

UNIX

Профессиональное программирование

3-е издание

У. Ричард Стивенс,
Стивен А. Раго

Advanced Programming in the UNIX® Environment

Third Edition

W. Richard Stevens
Stephen A. Rago

 Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

UNIX

Профессиональное программирование

3-е издание

У. Ричард Стивенс,
Стивен А. Раго



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск

2018

ББК 32.973.2-018.2

УДК 004.451

С80

У. Ричард Стивенс, Стивен А. Раго

С80 UNIX. Профессиональное программирование. 3-е изд. — СПб.: Питер, 2018. — 944 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0649-3

Эта книга заслуженно пользуется популярностью у серьезных программистов во всем мире, поскольку содержит самую важную и практическую информацию об управлении ядрами UNIX и Linux. Без этих знаний невозможно написать эффективный и надежный код.

От основ — файлы, каталоги и процессы — вы постепенно перейдете к более сложным вопросам, таким как обработка сигналов и терминальный ввод/вывод, многопоточная модель выполнения и межпроцессное взаимодействие с применением сокетов.

В общей сложности в этой книге охвачены более 70 интерфейсов, включая функции POSIX асинхронного ввода/вывода, циклические блокировки, барьеры и семафоры POSIX.

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 Addison-Wesley Longman. 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 978-0321637734 англ.

ISBN 978-5-4461-0649-3

© 2013 Pearson Education, Inc.

© Перевод на русский язык ООО Издательство «Питер», 2018

© Издание на русском языке, оформление ООО Издательство

«Питер», 2018

© Серия «Для профессионалов», 2018

Отзывы о втором издании

Обновление, выполненное Стивеном Раго (Stephen Rago), — это событие, которого давно и с нетерпением ждало все сообщество профессионалов, использующих в своей работе многоликое семейство UNIX и UNIX-подобных операционных систем. В этом издании исключены устаревшие и добавлены новейшие сведения. Содержание всех тем, примеров и прикладных программ обновлено в соответствии с последними версиями наиболее популярных реализаций UNIX и UNIX-подобных систем. И кроме того, сохранен стиль изложения классического оригинала.

*Мукеш Кэкер (Mukesh Kacker),
соучредитель и бывший технический директор Pronto Networks, Inc.*

Один из фундаментальных классических трудов, посвященных программированию для UNIX.

*Эрик С. Рэймонд (Eric S. Raymond),
автор книги *The Art of UNIX Programming**

Это подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стивену Раго удалось обновить и дополнить текст классической работы Стивенса, сохранив точность оригинала. Особенности прикладных программных интерфейсов разъясняются на простых и понятных примерах. В книге также описывается множество ловушек, о которых следует помнить при написании программ для различных реализаций UNIX, и показывается, как их можно избежать, опираясь на соответствующие стандарты, такие как POSIX 1003.1 (редакция от 2004 года) и Single UNIX Specification, Version 3.

*Эндрю Джози (Andrew Josey),
директор по сертификации The Open Group и председатель рабочей группы POSIX 1003.1*

Второе издание книги — жизненно необходимый справочник для любого, кто занимается разработкой программ для UNIX. Эту книгу я открываю первой, когда хочу изучить или вспомнить какие-либо из интерфейсов системы. Стивен Раго удачно переработал содержание книги и включил в нее сведения о новейших операционных системах, таких как GNU/Linux и Apple OS X, придерживаясь при этом стиля первого издания — как в смысле удобочитаемости, так и в смысле полноты изложения. Для нее всегда найдется место рядом с моим компьютером.

*Доктор Бенджамин Куперман (Dr. Benjamin Kuperman),
колледж г. Свартмор (Swarthmore)*

Отзывы о первом издании

Книга *Advanced Programming in the UNIX® Environment* обязательно должна быть у любого серьезного программиста, который пишет для UNIX на языке С. По своей основательности, глубине и ясности подачи материала она не имеет равных.

UniForum Monthly

Многочисленные читатели рекомендовали мне книгу *Advanced Programming in the UNIX® Environment* Ричарда Стивенса (издательство Addison-Wesley), и я благодарен им за это. Раньше я даже не слышал об этой книге, хотя она вышла еще в 1992 году. Получив экземпляр книги, я с первых же глав был очарован ею.

Open Systems Today

В книге *Advanced Programming in the UNIX® Environment* (Addison-Wesley), написанной Ричардом Стивенсом, вы найдете очень понятное и подробное описание внутреннего устройства UNIX. Она включает множество практических примеров, и я нахожу ее очень полезной при разработке системного программного обеспечения.

RS/Magazine

Краткое содержание

Вступительное слово ко второму изданию	20
Предисловие.....	22
Предисловие ко второму изданию	25
Предисловие к первому изданию	29
Глава 1. Обзор ОС UNIX	34
Глава 2. Стандарты и реализации UNIX.....	59
Глава 3. Файловый ввод/вывод	103
Глава 4. Файлы и каталоги.....	137
Глава 5. Стандартная библиотека ввода/вывода	192
Глава 6. Информация о системе и файлы данных.....	229
Глава 7. Окружение процесса	252
Глава 8. Управление процессами	284
Глава 9. Взаимоотношения между процессами	347
Глава 10. Сигналы	377
Глава 11. Потоки.....	454
Глава 12. Управление потоками	496
Глава 13. Процессы-демоны.....	538

Глава 14. Расширенные операции ввода/вывода.....	557
Глава 15. Межпроцессные взаимодействия	614
Глава 16. Межпроцессные взаимодействия в сети: сокеты	676
Глава 17. Расширенные возможности IPC.....	719
Глава 18. Терминальный ввод/вывод	761
Глава 19. Псевдотерминалы.....	807
Глава 20. Библиотека базы данных	837
Глава 21. Взаимодействие с сетевым принтером	887

Приложение А. Прототипы функцийwww.piter.com

Приложение В. Различные исходные текстыwww.piter.com

Приложение С. Варианты решения некоторых упражненийwww.piter.com

Приложения к книге доступны по ссылке: <https://goo.gl/AoCBfd>.

Оглавление

Вступительное слово ко второму изданию.....	20
Предисловие	22
Введение.....	22
Изменения в третьем издании.....	23
Благодарности	24
Предисловие ко второму изданию.....	25
Введение.....	25
Изменения во втором издании	26
Благодарности	27
Предисловие к первому изданию.....	29
Введение.....	29
Стандарты UNIX	29
Структура книги	29
Примеры в книге	30
Перечень систем, использовавшихся для тестирования примеров.....	31
Благодарности	32
От издательства	33
Глава 1. Обзор ОС UNIX.....	34
1.1. Введение	34
1.2. Архитектура UNIX	34
1.3. Вход в систему	35
1.4. Файлы и каталоги.....	37
1.5. Ввод и вывод.....	42
1.6. Программы и процессы.....	44
1.7. Обработка ошибок	48
1.8. Идентификация пользователя	50
1.9. Сигналы	52
1.10. Представление времени	54

1.11. Системные вызовы и библиотечные функции	55
1.12. Подведение итогов	58
Упражнения	58
Глава 2. Стандарты и реализации UNIX	59
2.1. Введение	59
2.2. Стандартизация UNIX	59
2.2.1. ISO C	59
2.2.2. IEEE POSIX	61
2.2.3. Single UNIX Specification	66
2.2.4. FIPS	70
2.3. Реализации UNIX	70
2.3.1. UNIX System V Release 4	71
2.3.2. 4.4BSD	71
2.3.3. FreeBSD	72
2.3.4. Linux	72
2.3.5. Mac OS X	73
2.3.6. Solaris	73
2.3.7. Прочие версии UNIX	73
2.4. Связь между стандартами и реализациями	74
2.5. Ограничения	74
2.5.1. Пределы ISO C	76
2.5.2. Пределы POSIX	77
2.5.3. Пределы XSI	81
2.5.4. Функции sysconf, pathconf и fpathconf	82
2.5.5. Неопределенные пределы времени выполнения	91
2.6. Необязательные параметры	94
2.7. Макроопределения проверки особенностей	99
2.8. Элементарные системные типы данных	100
2.9. Различия между стандартами	101
2.10. Подведение итогов	102
Упражнения	102
Глава 3. Файловый ввод/вывод	103
3.1. Введение	103
3.2. Дескрипторы файлов	103
3.3. Функции open и openat	104
3.4. Функция creat	108
3.5. Функция close	109
3.6. Функция lseek	109
3.7. Функция read	113
3.8. Функция write	114
3.9. Эффективность операций ввода/вывода	115

3.10. Совместное использование файлов	117
3.11. Атомарные операции	121
3.12. Функции dup и dup2.....	123
3.13. Функции sync, fsync и fdatasync	125
3.14. Функция fcntl	126
3.15. Функция ioctl.....	132
3.16. /dev/fd	133
3.17. Подведение итогов	135
Упражнения	135
Глава 4. Файлы и каталоги	137
4.1. Введение.....	137
4.2. Функции stat, fstat и lstat	137
4.3. Типы файлов	139
4.4. set-user-ID и set-group-ID.....	142
4.5. Права доступа к файлу	143
4.6. Принадлежность новых файлов и каталогов.....	146
4.7. Функции access и faccessat.....	147
4.8. Функция umask.....	149
4.9. Функции chmod, fchmod и fchmodat.....	151
4.10. Бит sticky	154
4.11. Функции chown, fchown, fchownat и lchown	155
4.12. Размер файла	157
4.13. Усечение файлов	158
4.14. Файловые системы.....	159
4.15. Функции link, linkat, unlink, unlinkat и remove	162
4.16. Функции rename и renameat.....	166
4.17. Символические ссылки.....	167
4.18. Создание и чтение символьических ссылок	170
4.19. Временные характеристики файлов	171
4.20. Функции futimens, utimensat и utimes.....	174
4.21. Функции mkdir, mkdirat и rmdir	177
4.22. Чтение каталогов.....	178
4.23. Функции chdir, fchdir и.getcwd	183
4.24. Специальные файлы устройств	186
4.25. Коротко о битах прав доступа к файлам	188
4.26. Подведение итогов	190
Упражнения	190
Глава 5. Стандартная библиотека ввода/вывода.....	192
5.1. Введение	192
5.2. Потоки и объекты FILE	192
5.3. Стандартные потоки ввода, вывода и сообщений об ошибках	194

5.4. Буферизация	194
5.5. Открытие потока	197
5.6. Чтение из потока и запись в поток	200
5.7. Построчный ввод/вывод	203
5.8. Эффективность стандартных функций ввода/вывода.....	204
5.9. Ввод/вывод двоичных данных	207
5.10. Позиционирование в потоке	208
5.11. Форматированный ввод/вывод.....	210
5.12. Подробности реализации.....	216
5.13. Временные файлы	220
5.14. Потоки ввода/вывода в памяти.....	223
5.15. Альтернативы стандартной библиотеке ввода/вывода	227
5.16. Подведение итогов	228
Упражнения	228
Глава 6. Информация о системе и файлы данных.....	229
6.1. Введение	229
6.2. Файл паролей.....	229
6.3. Теневые пароли	233
6.4. Файл групп	235
6.5. Идентификаторы дополнительных групп	236
6.6. Различия реализаций	238
6.7. Прочие файлы данных	239
6.8. Учет входов в систему	240
6.9. Информация о системе.....	241
6.10. Функции даты и времени	243
6.11. Подведение итогов	251
Упражнения	251
Глава 7. Окружение процесса.....	252
7.1. Введение	252
7.2. Функция main	252
7.3. Завершение работы процесса	252
7.4. Аргументы командной строки.....	257
7.5. Список переменных окружения.....	258
7.6. Организация памяти программы на языке С	259
7.7. Разделяемые библиотеки	261
7.8. Распределение памяти.....	262
7.9. Переменные окружения	266
7.10. Функции setjmp и longjmp	270
7.11. Функции getrlimit и setrlimit	277
7.12. Подведение итогов	282
Упражнения	282

Глава 8. Управление процессами	284
8.1. Введение	284
8.2. Идентификаторы процесса	284
8.3. Функция fork.....	286
8.4. Функция vfork.....	292
8.5. Функции exit.....	294
8.6. Функции wait и waitpid.....	296
8.7. Функция waitid	303
8.8. Функции wait3 и wait4.....	304
8.9. Гонка за ресурсами.....	305
8.10. Функции exec.....	308
8.11. Изменение идентификаторов пользователя и группы	315
8.12. Интерпретируемые файлы	321
8.13. Функция system	326
8.14. Учет использования ресурсов процессами	331
8.15. Идентификация пользователя	337
8.16. Планирование процессов.....	338
8.17. Временные характеристики процесса.....	342
8.18. Подведение итогов	345
Упражнения	345
Глава 9. Взаимоотношения между процессами	347
9.1. Введение	347
9.2. Вход с терминала	347
9.3. Вход в систему через сетевое соединение	353
9.4. Группы процессов.....	356
9.5. Сеансы	357
9.6. Управляющий терминал	359
9.7. Функции tcgetpgrp, tcsetpgrp и tcgetsid.....	361
9.8. Управление заданиями	362
9.9. Выполнение программ командной оболочкой	367
9.10. Осирачившие группы процессов	371
9.11. Реализация в FreeBSD.....	374
9.12. Подведение итогов	376
Упражнения	376
Глава 10. Сигналы	377
10.1. Введение	377
10.2. Концепция сигналов	377
10.3. Функция signal	389
10.4. Ненадежные сигналы.....	393
10.5. Прерванные системные вызовы	394

10.6. Реентерабельные функции	397
10.7. Семантика сигнала SIGCLD.....	400
10.8. Надежные сигналы. Терминология и семантика.....	403
10.9. Функции kill и raise.....	404
10.10. Функции alarm и pause.....	406
10.11. Наборы сигналов	412
10.12. Функция sigprocmask.....	414
10.13. Функция sigpending	415
10.14. Функция sigaction.....	418
10.15. Функции sigsetjmp и siglongjmp	424
10.16. Функция sigsuspend.....	428
10.17. Функция abort	434
10.18. Функция system	437
10.19. Функции sleep, nanosleep и clock_nanosleep.....	442
10.20. Функция sigqueue.....	446
10.21. Сигналы управления заданиями.....	447
10.22. Имена и номера сигналов	450
10.23. Подведение итогов	451
Упражнения	452
Глава 11. Потоки.....	454
11.1. Введение	454
11.2. Понятие потоков.....	454
11.3. Идентификация потоков	455
11.4. Создание потока	457
11.5. Завершение потока.....	460
11.6. Синхронизация потоков	468
11.6.1. Мьютексы.....	471
11.6.2. Предотвращение тупиковых ситуаций	474
11.6.3. Функция pthread_mutex_timedlock	479
11.6.4. Блокировки чтения-записи	480
11.6.5. Блокировки чтения-записи с тайм-аутом	485
11.6.6. Переменные состояния.....	485
11.6.7. Циклические блокировки	489
11.6.8. Барьеры	491
11.7. Подведение итогов	495
Упражнения	495
Глава 12. Управление потоками.....	496
12.1. Введение	496
12.2. Пределы для потоков	496
12.3. Атрибуты потока.....	497

12.4. Атрибуты синхронизации	502
12.4.1. Атрибуты мьютексов	502
12.4.2. Атрибуты блокировок чтения-записи	511
12.4.3. Атрибуты переменных состояния	512
12.4.4. Атрибуты барьеров	513
12.5. Реентерабельность	513
12.6. Локальные данные потоков	518
12.7. Принудительное завершение потоков	523
12.8. Потоки и сигналы	527
12.9. Потоки и fork	531
12.10. Потоки и операции ввода/вывода	535
12.11. Подведение итогов	536
Упражнения	536
Глава 13. Процессы-демоны	538
13.1. Введение	538
13.2. Характеристики демонов	538
13.3. Правила программирования демонов	541
13.4. Журналирование ошибок	544
13.5. Демоны в единственном экземпляре	549
13.6. Соглашения для демонов	551
13.7. Модель клиент-сервер	555
13.8. Подведение итогов	556
Упражнения	556
Глава 14. Расширенные операции ввода/вывода	557
14.1. Введение	557
14.2. Неблокирующий ввод/вывод	557
14.3. Блокировка записей	561
14.4. Мультиплексирование ввода/вывода	577
14.4.1. Функции select и pselect	580
14.4.2. Функция poll	585
14.5. Асинхронный ввод/вывод	588
14.5.1. Асинхронный вывод в System V	589
14.5.2. Асинхронный ввод/вывод в BSD	590
14.5.3. Асинхронный ввод/вывод в POSIX	590
14.6. Функции readv и writev	600
14.7. Функции readn и writen	602
14.8. Операции ввода/вывода с отображаемой памятью	604
14.9. Подведение итогов	612
Упражнения	612

Глава 15. Межпроцессные взаимодействия.....	614
15.1. Введение	614
15.2. Неименованные каналы	616
15.3. Функции popen и pclose	623
15.4. Сопроцессы	629
15.5. FIFO.....	634
15.6. XSI IPC.....	638
15.6.1. Идентификаторы и ключи	638
15.6.2. Структура прав доступа.....	640
15.6.3. Конфигурируемые пределы	641
15.6.4. Преимущества и недостатки	641
15.7. Очереди сообщений.....	643
15.8. Семафоры.....	649
15.9. Разделяемая память	656
15.10. Семафоры POSIX.....	664
15.11. Свойства взаимодействий типа клиент-сервер	670
15.12. Подведение итогов	673
Упражнения	674
Глава 16. Межпроцессные взаимодействия в сети: сокеты.....	676
16.1. Введение	676
16.2. Дескрипторы сокетов.....	676
16.3. Адресация	681
16.3.1. Порядок байтов.....	681
16.3.2. Форматы адресов	683
16.3.3. Определение адреса	685
16.3.4. Присваивание адресов сокетам	692
16.4. Установка соединения	694
16.5. Передача данных.....	698
16.6. Параметры сокетов.....	712
16.7. Экстренные данные.....	715
16.8. Неблокирующий и асинхронный ввод/вывод.....	716
16.9. Подведение итогов	717
Упражнения	718
Глава 17. Расширенные возможности IPC.....	719
17.1. Введение	719
17.2. Сокеты домена UNIX	719
17.3. Уникальные соединения.....	725
17.4. Передача дескрипторов файлов.....	731
17.5. Сервер открытия файлов, версия 1	742

17.6. Сервер открытия файлов, версия 2	748
17.7. Подведение итогов.....	759
Упражнения	759
Глава 18. Терминальный ввод/вывод	761
18.1. Введение	761
18.2. Обзор	761
18.3. Специальные символы ввода	770
18.4. Получение и изменение характеристик терминала.....	776
18.5. Флаги режимов терминала.....	777
18.6. Команда stty	784
18.7. Функции для работы со скоростью передачи.....	785
18.8. Функции управления линией связи	786
18.9. Идентификация терминала	787
18.10. Канонический режим	793
18.11. Неканонический режим.....	796
18.12. Размер окна терминала.....	803
18.13. termcap, terminfo и curses.....	805
18.14. Подведение итогов	806
Упражнения	806
Глава 19. Псевдотерминалы	807
19.1. Введение	807
19.2. Обзор	807
19.3. Открытие устройств псевдотерминалов	814
19.4. Функция pty_fork	819
19.5. Программа pty	822
19.6. Использование программы pty.....	826
19.7. Дополнительные возможности	833
19.8. Подведение итогов	835
Упражнения	835
Глава 20. Библиотека базы данных	837
20.1. Введение	837
20.2. Предыстория	837
20.3. Библиотека.....	838
20.4. Обзор реализации	841
20.5. Централизация или децентрализация?	845
20.6. Одновременный доступ	847
20.7. Сборка библиотеки	849
20.8. Исходный код	849
20.9. Производительность	878

20.10. Подведение итогов	884
Упражнения	885
Глава 21. Взаимодействие с сетевым принтером	887
21.1. Введение	887
21.2. Протокол печати через Интернет.....	887
21.3. Протокол передачи гипертекста	891
21.4. Очередь печати	892
21.5. Исходный код	894
21.6. Подведение итогов	942
Упражнения	942

Приложение А. Прототипы функцийwww.piter.com

Приложение В. Различные исходные текстыwww.piter.com

Приложение С. Варианты решения некоторых упражненийwww.piter.com

Приложения к книге доступны по ссылке: <https://goo.gl/AoCBfd>.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Приложения к книге доступны по ссылке: <https://goo.gl/AoCBfd>.

Моим родителям, Ленарду и Грейс

Вступительное слово ко второму изданию

В каждом интервью или в дискуссии после лекции мне задают один и тот же вопрос: «Ожидали ли вы, что UNIX продержится так долго?». Разумеется, в ответ я говорю одно и то же: «Нет, для нас это оказалось полной неожиданностью». Кто-то даже высчитал, что система в том или ином виде существует уже более половины всей жизни компьютерной индустрии.

Процесс развития индустрии был бурным и сложным. По сравнению с началом 70-х годов компьютерные технологии сильно изменились, особенно в области поддержки сетей, вездесущей графики и широкого распространения персональных компьютеров, но система сумела учесть и вобрать в себя все эти явления. Несмотря на то что сегодня в области настольных систем доминируют Microsoft и Intel, рынок определенно движется в направлении от единого поставщика к нескольким, а в последние годы все более ориентируется на открытые стандарты и свободно распространяемые продукты.

К счастью, система UNIX, если рассматривать ее как явление, а не только как торговую марку, не просто двигалась вперед, но сумела возглавить эти тенденции. В 70-х и 80-х годах XX века держателем авторских прав на исходные тексты UNIX была корпорация AT&T, но она всячески поощряла усилия по стандартизации системных интерфейсов и языков. Например, AT&T опубликовала описание интерфейса System V (System V Interface Definition, SVID), которое легло в основу стандарта POSIX. Так получилось, что UNIX легко смогла приспособиться к работе в сетевом окружении и, может быть, не так легко, но довольно эффективно — к работе с графикой. Кроме того, основные интерфейсы ядра UNIX и инструментальные средства уровня пользователя послужили технологической основой для появления движения за программное обеспечение с открытым исходным кодом. Важно отметить, что статьи и книги, посвященные системе UNIX, всегда были востребованы, даже когда реализация самой системы была закрыта. Примером может служить книга Мориса Баха (Maurice Bach) «The Design of the Unix Operating System». Честно говоря, я мог бы утверждать, что такая долговечность системы во многом обусловлена ее привлекательностью для талантливых авторов, которые стремились объяснить ее красоты и тайны. Брайан Керниган (Brian Kernighan) — один из них, а другой, конечно же, Рич Стивенс (Rich Stevens). Первое издание этой книги, а также серия его книг, посвященных сетевым технологиям, справедливо считаются одними из лучших и потому пользуются заслуженной популярностью.

Первое издание этой книги вышло в свет еще до того, как получили широкое распространение Linux и другие реализации UNIX с открытым исходным кодом, берущие начало из Беркли, а также когда большинство людей имели лишь модемное подключение к сети. Стив Раго (Steve Rago) переработал эту книгу, чтобы учесть изменения, произшедшие в компьютерных технологиях и в стандартах ISO и IEEE с момента выхода первой публикации. Поэтому все примеры в книге обновлены и вновь протестированы.

Это самое достойное второе издание классики.

Мюррей Хилл, Нью-Джерси

Деннис Ритчи

Март 2005

Предисловие

Введение

Прошло почти восемь лет, как я приступил к обновлению первого издания, и за эти годы многое изменилось.

- К моменту публикации второго издания организация The Open Group выпустила новую редакцию спецификации Single UNIX Specification от 2004 года, включив в нее два блока изменений. В 2008-м The Open Group выпустила очередную версию спецификации Single UNIX Specification, дополнив основные определения, добавив новые и удалив устаревшие интерфейсы. В нее была включена версия 7 спецификации Base Specification, и в 2009-м она была опубликована как версия POSIX.1 2008 года. В 2010-м была опубликована версия 4 спецификации Single UNIX Specification, куда вошло обновленное определение интерфейса библиотеки curses.
- Организация The Open Group сертифицировала как UNIX-системы версии 10.5, 10.6 и 10.8 операционной системы Mac OS X, выполняющиеся на аппаратной архитектуре Intel.
- Компания Apple Computer прекратила разработку Mac OS X для платформы PowerPC. Начиная с версии 10.6 (Snow Leopard), новые версии операционной системы выпускались только для платформы x86.
- Были открыты исходные коды операционной системы Solaris, чтобы повысить ее конкурентоспособность с популярной открытой моделью разработки, которой следуют FreeBSD, Linux и Mac OS X. Когда в 2010-м Oracle Corporation купила Sun Microsystems, она прекратила разработку OpenSolaris. Вследствие этого сообществом Solaris был образован открытый проект Illumos на основе OpenSolaris, в рамках которого было продолжено развитие системы. За дополнительной информацией обращайтесь по адресу <http://www.illumos.org>.
- В 2011-м был обновлен стандарт языка C, но так как еще не все системы переориентировались на него, мы продолжим ссылаться на стандарт 1999 года.

Особо следует отметить, что платформы, использовавшиеся в качестве примеров во втором издании, значительно устарели. В этом, третьем, издании я буду опираться на следующие платформы:

1. FreeBSD 8.0, наследница 4.4BSD, от Computer Systems Research Group из Калифорнийского университета в Беркли, работающая на 32-разрядном процессоре Intel Pentium.
2. Linux 3.2.0 (дистрибутив Ubuntu 12.04), свободно распространяемая UNIX-подобная операционная система, работающая на 64-разрядном процессоре Intel Core i5.

3. Apple Mac OS X, версия 10.6.8 (Darwin 10.8.0), работающая на 64-разрядном процессоре Intel Core 2 Duo. (Ядро Darwin основано на ядрах FreeBSD и Mach.) Я выбрал версию для платформы Intel, потому что последние версии Mac OS X больше не поддерживают платформу PowerPC. Недостаток этого выбора в том, что появился перекос в сторону Intel. А при обсуждении проблем разнородности весьма полезно иметь процессоры с различными характеристиками, такими как порядок следования байтов и размер целого числа.
4. Solaris 10 (производная от System V Release 4), выпущенная компанией Sun Microsystems (ныне Oracle) и работающая на 64-разрядном процессоре UltraSPARC IIi.

Изменения в третьем издании

Одним из значительных изменений в версии POSIX.1-2008 спецификации Single UNIX Specification является объявление устаревшими интерфейсов, имеющих отношение к STREAMS. Это первый шаг на пути удаления интерфейсов из будущих версий стандарта. По этой причине я скрепя сердце удалил из этого издания книги все, что относилось к STREAMS. Я не считаю это изменение удачным, потому что интерфейсы STREAMS выглядят более привлекательными на фоне сокетов и во многих отношениях являются более гибкими. Должен признать, что я не могу быть полностью беспристрастным, когда дело доходит до механизма STREAMS, но у меня нет причин оспаривать понижение его значимости в современных системах:

- Linux не включает поддержку STREAMS в базовую систему, хотя существуют пакеты (LiS и OpenSS7), добавляющие эту функциональность;
- Solaris 10 включает поддержку STREAMS, но уже Solaris 11 использует реализацию сокетов, не опирающуюся на интерфейсы STREAMS;
- Mac OS X не включает поддержку STREAMS;
- FreeBSD не включает поддержку STREAMS (и никогда не включала).

