий входящие клиентские соединения и создающий для их обработки дочерние процессы. Стоит отметить, что всю работу по аутентификации пользователя и запуску командной оболочки можно выполнять путем создания функции `ptyFork()` внутри каждого дочернего процесса еще одного потомка (внука), который запускает программу `login(1)`.

60.6. Добавьте в программу, разработанную в предыдущем шаге, код, обновляющий файлы с учетными данными в начале и конце сессии командной строки (см. главу 40).

60.7. Представьте, что мы выполняем программу, которая работает длительное время и генерирует вывод, направленный в файл или именованный канал. Например:

```
$ longrunner | grep str
```

У вышеописанного сценария есть одна проблема: по умолчанию пакет `stdio` сбрасывает стандартный вывод только при заполнении соответствующего буфера. Это значит, что вывод программы `longrunner` будет приходить «рывками», разделенными длинными отрезками времени. Данную проблему можно обойти, написав программу, которая делает следующее.

- Создает псевдотерминал.
- Выполняет программу, указанную в виде аргумента командной строки, используя стандартные файловые дескрипторы, подключенные ко вторичному устройству псевдотерминала.
- Считывает вывод из первичного устройства псевдотерминала и немедленно направляет его в стандартный вывод (`STDOUT_FILENO`, файловый дескриптор 1). В то же время она считывает ввод из терминала и записывает его в первичное устройство, чтобы он мог быть прочитан выполняемой программой.

Такую программу (назовем ее `unbuffer`) можно было бы применять следующим образом:

```
$ ./unbuffer longrunner | grep str
```

Напишите программу `unbuffer` (по большей части ее код похож на листинг 60.3).

60.8. Напишите программу, которая реализует сценарный язык, подходящий для управления редактором `vi` в пассивном режиме. Поскольку этот редактор рассчитан на работу в терминале, ваша программа должна использовать псевдотерминал.

Список используемых источников

1. *Aho, A. V., Kernighan, B. W., Weinberger, P. J.* The AWK Programming Language. Addison-Wesley, Reading, Massachusetts, 1988.
2. *Albitz, P., Liu, C.*DNS and BIND (5th edition). O'Reilly, Sebastopol, California, 2006.
3. *Anley, C., Heasman, J., Lindner, F., Richarte, G.* The Shellcoder's Handbook: Discovering and Exploiting Security Holes. Wiley, Indianapolis, Indiana, 2007.
4. *Bach, M.* The Design of the UNIX Operating System. Prentice Hall, Englewood Cliffs, New Jersey, 1986.
5. *Bhattiprolu, S., Biederman, E. W., Hallyn, S., Lezcano, D.*Virtual Servers and Checkpoint/Restart in Mainstream Linux // ACM SIGOPS Operating Systems Review. Vol. 42, Issue 5, July 2008, pages 104–113. http://www.mnis.fr/fr/services/virtualisation/pdf/cr.pdf.
6. *Bishop, M.*Computer Security: Art and Science. Addison-Wesley, Reading, Massachusetts, 2003.
7. *Bishop, M.* Introduction to Computer Security. Addison-Wesley, Reading, Massachusetts, 2005.
8. *Borisov, N., Johnson, R., Sastry, N., Wagner, D.* Fixing Races for Fun and Profit: How to abuse atime // Proceedings of the 14th USENIX Security Symposium, 2005. http://www.cs.berkeley.edu/~nks/papers/races-usenix05.pdf.
9. *Bovet, D. P., Cesati, M.* Understanding the Linux Kernel (3rd edition). O'Reilly, Sebastopol, California, 2005.
10. *Butenhof, D. R.* Programming with POSIX Threads. Addison-Wesley, Reading, Massachusetts, 1996. http://homepage.mac.com/dbutenhof/Threads/Threads.html.
11. *Chen, H., Wagner, D., Dean, D.* Setuid Demystified // Proceedings of the 11th USENIX Security Symposium, 2002. http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf.
12. *Comer, D.E.* Internetworking with TCP/IP. Vol. I: Principles, Protocols, and Architecture (4th edition). Prentice Hall, Upper Saddle River, New Jersey,2000. http://www.cs.purdue.edu/homes/dec/netbooks.html.
13. *Comer, D.E., Stevens, D.L.* Internetworking with TCP/IP. Vol. II: Design, Implementation, and Internals (3rd edition). Prentice Hall, Upper Saddle River, New Jersey, 1999.
14. *Comer, D.E., Stevens, D.L.* Internetworking with TCP/IP. Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version. Prentice Hall, Englewood Cliffs, New Jersey, 2000.
15. *Corbet, J.* The Orlov block allocator // Linux Weekly News, 5 November 2002. http://lwn.net/Articles/14633/.
16. *Corbet, J., Rubini, A., Kroah-Hartman, G.* Linux Device Drivers (3rd edition). O'Reilly, Sebastopol, California, 2005. http://lwn.net/Kernel/LDD3/.
17. *Crosby, S.A., Wallach, D. S.* Denial of Service via Algorithmic Complexity Attacks // Proceedings of the 12th USENIX Security Symposium, 2003. http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf.
18. *Deitel, H.M., Deitel, P. J., Choffnes, D. R.* Operating Systems (3rd edition). Prentice Hall, Upper Saddle River, New Jersey, 2004.
19. *Dijkstra, E.W.* Cooperating Sequential Processes // Programming Languages, ed. F. Genuys, Academic Press, New York, 1968.
20. *Drepper, U.* Futexes Are Tricky, 2004 (a). http://people.redhat.com/drepper/futex.pdf.
21. *Drepper, U.* How to Write Shared Libraries, 2004 (b). http://people.redhat.com/drepper/dsohowto.pdf.
22. *Drepper, U.* What Every Programmer Should Know About Memory, 2007. http://people.redhat.com/drepper/cpumemory.pdf.
23. *Drepper, U.* Defensive Programming for Red Hat Enterprise Linux, 2009. http://people.redhat.com/drepper/defprogramming.pdf.