После удаления разделов, связанных с механизмом STREAMS, появилась возможность добавить обсуждение новых тем, таких как POSIX-совместимый асинхронный ввод/вывод.

Второе издание книги охватывало ядро Linux версии 2.4. В этом издании я использую версию 3.2. Самое большое отличие между ними заключается в подсистеме управления потоками выполнения. В версии Linux 2.6 поддержка потоков выполнения была реализована на основе библиотеки Native POSIX Thread Library (NPTL), которая делает потоки выполнения в Linux более похожими на потоки в других системах.

Если говорить в общем, в это издание было включено более 70 новых интерфейсов, в том числе интерфейсов асинхронного ввода/вывода, циклические блокировки (spin locks), барьеры (barriers) и семафоры POSIX. Описание наиболее устаревших интерфейсов было удалено, кроме некоторых, распространенных повсеместно.

Благодарности

Я получил от читателей множество комментариев и замечаний после выхода второго издания и хочу поблагодарить их за то, что помогли повысить точность представленной информации. Далее перечислены те, кто оказались первыми: Сет Арнольд (Seth Arnold), Люк Баккен (Luke Bakken), Рик Баллард (Rick Ballard), Йоханнес Биттнер (Johannes Bittner), Дэвид Брондер (David Bronder), Влад Буслов (Vlad Buslov), Питер Батлер (Peter Butler), Ю-Чинг Чен (Yuching Chen), Майк Чэнг (Mike Cheng), Джим Коллинз (Jim Collins), Боб Кузинс (Bob Cousins), Уилл Деннис (Will Dennis), Томас Дики (Thomas Dickey), Лойк Домайнэ (Loïc Domaingé), Игорь Фуксман (Igor Fuksman), Алекс Гезерлис (Alex Gezerlis), М. Скотт Гордон (M. Scott Gordon), Тимоти Гойя (Timothy Goya), Тони Грэхем (Tony Graham), Майкл Хобгуд (Michael Hobgood), Майкл Керриск (Michael Kerrisk), Юнг-Ху Квон (Youngho Kwon), Ричард Ли (Richard Li), Сюйек Лиу (Xueke Liu), Юн Лонг (Yun Long), Дэн Мак-Грегор (Dan McGregor), Дилан Мак-Нами (Dylan McNamee), Грег Миллер (Greg Miller), Симон Морган (Simon Morgan), Гарри Ньютон (Harry Newton), Джим Олдфилд (Jim Oldfield), Скотт Парриш (Scott Parish), Звездан Петкович (Zvezdan Petkovic), Дэвид Рейсс (David Reiss), Константинос Сакутис (Konstantinos Sakoutis), Дэвид Смут (David Smoot), Дэвид Сомерс (David Somers), Андрей Ткачук (Andriy Tkachuk), Натан Уикс (Nathan Weeks), Флориан Веймер (Florian Weimer), Ку-Инг-Янг Сюй (Qingyang Xu) и Майкл Залокар (Michael Zalokar).

Технические рецензенты помогли повысить точность представленной в книге информации. Большое спасибо Стиву Альберту (Steve Albert), Богдану Барбу (Bogdan Barbu) и Роберту Даю (Robert Day). Особую благодарность хочу выразить Джиффи Клэр (Geoff Clare) и Эндрю Джози (Andrew Josey) за дополнительные сведения, касающиеся спецификации Single UNIX Specification, и помочь в уточнении информации в главе 2. Кроме того, хочу поблагодарить Кена Томпсона (Ken Thompson) за ответы на вопросы, связанные с историей развития UNIX.

Еще раз хочу поблагодарить сотрудников издательства Addison-Wesley, с которыми было приятно работать. Спасибо Киму Бодингхаймеру (Kim Boedigheimer), Ромми Френчу (Romny French), Джону Фуллеру (John Fuller), Джессике Голдстейн (Jessica Goldstein), Джулии Нэйхил (Julie Nahil) и Дебре Уильямс-Коули (Debra Williams-Cauley). Также выражая свою благодарность Джиллу Хоббсу (Jill Hobbs) за отличную работу по техническому редактированию.

Наконец, я благодарен моей семье, которая с пониманием отнеслась к тому, что я потратил массу времени на работу над этим изданием.

Как и прежде, исходный код примеров из этой книги доступен на сайте [www.apuebook.com](http://apuebook.com). Я буду благодарен всем читателям, кто пришлет по электронной почте свои комментарии, предложения и замечания об ошибках.

Уоррен, Нью-Джерси
Стивен Раго
Январь 2013
sar@apuebook.com

Предисловие ко второму изданию

Введение

Мой первый контакт с Ричем Стивенсом состоялся по электронной почте, когда я сообщил ему об опечатке в его книге «UNIX Network Programming»¹. Позже он говорил, что я оказался первым, кто прислал ему сообщение о найденной ошибке. До самой его смерти в 1999 году мы время от времени обменивались электронными письмами. Обычно это происходило, когда один из нас задавался каким-то вопросом и полагал, что другой мог бы на него ответить. Мы встречались с ним за обедом на конференциях USENIX и на лекциях, которые он читал.

Рич Стивенс был другом и настоящим джентльменом. Когда в 1993 году я написал книгу «UNIX System V Network Programming», я представлял ее как версию книги Рича «UNIX Network Programming», ориентированную на System V. Рич охотно взялся за рецензирование моих глав, воспринимая меня не как конкурента, но как коллегу. Мы часто обсуждали сотрудничество в работе над версией его книги «TCP/IP Illustrated»², посвященной STREAMS. Если бы события сложились по-иному, вероятно, мы сделали бы это, но Ричарда больше нет с нами, поэтому приближением к осуществлению совместной работы я могу считать обновление данной книги.

Когда издательство Addison-Wesley сообщило, что заинтересовано в обновлении книги Рича, я думал, что дополнений будет не очень много. Даже по прошествии 13 лет его работа остается актуальной. Но современный мир UNIX значительно отличается от того, каким он был во время выхода первого издания книги.

- Версии System V практически вытеснила операционная система Linux. Основные производители аппаратного обеспечения, поставляющие свою продукцию в комплекте с собственными версиями UNIX, либо выпустили версии своих продуктов для Linux, либо объявили о ее поддержке. ОС Solaris, вероятно, осталась последней наследницей System V Release 4, обладающей более или менее заметной долей рынка.
- После выхода 4.4BSD группа по проведению исследований в области информационных технологий (Computing Science Research Group, CSRG) из Ка-

¹ Стивенс У., Феннер Б., Рудофф Э. UNIX: разработка сетевых приложений / Пер. с англ. — СПб.: Питер, 2006.

² Ричард У. Стивенс. Протоколы TCP/IP. В подлиннике / Пер. с англ. — СПб.: БХВ-Санкт-Петербург, 2003.

лифорнийского университета в Беркли принял решение о завершении разработки операционной системы UNIX, однако существует несколько групп добровольцев, которые продолжают поддерживать открытые версии.

- Появление ОС Linux, поддерживаемой тысячами добровольцев, позволило любому обладать компьютером, работающим под управлением UNIX-подобной операционной системы, распространяемой с открытым исходным кодом для новейших устройств. Успех Linux выглядит не совсем обычно, учитывая, что существует несколько свободно распространяемых BSD-альтернатив.
- В очередной раз проявив себя в качестве инновационной компании, Apple Computer отказалась от устаревшей операционной системы Mac и заменила ее системой, созданной на основе Mach и FreeBSD.

В связи с этим я попытался дополнить сведения, содержащиеся в книге, чтобы охватить эти четыре платформы.

Когда в 1992 году Рич написал свою книгу «Advanced Programming in the UNIX Environment», я избавился от большей части имевшихся у меня руководств по программированию в UNIX и оставил на своем столе две книги: словарь и «Advanced Programming in the UNIX Environment». Надеюсь, что вы найдете это издание книги не менее полезным.

Изменения во втором издании

Работа Рича не потеряла актуальности. Я старался сохранить оригинальное изложение материала, но слишком многое произошло за последние 13 лет. Особен-но это относится к стандартам, которые затрагивают программные интерфейсы UNIX.

Везде, где это было необходимо, я дополнил описания интерфейсов, которые изменились в результате стандартизации. Особенно это заметно в главе 2, посвященной стандартам. В этом издании мы будем основываться на стандарте POSIX.1 2001 года как более универсальном по сравнению с версией 1990 года, на которой была основана первая редакция книги. Стандарт ISO C 1990 года был изменен и дополнен в 1999 году, и некоторые изменения коснулись интерфейсов, описываемых стандартом POSIX.1.

Сегодня спецификация POSIX.1 охватывает намного большее количество интерфейсов. Основные спецификации Single UNIX Specification (опубликованные The Open Group, ранее X/Open) вошли в состав POSIX.1. Теперь POSIX.1 включает в себя ряд стандартов 1003.1 и некоторые из предварительных стандартов, опубликованных ранее.

В соответствии с этим я дополнил книгу новыми главами, охватывающими новые темы. Понятия потоков и многопоточных приложений очень важны, поскольку они дают программистам более элегантный способ организации параллельных вычислений и асинхронной обработки.

Интерфейс сокетов теперь стал частью стандарта POSIX.1. Он обеспечивает еди-ный интерфейс межпроцессных взаимодействий (Interprocess Communication,

IPC), не зависящий от местонахождения процессов, и его обсуждение является естественным продолжением глав, посвященных IPC.

Я опустил рассмотрение большинства интерфейсов реального времени, которые появились в POSIX.1. Их лучше всего изучать по книгам, специально посвященным созданию приложений реального времени. Одну из таких книг вы найдете в списке литературы.

Я изменил некоторые примеры в последних главах, чтобы приблизить их к практическим задачам. Например, в наши дни немногие системы работают с PostScript-принтерами через последовательный или параллельный порт. Чаще встречается случай, когда доступ к таким принтерам осуществляется посредством сети, поэтому я изменил учебный пример взаимодействия с принтером, чтобы учесть это обстоятельство.

Глава, посвященная взаимодействию с модемом, потеряла актуальность. Однако чтобы оригинальный материал не был утерян окончательно, он выложен на веб-сайте книги в двух форматах: PostScript (<http://www.apuebook.com/lostchapter/modem.ps>) и PDF (<http://www.apuebook.com/lostchapter/modem.pdf>).

Все исходные тексты примеров из книги также доступны на сайте www.apuebook.com. Большая часть примеров была протестирована на четырех plataформах.

1. FreeBSD 5.2.1 — системе, производной от 4.4BSD, от Computer Systems Research Group из Калифорнийского университета в Беркли, работающей на процессоре Intel Pentium.
2. Linux 2.4.22 (дистрибутив Mandrake 9.2) — свободно распространяемой UNIX-подобной операционной системе, работающей на процессоре Intel Pentium.
3. Solaris 9 — производной от System V Release 4, от Sun Microsystems, работающей на 64-разрядном процессоре UltraSPARCII.
4. Darwin 7.4.0 — системе, основанной на FreeBSD и Mach, которая поддерживается Apple Mac, работающей на процессоре PowerPC.

Благодарности

Первое издание этой книги, которое сразу же стало классикой, было полностью написано Ричем Стивенсоном.

Вероятно, мне не удалось бы справиться с обновлением книги без поддержки моей семьи. Они stoически терпели груды разбросанных повсюду бумаг (их было гораздо больше, чем обычно), монополизацию мною большинства компьютеров в доме и огромное количество времени, когда мое лицо было скрыто за терминатором. Моя супруга Джин (Jeanne) даже помогла мне, установив Linux на одну из тестовых машин.

Технические рецензенты предложили немало улучшений и исправлений и помогли удостовериться в правильности содержимого книги. Большое спасибо Дэвиду Бозуму (David Bausum), Дэвиду Борхему (David Boreham), Кейту Бостику (Keith Bostic), Марку Эллису (Mark Ellis), Филу Говарду (Phil Howard), Эндрю Джо-

зи (Andrew Josey), Мукешу Кэкеру (Mukesh Kacker), Брайану Кернигану (Brian Kernighan), Бенгту Клебергу (Bengt Kleberg), Бену Куперману (Ben Kuperman), Эрику Рэймонду (Eric Raimond) и Энди Рудоффу (Andy Rudoff).

Я также хотел бы поблагодарить Энди Рудофа за ответы на вопросы, касающиеся ОС Solaris, и Денниса Ритчи за то, что он «перекапывал» старые работы в поисках фактов по истории UNIX. Еще раз хочу поблагодарить сотрудников издательства Addison-Wesley, с которыми было приятно работать. Спасибо Тиррелу Альбо (Tyrrell Albaugh), Мэри Франц (Mary Franz), Джону Фуллеру (John Fuller), Карен Геттман (Karen Gettman), Джессике Голдстейн (Jessica Goldstein), Норин Реджине (Noreen Regina) и Джону Уэйтю (John Wait). Мои благодарности Эвелин Пайл (Evelyn Pyle) за отличную работу по техническому редактированию.

Как и Рич, я также буду благодарен всем читателям, кто пришлет свои комментарии, предложения и замечания об ошибках.

Уоррен, Нью-Джерси

Стивен Раго

Апрель 2005

sar@apuebook.com

Предисловие к первому изданию

Введение

В этой книге описаны программные интерфейсы системы UNIX: интерфейс системных вызовов и многочисленные функции из стандартной библиотеки языка С. Она предназначена для всех, кто пишет программы, работающие под управлением UNIX.

Подобно большинству операционных систем, UNIX предоставляет работающим в ней программам разнообразные услуги: открытие и чтение файлов, запуск новых программ, выделение памяти, получение текущего времени и т. д. Все это называется *интерфейсом системных вызовов* (system call interface). Кроме того, стандартная библиотека языка С включает огромное количество функций, которые используются практически в любой программе, написанной на С (форматированный вывод значений переменных, сравнение строк и т. п.).

Интерфейс системных вызовов и библиотечные функции традиционно описываются во втором и третьем разделах «Unix Programmer's Manual» (Руководство программиста UNIX). Эта книга не дублирует указанные разделы. В ней вы найдете примеры и пояснения, которые отсутствуют в упомянутом руководстве.

Стандарты UNIX

Быстрый рост количества версий UNIX, наблюдавшийся в 80-е годы, был урегулирован различными международными стандартами, которые стали появляться с конца 80-х. К ним относится стандарт ANSI языка программирования С, семейство стандартов IEEE POSIX (которые продолжают развиваться по сей день) и руководство по обеспечению переносимости X/Open.

Данная книга описывает эти стандарты. И не просто описывает, а рассматривает их применительно к популярным реализациям — System V Release 4 и 4.4BSD. Здесь представлены соответствующие действительности описания, которых зачастую недостает самим стандартам и книгам, лишь описывающим стандарты.

Структура книги

Эта книга делится на шесть частей.

1. Обзор и знакомство с базовыми понятиями, связанными с программированием в UNIX, и с терминологией (глава 1). Обсуждение достижений в области стандартизации UNIX и различных реализаций UNIX (глава 2).

2. Ввод/вывод: небуферизованный ввод/вывод (глава 3), характеристики файлов и каталогов (глава 4), стандартная библиотека ввода/вывода (глава 5) и стандартные системные файлы (глава 6).
3. Процессы: окружение процессов в UNIX (глава 7), управление процессами (глава 8), взаимоотношения между различными процессами (глава 9) и сигналы (глава 10).
4. Дополнительно об операциях ввода/вывода: терминальный ввод/вывод (глава 11), расширенные операции ввода/вывода (глава 12) и процессы-демоны (глава 13).
5. IPC – взаимодействия между процессами (главы 14 и 15).
6. Примеры: библиотека базы данных (глава 16), управление PostScript-принтером (глава 17), программа работы с модемом (глава 18) и использование псевдотерминалов (глава 19).

При чтении книги нелишним будет знание языка С, равно как и некоторый опыт использования UNIX. Наличие опыта разработки программ для UNIX не требуется. Книга предназначена для программистов, знакомых с UNIX или с другими операционными системами и желающих детально изучить возможности, предоставляемые большинством реализаций UNIX.

Примеры в книге

Данная книга содержит множество примеров — примерно 10 000 строк исходного кода. Все примеры написаны на языке С. Кроме того, примеры написаны в соответствии со стандартом ANSI C. Желательно, чтобы при чтении этой книги у вас под рукой был текст «*Unix Programmer's Manual*» (Руководство программиста UNIX) для вашей операционной системы, так как мы часто будем ссылаться на него при обсуждении малопонятных или зависящих от реализации особенностей.

Обсуждение практически каждой функции или системного вызова будет сопровождаться небольшой законченной программой. Это намного проще, чем рассматривать те же функции в больших программах, и позволит исследовать аргументы и возвращаемые значения. Так как некоторые из маленьких программ представляют собой достаточно искусственные примеры, в книгу включено несколько больших примеров (главы 16, 17, 18 и 19). Они демонстрируют решение практических задач.

Все примеры программ были включены в текст книги прямо из файлов с исходными текстами. Копии всех примеров в виде файлов вы найдете на анонимном FTP-сервере <ftp://ftp.uu.net>, в архиве *published/books/stevens.advprog.tar.Z*. Вы можете загрузить эти примеры и поэкспериментировать с ними в своей операционной системе.

Перечень систем, использовавшихся для тестирования примеров

К сожалению, все операционные системы находятся в постоянном движении. UNIX не является исключением. Следующая диаграмма показывает процесс эволюции различных версий System V и 4.xBSD.

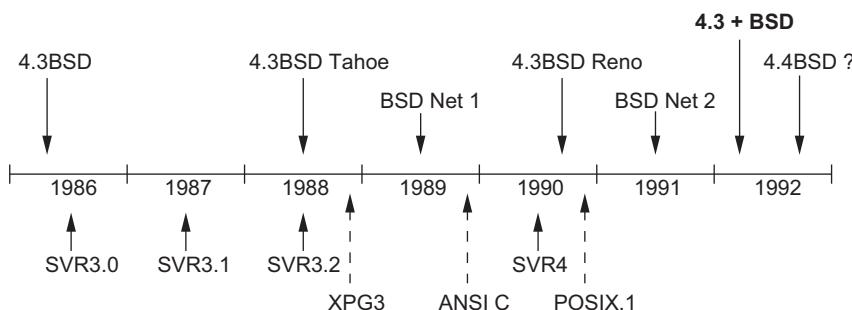


Рис. 1. Хронология эволюции различных версий System V

Под обозначением 4.xBSD подразумеваются различные операционные системы от Computer Systems Research Group из Калифорнийского университета в Беркли. Эта группа также занимается распространением BSD Net 1 и BSD Net 2 – операционных систем семейства 4.xBSD с открытым исходным кодом. Под обозначением SVRx подразумевается System V Release x от AT&T. XPG3 – это «X/Open Portability Guide, Issue 3» (Руководство X/Open по обеспечению переносимости, выпуск 3). ANSI C – это стандарт ANSI языка программирования С. POSIX.1 – это стандарт ISO и IEEE интерфейса UNIX-подобных операционных систем. Более подробно об этих стандартах и различных версиях UNIX мы поговорим в разделах 2.2 и 2.3.

В этой книге под обозначением 4.3+BSD будут подразумеваться версии UNIX, появившиеся между выпусками BSD Net 2 и 4.4BSD.

К моменту написания книги версия 4.4BSD еще не была выпущена, поэтому было бы преждевременно говорить об операционной системе 4.4BSD. Однако эту серию операционных систем нужно было как-то обозначить, поэтому повсюду в книге мы будем использовать обозначение 4.3+BSD.

Большинство примеров в книге протестированы в четырех версиях UNIX.

1. UNIX System V/386 Release 4.0 Version 2.0 («чистая SVR4») от U.H. Corp. (UHC), работающая на процессоре Intel 80386.
2. 4.3+BSD от Computer Systems Research Group, отделение информатики, Калифорнийский университет в Беркли, работающая на компьютере Hewlett Packard.

3. BSD/386 (производная от BSD Net 2) от Berkeley Software Design Inc., работающая на процессоре Intel 80386. Эта система практически полностью соответствует тому, что мы называем 4.3+BSD.
4. SunOS, версии 4.1.1 и 4.1.2 от Sun Microsystems (системы, в которых хорошо заметно наследие Беркли, но при этом много дополнений, пришедших из System V), работающие на SPARC-станциях SLC.

В книге представлены многочисленные тесты производительности с указанием проверяемых операционных систем.

Благодарности

Я многим обязан моей семье за любовь, поддержку и множество выходных дней, потерянных за последние полтора года. Создание книги — во многом заслуга семьи. Спасибо вам, Салли, Билл, Эллен и Дэвид.

Я особенно благодарен Брайану Кернигану за его помощь при работе над книгой. Его многочисленные рецензии рукописи и ненавязчивые рекомендации по улучшению стиля изложения, надеюсь, будут заметны в окончательном варианте. Стив Раго также отдал немало времени рецензированию рукописи и ответам на многие вопросы об устройстве и истории развития System V. Выражаю свою благодарность другим техническим рецензентам издательства Addison-Wesley, которые дали ценные комментарии по различным частям рукописи: Мори Баху (Maury Bach), Марку Эллису (Mark Ellis), Джифу Гитлину (Jeff Gitlin), Питеру Ханиману (Peter Honeymann), Джону Линдерману (John Linderman), Дугу Мак-Илрою (Doug McIlroy), Эви Немет (Evi Nemeth), Крейгу Парtridge (Craig Partridge), Дейву Пресотто (Dave Presotto), Гэри Уилсону (Gary Wilson) и Гэри Райту (Gary Wright).

Кейт Бостик (Keith Bostic) и Кирк Мак-Кьюсик (Kirk McKusick) из U.C. Berkeley CSRG предоставили учетную запись, которая использовалась для проверки примеров в последней версии BSD (также большое спасибо Питеру Салусу (Peter Salus)). Сэм Нэйтарос (Sam Nataros) и Йоахим Саксен (Joachim Sacksen) из UHC предоставили копию операционной системы SVR4 для тестирования примеров. Трент Хайн (Trent Hein) помог получить альфа- и бета-версии BSD/386.

Другие мои друзья на протяжении последних лет часто оказывали мне немаловажную помощь: Пол Лукина (Paul Lucchina), Джо Годсил (Joe Godsil), Джим Хог (Jim Hogue), Эд Танкус (Ed Tankus) и Гэри Райт (Gary Wright). Редактор из издательства Addison-Wesley, Джон Уэйт (John Wait), был моим большим другом на протяжении всего этого времени. Он никогда не жаловался на срыв сроков и постоянное увеличение числа страниц. Отдельное спасибо Национальной оптической астрономической обсерватории (NOAO) и особенно Сиднею Уольфу (Sidney Wolff), Ричарду Уольфу (Richard Wolff) и Стиву Гранди (Steve Grandi) за предоставленное машинное время.

Настоящие книги о UNIX пишутся в формате troff, и данная книга также следует этой проверенной временем традиции. Копия книги, готовая к тиражированию, была подготовлена автором с помощью пакета groff, созданного Джеймсом Кларком (James Clark). Большое ему спасибо за такой замечательный пакет и за его быстрый отклик на замеченные ошибки. Быть может, мне когда-нибудь удастся разобраться с концевыми сносками в troff.

Я буду благодарен всем читателям, которые пришлют свои комментарии, предложения и замечания об ошибках.

Таксон, Аризона

Ричард Стивенс

Апрель 1992

rsteven@kohala.com

<http://www.kohala.com/~rsteven>

1

Обзор ОС UNIX

1.1. Введение

Любая операционная система предоставляет работающим в ней программам разнообразные услуги: открытие и чтение файлов, запуск новых программ, выделение памяти, получение текущего времени и многое другое. В этой книге рассказывается об услугах, предоставляемых различными версиями операционной системы UNIX.

Строго линейное описание системы UNIX без опережающего использования терминов, которые фактически еще не были описаны, практически невозможно (и такое изложение, скорее всего, было бы скучным). Эта глава предлагает обзорную экскурсию по системе UNIX. Мы дадим краткие описания и примеры некоторых терминов и понятий, которые будут встречаться на протяжении всей книги. В последующих главах мы рассмотрим их более подробно. Эта глава также содержит обзор услуг, предоставляемых системой UNIX, для тех, кто мало знаком с ней.

1.2. Архитектура UNIX

Строго говоря, операционная система определяется как программное обеспечение, управляющее аппаратными ресурсами компьютера и предоставляющее среду для выполнения прикладных программ. Обычно это программное обеспечение называют *ядром* (kernel), так как оно имеет относительно небольшой объем и составляет основу системы. На рис. 1.1 изображена схема, отражающая архитектуру системы UNIX.

Интерфейс ядра — это слой программного обеспечения, называемый *системными вызовами* (заштрихованная область на рис. 1.1). Библиотеки функций общего пользования основываются на интерфейсе системных вызовов, но прикладная программа может свободно пользоваться как теми, так и другими (более подробно о библиотечных функциях и системных вызовах мы поговорим в разделе 1.11). Командная оболочка (shell) — это особое приложение, которое предоставляет интерфейс для запуска других приложений.

В более широком смысле операционная система — это ядро и все остальное программное обеспечение, которое делает компьютер пригодным к использованию и обеспечивает его индивидуальность. В состав этого программного обеспечения



Рис. 1.1. Архитектура системы UNIX

входят системные утилиты, прикладные программы, командные оболочки, библиотеки функций общего пользования и т. п.

Например, Linux – это ядро операционной системы GNU. Некоторые так и называют эту операционную систему – GNU/Linux, но чаще ее именуют просто Linux. Хотя, строго говоря, такое наименование не является правильным, но оно вполне понятно, если учесть двоякий смысл выражения *операционная система*. (И кроме того, оно обладает таким преимуществом, как краткость.)

1.3. Вход в систему

Имя пользователя

При входе в систему UNIX мы вводим имя пользователя и пароль. После этого система отыскивает введенное имя в файле паролей; обычно это файл /etc/passwd. Файл паролей содержит записи, состоящие из семи полей, разделенных двоеточием: имя пользователя, зашифрованный пароль, числовой идентификатор пользователя (205), числовой идентификатор группы (105), поле комментария, домашний каталог (/home/sar) и командный интерпретатор (/bin/ksh).

```
sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
```

Все современные системы хранят пароли в отдельном файле. В главе 6 мы рассмотрим эти файлы и некоторые функции доступа к ним.

Командные оболочки

Обычно после входа в систему на экран выводится некоторая системная информация, после чего можно вводить команды, предназначенные для командной оболочки. (В некоторых системах после ввода имени пользователя и пароля запускается графический интерфейс, но и в этом случае, как правило, можно получить доступ к командной оболочке, запустив командный интерпретатор в одном из окон.) *Командная оболочка* — это интерпретатор командной строки, который читает ввод пользователя и выполняет команды. Ввод пользователя обычно считывается из терминала (интерактивная командная оболочка) или из файла (который называется *сценарием командной оболочки*). В табл. 1.1 перечислены наиболее распространенные командные оболочки.

Таблица 1.1. Наиболее распространенные командные оболочки, используемые в UNIX

Название	Путь	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	/bin/sh	✓	✓	копия bash	✓
Bourne-again shell	/bin/bash	необязательно	✓	✓	✓
C shell	/bin/csh	ссылка на tcsh	необязательно	ссылка на tcsh	✓
Korn shell	/bin/ksh	необязательно	необязательно		✓
TENEX C shell	/bin/tcsh	✓	необязательно	✓	✓

Информацию о том, какой командный интерпретатор следует запустить, система извлекает из записи в файле паролей.

Командный интерпретатор Bourne shell был разработан в Bell Labs Стивом Борном (Steve Bourne). Он входит в состав практически всех существующих версий UNIX, начиная с Version 7. Управляющие конструкции в Bourne shell чем-то напоминают язык программирования Algol 68.

Командный интерпретатор C shell был разработан в Беркли Биллом Джоем (Bill Joy) и входит в состав всех версий BSD. Кроме того, C shell включен в состав System V/386 Release 3.2 от AT&T, а также в System V Release 4 (SVR4). (В следующей главе мы расскажем об этих версиях UNIX подробнее.) Командная оболочка C shell разработана на основе оболочки 6-й редакции UNIX, а не Bourne shell. Управляющие конструкции этого интерпретатора напоминают язык программирования C, и кроме того, он поддерживает дополнительные особенности, отсутствующие в Bourne shell: управление заданиями, историю команд и возможность редактирования командной строки.

Командный интерпретатор Korn shell можно считать наследником Bourne shell. Он впервые появился в составе SVR4. Разработанный в Bell Labs Дэвидом Корном (David Korn), он может работать в большинстве версий UNIX, но до выхода SVR4 обычно распространялся как дополнение за отдельную плату и поэтому не получил такого широкого распространения, как предыдущие два интерпретатора.

Сохраняет обратную совместимость с Bourne shell и предоставляет те же возможности, благодаря которым C shell стал таким популярным: управление заданиями, возможность редактирования командной строки и пр.

Bourne-again shell – командный интерпретатор GNU, входящий в состав всех версий ОС Linux. Был разработан в соответствии со стандартом POSIX и в то же время сохраняет совместимость с Bourne shell, а также поддерживает особенности, присущие C shell и Korn shell.

Командный интерпретатор TENEX C shell – это расширенная версия C shell. Заимствовал некоторые особенности, такие как автодополнение команд, из ОС TENEX, разработанной в 1972 году в компании Bolt Beranek and Newman. TENEX C shell расширяет возможности C shell и часто используется в качестве его замены.

Стандарт POSIX 1003.2 определяет перечень возможностей, которыми должен обладать командный интерпретатор. Эта спецификация была основана на особенностях Korn shell и Bourne shell.

В разных дистрибутивах Linux по умолчанию используются разные командные интерпретаторы. В одних по умолчанию используется командный интерпретатор Bourne-again, в других – BSD-версия Bourne shell – dash (Debian Almquist shell, первоначально написан Кеннетом Альмквистом (Kenneth Almquist) и затем перенесен в Linux). В системах FreeBSD по умолчанию используется производная от Almquist shell. В Mac OS X – Bourne-again shell. Операционная система Solaris унаследовала от BSD и System V все командные интерпретаторы, перечисленные в табл. 1.1. Свободно распространяемые версии большинства командных интерпретаторов можно найти в Интернете.

На протяжении всей книги в подобных примечаниях мы будем приводить исторические заметки и сравнения различных реализаций UNIX. Зачастую обоснования отдельных методов реализации становятся гораздо понятнее в историческом контексте.

В тексте книги приводятся многочисленные примеры взаимодействия с командным интерпретатором, демонстрирующие запуск разрабатываемых нами программ. В этих примерах используются возможности, общие для Bourne shell, Korn shell и Bourne-again shell.

1.4. Файлы и каталоги

Файловая система

Файловая система UNIX имеет иерархическую древовидную структуру, состоящую из каталогов и файлов. Начинается она с корневого каталога, имеющего имя из единственного символа /.

Каталог – это файл, содержащий каталожные записи. Логически каждую такую запись можно представить в виде структуры, состоящей из имени файла и дополнительной информации, описывающей атрибуты файла. К атрибутам относятся такие характеристики, как тип файла (обычный файл или каталог), размер, владелец, разрешения (доступность файла для других пользователей), время последнего изменения. Функции `stat` и `fstat` возвращают структуру с информацией обо всех атрибутах файла. В главе 4 мы исследуем атрибуты файла более детально.

Мы различаем логическое представление каталожной записи и фактический способ хранения этой информации на диске. Большинство реализаций файловых систем для UNIX не хранят атрибуты в каталожных записях из-за сложностей, связанных с их синхронизацией при наличии нескольких жестких ссылок на файл. Это станет понятным, когда мы будем обсуждать жесткие ссылки в главе 4.

Имя файла

Имена элементов каталога называются *именами файлов* (filenames). Только два символа не могут встречаться в именах файлов — прямой слеш (/) и нулевой символ (\0). Символ слеша разделяет компоненты, образующие строку пути к файлу (описывается ниже), а нулевой символ обозначает конец этой строки. Однако на практике, выбирая имена для файлов, лучше ограничиться подмножеством обычных печатных символов. (Мы ограничиваем набор допустимых символов, потому что некоторые специальные символы имеют особое значение для командного интерпретатора и при их использовании в именах файлов нам пришлось бы применять экранирование, а это влечет дополнительные сложности.) В действительности для совместимости стандарт POSIX.1 рекомендует использовать в именах файлов только следующие символы: буквы (a-z, A-Z), цифры (0-9), точку (.), дефис (-) и подчеркивание (_).

Всякий раз, когда создается новый каталог, в нем автоматически создаются две записи: . (точка) и .. (точка-точка). Запись «точка» соответствует текущий каталог, а записи «точка-точка» — родительский. В корневом каталоге запись «точка-точка» представляет тот же каталог, что и «точка».

В Research UNIX System и некоторых устаревших версиях UNIX System V длина имени файла ограничена 14 символами. В версиях BSD этот предел увеличен до 255 символов. Сегодня практически все файловые системы коммерческих версий UNIX поддерживают имена файлов с длиной не менее 255 символов.

Путь к файлу

Последовательность из одного или более имен файлов, разделенных слешем, образует строку *пути к файлу*. Эта строка может также начинаться с символа слеша, и тогда она называется строкой *абсолютного пути*, иначе — строкой *относительного пути*. Относительные пути начинаются от текущего каталога. Имя корня файловой системы (/) — особый случай строки абсолютного пути, которая не содержит ни одного имени файла.

Пример

Вывести список всех файлов в каталоге достаточно просто. Вот пример упрощенной реализации команды ls(1).

Листинг 1.1. Вывод списка всех файлов в каталоге

```
#include "apue.h"
#include <dirent.h>

int
```

```

main(int argc, char *argv[])
{
    DIR            *dp;
    struct dirent  *dirp;

    if (argc != 2)
        err_quit("Использование: ls имя_каталога");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("невозможно открыть %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}

```

Нотация `ls(1)` — обычный способ ссылки на определенную страницу справочного руководства UNIX. В данном случае она ссылается на страницу с описанием команды `ls` в первом разделе. Разделы справочного руководства обычно нумеруются цифрами от 1 до 8, а страницы в каждом разделе отсортированы по алфавиту. Здесь и далее мы будем исходить из предположения, что у вас под рукой имеется текст справочного руководства по вашей версии UNIX.

*Исторически сложилось так, что все восемь разделов были объединены в документ, который называется «*UNIX Programmer's Manual*» (Руководство программиста UNIX). Но поскольку количество страниц в руководстве постоянно растет, появилась тенденция к распределению разделов по отдельным руководствам: например, одно для пользователей, одно для программистов и одно для системных администраторов.*

В некоторых версиях UNIX разделы справочного руководства делятся на подразделы, обозначенные заглавными буквами. Так, описания всех стандартных функций ввода/вывода в AT&T [1990e] находятся в разделе 3S, например `fopen(3S)`. В некоторых системах в обозначениях разделов цифры заменены алфавитными символами, например «C» — для раздела с описаниями команд.

В наши дни большинство руководств распространяется в электронном виде. Если вы располагаете такой версией справочного руководства, просмотреть справку по команде `ls` можно так:

```
man 1 ls
```

или

```
man -sl ls
```

Программа в листинге 1.1 просто выводит список файлов в указанном каталоге и ничего больше. Назовем файл с исходным кодом `myls.c` и скомпилируем его в исполняемый файл с именем по умолчанию `a.out`:

```
cc myls.c
```

Исторически команда cc(1) запускает компилятор языка C. В системах, где используется компилятор GNU C, выполняемый файл компилятора носит имя gcc(1). В этих системах команда cc часто является ссылкой на gcc.

Примерный результат работы нашей программы:

```
$ ./a.out /dev
```

```
.
```

```
..
```

```
cdrom
```

```
stderr
```

```
stdout
```

```
stdin
```

```
fd
```

```
sda4
```

```
sda3
```

```
sda2
```

```
sda1
```

```
sda
```

```
tty2
```

```
tty1
```

```
console
```

```
tty
```

```
zero
```

```
null
```

и еще много строк...

```
mem
```

```
$ ./a.out /etc/ssl/private
```

невозможно открыть /etc/ssl/private: Permission denied

```
$ ./a.out /dev/tty
```

невозможно открыть /dev/tty: Not a directory

Далее в книге запуск программ и результаты их работы демонстрируются именно так: символы, вводимые с клавиатуры, выделяются **жирным монотипом**, а результат выполнения — **обычным монотипом**. Комментарии среди строк, выводимых в терминал, будут оформляться **курсивом**. Знак доллара, предшествующий вводимым с клавиатуры символам, — это приглашение командного интерпретатора. Мы всегда будем отображать приглашение в виде символа доллара.

Обратите внимание, что полученный список файлов не отсортирован по алфавиту. Команда `ls` сортирует имена файлов перед выводом.

Рассмотрим детальнее эту программу из 20 строк.

- В первой строке подключается наш собственный заголовочный файл `arie.h`. Мы будем использовать его почти во всех программах в этой книге. Этот заголовочный файл, в свою очередь, подключает некоторые стандартные заголовочные файлы и определяет множество констант и прототипов функций, которые будут встречаться в наших примерах. Листинг этого файла вы найдете в приложении B.
- Далее подключается системный заголовочный файл `dirent.h`, чтобы добавить объявления прототипов функций `opendir` и `readdir`, а также определение структуры `dirent`. В некоторых системах определения разбиты на несколько заголовочных файлов. Например, в дистрибутиве Linux Ubuntu 12.04 файл `/usr/include/dirent.h` объявляет прототипы функций и подключает файл `bits/dirent.h` с определением структуры `dirent` (и фактически хранится в каталоге `/usr/include/x86_64-linux-gnu/bits`).

- Функция `main` объявлена в соответствии со стандартом ISO C. (Более подробно об этом стандарте рассказывается в следующей главе.)
- В ней мы принимаем аргумент командной строки `argv[1]` — имя каталога, для которого требуется получить список файлов. В главе 7 мы увидим, как вызывается функция `main` и как программа получает доступ к аргументам командной строки и переменным окружения.
- Поскольку на практике в разных системах используется разный формат записей в каталогах, для получения информации мы использовали стандартные функции `opendir`, `readdir` и `closedir`.
- Функция `opendir` возвращает указатель на структуру `DIR`, который затем передается функции `readdir`. (Нас пока не интересует содержимое структуры `DIR`.) Затем в цикле вызывается функция `readdir`, которая читает очередную запись и возвращает указатель на структуру `dirent` или пустой указатель, если все записи были прочитаны. Все, что нам сейчас нужно в структуре `dirent`, — это имя файла (`d_name`). Это имя можно передать функции `stat` (раздел 4.2), чтобы определить атрибуты файла.
- Для обработки ошибочных ситуаций вызываются наши собственные функции: `err_sys` и `err_quit`. Функция `err_sys` в предыдущем примере выводит сообщение, описывающее возникшую ошибку («Permission denied» — «Доступ запрещен» или «Not a directory» — «Не является каталогом»). Исходный код этих функций и их описание приводятся в приложении B. В разделе 1.7 мы еще поговорим об обработке ошибок.
- В конце программы вызывается функция `exit` с аргументом 0. Она завершает выполнение программы. В соответствии с соглашениями 0 означает нормальное завершение программы, а значения в диапазоне от 1 до 255 свидетельствуют о наличии ошибки. В разделе 8.5 мы покажем, как любая программа, в том числе и наша, может получить код завершения другой программы.

Рабочий каталог

У каждого процесса имеется свой *рабочий каталог*, который иногда называют *текущим рабочим каталогом*. Это каталог, от которого откладыватся все относительные пути, используемые в программе. Процесс может изменить свой рабочий каталог вызовом функции `chdir`.

Например, относительный путь к файлу `doc/memo/joe` означает, что файл или каталог `joe` находится в каталоге `memo`, который находится в каталоге `doc`, который в свою очередь должен находиться в рабочем каталоге. Встретив такой путь, мы можем точно сказать, что `doc` и `memo` — это каталоги, но не можем утверждать, является `joe` каталогом или файлом. Путь `/usr/lib/lint` — это абсолютный путь к файлу или каталогу `lint` в каталоге `lib`, находящемся в каталоге `usr`, который в свою очередь находится в корневом каталоге.

Домашний каталог

Когда пользователь входит в систему, рабочим каталогом становится его *домашний каталог*. Домашний каталог пользователя определяется в соответствии с записью в файле паролей (раздел 1.3).

1.5. Ввод и вывод

Дескрипторы файлов

Дескрипторы файлов — это, как правило, небольшие целые положительные числа, используемые ядром для идентификации файлов, к которым обращается конкретный процесс. Всякий раз, когда процесс открывает существующий или создает новый файл, ядро возвращает его дескриптор, который затем используется для выполнения операций чтения/записи с файлом.

Стандартный ввод, стандартный вывод, стандартный вывод ошибок

По соглашению, все командные оболочки при запуске новой программы открывают для нее три файловых дескриптора: стандартного ввода, стандартного вывода и стандартного вывода ошибок. За исключением особых случаев, все три дескриптора по умолчанию связаны с терминалом, как в простой команде

```
ls
```

Большинство командных оболочек дает возможность перенаправить любой из этих дескрипторов в любой файл. Например

```
ls > file.list
```

выполнит команду `ls` и перенаправит стандартный вывод в файл с именем `file.list`.

Небуферизованный ввод/вывод

Небуферизованный ввод/вывод осуществляется функциями `open`, `read`, `write`, `lseek` и `close`. Все эти функции работают с дескрипторами файлов.

Пример

В листинге 1.2 приводится пример программы, которая читает данные из стандартного ввода и копирует их в стандартный вывод. С ее помощью можно выполнять копирование любых обычных файлов.

Листинг 1.2. Копирование со стандартного ввода на стандартный вывод

```
#include "apue.h"
#define BUFFSIZE 4096

int
main(void)
{
    int    n;
    char   buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка записи");

    if (n < 0)
```

```
    err_sys("ошибка чтения");
    exit(0);
}
```

Заголовочный файл `<unistd.h>`, подключаемый в `apue.h`, и константы `STDIN_FILENO` и `STDOUT_FILENO` являются частью стандарта POSIX (о котором мы будем много говорить в следующей главе). В этом файле объявлены прототипы функций для доступа к системным службам UNIX, в том числе к функциям `read` и `write`, используемым в этом примере.

Константы `STDIN_FILENO` и `STDOUT_FILENO`, объявленные в файле `<unistd.h>`, соответствуют дескрипторам файлов стандартного ввода и стандартного вывода. Обычные значения этих констант — соответственно 0 и 1, как определено стандартом POSIX.1, но мы будем пользоваться именованными константами для большей удобочитаемости.

Константу `BUFFSIZE` мы детально исследуем в разделе 3.9, где увидим, как различные значения могут влиять на производительность программы. Однако независимо от значения этой константы наша программа будет в состоянии выполнять копирование файла.

Функция `read` возвращает количество прочитанных байтов. Это число затем передается функции `write`, чтобы сообщить ей, сколько байтов нужно записать. Достигнув конца файла, функция `read` вернет 0 и программа завершится. Если в процессе чтения возникнет ошибка, `read` вернет -1. Большинство системных функций в случае ошибки возвращают -1.

Если скомпилировать программу в выполняемый файл с именем по умолчанию (`a.out`) и запустить ее:

```
./a.out > data
```

стандартный вывод будет перенаправлен в файл `data`, а стандартным вводом и стандартным выводом сообщений об ошибках будет терминал. Если выходной файл не существует, командная оболочка создаст его. Программа будет копировать строки, вводимые с клавиатуры, в стандартный вывод до тех пор, пока мы не введем символ «конец-файла» (обычно `Control-D`).

Если запустить программу так:

```
./a.out < infile > outfile
```

она скопирует файл `infile` в файл `outfile`.

Более подробно функции небуферизованного ввода/вывода будут описаны в главе 3.

Стандартные функции ввода/вывода

Стандартные функции ввода/вывода предоставляют буферизованный интерфейс к функциям небуферизованного ввода/вывода. Использование стандартных функций ввода/вывода избавляет от необходимости задумываться о выборе оптимального размера буфера, например о значении константы `BUFFSIZE` в листинге 1.2. Другое преимущество стандартных функций ввода/вывода — они значительно упрощают

обработку пользовательского ввода (что на каждом шагу встречается в прикладных программах UNIX). Например, функция `fgets` читает из файла строку целиком, в то время как функция `read` считывает указанное количество байтов. Как будет показано в разделе 5.4, стандартная библиотека ввода/вывода включает функции, с помощью которых можно управлять типом буферизации.

Самой известной стандартной функцией ввода/вывода является `printf`. В программах, вызывающих ее, обязательно должен быть подключен заголовочный файл `<stdio.h>` (мы всегда делаем это через подключение файла `apue.h`), определяющий прототипы всех стандартных функций ввода/вывода.

Пример

Программа в листинге 1.3 (более детально исследуется в разделе 5.8) подобна программе из предыдущего примера, использующей функции `read` и `write`. Она также копирует данные, полученные со стандартного ввода, в стандартный вывод и может выполнять копирование обычных файлов.

Листинг 1.3. Копирование стандартного ввода в стандартный вывод с использованием стандартных функций ввода/вывода

```
#include "apue.h"
int
main(void)
{
    int      c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Функция `getc` читает один символ, который затем записывается вызовом функции `putc`. Прочитав последний байт, `getc` вернет признак конца файла — константу `EOF` (определену в `<stdio.h>`). Константы `stdin` и `stdout` также определены в `<stdio.h>` и обозначают стандартный ввод и стандартный вывод.

1.6. Программы и процессы

Программа

Программа — это выполняемый файл, хранящийся на диске. Программа загружается в память и передается ядру для выполнения вызовом одной из шести функций семейства `exec`. Мы рассмотрим эти функции в разделе 8.10.

Процессы и идентификаторы процессов

Выполняющая программу называется *процессом*. Этот термин будет встречаться практически на каждой странице этой книги. В некоторых операционных системах

макс для обозначения выполняемой в данный момент программы используется термин *задача*.

UNIX присваивает каждому процессу уникальный числовой идентификатор, который называется *идентификатором процесса* (*process ID*, или *PID*). Идентификатор процесса — всегда целое неотрицательное число.

Пример

Программа в листинге 1.4 выводит собственный идентификатор процесса.

Листинг 1.4. Вывод идентификатора процесса

```
#include "apue.h"
int
main(void)
{
    printf("привет от процесса с идентификатором %d\n", (long)getpid());
    exit(0);
}
```

Если скомпилировать эту программу в файл `a.out` и запустить, она выведет примерно следующее:

```
$ ./a.out
привет от процесса с идентификатором 851
$ ./a.out
привет от процесса с идентификатором 854
```

После запуска эта программа вызовет `getpid`, чтобы получить идентификатор своего процесса. Как будет показано ниже, `getpid` возвращает значение типа `pid_t`. Его размер нам неизвестен, зато известно, что в соответствии с требованиями стандартов оно должно умещаться в длинное целое. Поскольку мы должны сообщить функции `printf` размер каждого аргумента, мы выполняем приведение типа фактического значения к более широкому типу (в данном случае к длинному целому). Хотя в большинстве систем значение идентификатора процесса укладывается в тип `int`, использование типа `long` обеспечивает дополнительную переносимость.

Управление процессами

За управление процессами отвечают три основные функции: `fork`, `exec` и `waitpid`. (Функция `exec` имеет шесть разновидностей, но мы часто будем ссылаться на них просто как на функцию `exec`.)

Пример

Особенности управления процессами в UNIX демонстрирует простая программа (листинг 1.5), которая читает команды со стандартного ввода и выполняет их. Это похоже на примитивную реализацию командной оболочки.

Листинг 1.5. Чтение команд со стандартного ввода и их выполнение

```
#include "apue.h"
#include <sys/wait.h>

int
```

```

main(void)
{
    char    buf[MAXLINE]; /* из apue.h */
    pid_t   pid;
    int     status;

    printf("%"); /* вывести приглашение (printf требует использовать */
               /* последовательность %%, чтобы вывести символ %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* заменить символ перевода строки */
        if ((pid = fork()) < 0)
            err_sys("ошибка вызова fork");
        } else if (pid == 0) { /* дочерний процесс */
            execvp(buf, buf, (char *)0);
            err_ret("невозможно выполнить: %s", buf);
            exit(127);
        }

        /* родительский процесс */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("ошибка вызова waitpid");
        printf("%");
    }
    exit(0);
}

```

Перечислим наиболее важные аспекты в этой 30-строчной программе.

- Для чтения строки из стандартного ввода используется функция `fgets`. Когда первым в строке вводится символ конца файла (обычно Control-D), `fgets` возвращает пустой указатель, цикл прерывается и процесс завершает работу. В главе 18 мы опишем все символы, имеющие специальное значение — признак конца файла, забой, удаление строки и пр., и покажем, как можно замещать.
- Каждая строка, возвращаемая функцией `fgets`, завершается символом перевода строки, за которым следует нулевой символ (\0), поэтому мы определили ее длину с помощью стандартной функции `strlen` и заменили перевод строки нулевым символом. Это необходимо, потому что функция `execvp` ожидает получить строку, завершающуюся нулевым символом, а не символом перевода строки.
- Вызов функции `fork` создает новый процесс — копию вызывающего процесса. Мы называем вызывающий процесс родительским процессом, а вновь созданный — дочерним. В родительском процессе функция `fork` возвращает идентификатор дочернего процесса, в дочернем процессе — 0. Поскольку `fork` создает новый процесс, можно сказать, что она вызывается один раз — родительским процессом, а возвращает управление дважды — в родительском и в дочернем процессах.
- Для запуска команды, прочитанной из стандартного ввода, в дочернем процессе вызывается функция `execvp`. Она замещает дочерний процесс новой программой из файла. Комбинация функций `fork/exec` — это своего рода двухступенчатый системный вызов, порождающий новый процесс. В UNIX эти два

этапа выделены в самостоятельные функции. Более подробно мы поговорим о них в главе 8.

- Поскольку дочерний процесс запускает новую программу с помощью `exec1p`, родительский процесс должен дождаться его завершения, прежде чем продолжить работу. Делается это с помощью вызова функции `waitpid`, которой передается идентификатор дочернего процесса — аргумент `pid`. Функция `waitpid` возвращает код завершения дочернего процесса (переменная `status`), но в нашей программе это значение не используется. Мы могли бы проверить его, чтобы узнать, как завершился дочерний процесс.
- Одно из основных ограничений этой программы заключается в невозможности передать аргументы выполняемой команде. Так, например, вы не сможете указать имя каталога, чтобы получить список файлов, хранящихся в нем. Мы можем выполнить команду `ls` только для рабочего каталога. Чтобы передать аргументы, необходимо проанализировать введенную строку, выделить аргументы в соответствии с некоторыми признаками (например, по символам пробела или табуляции) и затем передать их в виде отдельных аргументов функции `exec1p`. Тем не менее наша программа наглядно демонстрирует, как работают функции управления процессами.

Ниже показан вывод программы, полученный в нашей системе. Обратите внимание, что она выводит символ `%` в качестве приглашения, чтобы как-то отличить его от приглашения командной оболочки.

```
$ ./a.out
% date
Sat Jan 21 19:42:07 EST 2012
% who
sar      console Jan  1 14:59
sar      ttys000 Jan  1 14:59
sar      ttys001 Jan 15 15:28
% pwd
/home/sar/bk/apue/3e
% ls
Makefile
a.out
shell1.c
% ^D      Ввод символа конца файла
$         приглашение командной оболочки
```

Последовательность `^D` указывает на ввод управляющего символа. Управляющие символы — это специальные символы, которые формируются при нажатой и удерживаемой клавише `Control` или `Ctrl` (в зависимости от модели клавиатуры) и одновременном нажатии на другую клавишу. Символ `Control-D`, или `^D`, представляет признак конца файла. Мы встретим еще много управляющих символов, когда будем обсуждать терминальный ввод/вывод в главе 18.

Потоки выполнения и идентификаторы потоков

Обычно процесс имеет единственный поток выполнения — только одна последовательность машинных инструкций выполняется в одно и то же время. Со многими проблемами легче справиться, если решать различные части задачи одновременно

в нескольких потоках. Кроме того, в многопроцессорных системах различные потоки одного процесса могут выполняться параллельно.

Все потоки в процессе разделяют общее адресное пространство, файловые дескрипторы, стеки и прочие атрибуты процесса. Поскольку потоки могут обращаться к одной и той же области памяти, они должны синхронизировать доступ к разделяемым данным, чтобы избежать несогласованности.

Подобно процессам, каждый поток имеет свой числовой идентификатор. Однако идентификаторы потоков являются локальными для процесса. Они не имеют никакого значения для других процессов и служат для ссылки на конкретные потоки внутри процесса, когда требуется оказать управляющее воздействие.

Функции управления потоками отличны от функций управления процессами. Однако поскольку потоки были добавлены в UNIX намного позже появления модели процессов, эти две модели находятся в достаточно сложной взаимосвязи, как будет показано в главе 12.

1.7. Обработка ошибок

Часто при появлении ошибки функции системы UNIX возвращают отрицательное число, а в глобальную переменную `errno` записывают некоторое целое число, несущее дополнительную информацию об ошибке. Например, функция `open` возвращает файловый дескриптор — неотрицательное число — или `-1` в случае ошибки. Вообще через переменную `errno` функция `open` может вернуть 15 различных кодов ошибок, таких как отсутствие файла, недостаточность прав доступа и т. п. Некоторые функции следуют иному соглашению. Например, большинство функций, которые должны возвращать указатель на какой-либо объект, в случае ошибки возвращают пустой указатель.

Определения переменной `errno` и констант всех возможных кодов ошибок находятся в заголовочном файле `<errno.h>`. Имена констант начинаются с символа `E`. Кроме того, на первой странице второго раздела справочного руководства UNIX, которая называется `intro(2)`, обычно перечислены все константы кодов ошибок. Например, если переменная `errno` содержит код, равный значению константы `EACCES`, это означает, что возникли проблемы с правами доступа, например, при открытии файла.

В ОС Linux коды ошибок и соответствующие им имена констант перечислены на странице `errno(3)`.

Стандарты POSIX и ISO C определяют `errno` как символ, разворачивающийся в изменяемое левостороннее выражение (то есть выражение, которое может стоять слева от оператора присваивания) целого типа. Это может быть целое число, соответствующее коду ошибки, или функция, возвращающая указатель на код ошибки. Изначально переменная `errno` определялась как

```
extern int errno;
```

Но в многопоточной среде адресное пространство процесса совместно используется несколькими потоками и каждый поток должен обладать своей локальной

копией `errno`, чтобы исключить конфликты. ОС Linux, например, поддерживает многопоточный доступ к переменной `errno`, определяя ее так:

```
extern int *_errno_location(void);
#define errno (*_errno_location())
```

Необходимо знать два правила, касающиеся `errno`. Во-первых, значение `errno` никогда не очищается процедурой, если ошибка не происходит. Следовательно, проверять это значение надо лишь в тех случаях, когда значение, возвращаемое функцией, указывает, что произошла ошибка. Во-вторых, ни одна функция никогда не устанавливает значение `errno` в 0 и ни одна константа из определяемых в `<errno.h>` не имеет значения 0.

Для вывода сообщений об ошибках стандарт С предусматривает две функции.