24. *Erickson, J.M.* Hacking: The Art of Exploitation (2nd edition). No Starch Press, San Francisco, California, 2008.
25. *Floyd, S.* TCP and Explicit Congestion Notification // ACM Computer Communication Review. Vol. 24, No. 5, October 1994, pages 10–23. http://www.icir.org/floyd/papers/tcp_ecn.4.pdf.
26. *Franke, H., Russell, R., Kirkwood, M.* Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux // Proceedings of the Ottawa Linux Symposium, 2002. http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.
27. *Frisch, A.* Essential System Administration (3rd edition). O'Reilly, Sebastopol, California, 2002.
28. *Gallmeister, B.O.* POSIX.4: Programming for the Real World. O'Reilly, Sebastopol, California, 1995.
29. *Gammo, L., Brecht, T., Shukla, A., Pariag, D.* Comparing and Evaluating epoll, select, and poll Event Mechanisms // Proceedings of the Ottawa Linux Symposium, 2002. http://www.kernel.org/doc/ols/2004/ols2004v1-pages-215-226.pdf.
30. *Gancarz, M.* Linux and the Unix Philosophy. Digital Press, 2003.
31. *Garfinkel, S., Spafford, G., Schwartz, A.* Practical Unix and Internet Security (3rd edition). O'Reilly, Sebastopol, California, 2003.
32. *Gont, F.* Security Assessment of the Internet Protocol. UK Centre for the Protection of the National Infrastructure, 2008. http://www.cpni.gov.uk/Docs/InternetProtocol.pdf.
33. *Gont, F.* Security Assessment of the Transmission Control Protocol (TCP). CPNI Technical Note 3/2009. UK Centre for the Protection of the National Infrastructure, 2009 (a). http://www.cpni.gov.uk/Docs/tn-03-09-security-assessment-TCP.pdf.
34. *Gont, F., Yourtchenko, A.* On the implementation of TCP urgent data // Internet draft, 20 May 2009(b). http://www.gont.com.ar/drafts/urgent-data/.
35. *Goodheart, B., Cox, J.* The Magic Garden Explained: The Internals of UNIX SVR4. Prentice Hall, Englewood Cliffs, New Jersey, 1994.
36. *Goralski, W.* The Illustrated Network: How TCP/IP Works in a Modern Network. Morgan Kaufmann, Burlington, Massachusetts, 2009.
37. *Gorman, M.* Understanding the Linux Virtual Memory Manager. Prentice Hall, Upper Saddle River, New Jersey, 2004. http://www.phptr.com/perens.
38. *Grünbacher, A.* "POSIX Access Control Lists on Linux // Proceedings of USENIX 2003/Freenix Track, pages 259–272. http://www.suse.de/~agruen/acl/linux-acls/online/.
39. *Gutmann, P.* Secure Deletion of Data from Magnetic and Solid-State Memory // Proceedings of the 6th USENIX Security Symposium, 1996. http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html.
40. *Hallyn, S.* POSIX file capabilities: Parceling the power of root, 2007. http://www.ibm.com/developerworks/library/l-posixcap.html.
41. *Harbison, S., Steele, G.* C: A Reference Manual (5th edition). Prentice Hall, Englewood Cliffs, New Jersey, 2002.
42. *Herbert, T.F.* The Linux TCP/IP Stack: Networking for Embedded Systems. Charles River Media, Hingham, Massachusetts, 2004.
43. *Hubička, J.* Porting GCC to the AMD64 Architecture // Proceedings of the First Annual GCC Developers' Summit, 2003. http://www.ucw.cz/~hubicka/papers/amd64/index.html.
44. *Johnson, M.K., Troan, E.W.* Linux Application Development (2nd edition). Addison-Wesley, Reading, Massachusetts, 2005.
45. *Josey, A. (ed.).* The Single UNIX Specification, Authorized Guide to Version 3. The Open Group, 2004. http://www.unix-systems.org/version3/theguide.html; http://www.opengroup.org/bookstore/ catalog/.

46. *Kent, A., Mogul, J.C.*Fragmentation Considered Harmful // ACM Computer Communication Review, Vol. 17, No. 5, August 1987. http://www.acm.org/sigcomm/ccr/archive/1995/jan95/ccr-9501-mogulf1.pdf.
47. *Kernighan, B.W., Ritchie, D.M.* The C Programming Language (2nd edition). Prentice Hall, Englewood Cliffs, New Jersey, 1988.
48. *Kopparapu, C.* Load Balancing Servers, Firewalls, and Caches. John Wiley and Sons, 2002.
49. *Kozierok, C.M.* The TCP/IP Guide. No Starch Press, San Francisco, California, 2005. http://www.tcpipguide.com/.
50. *Kroah-Hartman, G.* udev — A Userspace Implementation of devfs // Proceedings of the 2003 Linux Symposium, 2003. http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf.
51. *Kumar, A., Cao, M., Santos, J., Dilger, A.* Ext4 block and inode allocator improvements // Proceedings of the 2008 Linux Symposium, Ottawa, Canada, 2008. http://ols.fedoraproject.org/OLS/Reprints-2008/kumar-reprint.pdf.
52. *Lemon, J.* Kqueue: A generic and scalable event notification facility // Proceedings of USENIX 2001/Freenix Track, 2001. http://people.freebsd.org/~jlemon/papers/kqueue_freenix.pdf.
53. *Lemon, J.* Resisting SYN flood DoS attacks with a SYN cache // Proceedings of USENIX BSDCon, 2002. http://people.freebsd.org/~jlemon/papers/syncache.pdf.
54. *Levine, J.* Linkers and Loaders. Morgan Kaufmann, San Francisco, California, 2000. http://www.iecc.com/linker/.
55. *Lewine, D.* POSIX Programmer's Guide. O'Reilly, Sebastopol, California, 1991.
56. *Liang, S.* The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, Reading, Massachusetts, 1999. http://java.sun.com/docs/books/jni/.
57. *Libes, D., Ressler, S.* Life with UNIX: A Guide for Everyone. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
58. *Lions, J.* Lions' Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications, San Jose, California, 1996.
59. *Love, R.* Linux Kernel Development (3rd edition). Addison-Wesley, Reading, Massachusetts, 2010.
60. *Lu, H.J.* ELF: From the Programmer's Perspective, 1995. http://www.trunix.org/programlama/os/elf-hl/Documentation/elf/elf.html.
61. *Mann, S., Mitchell, E.L.* Linux System Security (2nd edition). Prentice Hall, Englewood Cliffs, New Jersey, 2003.
62. *Matloff, N. Salzman, P.J.* The Art of Debugging with GDB, DDD, and Eclipse. No Starch Press, San Francisco, California, 2008.
63. *Maxwell, S.* Linux Core Kernel Commentary. Coriolis, Scottsdale, Arizona, 1999.
64. *McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.* A fast file system for UNIX // ACM Transactions on Computer Systems. Vol. 2, Issue 3 (August), 1984.
65. *McKusick, M.K.* Twenty years of Berkeley Unix // Open Sources: Voices from the Open Source Revolution, C. DiBona, S. Ockman, and M. Stone (eds.). O'Reilly, Sebastopol, California, 1999.
66. *McKusick, M.K., Bostic, K., Karels, M.J.* The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, Massachusetts, 1996.
67. *McKusick, M.K., Neville-Neil, G.V.* The Design and Implementation of the FreeBSD Operating System. Addison-Wesley, Reading, Massachusetts, 2005.
68. *Mecklenburg, R.* Managing Projects with GNU Make (3rd edition). O'Reilly, Sebastopol, California, 2005.
69. *Mills, D.L.* Network Time Protocol (Version 3) Specification, Implementation and Analysis // RFC 1305, March 1992. http://www.rfc-editor.org/rfc/rfc1305.txt.