```
#include <string.h>
char *strerror(int errnum);
```

Возвращает указатель на строку сообщения

Эта функция преобразует код ошибки `errnum`, обычно равный значению `errno`, в строку сообщения об ошибке и возвращает указатель на нее.

Функция `perror`, основываясь на значении `errno`, выводит сообщение об ошибке в стандартный вывод ошибок.

```
#include <stdio.h>
void perror(const char *msg);
```

Она выводит строку, на которую ссылается аргумент `msg`, двоеточие, пробел и текст сообщения об ошибке, соответствующий значению `errno`. Вывод заканчивается символом перевода строки.

Пример

В листинге 1.6 приводится пример использования этих функций.

Листинг 1.6. Демонстрация функций strerror и perror

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Скомпилировав и запустив эту программу, мы получим

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

Обратите внимание: мы передали функции `perror` имя выполняемого файла программы — `a.out`, находящееся в `argv[0]`. Это стандартное соглашение, принятое в UNIX. Если программа выполняется в составе конвейера, как показано ниже,

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

следуя этому соглашению, мы сможем точно определить, в какой из программ произошла ошибка.

Во всех примерах в этой книге вместо `strerror` или `perror` мы будем использовать собственные функции вывода сообщений об ошибках, которые можно найти в приложении B. Они принимают переменное количество аргументов, что позволяет легко обрабатывать ошибочные ситуации единственным выражением на языке C.

Восстановление после ошибок

Ошибки, определенные в `<errno.h>`, можно разделить на две категории: фатальные и нефатальные. Восстановление нормальной работы после фатальных ошибок невозможно. Самое большее, что можно сделать, — вывести сообщение об ошибке на экран или в файл журнала и завершить приложение. Нефатальные ошибки допускают нормальное продолжение работы. Большинство нефатальных ошибок по своей природе носит временный характер (например, нехватка ресурсов), и их можно избежать при меньшей загруженности системы.

К нефатальным ошибкам, связанным с нехваткой ресурсов, относятся: `EAGAIN`, `ENFILE`, `ENOBUFS`, `ENOLCK`, `ENOSPC`, `ENOSR`, `EWOULDBLOCK` и иногда `ENOMEM`. Ошибку `EBUSY` можно считать нефатальной, если она сообщает о занятости общего ресурса в настоящий момент времени. Иногда нефатальной может считаться ошибка `EINTR`, если она возникает в результате прерывания медленно работающего системного вызова (подробнее об этом рассказывается в разделе 10.5).

Для восстановления после вышеперечисленных ошибок обычно достаточно приостановить работу на короткое время и повторить попытку. Этот прием можно использовать в других ситуациях. Например, если ошибка свидетельствует о разрыве сетевого соединения, можно подождать некоторое время и затем попытаться восстановить соединение. В некоторых приложениях используется алгоритм экспоненциального увеличения времени задержки, когда пауза увеличивается с каждой следующей попыткой.

В конечном счете сам разработчик приложения решает, после каких ошибок возможно продолжение работы. Применяя разумную стратегию восстановления после ошибок, можно существенно повысить отказоустойчивость приложения и избежать аварийного завершения его работы.

1.8. Идентификация пользователя

Идентификатор пользователя

Идентификатор пользователя (*user ID*, или *UID*) из записи в файле паролей — это числовое значение, которое однозначно идентифицирует пользователя в системе. Идентификатор пользователя назначается системным администратором при соз-

дании учетной записи и не может изменяться пользователем. Как правило, каждому пользователю назначается уникальный идентификатор. Ниже мы узнаем, как ядро использует идентификатор пользователя для проверки прав на выполнение определенных операций.

Пользователь с идентификатором 0 называется *суперпользователем*, или *root*. В файле паролей этому пользователю обычно присвоено имя *root*. Этот пользователь обладает особыми суперпривилегиями. Как показано в главе 4, если процесс имеет привилегии суперпользователя, большинство проверок прав доступа к файлам просто не выполняется. Некоторые системные операции доступны только суперпользователю. Суперпользователь обладает неограниченной свободой действий в системе.

В клиентских версиях Mac OS X учетная запись суперпользователя заблокирована, в серверных версиях — разблокирована. Инструкции по разблокированию учетной записи суперпользователя можно найти на веб-сайте компании Apple: <http://support.apple.com/kb/HT1528>.

Идентификатор группы

Кроме всего прочего, запись в файле паролей содержит числовой *идентификатор группы* (*group ID*, или *GID*). Он также назначается системным администратором при создании учетной записи. Как правило, в файле паролей имеется несколько записей с одинаковым идентификатором группы. Обычно группы используются для распределения пользователей по проектам или отделам. Это позволяет организовать совместное использование ресурсов, например файлов, членами определенной группы. В разделе 4.5 показано, как назначить файлу такие права доступа, чтобы он был доступен всем членам группы и недоступен другим пользователям.

В системе существует файл групп, определяющий соответствия имен групп их числовым идентификаторам. Обычно этот файл называется */etc/group*.

Представление идентификаторов пользователя и группы в числовом виде сложилось исторически. Для каждого файла на диске файловая система хранит идентификаторы пользователя и группы его владельца. Поскольку каждый идентификатор представлен двухбайтным целым числом, для хранения обоих идентификаторов требуется всего четыре байта. Если бы вместо идентификаторов использовались полные имена пользователей и групп, потребовалось бы хранить на диске значительно больший объем информации. Кроме того, сравнение строк вместо сравнения целых чисел при проверках прав доступа выполнялось бы гораздо медленнее.

Однако человеку удобнее работать с осмысленными именами, чем с числовыми идентификаторами, поэтому файл паролей хранит соответствия между именами и идентификаторами пользователей, а файл групп — между именами и идентификаторами групп. Например, команда *ls -l* выведет имена владельцев файлов, используя файл паролей для преобразования числовых идентификаторов в соответствующие им имена пользователей.

В ранних версиях UNIX для представления идентификаторов использовались 16-разрядные целые числа, в современных версиях — 32-разрядные.

Пример

Программа в листинге 1.7 выводит идентификаторы пользователя и группы.

Листинг 1.7. Вывод идентификаторов пользователя и группы

```
#include "apue.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Для получения идентификаторов пользователя и группы используются функции `getuid` и `getgid`. Запуск программы дает следующие результаты:

```
$ ./a.out
uid = 205, gid = 105
```

Идентификаторы дополнительных групп

В дополнение к группе, идентификатор которой указан в файле паролей, большинство версий UNIX позволяют пользователю быть членом других групп. Впервые такая возможность появилась в 4.2BSD, где можно было определить до 16 дополнительных групп, к которым мог принадлежать пользователь. Во время входа в систему из файла `/etc/group` извлекаются первые 16 групп, в которых присутствует имя данного пользователя, и их идентификаторы назначаются идентификаторами *дополнительных групп*. Как показано в следующей главе, стандарт POSIX требует, чтобы операционная система поддерживала не менее 8 дополнительных групп для каждого процесса, однако большинство систем поддерживает не менее 16 таких групп.

1.9. Сигналы

Сигналы используются, чтобы известить процесс о наступлении некоторого состояния. Например, если процесс попытается выполнить деление на ноль, он получит уведомление в виде сигнала `SIGFPE` (floating-point exception — ошибка выполнения операции с плавающей запятой). Процесс может реагировать на сигнал тремя способами.

1. Игнорировать сигнал. Такая реакция не рекомендуется для сигналов, указывающих на аппаратную ошибку (такую, как деление на ноль или обращение к памяти, находящейся вне адресного пространства процесса), поскольку результат в этом случае непредсказуем.
2. Разрешить выполнение действия по умолчанию. В случае деления на ноль по умолчанию происходит аварийное завершение процесса.
3. Определить функцию, которая будет вызвана для обработки сигнала (такие функции называют перехватчиками сигналов). Определив такую функцию,

можно отслеживать получение сигнала и реагировать на него по своему усмотрению.

Сигналы порождаются во многих ситуациях. Две клавиши терминала, известные как *клавиша прерывания* (Control-C или DELETE) и *клавиша выхода* (часто Control-\), используются для прерывания работы текущего процесса. Генерировать сигнал можно также вызовом функции `kill`. С ее помощью один процесс может послать сигнал другому. Естественно, эта операция имеет свои ограничения: чтобы послать сигнал процессу, мы должны быть его владельцем (или суперпользователем).

Пример

Вспомните пример простейшей командной оболочки в листинге 1.5. Если запустить эту программу и нажать клавишу прерывания (Control-C), процесс завершит работу, так как реакция по умолчанию на этот сигнал, называемый `SIGINT`, заключается в завершении процесса. Процесс не сообщил ядру, что реакция на сигнал должна отличаться от действия по умолчанию, поэтому он завершается.

Чтобы перехватить сигнал, программа должна вызвать функцию `signal`, передав ей имя функции, которая должна вызываться при получении сигнала `SIGINT`. В следующем примере эта функция называется `sig_int`. Она просто выводит на экран сообщение и новое приглашение к вводу. Добавив 11 строк в программу из листинга 1.5, мы получим версию в листинге 1.8 (добавленные строки обозначены символами «+»).

Листинг 1.8. Чтение команд со стандартного ввода и их выполнение

```
#include "apue.h"
#include <sys/wait.h>

+ static void sig_int(int); /* наша функция-перехватчик */
+
int
main(void)
{
    char    buf[MAXLINE]; /* из apue.h */
    pid_t   pid;
    int     status;

+    if (signal(SIGINT, sig_int) == SIG_ERR)
+        err_sys("ошибка вызова signal");

+    printf("% "); /* вывести приглашение (printf требует использовать */
+           /* последовательность %, чтобы вывести символ %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* заменить символ перевода строки */

        if ((pid = fork()) < 0) {
            err_sys("ошибка вызова fork");
        } else if (pid == 0) { /* дочерний процесс */
            execlp(buf, buf, (char *)0);
            err_ret("невозможно выполнить: %s", buf);
            exit(127);
        }
    }
}
```

```

/* родительский процесс */
if ((pid = waitpid(pid, &status, 0)) < 0)
    err_sys("ошибка вызова waitpid");
printf("% ");
}
exit(0);
}
+
+ void
+ sig_int(int signo)
+ {
+     printf("прервано\n%");
+ }

```

В главе 10 мы детально рассмотрим сигналы, поскольку с ними работает большинство серьезных приложений.

1.10. Представление времени

Исторически в системе UNIX поддерживается два способа представления времени.

1. Календарное время. Значения в этом представлении хранят число секунд, прошедших с начала Эпохи — 00:00:00 1 января 1970 года по согласованному всемирному времени (Coordinated Universal Time, UTC). (Старые руководства описывают UTC как Greenwich Mean Time — время по Гринвичу.) Эти значения используются, например, для представления времени последнего изменения файла.

Для хранения времени в этом представлении используется тип данных `time_t`.

2. Время работы процесса. Оно еще называется процессорным временем и измеряет ресурсы центрального процессора, использованные процессом. Эти значения измеряются в тактах (ticks). Исторически сложилось так, что в различных системах в одной секунде может быть 50, 60 или 100 тактов.

Для хранения времени в этом представлении используется тип данных `clock_t`. (В разделе 2.5.4 мы покажем, как узнать количество тактов в секунде вызовом `sysconf`.)

В разделе 3.9 мы увидим, что при измерении времени выполнения процесса система UNIX хранит три значения:

- общее время (Clock time);
- пользовательское время (User CPU time);
- системное время (System CPU time).

Общее время (иногда называют *временем настенных часов*) — отрезок времени, затраченный процессом от момента запуска до завершения. Это значение зависит от общего количества процессов, выполняемых в системе. Всякий раз, когда нас интересует общее время, измерения должны делаться на незагруженной системе.

Пользовательское время — это время, затраченное на исполнение машинных инструкций самой программы. Системное время — это время, затраченное на вы-

полнение машинных инструкций в ядре от имени процесса. Например, всякий раз, когда процесс обращается к системному вызову, такому как `read` или `write`, ему приписывается время, затраченное ядром на выполнение запроса. Сумму пользовательского и системного времени часто называют *процессорным временем* (CPU time).

Измерить общее, пользовательское и системное время просто: выполните команду `time(1)`, передав ей в качестве аргумента команду, время работы которой требуется измерить. Например:

```
$ cd /usr/include  
$ time -p grep _POSIX_SOURCE /*.*.h > /dev/null  
  
real    0m0.81s  
user    0m0.11s  
sys     0m0.07s
```

Формат вывода результатов зависит от командной оболочки, поскольку некоторые из них вместо утилиты `/usr/bin/time` используют встроенную функцию, измеряющую время выполнения заданной команды.

В разделе 8.17 мы увидим, как получить все три значения из запущенного процесса. Собственно тема даты и времени будет рассматриваться в разделе 6.10.

1.11. Системные вызовы и библиотечные функции

Любая операционная система дает прикладным программам возможность обращаться к системным службам. Во всех реализациях UNIX имеется строго определенное число точек входа в ядро, которые называются *системными вызовами* (вспомните рис. 1.1). Седьмая версия Research UNIX System имела около 50 системных вызовов, 4.4BSD — около 110, а SVR4 — примерно 120. В Linux 3.2.0 имеется 380 системных вызовов, а в FreeBSD 8.0 их более 450.

Интерфейс системных вызовов всегда документируется во втором разделе «Руководства программиста UNIX». Он определяется на языке C независимо от конкретных реализаций, использующих системные вызовы в той или иной системе. В этом отличие от многих старых систем, которые традиционно определяли точки входа в ядро на языке ассемблера.

В системе UNIX для каждого системного вызова предусматривается одноименная функция в стандартной библиотеке языка C. Пользовательский процесс вызывает эту функцию как обычно, а она вызывает соответствующую службу ядра, применяя способ обращения, принятый в данной системе. Например, функция может поместить один или более своих аргументов в регистры общего назначения и затем выполнить некоторую машинную инструкцию, которая сгенерирует программное прерывание. В нашем случае мы можем рассматривать системные вызовы как обычные функции языка C.

Раздел 3 «Руководства программиста UNIX» описывает функции общего назначения, доступные программисту. Эти функции не являются точками входа в ядро,

хотя могут обращаться к нему посредством системных вызовов. Например, функция `printf` может использовать системный вызов `write` для вывода строки, но функции `strcpy` (копирование строки) и `atoi` (преобразование ASCII-строки в число) не выполняют системных вызовов.

С точки зрения разработчика системы, между системным вызовом и библиотечной функцией имеются коренные различия. Но с точки зрения пользователя, эти различия носят непринципиальный характер. В контексте нашей книги системные вызовы и библиотечные функции можно представлять как обычные функции языка С. И те и другие предназначены для обслуживания прикладных программ. Однако при этом нужно понимать, что библиотечные функции можно заменить, если в этом возникнет необходимость, а системные вызовы — нет.

Рассмотрим в качестве примера функцию выделения памяти `malloc`. Существует масса способов распределения памяти и алгоритмов «сборки мусора» (метод наилучшего приближения, метод первого подходящего и т. д.). Но нет единой методики, оптимальной абсолютно для всех возможных ситуаций. Системный вызов `sbrk(2)`, который занимается выделением памяти, не является диспетчером памяти общего назначения. Он лишь увеличивает или уменьшает адресное пространство процесса на заданное количество байтов, а управление этим пространством возлагается на сам процесс. Функция `malloc(3)` реализует одну конкретную модель распределения памяти. Если она нам не нравится по каким-то причинам, мы можем написать собственную функцию `malloc`, которая, вероятно, будет обращаться к системному вызову `sbrk`. На самом деле многие программные пакеты реализуют собственные алгоритмы распределения памяти с использованием системного вызова `sbrk`. На рис. 1.2 показаны взаимоотношения между приложением, функцией `malloc` и системным вызовом `sbrk`.

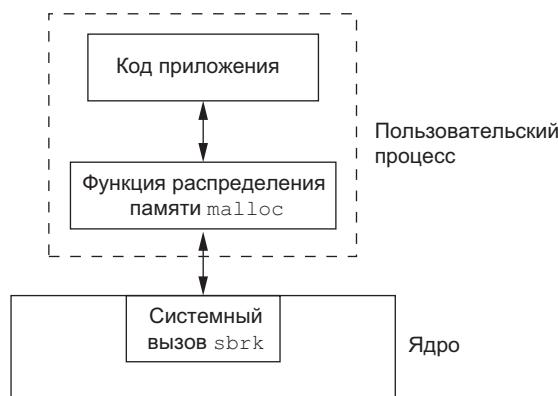


Рис. 1.2. Разделение обязанностей функции `malloc` и системного вызова `sbrk`

Здесь мы видим четкое разделение обязанностей: системный вызов выделяет дополнительную область памяти от имени процесса, а библиотечная функция `malloc` распоряжается этой областью.

Еще один пример, иллюстрирующий различия между системным вызовом и библиотечной функцией, — интерфейс определения текущей даты и времени. В некоторых операционных системах имеется два системных вызова: один возвращает время, другой — дату. Любая специальная обработка, такая как переход на летнее время, выполняется ядром или требует вмешательства человека. UNIX предоставляет единственный системный вызов, возвращающий количество секунд, прошедших с начала Эпохи — 0 часов 00 минут 1 января 1970 года по согласованному всемирному времени (UTC). Любая интерпретация этого значения, например представление в удобном для человека виде с учетом поясного времени, полностью возлагается на пользовательский процесс. Стандартная библиотека языка C содержит функции практически для любых случаев. Они, например, реализуют различные алгоритмы, учитывающие переход на зимнее или летнее время. Прикладная программа может обращаться к системному вызову и к библиотечной функции. Кроме того, следует помнить, что библиотечные функции, в свою очередь, также могут обращаться к системным вызовам. Это наглядно показано на рис. 1.3.

Другое отличие системных вызовов от библиотечных функций заключается в том, что системные вызовы обеспечивают лишь минимально необходимую функциональность, тогда как библиотечные функции часто обладают более широкими возможностями. Мы уже видели это различие на примере сравнения системного вызова `sbrk` с библиотечной функцией `malloc`. Мы еще столкнемся с этим различием, когда будем сравнивать функции небуферизованного ввода/вывода (глава 3) и стандартные функции ввода/вывода (глава 5).

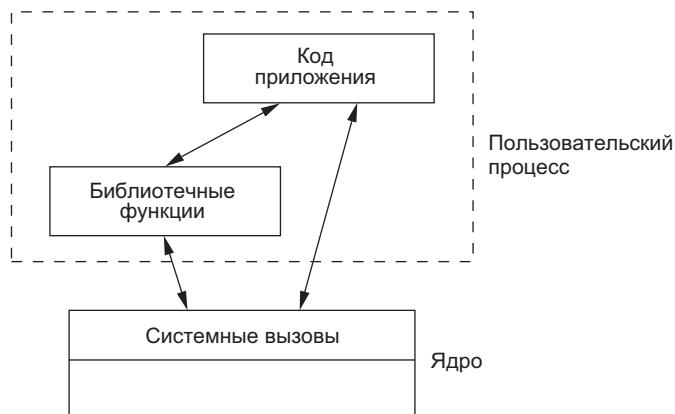


Рис. 1.3. Разделение обязанностей функции malloc и системного вызова sbrk

Системные вызовы управления процессами (`fork`, `exec` и `waitpid`) обычно вызываются пользовательским процессом напрямую. (Вспомните простую командную оболочку в листинге 1.5.) Но существуют также библиотечные функции, которые служат для упрощения самых распространенных случаев: например, функции `system` и `popen`. В разделе 8.13 мы увидим реализацию функции `system`, выпол-

ненную на основе системных вызовов управления процессами. В разделе 10.18 мы дополним этот пример обработкой сигналов.

Чтобы охарактеризовать интерфейс системы UNIX, используемый большинством программистов, мы должны будем описать не только системные вызовы, но и некоторые библиотечные функции. Описав, к примеру, только системный вызов `sbrk`, мы оставили бы без внимания более удобную для программиста функцию `malloc`, которая применяется во множестве приложений. В этой книге под термином *функция* мы будем подразумевать и системные вызовы, и библиотечные функции, за исключением случаев, когда потребуется подчеркнуть имеющиеся отличия.

1.12. Подведение итогов

Эта глава представляет собой обзорную экскурсию по системе UNIX. Мы дали определение ряда фундаментальных понятий, с которыми столкнемся еще не раз, и привели примеры небольших программ, чтобы вы могли представить, о чем пойдет речь в этой книге.

Следующая глава рассказывает о стандартизации UNIX и о влиянии деятельности в этой области на ее развитие. Стандарты, особенно ISO C и POSIX.1, будут постоянно встречаться на протяжении всей книги.

Упражнения

- 1.1 В своей системе проверьте и убедитесь, что каталоги «.» и «..» являются различными каталогами, за исключением корневого каталога.
- 1.2 Просмотрите еще раз результат работы примера в листинге 1.4 и скажите, куда пропали процессы с идентификаторами 852 и 853.
- 1.3 Аргумент функции `perror` в разделе 1.7 определен с атрибутом `const` (в соответствии со стандартом ISO C), в то время как целочисленный аргумент функции `strerror` определен без этого атрибута. Почему?
- 1.4 Если предположить, что календарное время хранится в виде 32-разрядного целого числа со знаком, в каком году наступит переполнение? Какими способами можно отдалить дату переполнения? Будут ли найденные решения совместимы с существующими приложениями?
- 1.5 Если предположить, что время работы процесса хранится в виде 32-разрядного целого числа со знаком и система отсчитывает 100 тактов в секунду, через сколько дней наступит переполнение счетчика?

2

Стандарты и реализации UNIX

2.1. Введение

Немалая работа была проделана для стандартизации системы UNIX и языка программирования С. Хотя приложения всегда обладали высокой переносимостью между разными версиями UNIX, тем не менее появление многочисленных версий UNIX в течение 80-х годов привело к тому, что крупные пользователи, такие как правительство США, были вынуждены призвать разработчиков к выработке стандартов.

В этой главе мы сначала рассмотрим различные попытки стандартизации, предпринимавшиеся за последние два с половиной десятилетия, а затем обсудим их влияние на реализации UNIX, которые обсуждаются в данной книге. Важной частью любых работ по стандартизации является спецификация различных ограничений, которые должны быть установлены для каждой реализации, поэтому мы рассмотрим эти ограничения и различные способы определения их значений.

2.2. Стандартизация UNIX

2.2.1. ISO C

В конце 1989 года был одобрен стандарт ANSI для языка программирования С – X3.159-1989. Этот стандарт также был принят как международный стандарт ISO/IEC 9899:1990. Аббревиатура ANSI расшифровывается как American National Standards Institute (Американский национальный институт стандартов, представляющий США в Международной организации по стандартизации – International Organization for Standardization, ISO). Аббревиатура IEC означает International Electrotechnical Commission (Международная электротехническая комиссия).

Стандарт языка С теперь поддерживается и развивается международной рабочей группой ISO/IEC по стандартизации языка программирования С, известной как ISO/IEC JTC1/SC22/WG14, или сокращенно WG14. Цель стандарта ISO C обеспечить переносимость программ на языке С между самыми разными операционными системами, не только UNIX. Этот стандарт определяет не только синтаксис и семантику языка, но также состав стандартной библиотеки [ISO 1999, глава 7; Plauger 1992; Kernighan and Ritchie 1988, приложение B]. Эта библиотека имеет большое значение, потому что все современные версии UNIX, в том числе опи-

санные в этой книге, обязаны предоставлять библиотеки функций, определяемых стандартом языка C.

В 1999 году стандарт ISO C был обновлен и одобрен как ISO/IEC 9899:1999. Он в значительной степени улучшил поддержку приложений, выполняющих числовую обработку. Изменения не затронули стандарты POSIX, описываемые в этой книге, кроме добавления ключевого слова `restrict` к некоторым прототипам функций. Это ключевое слово сообщает компилятору, какие ссылки по указателю можно оптимизировать, отмечая объекты, доступ к которым осуществляется из функций только посредством данного указателя.

Начиная с 1999 года опубликованы три технические поправки, исправляющие ошибки в стандарте ISO C: в 2001, 2004 и 2007 годах. В большинстве случаев между одобрением стандарта и модификацией программного обеспечения, учитывающей изменения в стандартах, проходит какое-то время. По мере своего развития все системы компиляции добавляют или совершенствуют поддержку последней версии стандарта ISO C.

Информацию о текущем уровне соответствия gcc стандарту ISO C от 1999 года можно найти по адресу <http://www.gnu.org/software/gcc/c99status.html>. Хотя стандарт языка C был обновлен и дополнен в 2011 году, в этой книге будет рассматриваться только версия 1999 года, так как другие стандарты пока не учитывают соответствующие изменения.

Библиотеку ISO C можно разбить на 24 раздела, основываясь на именах заголовочных файлов, определяемых стандартом (табл. 2.1). Стандарт POSIX.1 включает эти файлы и, кроме того, определяет ряд дополнительных заголовочных файлов. Как видно в табл. 2.1, все эти заголовочные файлы поддерживаются четырьмя реализациями (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10), описываемыми далее в этой главе.