70. *Mochel, P.* The sysfs Filesystem // Proceedings of the Ottawa Linux Symposium, 2002.
71. *Mosberger, D., Eranian, S.* IA-64 Linux Kernel: Design and Implementation. Prentice Hall, Upper Saddle River, New Jersey, 2002.
72. *Peek, J., Todino-Gonguet, G., Strang, J.* Learning the UNIX Operating System (5th edition). O'Reilly, Sebastopol, California, 2001.
73. *Peikari, C., Chuvakin, A.* Security Warrior. O'Reilly, Sebastopol, California, 2004.
74. *Plauger, P.J.* The Standard C Library. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
75. *Quarterman, J.S., Wilhelm, S.* UNIX, Posix, and Open Systems: The Open Standards Puzzle. Addison-Wesley, Reading, Massachusetts,1993.
76. *Ritchie, D.M.* The Evolution of the UNIX Time-sharing System // AT&T Bell Laboratories Technical Journal, 63, No. 6 Part 2 (October 1984), pages 1577–93.
77. *Ritchie, D.M., Thompson, K.L.* The Unix Time-Sharing System // Communications of the ACM, 17 (July 1974), pages 365–375.
78. *Robbins, K.A., Robbins, S.* UNIX Systems Programming: Communication, Concurrency, and Threads (2nd edition). Prentice Hall, Upper Saddle River, New Jersey, 2003.
79. *Rochkind, M.J.* Advanced UNIX Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1985.
80. *Rochkind, M.J.* Advanced UNIX Programming (2nd edition). Addison-Wesley, Reading, Massachusetts, 2004.
81. *Rosen, L.* Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall, Upper Saddle River, New Jersey, 2005.
82. *St. Laurent, A.M.* Understanding Open Source and Free Software Licensing. O'Reilly, Sebastopol, California, 2004.
83. *Salus, P.H.* A Quarter Century of UNIX. Addison-Wesley, Reading, Massachusetts, 1994.
84. *Salus, P.H.* The Daemon, the Gnu, and the Penguin. Addison-Wesley, Reading, Massachusetts, 2008. http://www.groklaw.net/staticpages/index.php?page=20051013231901859.
85. *Sarolahti, P., Kuznetsov, A.* Congestion Control in Linux TCP // Proceedings of USENIX 2002 / Freenix Track, 2002. http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf.
86. *Schimmel, C.* UNIX Systems for Modern Architectures. Addison-Wesley, Reading, Massachusetts, 1994.
87. *Snader, J.C.* Effective TCP/IP Programming: 44 tips to improve your network programming. Addison-Wesley, Reading, Massachusetts, 2000.
88. *Stevens, W.R.* Advanced Programming in the UNIX Environment. Addison-Wesley, Reading, Massachusetts, 1992. http://www.kohala.com/start/.
89. *Stevens, W.R.* UNIX Network Programming, Volume 1 (2nd edition): Networking APIs: Sockets and XTI. Prentice Hall, Upper Saddle River, New Jersey, 1998.
90. *Stevens, W.R.* UNIX Network Programming, Volume 2 (2nd edition): Interprocess Communications. Prentice Hall, Upper Saddle River, New Jersey, 1999.
91. *Stevens, W.R.* TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley, Reading, Massachusetts, 1994.
92. *Stevens, W.R.* TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley, Reading, Massachusetts, 1996.
93. *Stevens, W.R., Fenner, B., Rudoff, A.M.* UNIX Network Programming, Volume 1 (3rd edition): The Sockets Networking API. Addison-Wesley, Boston, Massachusetts, 2004.
94. *Stevens, W.R., Rago, S.A.* Advanced Programming in the UNIX Environment (2nd edition). Addison-Wesley, Boston, Massachusetts, 2005.
95. *Stewart, R.R., Xie, Q.* Stream Control Transmission Protocol (SCTP). Addison-Wesley, Reading, Massachusetts, 2001.
96. *Stone, J., Partridge, C.* When the CRC and the TCP Checksum Disagree // Proceedings of SIGCOMM, 2000. http://www.acm.org/sigcomm/sigcomm2000/conf/abstract/9-1.htm.

97. *Strang, J.* Programming with Curses. O'Reilly, Sebastopol, California, 1986.
98. *Strang, J., Mui, L., O'Reilly, T.* Termcap & Terminfo (3rd edition). O'Reilly, Sebastopol, California, 1988.
99. *Tanenbaum, A.S.* Modern Operating Systems (3rd edition). Prentice Hall, Upper Saddle River, New Jersey, 2007.
100. *Tanenbaum, A.S.* Computer Networks (4th edition). Prentice Hall, Upper Saddle River, New Jersey, 2002.
101. *Tanenbaum, A.S., Woodhull, A.S.* Operating Systems: Design And Implementation (3rd edition). Prentice Hall, Upper Saddle River, New Jersey, 2006.
102. Torvalds, L.B., and Diamond, D. 2001. Just for Fun: The Story of an Accidental Revolutionary. HarperCollins, New York, New York.
103. *Tsafrir, D., da Silva, D., Wagner, D.* The Murky Issue of Changing Process Identity: Revising 'Setuid Demystified' // ;login: The USENIX Magazine, June 2008. http://www.usenix.org/publications/login/2008-06/pdfs/tsafrir.pdf.
104. *Vahalia, U.* UNIX Internals: The New Frontiers. Prentice Hall, Upper Saddle River, New Jersey, 1996.
105. *van der Linden, P.* Expert C Programming—Deep C Secrets. Prentice Hall, Englewood Cliffs, New Jersey, 1994.
106. *Vaughan, G.V., Elliston, B., Tromey, T., Taylor, I.L.* GNU Autoconf, Automake, and Libtool. New Riders, Indianapolis, Indiana, 2000. http://sources.redhat.com/autobook/.
107. *Viega, J., McGraw, G.* Building Secure Software. Addison-Wesley, Reading, Massachusetts, 2002.
108. *Viro, A., Pai, R.* Shared-Subtree Concept, Implementation, and Applications in Linux // Proceedings of the Ottawa Linux Symposium, 2006. http://www.kernel.org/doc/ols/2006/ols2006v2-pages-209-222.pdf.
109. *Watson, R.N.M.* Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD // Proceedings of BSDCon, 2000. http://www.trustedbsd.org/trustedbsd-bsdcon-2000.pdf.
110. *Williams, S.* 2002. Free as in Freedom: Richard Stallman's Crusade for Free Software. O'Reilly, Sebastopol, California.
111. *Wright, G.R., Stevens, W.R.* TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley, Reading, Massachusetts, 1995.

Майкл Керриск

Linux API. Исчерпывающее руководство

Перевели с английского *Н. Вильчинский, С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*Д. Новикова, Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*И. Низамов, Е. Павлович, Т. Радецкая*
Верстка	*А. Барцевич*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 10.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 04.10.17. Формат 70x100/16. Бумага писчая. Усл. п. л. 100,620. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**