Перечень заголовочных файлов ISO C зависит от версии компилятора языка C, используемой в той или иной операционной системе. Изучая табл. 2.1, имейте в виду, что FreeBSD 8.0 распространяется с gcc версии 4.2.1, Solaris 10 — с gcc версии 3.4.3 (в дополнение к собственному компилятору C, входящему в состав Sun Studio), Ubuntu 12.04 (Linux 3.2.0) — с gcc версии 4.6.3, а Mac OS X 10.6.8 — с двумя версиями gcc, 4.0.1 и 4.2.1.

Таблица 2.1. Перечень заголовочных файлов, определяемых стандартом ISO C

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<code><cassert.h></code>	✓	✓	✓	✓	Проверка программных утверждений
<code><complex.h></code>	✓	✓	✓	✓	Поддержка арифметики комплексных чисел
<code><cctype.h></code>	✓	✓	✓	✓	Типы символов
<code><errno.h></code>	✓	✓	✓	✓	Коды ошибок (раздел 1.7)
<code><fenv.h></code>	✓	✓	✓	✓	Окружение операций с плавающей запятой
<code><float.h></code>	✓	✓	✓	✓	Арифметика с плавающей запятой

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<inttypes.h>	✓	✓	✓	✓	Преобразования целочисленных типов
<iso646.h>	✓	✓	✓	✓	Альтернативные макросы операторов отношений
<limits.h>	✓	✓	✓	✓	Константы реализации (раздел 2.5)
<locale.h>	✓	✓	✓	✓	Классы региональных настроек (локалей)
<math.h>	✓	✓	✓	✓	Математические константы
<setjmp.h>	✓	✓	✓	✓	Нелокальные переходы (раздел 7.10)
<signal.h>	✓	✓	✓	✓	Сигналы (глава 10)
<stdarg.h>	✓	✓	✓	✓	Списки аргументов переменной длины
<stdbool.h>	✓	✓	✓	✓	Логический тип и значения
<stddef.h>	✓	✓	✓	✓	Стандартные определения
<stdint.h>	✓	✓	✓	✓	Целочисленные типы
<stdio.h>	✓	✓	✓	✓	Стандартная библиотека ввода/вывода (глава 5)
<stdlib.h>	✓	✓	✓	✓	Функции общего назначения
<string.h>	✓	✓	✓	✓	Операции над строками
<tgmath.h>	✓	✓	✓	✓	Макроопределения математических операций
<time.h>	✓	✓	✓	✓	Время и дата (раздел 6.10)
<wchar.h>	✓	✓	✓	✓	Расширенная поддержка многобайтных символов
<wctype.h>	✓	✓	✓	✓	Классификация и функции преобразования многобайтных символов

2.2.2. IEEE POSIX

POSIX — это семейство стандартов, разработанных организацией IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров электроники и радиотехники). Аббревиатура POSIX расшифровывается как Portable Operating System Interface (Интерфейс переносимой операционной системы). Изначально это название относилось только к стандарту IEEE 1003.1-1988 (интерфейс операционной системы), но позднее оно стало объединять множество других стандартов и предварительных стандартов проекта под номером 1003, в том числе на командную оболочку и утилиты (1003.2).

Основной интерес для нас представляет стандарт на интерфейс переносимой операционной системы 1003.1, цель которого состоит в повышении переносимости

приложений между различными версиями UNIX. Этот стандарт определяет набор служб, которые должна предоставлять операционная система, если она претендует на звание «POSIX-совместимой». Хотя стандарт 1003.1 базируется на операционной системе UNIX, он не ограничивается UNIX и UNIX-подобными операционными системами. Действительно, некоторые производители proprietарных операционных систем утверждают, что их системы являются POSIX-совместимыми, в то же время сохраняя все свои proprietарные особенности.

Поскольку стандарт 1003.1 определяет *интерфейс*, а не *реализацию*, между системными вызовами и библиотечными функциями не делается никаких различий. Стандарт именует все процедуры *функциями*.

Стандарты продолжают непрерывно развиваться, и 1003.1 не является исключением. Версия этого стандарта 1988 года, IEEE Standard 1003.1-1988, была дополнена и представлена на рассмотрение Международной организации по стандартизации (ISO). Текст стандарта был полностью переработан, хотя при этом не было добавлено каких-либо новых интерфейсов или особенностей. Окончательный документ был опубликован как IEEE Std 1003.1-1990 [IEEE 1990]. Он также является международным стандартом ISO/IEC 99451: 1990. Обычно этот стандарт называют POSIX.1, и в этой книге также используется это обозначение.

Рабочая группа IEEE 1003.1 продолжила внесение изменений в стандарт. В 1996 году была издана пересмотренная версия стандарта IEEE 1003.1. Она включала в себя стандарт 1003.1-1990, стандарт на расширения реального времени 1003.1b-1993 и интерфейсы многопоточного программирования под названием *pthreads* для потоков POSIX. Эта версия стандарта была опубликована как International Standard ISO/IEC 9945-1:1996. В 1999 году с выходом стандарта IEEE Standard 1003.1d-1999 были добавлены улучшенные интерфейсы реального времени. Год спустя был опубликован стандарт IEEE Standard 1003.1j-2000, в котором появились дополнительные, улучшенные интерфейсы реального времени. В этом же году вышел стандарт IEEE Standard 1003.1q-2000, добавивший расширения трассировки событий.

Версия стандарта 1003.1 2001 года отличалась от предшествующих версий тем, что объединила некоторые поправки из стандартов 1003.1, 1003.2 и часть Single UNIX Specification (SUS — единая спецификация UNIX) версии 2 (подробнее об этом стандарте рассказывается ниже). В окончательный вариант IEEE Standard 1003.1-2001 вошли следующие стандарты:

- ISO/IEC 9945-1 (IEEE Standard 1003.1-1996), включающий:
 - IEEE Standard 1003.1-1990;
 - IEEE Standard 1003.1b-1993 (расширения реального времени);
 - IEEE Standard 1003.1c-1995 (*pthreads*);
 - IEEE Standard 1003.1i-1995 (список технических исправлений);
- IEEE P1003.1a предварительный стандарт (пересмотр системных интерфейсов);
- IEEE Standard 1003.1d-1999 (улучшенные расширения реального времени);
- IEEE Standard 1003.1j-2000 (дополнительные улучшенные расширения реального времени);

- IEEE Standard 1003.1q-2000 (трассировка);
- Части стандарта IEEE Standard 1003.1g-2000 (независимые от протокола интерфейсы);
- ISO/IEC 9945-2 (IEEE Standard 1003.2-1993);
- IEEE P1003.2b предварительный стандарт (оболочка и дополнительные утилиты);
- IEEE Standard 1003.2d-1994 (пакетные расширения);
- Основные спецификации Single UNIX Specification версии 2, включая:
 - System Interface Definitions, Issue 5 (определения системных интерфейсов, выпуск 5);
 - Commands and Utilities, Issue 5 (команды и утилиты, выпуск 5);
 - System Interfaces and Headers, Issue 5 (системные интерфейсы и заголовочные файлы, выпуск 5);
- Open Group Technical Standard, Networking Services, Issue 5.2 (технический стандарт на сетевые службы, выпуск 5.2);
- ISO/IEC 9899:1999, Programming Languages – C (языки программирования – C).

В 2004 году спецификация POSIX.1 была дополнена техническими исправлениями. В 2008-м были произведены более обширные исправления и выпущены как Base Specifications Issue 7 (базовые спецификации, выпуск 7). Эта версия была одобрена организацией ISO в конце 2008-го и опубликована в 2009-м под названием International Standard ISO/IEC 9945:2009. Этот стандарт основывается на некоторых других стандартах:

- IEEE Standard 1003.1, издание 2004 года;
- Open Group Technical Standard, 2006, Extended API Set, Parts 1–4 (расширенные программные интерфейсы, части 1–4);
- ISO/IEC 9899:1999, включающий исправления.

В табл. 2.2, 2.3 и 2.4 приводятся списки обязательных и дополнительных заголовочных файлов, предусматриваемых стандартом POSIX.1. Поскольку POSIX.1 включает стандартные библиотечные функции ISO C, он также требует наличия заголовочных файлов, перечисленных в табл. 2.1. Все четыре таблицы представляют собой перечень заголовочных файлов, которые включены в обсуждаемые здесь реализации операционных систем.

В этой книге мы описываем версию стандарта POSIX.1 от 2008 года, включающую в себя функции, определенные стандартом ISO C. Его интерфейсы подразделяются на обязательные для реализации и дополнительные. Кроме того, дополнительные интерфейсы по своей функциональности подразделяются на 40 категорий. Категории еще не устаревших интерфейсов программирования перечислены в табл. 2.5 с соответствующими им кодами. Коды – это двух- или трехсимвольные сокращения, которые помогают идентифицировать функциональную область интерфейса. С помощью этих кодов в тексте справочного руководства отмечаются места, где описываемые интерфейсы зависят от поддержки соответствующего

дополнения. Многие из дополнительных интерфейсов относятся к расширениям реального времени.

Таблица 2.2. Перечень обязательных заголовочных файлов, определяемых стандартом POSIX

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<aio.h>	✓	✓	✓	✓	Асинхронный ввод/вывод
<cpio.h>	✓	✓	✓	✓	Архиватор <code>cpio</code>
<dirent.h>	✓	✓	✓	✓	Работа с каталогами (раздел 4.22)
<dlfcn.h>	✓	✓	✓	✓	Динамическое связывание
<fcntl.h>	✓	✓	✓	✓	Управление файлами (раздел 3.14)
<fnmatch.h>	✓	✓	✓	✓	Шаблоны имен файлов
<glob.h>	✓	✓	✓	✓	Шаблоны путей в файловой системе
<grp.h>	✓	✓	✓		Файл групп (раздел 6.4)
<iconv.h>	✓	✓	✓		Преобразование кодировок символов
<langinfo.h>	✓	✓	✓	✓	Константы сведений о языках
<monetary.h>	✓	✓	✓	✓	Типы и функции для финансовых вычислений
<netdb.h>	✓	✓	✓	✓	Операции с распределенной базой системных данных
<nl_types.h>	✓	✓	✓	✓	Каталоги с сообщениями
<poll.h>	✓	✓	✓	✓	Функция <code>poll</code> (раздел 14.4.2)
<pthread.h>	✓	✓	✓	✓	Потоки (главы 11 и 12)
<pwd.h>	✓	✓	✓	✓	Файл паролей (раздел 6.2)
<regex.h>	✓	✓	✓	✓	Регулярные выражения
<sched.h>	✓	✓	✓	✓	Планировщик
<semaphore.h>	✓	✓	✓	✓	Семафоры
<strings.h>	✓	✓	✓	✓	Операции над строками
<tar.h>	✓	✓	✓	✓	Архиватор <code>tar</code>
<termios.h>	✓	✓	✓	✓	Терминальный ввод/вывод (глава 18)
<unistd.h>	✓	✓	✓	✓	Символьные константы

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<wordexp.h>	✓	✓		✓	Дополнение слов по шаблону
<arpa/inet.h>	✓	✓	✓	✓	Сеть Интернет (глава 16)
<net/if.h>	✓	✓	✓	✓	Локальные сетевые интерфейсы (глава 16)
<netinet/in.h>	✓	✓	✓	✓	Семейство адресов Интернета (раздел 16.3)
<netinet/tcp.h>	✓	✓	✓	✓	Определения протокола TCP
<sys/mman.h>	✓	✓	✓	✓	Управление памятью
<sys/select.h>	✓	✓	✓	✓	Функция <code>select</code> (раздел 14.4.1)
<sys/socket.h>	✓	✓	✓	✓	Интерфейс сокетов (глава 16)
<sys/stat.h>	✓	✓	✓	✓	Получение сведений о файлах (глава 4)
<sys/statvfs.h>	✓	✓	✓	✓	Информация о файловой системе
<sys/times.h>	✓	✓	✓	✓	Время работы процесса (раздел 8.17)
<sys/types.h>	✓	✓	✓	✓	Примитивы системных типов данных (раздел 2.8)
<sys/un.h>	✓	✓	✓	✓	Определения, касающиеся сокетов домена UNIX (раздел 17.2)
<sys/utsname.h>	✓	✓	✓	✓	Название системы (раздел 6.9)
<sys/wait.h>	✓	✓	✓	✓	Управление процессами (раздел 8.6)

Стандарт POSIX.1 не определяет понятие суперпользователя. Вместо этого он требует, чтобы некоторые операции были доступны только при наличии «соответствующих привилегий», но определение этого термина POSIX.1 оставляет на усмотрение конкретной реализации. Версии UNIX, разработанные в соответствии с принципами безопасности Министерства обороны США, имеют многоуровневую систему безопасности. В этой книге мы будем пользоваться традиционной терминологией и называть такие действия требующими привилегий суперпользователя.

По прошествии более чем 20 лет сформировались стандарты, которые можно считать достаточно зрелыми и устоявшимися. Стандарт POSIX.1 поддерживается открытой рабочей группой, известной как Austin Group (<http://www.opengroup.org/austin>). Чтобы стандарты оставались актуальными, время от времени они должны подтверждаться или обновляться.

Таблица 2.3. Заголовочные файлы расширений XSI, определяемые стандартом POSIX

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<fmtmsg.h>	✓	✓	✓	✓	Вывод сообщений в форматированном виде
<ftw.h>	✓	✓	✓	✓	Обход дерева файлов (раздел 4.21)
<libgen.h>	✓	✓	✓	✓	Определения для функций поиска по шаблону
<ndbm.h>	✓		✓	✓	Операции с базой данных (стандарта dbm)
<search.h>	✓	✓	✓	✓	Поиск по таблицам
<syslog.h>	✓	✓	✓	✓	Системное журналирование (раздел 13.4)
<utmpx.h>		✓	✓	✓	Работа с учетными записями пользователей
<sys/ipc.h>	✓	✓	✓	✓	IPC (раздел 15.6)
<sys/msg.h>	✓	✓	✓	✓	Очереди сообщений (15.7)
<sys/resource.h>	✓	✓	✓	✓	Операции с ресурсами (раздел 7.11)
<sys/sem.h>	✓	✓	✓	✓	Семафоры (15.8)
<sys/shm.h>	✓	✓	✓	✓	Разделяемая память (раздел 15.9)
<sys/time.h>	✓	✓	✓	✓	Типы данных для хранения времени
<sys/uio.h>	✓	✓	✓	✓	Векторные операции ввода/вывода (раздел 14.7)

2.2.3. Single UNIX Specification

Single Unix Specification (Единая спецификация UNIX) — это надмножество стандарта POSIX.1 и определяет дополнительные интерфейсы, расширяющие возможности, предоставляемые базовой спецификацией POSIX.1. Стандарт POSIX.1 является эквивалентом раздела Base Specification (базовые спецификации) спецификации Single UNIX Specification.

Расширение *X/Open System Interface (XSI)* определяет дополнительные интерфейсы POSIX.1, которые должны поддерживаться реализацией, чтобы она получила право именоваться «XSI-совместимой». В их число входят: синхронизация файлов, адрес и размер стека потока, синхронизация потоков между процессами и символьная константа _XOPEN_UNIX (все они отмечены в табл. 2.5 как «Обязательные для SUS»). Только XSI-совместимые реализации могут называться операционными системами UNIX.

Торговая марка *UNIX* принадлежит *The Open Group*, которая использует единую спецификацию *UNIX* для определения интерфейсов, обязательных для реализации в системе, чтобы она получила право называться системой *UNIX*. Для получения лицензии на право использования торговой марки *UNIX* реализация должна пройти серию тестов на соответствие.

Таблица 2.4. Необязательные заголовочные файлы, определяемые стандартом POSIX

Заголовочный файл	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<mqueue.h>	✓	✓	✓	✓	Очереди сообщений
<spawn.h>	✓	✓	✓	✓	Интерфейс запуска программ в системах реального времени

Таблица 2.5. Необязательные группы интерфейсов POSIX.1 и их коды

Код	Обязательные для SUS	Символическая константа	Описание
ADV		_POSIX_ADVISORY_INFO	Консультативная информация (расширение реального времени)
CPT		_POSIX_CPUTIME	Измерение времени работы процесса (расширение реального времени)
FSC	✓	_POSIX_FSYNC	Синхронизация файлов
IP6		_POSIX_IPV6	Интерфейсы IPv6
ML		_POSIX_MEMLOCK	Блокировка памяти процесса (расширение реального времени)
MLK		_POSIX_MEMLOCK_RANGE	Блокировка области памяти (расширение реального времени)
MON		_POSIX_MONOTONIC_CLOCK	Монотонные часы (расширение реального времени)
MSG		_POSIX_MESSAGE_PASSING	Передача сообщений (расширение реального времени)
MX		_STDC_IEC_559_-	Дополнение с плавающей запятой, соответствующее IEC 60599
PIO		_POSIX_PRIORITIZED_IO	Приоритетный ввод/вывод
PS		_POSIX_PRIORITY_SCHEDULING	Планирование процессов (расширение реального времени)
RPI		_POSIX_THREAD_ROBUST_PRIO_INHERIT	Наследование приоритета надежных мьютексов (расширение реального времени)
RPP		_POSIX_THREAD_ROBUST_PRIO_PROTECT	Защита приоритета надежных мьютексов (расширение реального времени)

Таблица 2.5 (окончание)

Код	Обязательные для SUS	Символическая константа	Описание
RS		_POSIX_RAW_SOCKETS	Низкоуровневые сокеты
SHM		_POSIX_SHARED_MEMORY_OBJECTS	Объекты разделяемой памяти (расширение реального времени)
SIO		_POSIX_SYNCHRONIZED_IO	Синхронизированный ввод/вывод (расширение реального времени)
SPN		_POSIX_SPAWN	Запуск процессов (расширение реального времени)
SS		_POSIX_SPORADIC_SERVER	Сервер непериодических (спoradicеских) процессов (расширение реального времени)
TCT		_POSIX_THREAD_CPUTIME	Измерение процессорного времени для потоков (расширение реального времени)
TPI		_POSIX_THREAD_PRIO_INHERIT	Наследование приоритета потока (расширение реального времени)
TPP		_POSIX_THREAD_PRIO_PROTECT	Защита приоритета потока (расширение реального времени)
TPS		_POSIX_THREAD_PRIORITY_SCHEDULING	Планирование выполнения потоков (расширение реального времени)
TSA	✓	_POSIX_THREAD_ATTR_STACKADDR	Адрес стека потока
TSH	✓	_POSIX_THREAD_PROCESS_SHARED	Синхронизация потоков между процессами
TSP		_POSIX_THREAD_SPORADIC_SERVER	Сервер непериодических (спорадических) потоков (расширение реального времени)
TSS	✓	_POSIX_THREAD_ATTR_STACKSIZE	Размер стека потока
TYM		_POSIX_TYPED_MEMORY_OBJECTS	Типизированная память (расширение реального времени)
XSI	✓	_XOPEN_UNIX	Интерфейсы расширений X/Open

Интерфейсы, необязательные для XSI-совместимых систем, делятся на *необязательные группы* по функциональному признаку:

- Шифрование: обозначаются символьной константой `_XOPEN_CRYPT`;
- Расширения реального времени: обозначаются символьной константой `_XOPEN_REALTIME`;

- Дополнения реального времени;
- Потоки реального времени: обозначаются символьной константой `_XOPEN_REALTIME_THREADS`;
- Дополнения к потокам реального времени.

Единая спецификация UNIX (SUS) публикуется организацией The Open Group, сформировавшейся в 1996 году в результате слияния X/Open и Open Software Foundation (OSF). X/Open принадлежит издание «X/Open Portability Guide» (Руководство X/Open по переносимости), которое заимствовало определенные стандарты и заполнило пробелы, связанные с отсутствующими функциональными возможностями. Целью этих руководств было повышение переносимости прикладных программ, которое стало возможным благодаря простому следованию опубликованным стандартам.

Первая версия Single UNIX Specification была издана X/Open в 1994 году. Она известна также под названием «Spec 1170», поскольку содержала примерно 1170 интерфейсов. Своими корнями она уходит в инициативу Common Open Software Environment (COSE – Общая открытая программная среда), цель которой состояла в том, чтобы еще больше повысить переносимость приложений между различными реализациями UNIX. Группа COSE – Sun, IBM, HP, Novell/USL и OSF – шагнула значительно дальше простого одобрения стандартов. Дополнительно она исследовала интерфейсы, обычно используемые коммерческими приложениями. В результате были отобраны 1170 интерфейсов и включены в X/Open Common Application Environment, Issue 4 (CAE – Общая среда приложений, известная также как XPG4, поскольку исторически ее предшественником было руководство X/Open Portability Guide), System V Interface Definition, Issue 3 (SVID – Определение интерфейса System V) и OSF Application Environment Specification (AES – Спецификация среды приложений).

Вторая версия Single UNIX Specification была издана The Open Group в 1997 году. В новую версию была добавлена поддержка потоков, интерфейсов реального времени, 64-разрядной арифметики, файлов большого размера и многобайтных символов.

Третья версия Single UNIX Specification (сокращенно – SUSv3) была опубликована The Open Group в 2001 году. Базовые спецификации SUSv3 те же, что и в стандарте IEEE Standard 1003.1-2001, и разделяются на четыре категории: «Основные определения», «Системные интерфейсы», «Командная оболочка и утилиты» и «Обоснование». SUSv3 также включает в себя X/Open Curses Issue 4, Version 2, но эта спецификация не является частью POSIX.1.

В 2002 году Международная организация по стандартизации одобрила эту версию как международный стандарт ISO/IEC 9945:2002. В 2003 году The Open Group снова обновила стандарт 1003.1, добавив в него исправления технического характера, после чего ISO одобрила его как ISO/IEC 9945:2003. В апреле 2004 года The Open Group опубликовала Single UNIX Specification, Version 3, 2004 Edition. В нее были включены дополнительные технические исправления основного текста стандарта.

В 2008 году в спецификацию Single UNIX Specification внесены исправления, добавлены новые и удалены устаревшие интерфейсы, а некоторые интерфейсы

отмечены как устаревшие, в подготовке к удалению в будущем. Кроме того, некоторым, прежде необязательным, интерфейсам присвоен статус обязательных, включая асинхронный ввод/вывод, барьеры, выбор тактового генератора, отображение файлов в память, защита памяти, блокировки чтения/записи, сигналы реального времени, семафоры POSIX, циклические блокировки (spin locks), потокобезопасные функции, потоки выполнения, тайм-ауты и таймеры. Получившийся стандарт известен как Base Specifications Issue 7 (базовые спецификации, выпуск 7) и в точности соответствует POSIX.1-2008. Организация The Open Group объединила эту версию с обновленной спецификацией X/Open Curses и в 2010 году выпустила их в виде версии 4 спецификации Single UNIX. Мы будем называть ее SUSv4.

2.2.4. FIPS

Аббревиатура *FIPS* означает Federal Information Processing Standard (Федеральный стандарт обработки информации). Этот стандарт опубликован правительством США, которое использовало его при покупке компьютерных систем. Стандарт FIPS 151-1 (апрель 1989 года) основан на IEEE Std. 1003.1-1988 и на проекте стандарта ANSI C. За ним последовал FIPS 151-2 (май 1993 года) на основе IEEE Standard 1003.1-1990. FIPS 151-2 требовал наличия некоторых возможностей, которые стандартом POSIX.1 объявлены необязательными. Все они стали обязательными в стандарте POSIX.1-2001.

В результате любой производитель, желавший продавать POSIX.1-совместимые компьютерные системы американскому правительству, должен поддерживать некоторые дополнительные особенности POSIX.1. Позднее стандарт POSIX.1 FIPS был отменен, поэтому мы больше не будем возвращаться к нему в этой книге.

2.3. Реализации UNIX

В предыдущем разделе были описаны ISO C, IEEE POSIX и Single UNIX Specification — три стандарта, разработанные независимыми организациями. Однако стандарты — это лишь спецификации интерфейса. А как они связаны с реальностью? Производители воплощают эти стандарты в конкретные реализации. Нам интересны не только сами стандарты, но и их воплощение.

В разделе 1.1 [McKusick et al., 1996] приводится подробная (и отлично иллюстрированная) история генеалогического дерева UNIX. Все началось с 6-й (1976) и 7-й (1979) редакций UNIX Time-Sharing System для PDP-11 (обычно они именуются Version 6 и Version 7). Они стали первыми версиями, получившими широкое распространение за пределами Bell Laboratories. Начали самостоятельно развиваться три ветви UNIX.

1. Одна — в AT&T — привела к появлению System III и System V (так называемые коммерческие версии UNIX).
2. Другая — в Калифорнийском университете города Беркли — привела к появлению реализаций 4.xBSD.

3. Третья — исследовательская версия UNIX, которая продолжала разрабатываться в исследовательском центре вычислительной техники (Computing Science Research Center) в AT&T Bell Laboratories, — привела к появлению UNIX Time-Shared System 8-й и 9-й редакций и завершилась выходом 10-й редакции в 1990 году.

2.3.1. UNIX System V Release 4

Версия UNIX System V Release 4 (SVR4) была выпущена подразделением AT&T — UNIX System Laboratories (USL, ранее — UNIX Software Operation). Версия SVR4 объединила функциональность AT&T UNIX System Release 3.2 (SVR3.2), SunOS — операционной системы от Sun Microsystems, 4.3BSD, выпущенной Калифорнийским университетом, и Xenix — операционной системы от корпорации Microsoft — в единую операционную систему. (Изначально Xenix разрабатывалась на основе 7-й редакции и позднее вобрала в себя многие особенности, присущие System V.) Исходные тексты SVR4 были опубликованы в конце 1989 года, а первые копии стали доступны конечным пользователям в 1990 году. Реализация SVR4 соответствовала как стандарту POSIX 1003.1, так и X/Open Portability Guide, Issue 3 (XPG3). Корпорация AT&T также опубликовала «System V Interface Definition» (SVID, Определение интерфейса System V) [AT&T, 1989]. Выпуск 3 SVID определил функциональные возможности, которые должны поддерживаться операционной системой, чтобы она могла быть квалифицирована как реализация, соответствующая System V Release 4. Как и в случае с POSIX.1, SVID определяет интерфейс, но не реализацию. В SVID не проводится различий между системными вызовами и библиотечными функциями. Чтобы обнаружить эти различия, необходимо обращаться к справочному руководству по фактической реализации SVR4 [AT&T, 1990e].

2.3.2. 4.4BSD

Версии Berkeley Software Distribution (BSD) разрабатывались и распространялись Computer Systems Research Group (CSRG) — Группой исследования компьютерных систем Калифорнийского университета в Беркли. Версия 4.2BSD была выпущена в 1983-м, а 4.3BSD — в 1986 году. Обе версии работали на мини-компьютерах VAX. Следующая версия, 4.3BSD Tahoe, была выпущена в 1988 году и также работала на специфическом мини-компьютере под названием Tahoe. (Книга Leffler (Leffler) и др. [1989] описывает версию 4.3BSD Tahoe.) В 1990 году последовала версия 4.3BSD Reno, которая поддерживала большую часть функциональных возможностей, определяемых стандартом POSIX.1.

Изначально BSD-системы содержали исходный код, запатентованный AT&T, и подпадали под действие лицензий AT&T. Чтобы получить исходный код BSD-системы, требовалась лицензия AT&T на UNIX. С тех пор положение вещей изменилось, так как на протяжении нескольких лет все больше исходного кода AT&T замещалось кодом сторонних разработчиков; кроме того, в системе появилось много новых функциональных возможностей, исходный код которых был получен из других источников.

В 1989 году большая часть кода в версии 4.3BSD Tahoe, не принадлежащего AT&T, была идентифицирована и выложена в публичный доступ под названием BSD Networking Software, Release 1.0. Затем, в 1991 году, последовал второй выпуск BSD Networking Software (Release 2.0), который был развитием версии 4.3BSD Reno. Основная цель состояла в том, чтобы освободить большую часть или даже всю систему 4.4BSD от любых лицензионных ограничений AT&T, сделав исходные тексты общедоступными.

Версия 4.4BSD-Lite должна была стать заключительным релизом CSRG. Однако ее официальный выпуск был отложен из-за юридических споров с USL. Как только в 1994 году юридические разногласия были устраниены, вышла полностью свободная 4.4BSD-Lite и с этого момента, чтобы получить ее, не требовалось приобретать какие-либо лицензии на исходные тексты UNIX. Вслед за этим, в 1995 году, CSRG выпустила вторую, исправленную версию. Второй выпуск 4.4BSD-Lite стал заключительной версией BSD от CSRG. (Эта версия BSD описана в книге Мак-Кьюсика [1996].)

Разработка операционной системы UNIX в Беркли началась с PDP-11, переместилась на мини-компьютеры VAX и затем на так называемые рабочие станции. В первой половине 90-х годов была добавлена поддержка популярных персональных компьютеров, собранных на базе микропроцессора 80386, что привело к появлению версии 386BSD. Реализация была выполнена Биллом Джолитцом (Bill Jolitz) и описана в серии ежемесячных статей в журнале «Dr. Dobb's Journal» за 1991 год. Значительная часть исходного кода была заимствована из BSD Networking Software, Release 2.0.

2.3.3. FreeBSD

Операционная система FreeBSD базируется на 4.4BSD-Lite. Проект FreeBSD образован с целью дальнейшего развития линейки BSD-систем после того, как в Беркли было принято решение о прекращении работ над BSD-версиями операционной системы UNIX и проект 386BSD оказался заброшенным.

Все программное обеспечение, разработанное в рамках проекта FreeBSD, является свободно распространяемым как в исходных текстах, так и в виде бинарных дистрибутивов. ОС FreeBSD 8.0 стала одной из четырех платформ, на которых тестировались примеры для данной книги.

Существует еще несколько свободных операционных систем, основанных на BSD. Проект NetBSD (<http://www.netbsd.org>) аналогичен проекту FreeBSD, основной акцент в нем сделан на переносимости между различными аппаратными платформами. Проект OpenBSD (<http://www.openbsd.org>) также аналогичен FreeBSD, но с акцентом на безопасность.

2.3.4. Linux

Linux — это операционная система, которая предоставляет все богатства программного окружения UNIX и свободно распространяется в соответствии с Общественной лицензией GNU (GNU Public License). Популярность Linux — это

нечто феноменальное в компьютерной индустрии. Linux часто отличается тем, что первой из операционных систем начинает поддерживать новейшие аппаратные решения.

ОС Linux была создана Линусом Торвальдсом в 1991 году в качестве замены ОС MINIX. Семена дали быстрые всходы, и множество разработчиков по всему миру добровольно взялись за работу по ее улучшению.

Дистрибутив Ubuntu 12.04 стал одной из систем, на которых тестировались примеры из этой книги. В этом дистрибутиве используется ядро Linux версии 3.2.0.

2.3.5. Mac OS X

Mac OS X отличается от предыдущих версий этой системы тем, что основана на совершенно иных технологиях. Ее ядро называется «Darwin» и представляет собой комбинацию ядра Mach [Accetta et al., 1986], ОС FreeBSD и объектно-ориентированного фреймворка для драйверов и других расширений ядра. Версия Mac OS X 10.5, портированная на архитектуру Intel, была сертифицирована как UNIX-система. (Дополнительную информацию о сертификации UNIX-систем вы найдете по адресу <http://www.opengroup.org/certification/idx/unix.html>.)

Mac OS X 10.6.8 (Darwin 10.8.0) использовалась как одна из тестовых платформ при написании этой книги.

2.3.6. Solaris

Solaris — это разновидность ОС UNIX, разработанная в Sun Microsystems. Основанная на System V Release 4, она совершенствовалась инженерами из Sun Microsystems более 10 лет. Это единственный коммерчески успешный потомок SVR4, формально сертифицированный как UNIX-система.

В 2005 году Sun Microsystems открыла большую часть исходных текстов ОС Solaris в рамках проекта открытой ОС OpenSolaris с целью создать сообщество сторонних разработчиков.

Версия Solaris 10 UNIX использовалась при написании этой книги в качестве одной из тестовых платформ.

2.3.7. Прочие версии UNIX

Среди прочих операционных систем, сертифицированных как UNIX-системы, можно назвать:

- AIX, версия UNIX от IBM;
- HP-UX, версия UNIX от Hewlett-Packard;
- IRIX, UNIX-система, распространяемая компанией Silicon Graphics;
- UnixWare, версия UNIX, которая происходит от SVR4 и продается корпорацией SCO.

2.4. Связь между стандартами и реализациями

Упомянутые стандарты определяют подмножество любой фактически существующей системы. Основное внимание в этой книге будет уделяться четырем операционным системам: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10. Несмотря на то что только Mac OS X и Solaris могут называться UNIX-системами, все четыре предоставляют похожую программную среду. Поскольку все четыре системы в различной степени являются POSIX-совместимыми, мы сосредоточимся на тех функциональных возможностях, которые считаются обязательными в соответствии со стандартом POSIX.1, указывая любые различия между POSIX и фактической реализацией в этих системах. Особенности и технические приемы, характерные только для конкретной реализации, будут отмечены особо. Мы также обратим внимание на любые особенности, обязательные для UNIX-систем, но необязательные для POSIX-совместимых систем.

Следует отметить, что реализации обеспечивают обратную совместимость по функциональным особенностям с более ранними версиями, такими как SVR3.2 и 4.3BSD. Например, Solaris поддерживает неблокирующие операции ввода/вывода (`O_NONBLOCK`), определяемые стандартом POSIX.1, и традиционный для System V метод (`O_NDELAY`). В этой книге мы будем говорить о тех характеристиках, которые предписываются стандартом POSIX.1, и при этом лишь иногда упоминать нестандартные особенности, сохраняемые для обратной совместимости. Например, SVR3.2 и 4.3BSD реализуют механизм надежных сигналов способом, который отличается от стандарта POSIX.1. В главе 10 мы опишем только сигналы POSIX.1.

2.5. Ограничения

Реализации определяют множество системных кодов и констант. Многие из них жестко зашиты в тексты программ, для получения других используются специальные методы. Благодаря описанной выше деятельности по стандартизации, сейчас преобладают более универсальные методы определения значений констант и предусматриваемых реализациями ограничений, что очень помогает в разработке переносимого программного обеспечения.

Существует два типа ограничений.

1. Пределы времени компиляции (например, наибольшее значение, которое может принимать переменная типа `short int`).
2. Пределы времени выполнения (например, максимальная длина имени файла).

Пределы времени компиляции могут определяться в заголовочных файлах, которые подключаются программой на этапе компиляции. Но пределы времени выполнения требуют, чтобы процесс получил их значения, вызвав соответствующие функции.

Таблица 2.6. Пределы значений целочисленных типов из файла <limits.h>

Имя	Описание	Минимальное значение	Типовое значение
CHAR_BIT	Количество бит на символ	8	8
CHAR_MAX	Максимальное значение типа <code>char</code>	(см. ниже)	127
CHAR_MIN	Минимальное значение типа <code>char</code>	(см. ниже)	-128
SCHAR_MAX	Максимальное значение типа <code>signed char</code>	127	127
SCHAR_MIN	Минимальное значение типа <code>signed char</code>	-127	-128
UCHAR_MAX	Максимальное значение типа <code>unsigned char</code>	255	255
INT_MAX	Максимальное значение типа <code>int</code>	32 767	2 147 483 647
INT_MIN	Минимальное значение типа <code>int</code>	-32 767	-2 147 483 648
UINT_MAX	Максимальное значение типа <code>unsigned int</code>	65 535	4 294 967 295
SHRT_MAX	Максимальное значение типа <code>short</code>	32 767	32 767
SHRT_MIN	Минимальное значение типа <code>short</code>	-32 767	-32 768
USHRT_MAX	Максимальное значение типа <code>unsigned short</code>	65 535	65 535
LONG_MAX	Максимальное значение типа <code>long</code>	2 147 483 647	2 147 483 647
LONG_MIN	Минимальное значение типа <code>long</code>	-2 147 483 647	-2 147 483 648
ULONG_MAX	Максимальное значение типа <code>unsigned long</code>	4 294 967 295	4 294 967 295
LLONG_MAX	Максимальное значение типа <code>long long</code>	9 223 372 036 854 775 807	9 223 372 036 854 775 807
LLONG_MIN	Минимальное значение типа <code>long long</code>	-9 223 372 036 854 775 807	-9 223 372 036 854 775 808
ULLONG_MAX	Максимальное значение типа <code>unsigned long long</code>	18 446 744 073 709 551 615	18 446 744 073 709 551 615
MB_LEN_MAX	Максимальное число байтов в многобайтных символах	1	6

Кроме того, некоторые пределы в одной реализации имеют фиксированные значения и могут определяться статически, в заголовочных файлах. В других реализациях они могут варьироваться, из-за чего для получения их значений требуется обращаться к соответствующим функциям во время выполнения. Примером предела такого типа может служить максимальная длина имени файла. System V до появления SVR4 ограничивала длину имени файла 14 символами, тогда как BSD-системы увеличили это значение до 255 символов. Сегодня большинство реализаций UNIX поддерживают множество различных типов файловых систем, и каждая из них имеет свои пределы — это случай предела времени выполнения, который зависит от того, в какой файловой системе находится рассматриваемый файл. Например, корневая файловая система может ограничивать длину имени файла 14 символами, тогда как в другой файловой системе это ограничение может составлять 255 символов.

Для решения этих проблем существует три типа ограничений.

1. Пределы времени компиляции (заголовочные файлы).
2. Пределы времени выполнения, не связанные с файлами или каталогами (функция `sysconf`).
3. Пределы времени выполнения, связанные с файлами или каталогами (функции `pathconf` и `fpathconf`).

Еще большая путаница возникает, если конкретный предел времени выполнения не изменяется в данной системе. В этом случае он может быть определен статически в заголовочном файле. Но если он не определен в заголовочном файле, тогда приложение должно вызвать одну из трех функций `conf` (которые вскоре будут описаны), чтобы определить его значение во время выполнения.

2.5.1. Пределы ISO C

Все пределы, определяемые стандартом ISO C, являются пределами времени компиляции. В табл. 2.6 приведены пределы, задаваемые стандартом языка C и определенные в файле `<limits.h>`. Эти константы всегда определяются заголовочным файлом и не изменяются. В третьей колонке указаны минимально допустимые значения, определяемые стандартом ISO C. Они были выбраны с учетом 16-разрядной целочисленной арифметики с поразрядным дополнением до единицы. В четвертой колонке приводятся значения для системы Linux, использующей 32-разрядную целочисленную арифметику с поразрядным дополнением до двойки. Обратите внимание, что для целочисленных типов без знака не приводится минимальное значение, так как оно всегда будет равно 0. В 64-разрядных системах максимальное значение типа `long` соответствует максимальному значению типа `long long`.

Одно из различий между системами, с которым мы столкнемся: как система представляет тип `char` со знаком или без него. В четвертой колонке табл. 2.6 мы видим, что в данной системе тип `char` представлен как целое со знаком. Значение константы `CHAR_MIN` эквивалентно `SCHAR_MIN`, а `CHAR_MAX` эквивалентно `SCHAR_MAX`. Если тип `char` в системе представляется как целое без знака, следовательно, значение `CHAR_MIN` будет равно 0, а `CHAR_MAX` — `UCHAR_MAX`.

Предельные значения для типов чисел с плавающей запятой аналогично определяются в заголовочном файле `<float.h>`. Каждый, кто всерьез занимается вычислениями с плавающей запятой, должен ознакомиться с содержимым этого файла. Хотя стандарт ISO C определяет минимально допустимые значения для целочисленных типов, POSIX.1 вносит свои дополнения в стандарт языка С. Чтобы соответствовать требованиям POSIX.1, реализация должна поддерживать константу `INT_MAX` со значением 2 147 483 647, константу `INT_MIN` со значением -2 147 483 647 и константу `UINT_MAX` со значением 4 294 967 295. Так как POSIX.1 требует от реализаций обеспечить поддержку 8-разрядного типа `char`, константа `CHAR_BIT` должна иметь значение 8, `SCHAR_MIN` — значение -128, `SCHAR_MAX` — значение 127 и `UCHAR_MAX` — значение 255.

Еще одна константа из стандарта ISO C, с которой мы встретимся, — это `FOPEN_MAX`. Она определяет гарантированное системой минимальное количество стандартных потоков ввода/вывода, которые могут быть открыты одновременно. Это значение хранится в заголовочном файле `<stdio.h>` и не может быть меньше 8. Согласно стандарту POSIX.1 константа `STREAM_MAX`, если таковая определена, должна иметь то же значение.

В файле `<stdio.h>` стандарт ISO C определяет также константу `TMP_MAX`. Это максимальное количество уникальных имен файла, которые могут быть сгенерированы функцией `tmpnam`. Более подробно мы поговорим об этом в разделе 5.13.

Хотя стандарт ISO C определяет константу `FILENAME_MAX`, мы постараемся не использовать ее, потому что в стандарте POSIX.1 определена более удачная альтернатива (`NAME_MAX` и `PATH_MAX`). Мы познакомимся с этими константами чуть ниже. В табл. 2.7 приводятся значения `FOPEN_MAX` и `TMP_MAX` для всех четырех платформ, обсуждаемых в данной книге.

Таблица 2.7. Пределы, определяемые стандартом ISO для различных платформ

Предел	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>FOPEN_MAX</code>	20	16	20	20
<code>TMP_MAX</code>	308 915 776	238 328	308 915 776	17 576
<code>FILENAME_MAX</code>	1 024	4 096	1 024	1 024

2.5.2. Пределы POSIX

Стандарт POSIX.1 определяет многочисленные константы, связанные с предельными значениями. К сожалению, это один из самых запутанных аспектов POSIX.1. Хотя POSIX.1 определяет огромное количество констант и предельных значений, мы остановимся лишь на тех из них, которые затрагивают базовые интерфейсы POSIX.1. Эти пределы и константы делятся на следующие категории.

1. Пределы числовых значений: `LONG_BIT`, `SSIZE_MAX` и `WORD_BIT`.
2. Минимальные значения: 25 констант, перечисленных в табл. 2.8.
3. Максимальное значение: `_POSIX_CLOCKRES_MIN`.

4. Значения, которые можно увеличить во время выполнения: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX и RE_DUP_MAX.
5. Значения, не изменяемые во время выполнения, возможно, неопределенные: 17 констант, перечисленных в табл. 2.9 (плюс еще четыре константы, описываемые в разделе 12.2, и три константы, описываемые в разделе 14.5).
6. Другие неизменяемые значения: NL_ARGMAX, NL_MSGMAX, NL_SETMAX и NL_TEXTMAX.
7. Изменяемые значения, связанные с размером строки пути: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF и SYMLINK_MAX.

Таблица 2.8. Минимальные значения из файла `<limits.h>`, определяемые стандартом POSIX.1

Имя	Описание: минимально допустимое значение для максимального значения...	Значение
<code>_POSIX_ARG_MAX</code>	Длины аргументов функций <code>exec</code>	4096
<code>_POSIX_CHILD_MAX</code>	Количества дочерних процессов на реальный идентификатор пользователя	25
<code>_POSIX_DELAYTIMER_MAX</code>	Количества переполнений каждого таймера	32
<code>_POSIX_HOST_NAME_MAX</code>	Длины имени сетевого узла, возвращаемого функцией <code>gethostname</code>	255
<code>_POSIX_LINK_MAX</code>	Количества ссылок на один файл	8
<code>_POSIX_LOGIN_NAME_MAX</code>	Длины имени пользователя	9
<code>_POSIX_MAX_CANON</code>	Количества байтов в канонической входной очереди терминала	255
<code>_POSIX_MAX_INPUT</code>	Количества байтов, доступных во входной очереди терминала	255
<code>_POSIX_NAME_MAX</code>	Количества байтов в имени файла, за исключением завершающего нулевого символа	14
<code>_POSIX_NGROUPS_MAX</code>	Количества идентификаторов дополнительных групп на процесс	8
<code>_POSIX_OPEN_MAX</code>	Количества открытых файлов на процесс	20
<code>_POSIX_PATH_MAX</code>	Длины строки пути к файлу, включая завершающий нулевой символ	256
<code>_POSIX_PIPE_BUF</code>	Количества байтов, которые можно атомарно записать в канал	512
<code>_POSIX_RE_DUP_MAX</code>	Количества повторяющихся вхождений для основного регулярного выражения, принимаемого функциями <code>regexec</code> и <code>regcomp</code> , при использовании интервальной нотации <code>\{m, n\}</code>	255
<code>_POSIX_RTSIG_MAX</code>	Количества сигналов реального времени, зарезервированных для приложения	8
<code>_POSIX_SEM_NSEMS_MAX</code>	Количества семафоров, одновременно используемых процессом	256

Имя	Описание: минимально допустимое значение для максимального значения...	Значение
_POSIX_SEM_VALUE_MAX	Значения, которые может хранить семафор	32 767
_POSIX_SIGQUEUE_MAX	Количества сигналов, которые процесс может поставить в очередь ожидания обработки	32
_POSIX_SSIZE_MAX	Значения, которые можно сохранить в переменной типа <code>ssize_t</code>	32 767
_POSIX_STREAM_MAX	Количества стандартных потоков ввода/вывода на процесс	8
_POSIX_SYMLINK_MAX	Количества байтов в символьической ссылке	255
_POSIX_SYMLOOP_MAX	Количества переходов по символическим ссылкам, допустимого в строке пути	8
_POSIX_TIMER_MAX	Количества таймеров в процессе	32
_POSIX_TTY_NAME_MAX	Длины имени терминального устройства, включая завершающий нулевой символ	9
_POSIX_TZNAME_MAX	Количества байтов в имени временной зоны	6

Некоторые из этих пределов и констант могут быть определены в файле `<limits.h>`, другие могут быть не определены — в зависимости от некоторых условий. Пределы и константы, которые необязательно должны быть определены, мы рассмотрим в разделе 2.5.4, когда будем говорить о функциях `sysconf`, `pathconf` и `fpathconf`.

В табл. 2.8 перечислено 25 неизменяемых минимальных значений. Эти значения являются неизменяемыми и не зависят от конкретной реализации операционной системы. Они задают самые строгие ограничения на функциональные возможности. Реализации, претендующие на звание POSIX-совместимых, должны обеспечивать значения не ниже указанных. Именно поэтому они называются минимально допустимыми, хотя в их именах присутствует окончание `MAX`. Кроме того, чтобы обеспечить максимальную переносимость, приложения, строго следующие стандарту, не должны требовать более высоких значений. Описания этих констант мы будем приводить по мере знакомства с ними.

Приложения, строго следующие стандарту POSIX, отличаются от просто POSIX-совместимых приложений. Последние используют только интерфейсы, определяемые стандартом IEEE Standard 1003.1-2008. Приложение, строго следующее стандарту, — это POSIX-совместимое приложение, которое не полагается ни на какое не определенное в стандарте поведение, не использует никаких устаревающих интерфейсов и не требует значений констант больших, чем минимумы, перечисленные в табл. 2.8.

К сожалению, некоторые из этих неизменяемых минимальных значений слишком малы, чтобы использоваться на практике. Например, большинство современных UNIX-систем позволяет открывать намного больше 20 файлов на процесс. Минимальный предел 256 для `_POSIX_PATH_MAX` также слишком мал. Длина строки пути может превысить это значение, то есть мы не можем использовать константы `_POSIX_OPEN_MAX` и `_POSIX_PATH_MAX` в качестве размеров массивов на этапе компиляции.

Таблица 2.9. Значения времени выполнения из файла `<limits.h>`, определяемые стандартом POSIX.1

Имя предела	Описание	Минимально допустимое значение
<code>ARG_MAX</code>	Максимальная длина аргументов функций семейства <code>exec</code> (в байтах)	<code>_POSIX_ARG_MAX</code>
<code>ATEXIT_MAX</code>	Максимальное количество функций, которые можно зарегистрировать с помощью функции <code>atexit</code>	32
<code>CHILD_MAX</code>	Максимальное число дочерних процессов на один реальный идентификатор пользователя	<code>_POSIX_CHILD_MAX</code>
<code>DELAYTIMER_MAX</code>	Максимальное количество переполнений таймера	<code>_POSIX_DELAYTIMER_MAX</code>
<code>HOST_NAME_MAX</code>	Максимальная длина имени сетевого узла, возвращаемого функцией <code>gethostname</code>	<code>_POSIX_HOST_NAME_MAX</code>
<code>LOGIN_NAME_MAX</code>	Максимальная длина имени пользователя	<code>_POSIX_LOGIN_NAME_MAX</code>
<code>OPEN_MAX</code>	Значение, на единицу больше максимального, которое можно присвоить файловому дескриптору	<code>_POSIX_OPEN_MAX</code>
<code>PAGESIZE</code>	Размер страницы памяти системы (в байтах)	1
<code>RTSIG_MAX</code>	Максимальное количество сигналов реального времени, зарезервированных для приложения	<code>_POSIX_RTSIG_MAX</code>
<code>SEM_NSEMS_MAX</code>	Максимальное количество семафоров, одновременно используемых процессом	<code>_POSIX_SEM_NSEMS_MAX</code>
<code>SEM_VALUE_MAX</code>	Максимальное значение семафора	<code>_POSIX_SEM_VALUE_MAX</code>
<code>SIGQUEUE_MAX</code>	Максимальное количество сигналов, которые процесс может поставить в очередь	<code>_POSIX_SIGQUEUE_MAX</code>
<code>STREAM_MAX</code>	Максимальное число стандартных потоков ввода/вывода, которые процесс может открыть одновременно	<code>_POSIX_STREAM_MAX</code>
<code>SYMLINK_MAX</code>	Максимальное количество символьических ссылок, которые можно пройти в процессе анализа пути к файлу	<code>_POSIX_SYMLINK_MAX</code>
<code>TIMER_MAX</code>	Максимальное количество таймеров в процессе	<code>_POSIX_TIMER_MAX</code>
<code>TTY_NAME_MAX</code>	Максимальная длина имени терминального устройства, включая завершающий нулевой символ	<code>_POSIX_TTY_NAME_MAX</code>
<code>TZNAME_MAX</code>	Количество байтов в имени временной зоны	<code>_POSIX_TZNAME_MAX</code>

В табл. 2.8 каждому из 25 неизменяемых минимальных значений соответствует значение, зависящее от реализации, имя которого отличается отсутствием приставки `_POSIX_`. Константы без приставки `_POSIX_` предназначены для хранения

фактических значений, поддерживаемых конкретной реализацией. (Эти 25 констант, значения которых определяются реализацией, перечислены в пунктах 1, 4, 5 и 7 списка, приведенного выше: 2 значения, которые можно увеличить во время выполнения, 15 неизменяемых значений времени выполнения и 7 изменяемых значений, связанных с размером строки пути.) Основная проблема в том, что не все 25 значений, зависящих от реализации, обязательно будут определены в заголовочном файле `<limits.h>`.

Например, определение конкретного значения может отсутствовать в заголовочном файле, если его фактическая величина для данного процесса зависит от объема памяти в системе. Если значения не определены в заголовочном файле, их нельзя использовать для задания границ массивов на этапе компиляции. Поэтому стандарт POSIX.1 определяет функции `sysconf`, `pathconf` и `fpathconf`, с помощью которых можно определить фактические значения пределов во время выполнения. Однако существует еще одна проблема: некоторые значения определены стандартом POSIX.1 как «возможно неопределенные» (следовательно, бесконечные). Это означает, что на практике значение не имеет верхней границы. Для Solaris, например, количество функций, которые можно зарегистрировать с помощью `atexit` для вызова по завершении процесса, ограничено только объемом доступной памяти. Поэтому предел `ATEXIT_MAX` для Solaris считается не определенным. Мы еще вернемся к этой проблеме в разделе 2.5.5.

2.5.3. Пределы XSI

Стандарт XSI тоже определяет ряд констант, значения которых зависят от реализации.

1. Минимальные значения: 5 констант, перечисленных в табл. 2.10.
2. Числовые неизменяемые пределы времени выполнения, возможно неопределенные: `IOV_MAX` и `PAGE_SIZE`.

Минимальные значения перечислены в табл. 2.10. Две последние константы иллюстрируют ситуацию, когда минимумы, объявленные в стандарте POSIX.1, слишком

Таблица 2.10. Минимальные значения из файла `<limits.h>`, определяемые стандартом XSI

Имя	Описание	Минимально допустимое значение	Типовое значение
<code>NL_LANGMAX</code>	Максимальный размер переменной окружения <code>LANG</code> в байтах	14	14
<code>NZERO</code>	Приоритет процесса по умолчанию	20	20
<code>_XOPEN_IOV_MAX</code>	Максимальное количество структур <code>iovec</code> , которое можно передать функциям <code>readv</code> и <code>writev</code>	16	16
<code>_XOPEN_NAME_MAX</code>	Максимальная длина имени файла (в байтах)	255	255
<code>_XOPEN_PATH_MAX</code>	Максимальная длина строки пути к файлу (в байтах)	1024	1024

малы (вероятно, чтобы сделать возможной реализацию POSIX-совместимых систем для встраиваемых устройств), поэтому для использования в XSI-совместимых системах были добавлены увеличенные минимальные значения.

2.5.4. Функции `sysconf`, `pathconf` и `fpathconf`

Мы перечислили различные минимальные значения, которые должны поддерживаться реализацией, но как узнать фактические пределы, которые поддерживает конкретная система? Как уже упоминалось выше, некоторые из этих пределов можно определить на этапе компиляции, другие — во время выполнения. Мы также говорили, что некоторые из них являются неизменяемыми в данной системе, тогда как другие, связанные с файлами или каталогами, могут изменяться. На этапе выполнения значения пределов можно получить с помощью одной из следующих трех функций.

```
#include <unistd.h>
long sysconf(int name);
long pathconf(const char *pathname, int name);
long fpathconf(int fd, int name);
```

Все три возвращают значение соответствующего предела в случае успеха или `-1` в случае ошибки (см. ниже)

Различие между двумя последними функциями состоит в том, что первая получает в аргументе строку пути к файлу, а вторая — файловый дескриптор.

В табл. 2.11 перечисляются значения аргумента *name*, которые можно передать функции `sysconf` для идентификации пределов времени выполнения. Для идентификации пределов этой функции передаются константы, имена которых начинаются с префикса `_SC_`. В табл. 2.12 перечисляются значения аргумента *name* для функций `pathconf` и `fpathconf`. Для идентификации пределов времени выполнения этим функциям передаются константы, имена которых начинаются с префикса `_PC_`.

Таблица 2.11. Пределы и идентификаторы для аргумента *name* функции `sysconf`

Имя предела	Описание	Аргумент <i>name</i>
<code>ARG_MAX</code>	Максимальная длина аргументов функций семейства <code>exec</code> (в байтах)	<code>_SC_ARG_MAX</code>
<code>ATEXIT_MAX</code>	Максимальное количество функций, которые можно зарегистрировать с помощью функции <code>atexit</code>	<code>_SC_ATEXIT_MAX</code>
<code>CHILD_MAX</code>	Максимальное число процессов на один реальный идентификатор пользователя	<code>_SC_CHILD_MAX</code>

Имя предела	Описание	Аргумент <code>name</code>
Количество тактов системных часов в секунду	Количество тактов системных часов в секунду	<code>_SC_CLK_TCK</code>
<code>COLL_WEIGHTS_MAX</code>	Максимальное количество весовых коэффициентов для одного элемента категории <code>LC_COLLATE</code> в файле региональных настроек	<code>_SC_COLL_WEIGHTS_MAX</code>
<code>DELAYTIMER_MAX</code>	Максимальное количество переполнений таймера	<code>_SC_DELAYTIMER_MAX</code>
<code>HOST_NAME_MAX</code>	Максимальная длина имени сетевого узла, возвращаемого функцией <code>gethostname</code>	<code>_SC_HOST_NAME_MAX</code>
<code>IOV_MAX</code>	Максимальное количество структур <code>iovec</code> , которое можно передать функциям <code>readv</code> и <code>writev</code>	<code>_SC_IOV_MAX</code>
<code>LINE_MAX</code>	Максимальная длина строки ввода, принимаемой утилитами	<code>_SC_LINE_MAX</code>
<code>LOGIN_NAME_MAX</code>	Максимальная длина имени пользователя	<code>_SC_LOGIN_NAME_MAX</code>
<code>NGROUPS_MAX</code>	Максимальное количество идентификаторов дополнительных групп на процесс	<code>_SC_NGROUPS_MAX</code>
<code>OPEN_MAX</code>	Значение, на единицу большее максимального, которое можно присвоить файловому дескриптору	<code>_SC_OPEN_MAX</code>
<code>PAGESIZE</code>	Системный размер страницы памяти (в байтах)	<code>_SC_PAGESIZE</code>
<code>PAGE_SIZE</code>	Системный размер страницы памяти (в байтах)	<code>_SC_PAGE_SIZE</code>
<code>RE_DUP_MAX</code>	Максимальное количество повторяющихся вхождений для основного регулярного выражения, принимаемого функциями <code>regexec</code> и <code>regcomp</code> , при использовании интервальной нотации <code>\{m, n\}</code>	<code>_SC_RE_DUP_MAX</code>
<code>RTSIG_MAX</code>	Максимальное количество сигналов реального времени, зарезервированных для приложения	<code>_SC_RTSIG_MAX</code>
<code>SEM_NSEMS_MAX</code>	Максимальное количество семафоров, одновременно используемых процессом	<code>_SC_SEM_NSEMS_MAX</code>
<code>SEM_VALUE_MAX</code>	Максимальное значение семафора	<code>_SC_SEM_VALUE_MAX</code>
<code>SIGQUEUE_MAX</code>	Максимальное количество сигналов, которые процесс может поставить в очередь	<code>_SC_SIGQUEUE_MAX</code>
<code>STREAM_MAX</code>	Максимальное число стандартных потоков ввода/вывода на процесс в любой конкретный момент времени; если определен, должен иметь значение, равное <code>FOPEN_MAX</code>	<code>_SC_STREAM_MAX</code>
<code>SYMLOOP_MAX</code>	Максимальное количество символьических ссылок, которые можно пройти в процессе анализа пути к файлу	<code>_SC_SYMLOOP_MAX</code>

Таблица 2.11 (окончание)

Имя предела	Описание	Аргумент name
TIMER_MAX	Максимальное количество таймеров в процессе	_SC_TIMER_MAX
TTY_NAME_MAX	Максимальная длина имени терминального устройства, включая завершающий нулевой символ	_SC_TTY_NAME_MAX
TZNAME_MAX	Количество байтов в имени временной зоны	_SC_TZNAME_MAX

Таблица 2.12. Пределы и идентификаторы для аргумента name функций pathconf и fpathconf

Имя предела	Описание	Аргумент name
FILESIZEBITS	Минимальное количество битов, необходимое для представления максимального размера обычного файла, допустимого для заданного каталога, в виде целого значения со знаком	_PC_FILESIZEBITS
LINK_MAX	Максимальное значение счетчика ссылок на один файл	_PC_LINK_MAX
MAX_CANON	Максимальное количество байтов в канонической входной очереди терминала	_PC_MAX_CANON
MAX_INPUT	Количества байтов, доступных во входной очереди терминала	_PC_MAX_INPUT
NAME_MAX	Максимальная длина имени файла в байтах (за исключением завершающего нулевого символа)	_PC_NAME_MAX
PATH_MAX	Максимальная длина строки пути к файлу, включая завершающий нулевой символ	_PC_PATH_MAX
PIPE_BUF	Максимальное количество байтов, которые можно записать в канал атомарно	_PC_PIPE_BUF
_POSIX_TIMESTAMP_RESOLUTION	Точность в наносекундах представления времени в атрибутах файлов	_PC_TIMESTAMP_RESOLUTION
SYMLINK_MAX	Количество байтов в символьской ссылке	_PC_SYMLINK_MAX

Рассмотрим внимательнее разные значения, возвращаемые этими тремя функциями.

1. Все три функции возвращают значение -1 и код ошибки `EINVAL` в переменной `errno`, если аргумент `name` содержит имя неподдерживаемого предела. В третьей колонке табл. 2.11 и табл. 2.12 даны имена пределов, которые будут использоваться на протяжении всей книги.
2. Для некоторых пределов могут возвращаться определенные числовые значения (≥ 0) либо признак неопределенности — значение -1 , но при этом значение `errno` не изменяется.
3. Значение предела `_SC_CLK_TCK` — количество тактов системных часов в секунду; эта величина используется при работе со значениями, возвращаемыми функцией `times` (раздел 8.17).

Ниже перечислены ограничения, накладываемые на аргумент `pathname` функции `pathconf` и аргумент `fd` функции `fpathconf`. Несоблюдение любого из этих ограничений может привести к непредсказуемым результатам.

1. Файл, к которому относятся параметры `_PC_MAX_CANON` и `_PC_MAX_INPUT`, должен быть файлом терминального устройства.
2. Файл, к которому относятся параметры `_PC_LINK_MAX` и `_PC_TIMESTAMP_RESOLUTION`, должен быть файлом или каталогом. Значение, возвращаемое для каталога, применимо только к самому каталогу, но не к файлам, находящимся в нем.
3. Файл, к которому относятся параметры `_PC_FILESIZEBITS` и `_PC_NAME_MAX`, должен быть каталогом. Возвращаемое значение относится к именам файлов в этом каталоге.
4. Файл, к которому относится параметр `_PC_PATH_MAX`, должен быть каталогом. Возвращаемое значение представляет максимальную длину относительного пути, когда заданный каталог является рабочим каталогом. (К сожалению, эта величина не отражает фактическую максимальную длину абсолютного пути, которую мы в действительности хотим узнать. Мы еще вернемся к этой проблеме в разделе 2.5.5.)
5. Файл, к которому относится параметр `_PC_PIPE_BUF`, должен быть неименованным каналом, именованным каналом или каталогом. В первых двух случаях возвращаемое значение относится к самим каналам. В случае каталога ограничение относится к любым именованным каналам, созданным в этом каталоге.
6. Файл, к которому относится параметр `_PC_SYMLINK_MAX`, должен быть каталогом. Возвращаемое значение — максимальная длина строки, которую может хранить символьская ссылка в этом каталоге.

Пример

Программа на языке `awk`(1), представленная в листинге 2.1, генерирует программу на языке C, которая, в свою очередь, выводит значения всех идентификаторов функций `pathconf` и `sysconf`.

Листинг 2.1. Генерация программы на языке C, которая выводит значения всех конфигурационных ограничений

```
#!/usr/bin/awk -f
BEGIN {
    printf("#include \"apue.h\"\n")
    printf("#include <errno.h>\n")
    printf("#include <limits.h>\n")
    printf("\n")
    printf("static void pr_sysconf(char *, int);\n")
    printf("static void pr_pathconf(char *, char *, int);\n")
    printf("\n")
    printf("int\n")
    printf("main(int argc, char *argv[])\n")
    printf("{\n")
    printf("\tif (argc != 2)\n")
    printf("\tterr_quit(\"Использование: a.out <dirname>\");\n")
    FS="\t"
    while (getline <"sysconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s определен как %ld\\n\", (long)%s+0);\\n",
               $1, $1)
        printf("#else\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\\n", $1)
        printf("#endif\\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_sysconf(\"%s =\", %s);\\n", $1, $2)
        printf("#else\\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\\n", $2)
        printf("#endif\\n")
    }
    close("sysconf.sym")
    while (getline <"pathconf.sym" > 0) {
        printf("#ifdef %s\n", $1)
        printf("\tprintf(\"%s определен как %ld\\n\", (long)%s+0);\\n",
               $1, $1)
        printf("#else\\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\\n", $1)
        printf("#endif\\n")
        printf("#ifdef %s\n", $2)
        printf("\tpr_pathconf(\"%s =\", argv[1], %s);\\n", $1, $2)
        printf("#else\\n")
        printf("\tprintf(\"идентификатор %s не найден\\n\");\\n", $2)
        printf("#endif\\n")
    }
    close("pathconf.sym")
    exit
}
END {
    printf("\texit(0);\\n")
    printf("{}\\n\\n")
    printf("static void\\n")
    printf("pr_sysconf(char *mesg, int name)\\n")
    printf("{\\n")
    printf("\tlong val;\\n\\n")
    printf("\tfputs(mesg, stdout);\\n")
    printf("\terrno = 0;\\n")
    printf("\tif ((val = sysconf(name)) < 0) {\\n")

```

```

printf("\t\tif (errno != 0) {\n")
printf("\t\t\tif (errno == EINVAL)\n")
printf("\t\t\t\ttfputs(\" (не поддерживается)\\n\", stdout);\n")
printf("\t\t\telse\n")
printf("\t\t\t\tterr_sys(\"ошибка вызова sysconf\");\n")
printf("\t\t\telse {\n")
printf("\t\t\t\ttfputs(\" (нет ограничений)\\n\", stdout);\n")
printf("\t\t\t}\n")
printf("\t\t}\n")
printf("static void\n")
printf("pr_pathconf(char *mesg, char *path, int name)\n")
printf("{\n")
printf("\tlong val;\n")
printf("\n")
printf("\tfputs(mesg, stdout);\n")
printf("\terrno = 0;\n")
printf("\tif ((val = pathconf(path, name)) < 0) {\n")
printf("\t\tif (errno != 0) {\n")
printf("\t\t\tif (errno == EINVAL)\n")
printf("\t\t\t\ttfputs(\" (не поддерживается)\\n\", stdout);\n")
printf("\t\t\telse\n")
printf("\t\t\t\tterr_sys(\"ошибка вызова pathconf, path = %%s\", path);\n")
printf("\t\t\t}\n")
printf("\t\tprintf(\" (нет ограничений)\\n\", stdout);\n")
printf("\t\t}\n")
printf("\t\telse {\n")
printf("\t\t\tprintf(\" %%ld\\n\", val);\n")
printf("\t\t}\n")
printf("}\n")
}

```

Программа на языке awk читает два входных файла — `pathconf.sym` и `sysconf.sym` — с перечнем пределов и идентификаторов, разделенных символами табуляции. Не на каждой платформе определены все идентификаторы, поэтому программа на языке awk заключает все вызовы `pathconf` и `sysconf` в директивы условной компиляции `#ifdef`.

Например, программа трансформирует строку входного файла, которая выглядит следующим образом:

```
NAME_MAX      _PC_NAME_MAX
```

в следующий код на языке C:

```

#ifndef NAME_MAX
    printf("NAME_MAX определен как %d\n", NAME_MAX+0);
#else
    printf("идентификатор NAME_MAX не найден\n");
#endif
#ifndef _PC_NAME_MAX
    pr_pathconf("NAME_MAX =", argv[1], _PC_NAME_MAX);
#else
    printf("идентификатор _PC_NAME_MAX не найден\n");
#endif

```

Программа в листинге 2.2 сгенерирована предыдущей программой на awk. Она выводит значения всех пределов, корректно обрабатывая случаи, когда идентификатор не определен.

Листинг 2.2. Вывод всех возможных значений sysconf и pathconf

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("Использование: a.out <каталог>");

#ifndef ARG_MAX
    printf("ARG_MAX определен как %d\n", ARG_MAX+0);
#else
    printf("идентификатор ARG_MAX не найден\n");
#endif
#ifndef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("идентификатор _SC_ARG_MAX не найден\n");
#endif

/* аналогично обрабатываются остальные идентификаторы sysconf... */

#ifndef MAX_CANON
    printf("MAX_CANON определен как %d\n", MAX_CANON+0);
#else
    printf("идентификатор MAX_CANON не найден\n");
#endif
#ifndef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("идентификатор _PC_MAX_CANON не найден\n");
#endif

/* аналогично обрабатываются остальные идентификаторы pathconf... */
    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
```

```

        fputs(" (не поддерживается)\n", stdout);
    else
        err_sys("ошибка вызова sysconf");
} else {
    fputs(" (нет ограничений)\n", stdout);
}
} else {
    printf("%ld\n", val);
}
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long      val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (не поддерживается)\n", stdout);
            else
                err_sys("ошибка вызова pathconf, path = %s", path);
        } else {
            fputs(" (нет ограничений)\n", stdout);
        }
    } else {
        printf("%ld\n", val);
    }
}
}

```

В табл. 2.13 приводятся результаты работы программы из листинга 2.2, в каждой из четырех систем, обсуждаемых в данной книге. «Нет идентификатора» означает, что данная платформа не имеет соответствующего идентификатора _SC или _PC, с помощью которого можно было бы узнать значение константы. В этом случае предел считается неопределенным. В противоположность этому обозначение «Не поддерживается» говорит о том, что идентификатор определен, но он не распознается функциями *pathconf* и *sysconf*. «Нет ограничений» означает, что система не задает этот предел, но это не значит, что предела нет вообще.

Следует отметить, что для некоторых пределов могут возвращаться некорректные значения. Например, в Linux для предела SYMLOOP_MAX сообщается, что он не ограничен, но, исследовав исходные тексты, можно увидеть, что в действительности количество переходов по символическим ссылкам ограничивается «жестко зашитым» значением 40 (функция follow_link в файле fs/namei.c).

Еще одна причина неточностей в Linux обусловлена тем, что функции pathconf и fpathconf реализованы в библиотеке языка C. Значения пределов, возвращаемые этими функциями, зависят от фактически используемой файловой системы, поэтому, если текущая файловая система неизвестна библиотеке, функции возвращают предполагаемое значение.

В разделе 4.14 мы увидим, что UFS – это реализация файловой системы Berkeley Fast File System для SVR4, а PCFS – реализация файловой системы MS-DOS FAT для Solaris.

Таблица 2.13. Примеры конфигурационных пределов

Предел	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				файловая система UFS	файловая система PCFS
ARG_MAX	262 144	2 097 152	262 144	2 096 640	2 096 640
ATEXIT_MAX	32	2 147 483 647	2 147 483 647	Нет огра- ничений	Нет огра- ничений
CHARCLASS_NAME_MAX	Нет иденти- фикатора	2 048	14	14	14
CHILD_MAX	1 760	47 211	266	8 021	8021
Количество тактов системных часов в секунду	128	100	100	100	100
COLL_WEIGHTS_MAX	0	255	2	10	10
FILESIZEBITS	64	64	64	41	Не поддер- живается
HOST_NAME_MAX	255	64	255	255	255
IOV_MAX	1024	1024	1024	16	16
LINE_MAX	2048	2048	2048	2048	2048
LINK_MAX	32 768	65 000	32 767	32 767	1
LOGIN_NAME_MAX	17	256	256	9	9
MAX_CANON	255	255	1024	256	256
MAX_INPUT	255	255	1024	512	512
NAME_MAX	255	255	255	255	8
NGROUPS_MAX	1023	65 536	16	16	16
OPEN_MAX	3520	1024	256	256	256
PAGESIZE	4096	4096	4096	8192	8192
PAGE_SIZE	4096	4096	4096	8192	8192
PATH_MAX	1024	4096	1024	1024	1024
PIPE_BUF	512	4096	512	5120	5120
RE_DUP_MAX	255	32 767	255	255	255
STREAM_MAX	3520	16	20	256	256
SYMLINK_MAX	1024	Нет ограни- чений	255	1024	1024
SYMLOOP_MAX	32	Нет ограни- чений	32	20	20
TTY_NAME_MAX	255	32	255	128	128
TZNAME_MAX	255	6	255	Нет огра- ничений	Нет огра- ничений

2.5.5. Неопределенные пределы времени выполнения

Выше упоминалось, что некоторые пределы могут быть не определены. Проблема в том, что если они не определены в заголовочном файле `<limits.h>`, их нельзя использовать на этапе компиляции. Но они могут оставаться неопределенными даже во время выполнения! Давайте рассмотрим два конкретных случая: размещение в памяти строки пути и определение количества файловых дескрипторов.

Строка пути

Многим программам приходится выделять память для хранения строки пути. Обычно память выделяется на этапе компиляции; в этом случае в качестве размеров массивов выбираются некоторые «магические» числа (немногие из которых верны), например 256, 512, 1024 или стандартная константа `BUFSIZ`. В операционной системе 4.3BSD правильное значение представляет константа `MAXPATHLEN`, определяемая в заголовочном файле `<sys/param.h>`, но большинство приложений, написанных под 4.3BSD, ее не используют.

Для таких случаев стандартом POSIX.1 предусматривается константа `PATH_MAX`, но ее значение может оказаться неопределенным. В листинге 2.3 приводится функция, которая будет использоваться в этой книге для определения объема памяти, необходимого для размещения строки пути.

Если константа `PATH_MAX` определена в файле `<limits.h>`, используется ее значение. Если нет, необходимо вызвать функцию `pathconf`. Значение, возвращаемое этой функцией, — это максимальный размер строки относительного пути для случая, когда первый аргумент является рабочим каталогом, поэтому мы указываем в первом аргументе корневой каталог и прибавляем к полученному результату единицу. Если `pathconf` сообщает, что константа `PATH_MAX` не определена, остается лишь надеяться на удачу и выбрать достаточно большое число самостоятельно.

Версии POSIX.1, до выхода редакции 2001 года, не уточняли, должна ли константа `PATH_MAX` учитывать завершающий нулевой символ в конце строки пути. Если реализация операционной системы соответствует одному из этих ранних стандартов и не соответствует какой-либо версии Single UNIX Specification (требующей учитывать наличие нулевого символа), следует на всякий случай добавить единицу к полученному объему памяти.

Выбор того или иного алгоритма в случае неопределенного результата зависит от того, как используется выделяемая память. Если память выделяется для вызова функции `getcwd`, например, чтобы получить абсолютное имя рабочего каталога (раздел 4.23), тогда, если выделенный объем памяти окажется слишком мал, мы получим признак ошибки и код `ERANGE` в `errno`. В такой ситуации можно увеличить объем памяти, выделенной под строку, вызвав функцию `realloc` (раздел 7.8 и упражнение 4.16), и повторить попытку. При необходимости можно продолжать увеличивать размер строки, пока вызов `getcwd` не завершится успехом.

Листинг 2.3. Динамическое выделение памяти для строки пути

```
#include "apue.h"
#include <errno.h>
#include <limits.h>
```

```

#define PATH_MAX
static long pathmax = PATH_MAX;
#else
static long pathmax = 0;
#endif

static long posix_version = 0;
static long xsi_version = 0;

/* Если константа PATH_MAX не определена */
/* адекватность следующего числа не гарантируется */
#define PATH_MAX_GUESS 1024

char *
path_alloc(size_t *sizep) /* если удалось выделить память, */
{                         /* возвращает также выделенный объем */
    char      *ptr;
    size_t   size;

    if (posix_version == 0)
        posix_version = sysconf(_SC_VERSION);

    if (xsi_version == 0)
        xsi_version = sysconf(_SC_XOPEN_VERSION);

    if (pathmax == 0) { /* первый вызов функции */
        errno = 0;
        if ((pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS; /* если константа не определена */
            else
                err_sys("ошибка вызова pathconf с параметром _PC_PATH_MAX");
        } else {
            pathmax++; /* добавить 1, так как путь относительно корня */
        }
    }

    /*
     * До версии POSIX.1-2001 не гарантируется, что PATH_MAX включает
     * завершающий нулевой байт. То же для XPG3.
     */
    if ((posix_version < 200112L) && (xsi_version < 4))
        size = pathmax + 1;
    else
        size = pathmax;

    if ((ptr = malloc(size)) == NULL)
        err_sys("malloc error for pathname");

    if (sizep != NULL)
        *sizep = size;
    return(ptr);
}

```

Максимальное количество открытых файлов

Как правило, процесс-демон, то есть процесс, который выполняется в фоновом режиме и не связан с терминальным устройством, закрывает все открытые файлы. Некоторые программы предусматривают следующую последовательность дей-

ствий, исходя из предположения, что в заголовочном файле `<sys/param.h>` определена константа `NOFILE`:

```
#include <sys/param.h>
for (i = 0; i < NOFILE; i++)
    close(i);
```

В других программах используется константа `_NFILE`, которая определена в некоторых версиях `<stdio.h>` как верхний предел. В третьих в качестве верхнего предела жестко задано число 20. Однако ни один из этих способов не переносим. Мы могли бы надеяться на константу `OPEN_MAX`, определяемую стандартом POSIX.1, чтобы переносимым образом узнать значение этого предела, но если она не определена, проблема останется нерешенной. Например, цикл в следующем фрагменте не выполнится ни разу, если `OPEN_MAX` не определена, так как `sysconf` вернет `-1`:

```
#include <unistd.h>
for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

Лучшее, что можно предпринять в такой ситуации, — закрыть все дескрипторы до некоторого произвольного предела, например 256. Как и в случае со строкой пути, такой подход не гарантирует желаемого результата во всех возможных случаях, но это лучшее, что можно сделать. Продемонстрируем его в листинге 2.4.

Листинг 2.4. Определение количества файловых дескрипторов

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifndef OPEN_MAX
static long openmax = OPEN_MAX;
#else
static long openmax = 0;
#endif

/*
 * Если константа PATH_MAX не определена */
 * адекватность следующего числа не гарантируется
 */
#define OPEN_MAX_GUESS 256

long
open_max(void)
{
    if (openmax == 0) { /* первый вызов функции */
        errno = 0;
        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS; /* неопределенный предел */
            else
                err_sys("ошибка вызова sysconf с параметром _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

Легко поддаться искушению просто вызывать функцию `close`, пока она не вернет признак ошибки, но дело в том, что по коду ошибки `EBADF`, которую возвращает `close`, нельзя сказать, была это попытка закрыть неправильный дескриптор или дескриптор просто не был открыт. Если бы мы реализовали такой алгоритм, в случае, когда дескриптор 10 был бы открыт, а дескриптор 9 — нет, выполнение цикла остановилось бы на дескрипторе 9 и дескриптор 10 остался бы незакрытым. С другой стороны, функция `dup` (раздел 3.12) возвращает признак ошибки, если превышен предел `OPEN_MAX`, но создание сотен копий дескриптора — слишком экстремальный способ выяснения значения искомого предела.

Некоторые реализации возвращают значение `LONG_MAX` для пределов, которые в действительности неограничены. Так обстоит дело с пределом `ATEXIT_MAX` в Linux (табл. 2.13). Вообще такой подход нельзя назвать приемлемым, потому что он может привести к непредсказуемой работе программ.

Например, с помощью команды `ulimit`, встроенной в командный интерпретатор Bourne-again shell, можно изменить максимальное количество открытых файлов на процесс. Вообще, если предел фактически не ограничен, для выполнения этой операции потребуются привилегии суперпользователя. Но если сделать верхний предел практически неограниченным, функция `sysconf` будет возвращать `LONG_MAX` в качестве предела `OPEN_MAX`. Тогда программа, которая ориентируется на значение верхнего предела, как в листинге 2.4, будет затрачивать огромное количество времени на попытки закрыть 2 147 483 647 дескрипторов, большинство из которых даже не открывались.

Системы, поддерживающие расширения XSI стандарта Single UNIX Specification, предоставляют функцию `getrlimit(2)` (раздел 7.11). Она может использоваться для получения максимального количества открытых дескрипторов на процесс. Таким способом можно узнать, определено ли ограничение на количество открытых файлов, и избежать дальнейших проблем.

Значение `OPEN_MAX`, согласно определению в стандарте POSIX, относится к разряду неизменяемых во время выполнения. Это значит, что данный предел не изменяется в течение всей жизни процесса. Однако в системах, поддерживающих расширения XSI, его можно изменить вызовом функции `setrlimit(2)` (раздел 7.11). (Значение этого предела также можно изменить командой `limit` оболочки C shell или командой `ulimit` оболочек Bourne, Bourne-again, Debian Almquist и Korn.) В системах, поддерживающих данную возможность, функцию в листинге 2.4 можно было бы изменить так, чтобы она вызывала `sysconf` при каждом обращении к ней, а не только в первом вызове.

2.6. Необязательные параметры

Мы уже видели список необязательных параметров, определяемых стандартом POSIX.1 (табл. 2.5), и обсуждали необязательные категории XSI в разделе 2.2.3. Для написания переносимых приложений, зависящих от любой из этих необязательных особенностей, необходим переносимый способ определения поддержки заданного необязательного параметра.

Так же как в случае с пределами (раздел 2.5), POSIX.1 определяет три способа сделать это:

1. Параметры времени компиляции определяются в файле `<unistd.h>`.
2. Параметры времени выполнения, не связанные с файлами или каталогами, идентифицируются функцией `sysconf`.
3. Параметры времени выполнения, связанные с файлами или каталогами, можно получить с помощью функции `pathconf` или `fpathconf`.

В перечень необязательных параметров входят символьные константы из третьей колонки табл. 2.5, а также символьные константы из табл. 2.14 и 2.15. Если символьная константа не определена, мы должны использовать функции `sysconf`, `pathconf` или `fpathconf`, чтобы узнать, поддерживается ли заданный необязательный параметр. В этом случае через аргумент `name` функции передается имя, сформированное заменой префикса `_POSIX` на `_SC` или `_PC`. Для констант с именами, начинающимися с префикса `_XOPEN`, в аргументе `name` передается идентификатор, сформированный путем добавления префикса `_SC` или `_PC`. Например, если константа `_POSIX_RAW_SOCKETS` не определена, чтобы узнать, поддерживает ли система низкоуровневые сокеты, можно вызвать функцию `sysconf`, передав ей в аргументе `name` идентификатор `_SC_RAW_SOCKETS`. Далее, если константа `_XOPEN_UNIX` не определена, чтобы узнать, поддерживает ли система расширения XSI, можно вызвать функцию `sysconf`, передав ей в аргументе `name` идентификатор `_SC_XOPEN_UNIX`.

Таблица 2.14. Необязательные параметры и идентификаторы для функций `pathconf` и `fpathconf`

Имя параметра	Описание	Аргумент <code>name</code>
<code>_POSIX_CHOWN_RESTRICTED</code>	Указывает, ограничено ли действие функции <code>chown</code>	<code>_PC_CHOWN_RESTRICTED</code>
<code>_POSIX_NO_TRUNC</code>	Указывает, приводит ли к ошибке применение путей длиннее, чем <code>NAME_MAX</code>	<code>_PC_NO_TRUNC</code>
<code>_POSIX_VDISABLE</code>	Если определен, действие специальных терминальных символов может быть запрещено этим значением	<code>_PC_VDISABLE</code>
<code>_POSIX_ASYNC_IO</code>	Указывает, поддерживаются ли операции асинхронного ввода/вывода для заданного файла	<code>_PC_ASYNC_IO</code>
<code>_POSIX_PRIO_IO</code>	Указывает, поддерживаются ли приоритетные операции ввода/вывода для заданного файла	<code>_PC_PRIO_IO</code>
<code>_POSIX_SYNC_IO</code>	Указывает, поддерживаются ли операции синхронизированного ввода/вывода для заданного файла	<code>_PC_SYNC_IO</code>
<code>_POSIX2_SYMLINKS</code>	Указывает, поддерживаются ли символические ссылки для каталогов	<code>_PC_2_SYMLINKS</code>

Для каждого параметра есть три возможных варианта определения его поддержки системой.

1. Если символьная константа не определена или определена со значением `-1`, значит, данная функциональная возможность не поддерживается системой на этапе компиляции. Старое приложение может выполняться в более новой системе, где эта возможность поддерживается, поэтому проверка во время выполнения может показать наличие поддержки, даже если она была недоступна на этапе компиляции.
2. Если символьическая константа определена и имеет значение больше нуля, значит, данная функциональная возможность поддерживается.
3. Если символьическая константа определена и имеет значение, равное нулю, значит, следует использовать функции `sysconf`, `pathconf` или `fpathconf`, чтобы узнать, поддерживается ли заданная функциональная возможность.

В табл. 2.14 перечислены константы, используемые в вызовах функций `pathconf` и `fpathconf`. В табл. 2.15 перечислены неустаревшие параметры и соответствующие им константы, используемые при обращении к функции `sysconf`, в дополнение к перечисленным в табл. 2.5. Обратите внимание, что мы опустили параметры, связанные со вспомогательными командами.

Как и в случае с системными пределами, есть несколько моментов, связанных с интерпретацией параметров функциями `sysconf`, `pathconf` и `fpathconf`.

1. Возвращаемое значение для параметра `_SC_VERSION` указывает год (первые четыре цифры) и месяц (последние две цифры) публикации стандарта. Это может быть число `198808L`, `199009L`, `199506L` или иное для более поздних версий. Так, третьей версии Single UNIX Specification соответствует значение `200112L` (редакция стандарта POSIX.1 2001 года). Четвертой версии стандарта Single UNIX Specification (редакция стандарта POSIX.1 2008 года) соответствует значение `200809L`.
2. Возвращаемое значение для параметра `_SC_XOPEN_VERSION` указывает версию XSI, которой соответствует система. Третьей версии Single UNIX Specification соответствует значение `600`. Четвертой версии Single UNIX Specification (редакция стандарта POSIX.1 2008 года) – значение `700`.
3. Возвращаемые значения для параметров `_SC_JOB_CONTROL`, `_SC_SAVED_IDS` и `_PC_VDISABLE` сейчас не относятся к дополнительным функциональным возможностям. XPG4 и ранние версии Single UNIX Specification требовали обязательной поддержки этих особенностей, но только начиная с третьей версии Single UNIX Specification они перешли в разряд обязательных. Эти идентификаторы сохранены для обратной совместимости.
4. Платформы, соответствующие требованиям POSIX.1-2008, также должны поддерживать следующие параметры:
 - `_POSIX_ASYNCHRONOUS_IO`
 - `_POSIX_BARRIERS`

- `_POSIX_CLOCK_SELECTION`
- `_POSIX_MAPPED_FILES`
- `_POSIX_MEMORY_PROTECTION`
- `_POSIX_READER_WRITER_LOCKS`
- `_POSIX_REALTIME_SIGNALS`
- `_POSIX_SEMAPHORES`
- `_POSIX_SPIN_LOCKS`
- `_POSIX_THREAD_SAFE_FUNCTIONS`
- `_POSIX_THREADS`
- `_POSIX_TIMEOUTS`
- `_POSIX_TIMERS`

Этим константам присвоено значение `200809L`. Соответствующие им константы `_SC` также сохранены для обратной совместимости.

1. Для параметров `_PC_CHOWN_RESTRICTED` и `_PC_NO_TRUNC` возвращается значение `-1` без изменения `errno`, если функциональная возможность не поддерживается для указанного значения аргумента *pathname* или *fd*. Во всех POSIX-совместимых системах возвращаемое значение будет больше нуля (указывая, что данная возможность поддерживается).
2. Файл, к которому относится параметр `_PC_CHOWN_RESTRICTED`, должен быть файлом или каталогом. Если это каталог, данная функциональная возможность применяется к файлам в этом каталоге.
3. Файл, к которому относятся параметры `_PC_NO_TRUNC` и `_PC_2_SYMLINKS`, должен быть каталогом.
4. Возвращаемое значение для параметра `_PC_NO_TRUNC` применяется к именам файлов в этом каталоге.
5. Файл, к которому относится параметр `_PC_VDISABLE`, должен быть файлом терминального устройства.
6. Файл, к которому относятся параметры `_PC_ASYNC_IO`, `_PC_PRIO_IO` и `_PC_SYNC_IO`, не должен быть каталогом.

Таблица 2.15. Необязательные параметры и их идентификаторы для функции `sysconf`

Имя параметра	Указывает, поддерживает ли система...	Аргумент name
<code>_POSIX_ASYNCHRONOUS_IO</code>	Асинхронный ввод/вывод POSIX	<code>_SC_ASYNCHRONOUS_IO</code>
<code>_POSIX_BARRIERS</code>	Барьеры	<code>_SC_BARRIERS</code>
<code>_POSIX_CLOCK_SELECTION</code>	Выбор часов	<code>_SC_CLOCK_SELECTION</code>
<code>_POSIX_JOB_CONTROL</code>	Управление заданиями	<code>_SC_JOB_CONTROL</code>
<code>_POSIX_MAPPED_FILES</code>	Отображение файлов в память	<code>_SC_MAPPED_FILES</code>

Таблица 2.15 (окончание)

Имя параметра	Указывает, поддерживает ли система...	Аргумент name
_POSIX_MEMORY_PROTECTION	Возможность защиты памяти	_SC_MEMORY_PROTECTION
_POSIX_READER_WRITER_LOCKS	Блокировки чтения/записи	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	Сигналы реального времени	_SC_REALTIME_SIGNALS
_POSIX_SAVED_IDS	Сохраненные идентификаторы пользователя и группы	_SC_SAVED_IDS
_POSIX_SEMAPHORES	Семафоры POSIX	_SC_SEMAPHORES
_POSIX_SHELL	Стандартную командную оболочку POSIX	_SC_SHELL
_POSIX_SPIN_LOCKS	Циклические блокировки (spin-locks)	_SC_SPIN_LOCKS
_POSIX_THREAD_SAFE_FUNCTIONS	Потокобезопасные функции	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREADS	Потоки выполнения	_SC_THREADS
_POSIX_TIMEOUTS	Версии выбранных функций с ограничением времени ожидания	_SC_TIMEOUTS
_POSIX_TIMERS	Таймеры	_SC_TIMERS
_POSIX_VERSION	Указывает версию POSIX.1	_SC_VERSION
_XOPEN_CRYPT	Группу интерфейсов шифрования XSI	_SC_XOPEN_CRYPT
_XOPEN_REALTIME	Группу интерфейсов реального времени XSI	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	Группу интерфейсов потоков реального времени XSI	_SC_XOPEN_REALTIME_THREADS
_XOPEN_SHM	Группу интерфейсов XSI разделяемой памяти	_SC_XOPEN_SHM
_XOPEN_VERSION	Указывает версию XSI	_SC_XOPEN_VERSION

В табл. 2.16 приведены некоторые конфигурационные параметры и соответствующие им значения для четырех платформ, обсуждаемых в данной книге. Обозначение «Не поддерживается» говорит о том, что соответствующая константа определена, но имеет значение -1 или имеет значение 0 , но вызов функции `sysconf` или `pathconf` возвращает -1 . Обратите внимание, что некоторые системы еще не соответствуют последней версии Single UNIX Specification.

Обратите внимание, что в операционной системе Solaris функция `pathconf` возвращает значение -1 для параметра `_PC_NO_TRUNC`, если вызывается для файла, находящегося в файловой системе PCFS. Эта файловая система поддерживает формат DOS (для дискет) и без предупреждения усекает имена файлов до формата 8.3, как того требует файловая система DOS.

Таблица 2.16. Примеры значений конфигурационных параметров

Предел	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				файловая система UFS	файловая система PCFS
_POSIX_CHOWN_RESTRICTED	1	1	200112	1	1
_POSIX_JOB_CONTROL	1	1	200112	1	1
_POSIX_NO_TRUNC	1	1	200112	1	Не поддер- живается
_POSIX_SAVED_IDS	Не поддер- живается	1	200112	1	1
_POSIX_THREADS	200112	200809	200112	200112	200112
_POSIX_VDISABLE	255	0	255	0	0
_POSIX_VERSION	200112	200809	200112	200112	200112
_XOPEN_UNIX	Не поддер- живается	1	1	1	1
_XOPEN_VERSION	Не поддер- живается	700	600	600	600

2.7. Макроопределения проверки особенностей

В заголовочных файлах определяется множество символов стандарта POSIX.1 и XSI. Большинство реализаций добавляют в эти файлы собственные определения в дополнение к тем, что описываются стандартами POSIX.1 и XSI. Если возникает необходимость скомпилировать программу так, чтобы она зависела только от определений POSIX и не конфликтовала с определениями, зависящими от реализации, необходимо определить константу _POSIX_C_SOURCE. Эта константа используется во всех заголовочных файлах стандарта POSIX.1 для исключения любых зависящих от реализации определений.

Ранние версии стандарта POSIX.1 определяли константу _POSIX_SOURCE. В редакции POSIX.1 2001 года ее заменила константа _POSIX_C_SOURCE.

Константы _POSIX_C_SOURCE и _XOPEN_SOURCE называются *макроопределениями проверки особенностей*. Все подобные макроопределения начинаются с символа подчеркивания. Обычно они используются в командной строке компилятора cc, например

```
cc -D_POSIX_C_SOURCE=200809L file.c
```

В таком случае макрос будет определен до подключения любого заголовочного файла. Чтобы использовать только определения стандарта POSIX.1, также можно в первой строке файла с исходным текстом программы указать следующее определение:

```
#define _POSIX_C_SOURCE 200809L
```

Чтобы приложениям стали доступны функциональные особенности, определяемые версией 4 Single UNIX Specification, нужно определить константу `_XOPEN_SOURCE` со значением 700. Помимо включения дополнительных интерфейсов XSI, это даст тот же эффект, что и определение константы `_POSIX_C_SOURCE` со значением 200809L, когда речь идет о функциональности, определяемой стандартом POSIX.1.

Стандарт Single UNIX Specification определяет утилиту `c99` в качестве интерфейса к среде компиляции языка C. Вот как с ее помощью можно скомпилировать файл:

```
c99 -D_XOPEN_SOURCE=700 file.c -o file
```

Чтобы разрешить компилятору `gcc` использовать расширения 1999 ISO C, можно добавить в командную строку параметр `-std=c99`, как показано ниже:

```
gcc -D_XOPEN_SOURCE=700 -std=c99 file.c -o file
```

2.8. Элементарные системные типы данных

Исторически некоторые типы данных в языке C были связаны с некоторыми переменными системы UNIX. Например, старшие и младшие номера устройств исторически хранились в виде 16-разрядного целого числа, где 8 разрядов отводилось для старшего номера устройства и 8 разрядов — для младшего. Но большинству крупных систем необходима возможность определения более 256 различных номеров устройств, поэтому потребовалось предусмотреть иной подход к нумерации. (Так, в 32-разрядной версии Solaris для хранения номеров устройств используются 32 разряда — 14 разрядов для старшего и 18 для младшего номера устройства.)

Заголовочный файл `<sys/types.h>` определяет ряд зависящих от реализации типов данных, которые называются *элементарными системными типами данных*. Кроме того, некоторые из этих типов данных объявляются и в других заголовочных файлах. Все они объявлены посредством директивы `typedef`. Названия их обычно завершаются символами `_t`. В табл. 2.17 перечислено большинство элементарных типов, с которыми мы будем сталкиваться в этой книге.

Таблица 2.17. Некоторые наиболее распространенные элементарные типы

Тип	Описание
<code>clock_t</code>	Счетчик тактов системных часов (время работы процесса, раздел 1.10)
<code>comp_t</code>	Счетчик тактов в упакованном виде (раздел 8.14)
<code>dev_t</code>	Номер устройства (старший и младший, раздел 4.24)
<code>fd_set</code>	Набор файловых дескрипторов (раздел 14.4.1)
<code>fpos_t</code>	Позиция в файле (раздел 5.10)
<code>gid_t</code>	Числовой идентификатор группы
<code>ino_t</code>	Номер индексного узла (i-node, раздел 4.14)
<code>mode_t</code>	Тип файла, режим создания файла (раздел 4.5)

Тип	Описание
<code>nlink_t</code>	Счетчик ссылок для записей в каталоге (раздел 4.14)
<code>off_t</code>	Размер файла и смещение в файле (со знаком) (<code>lseek</code> , раздел 3.6)
<code>pid_t</code>	Числовой идентификатор процесса и идентификатор группы процессов (со знаком, разделы 8.2 и 9.4)
<code>pthread_t</code>	Числовой идентификатор потока выполнения (раздел 11.3)
<code>ptrdiff_t</code>	Разность двух указателей (со знаком)
<code>rlim_t</code>	Предельное значение для ресурса (раздел 7.11)
<code>sig_atomic_t</code>	Тип данных, доступ к которому может выполняться атомарно (раздел 10.15)
<code>sigset_t</code>	Набор сигналов (раздел 10.11)
<code>size_t</code>	Размер объекта (например, строки) (без знака) (раздел 3.7)
<code>ssize_t</code>	Возвращаемый функциями результат, представляющий счетчик байтов (со знаком) (<code>read</code> , <code>write</code> , раздел 3.7)
<code>time_t</code>	Счетчик секунд календарного времени (раздел 1.10)
<code>uid_t</code>	Числовой идентификатор пользователя
<code>wchar_t</code>	Может представлять символы в любой кодировке

Эти типы определены так, что при написании программ нет необходимости погружаться в детали конкретной реализации, которые могут меняться от системы к системе. Мы будем описывать, как используется каждый из этих типов, по мере необходимости.

2.9. Различия между стандартами

В общем и целом, различные стандарты прекрасно уживаются друг с другом. В основном мы будем обращать внимание на различия между стандартами ISO C и POSIX.1, поскольку раздел Base Specifications стандарта Single UNIX Specification является надмножеством стандарта POSIX.1. Ниже перечисляются некоторые отличия.

Стандарт ISO C определяет функцию `clock`, возвращающую количество процессорного времени, использованного процессом. Возвращаемое значение имеет тип `clock_t`, но стандарт ISO не определяет единицы измерения. Чтобы преобразовать это значение в секунды, необходимо разделить его на константу `CLOCKS_PER_SEC`, объявленную в заголовочном файле `<time.h>`. Стандарт POSIX.1 определяет функцию `times`, возвращающую как процессорное время (для вызывающего процесса и для всех его дочерних процессов, завершивших свою работу), так и общее время. Все эти значения имеют тип `clock_t`. С помощью функции `sysconf` необходимо получить количество тактов в секунду и затем использовать его для перевода значений типа `clock_t` в секунды. Получается, что одна и та же характеристика — количество тактов в секунду — определяется стандартами ISO C и POSIX.1 по-разному. Кроме того, оба стандарта используют один и тот же тип данных (`clock_t`) для хранения

разных значений. Разницу можно наблюдать в Solaris, где функция `clock` возвращает время в микросекундах (следовательно, константа `CLOCKS_PER_SEC` имеет значение 1 000 000), тогда как функция `sysconf` возвращает значение 100 (количество тактов в секунду). Поэтому нужно с особым вниманием относиться к переменным типа `clock_t`, чтобы не смешивать разные единицы измерения.

Конфликт возможен также в случаях, когда стандарт ISO C определяет некоторую функцию, но не так строго, как это делает стандарт POSIX.1. Так, например, обстоит дело с функциями, которые требуют иной реализации в среде POSIX (многозадачной), чем в среде ISO C (где очень немногое можно предположить о целевой операционной системе). Тем не менее большинство POSIX-совместимых систем реализуют функции в соответствии со стандартом ISO C для сохранения совместимости. Примером может служить функция `signal`. Если мы по незнанию используем функцию `signal` из Solaris (надеясь написать переносимый код, который будет работать в среде ISO C и в устаревших версиях UNIX), то получим семантику, отличную от той, которую имеет функция `sigaction`, определяемая стандартом POSIX.1. Более подробно о функции `signal` мы поговорим в главе 10.

2.10. Подведение итогов

Очень многое произошло в сфере стандартизации программной среды UNIX за прошедшие два с половиной десятилетия. Мы описали наиболее важные стандарты — ISO C, POSIX и Single UNIX Specification — и их влияние на четыре реализации UNIX, обсуждаемые в данной книге: FreeBSD, Linux, Mac OS X и Solaris. Эти стандарты пытаются определить конфигурационные параметры, которые могут варьироваться от системы к системе, и мы видели, что они далеки от совершенства. В этой книге мы еще не раз столкнемся со многими из них.

В списке использованной литературы указывается, как получить копии стандартов, описанных в этой главе.

Упражнения

- 1.1 Мы упоминали в разделе 2.8, что некоторые из элементарных системных типов данных определены более чем в одном заголовочном файле. Например, в FreeBSD 8.0 тип `size_t` определен в 29 различных файлах. Поскольку программа может подключить все 29 файлов, а стандарт ISO C не допускает множественного определения одного и того же типа, подумайте, как должны быть написаны эти заголовочные файлы.
- 1.2 Просмотрите заголовочные файлы в своей системе и перечислите фактические типы данных, используемые для реализации элементарных системных типов.
- 1.3 Измените программу из листинга 2.4 так, чтобы избежать лишней работы, когда функция `sysconf` возвращает значение `LONG_MAX` для предела `OPEN_MAX`.

3

Файловый ввод/вывод

3.1. Введение

Обсуждение системы UNIX мы начнем с операций файлового ввода/вывода, таких как открытие файла, чтение из файла, запись в файл и т. д. Большинство операций файлового ввода/вывода в UNIX можно выполнить с помощью всего пяти функций: `open`, `read`, `write`, `lseek` и `close`. Затем мы рассмотрим, как изменение размера буфера влияет на производительность функций `read` и `write`.

Функции, описываемые в этой главе, часто называют функциями *небуферизованного ввода/вывода* в противоположность стандартным функциям ввода/вывода, о которых пойдет речь в главе 5. Термин *небуферизованный* означает, что каждая операция чтения или записи обращается к системному вызову в ядре. Функции небуферизованного ввода/вывода не являются частью стандарта ISO C, но они определены стандартами POSIX.1 и Single UNIX Specification.

Всякий раз, когда речь заходит о совместном использовании ресурсов несколькими процессами, особую важность приобретает понятие атомарного выполнения операций. Мы рассмотрим это понятие применительно к операциям файлового ввода/вывода и аргументам функции `open`. Далее мы увидим, как осуществляется одновременный доступ к файлам из нескольких процессов и какие структуры данных ядра с этим связаны. Затем перейдем к функциям `dup`, `fcntl`, `sync`, `fsync` и `ioctl`.

3.2. Дескрипторы файлов

Все открытые файлы представлены в ядре файловыми дескрипторами. Файловый дескриптор — это неотрицательное целое число. Когда процесс открывает существующий файл или создает новый, ядро возвращает ему файловый дескриптор. Чтобы выполнить запись в файл или чтение из него, нужно передать функции `read` или `write` его файловый дескриптор, полученный вызовом функции `open` или `creat`.

В соответствии с соглашениями командные оболочки UNIX ассоциируют файловый дескриптор 0 с устройством стандартного ввода процесса, 1 — с устройством стандартного вывода и 2 — с устройством стандартного вывода сообщений об ошибках. Это соглашение используется командными оболочками и большин-

ством приложений, но не является особенностью ядра UNIX. Тем не менее многие приложения не смогли бы работать, если бы это соглашение было нарушено.

Хотя значения этих дескрипторов определены стандартом POSIX.1, в POSIX-совместимых приложениях вместо фактических значений 0, 1 и 2 следует использовать константы `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`. Определения этих констант находятся в заголовочном файле `<unistd.h>`.

Под файловые дескрипторы отводится диапазон чисел от 0 до `OPEN_MAX-1`. (Вспомните табл. 2.11.) В ранних реализациях UNIX максимальным значением файлового дескриптора было число 19, что позволяло каждому процессу держать открытыми до 20 файлов, но многие системы увеличили этот предел до 63.

В операционных системах FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 этот предел практически бесконечен и ограничен лишь объемом памяти в системе, размером целых чисел и прочими жесткими и мягкими ограничениями, задаваемыми администратором системы.

3.3. Функции `open` и `openat`

Файл создается или открывается функцией `open` или `openat`.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Обе возвращают дескриптор файла в случае успеха,
–1 — в случае ошибки

Последний аргумент обозначен многоточием (...), таким способом стандарт ISO C указывает, что количество остальных аргументов и их типы могут варьироваться. В этих функциях последний аргумент используется только при создании нового файла, о чем мы поговорим немного позже. Этот аргумент мы привели в прототипе функции как комментарий.

Аргумент `path` представляет имя файла, который будет открыт или создан. Эти функции могут принимать большое количество параметров, определяемых аргументом `oflag`. Значение этого аргумента формируется объединением по ИЛИ (OR) одной или более констант, определяемых в заголовочном файле `<fcntl.h>` и перечисленных ниже:

O_RDONLY Файл открывается только на чтение.

O_WRONLY Файл открывается только на запись.

O_RDWR Файл открывается для чтения и для записи.

В большинстве реализаций для совместимости с устаревшим программным обеспечением константа O_RDONLY определяется как 0, O_WRONLY — как 1 и O_RDWR — как 2.

O_EXEC Файл открывается только для выполнения.

O_SEARCH Файл открывается только для поиска (применяется к каталогам).

Константа **O_SEARCH** предусмотрена, чтобы позволить проверить права доступа к каталогу в момент его открытия. Последующие операции с файловым дескриптором каталога, открытого для поиска, не влекут повторной переоценки прав доступа. Ни одна из операционных систем, рассматриваемых в этой книге, пока не поддерживает флаг **O_SEARCH**.

Должна быть указана одна и только одна из этих пяти констант. Далее приводится список констант, присутствие которых в аргументе *oflag* необязательно.

O_APPEND Запись производится в конец файла. Более подробное описание этого флага мы дадим чуть позже, в разделе 3.11.

O_CLOEXEC Устанавливает флаг **FD_CLOEXEC** дескриптора файла. Подробнее этот флаг будет рассматриваться в разделе 3.14.

O_CREAT Если файл не существует, он будет создан. Этот флаг требует наличия третьего аргумента функции **open** (*mode*), который определяет значения битов прав доступа к создаваемому файлу. (В разделе 4.5, где рассказывается о правах доступа, мы увидим, как определяется значение аргумента *mode* и какое влияние на него оказывает значение параметра *umask* процесса.)

O_DIRECTORY Вызывает ошибку, если *path* не является именем каталога.

O_EXCL Вызывает ошибку, если файл уже существует и задан флаг **O_CREAT**. При такой комбинации флагов атомарно выполняется проверка существования файла и его создание, если файл не существует. Более подробно атомарные операции описываются в разделе 3.11.

O_NOCTTY Если аргумент *path* ссылается на файл терминального устройства, это устройство не назначается управляющим терминалом вызывающего процесса. Подробнее об управляющих терминалах рассказывается в разделе 9.6.

O_NOFOLLOW Вызывает ошибку, если *path* является символьской ссылкой. Подробнее о символьских ссылках рассказывается в разделе 4.17.

O_NONBLOCK Если аргумент *path* ссылается на именованный канал (FIFO), специальный файл блочного устройства или специальный файл символьного устройства, этот флаг задает неблокирующий режим открытия файла и последующих операций ввода/вывода. Подробнее этот режим описывается в разделе 14.2.

В ранних выпусках System V появился флаг **O_NDELAY**. Он подобен флагу **O_NONBLOCK**, но вносит двусмысленность в трактовку значения, возвращаемого функцией **read**. При наличии флага **O_NODELAY** функция **read** возвращает 0 в случае отсутствия данных в именованном или неименованном канале либо в файле устройства, но тогда возникает конфликт со значением 0, которое возвращается по достижении конца файла. В системах, основанных на SVR4, сохранилась поддержка флага **O_NODELAY** с устаревшей семантикой, однако все новые приложения должны использовать флаг **O_NONBLOCK**.

O_SYNC Каждый вызов функции **write** ожидает завершения физической операции ввода/вывода, включая операцию обновления атрибутов файла. Этот флаг будет использоваться в разделе 3.14.

O_TRUNC Если файл существует и успешно открывается для записи либо для чтения и записи, его размер усекается до нуля.

O_TTY_INIT При открытии терминального устройства, которое еще не было открыто, устанавливает нестандартные параметры `termios` в значения, соответствующие стандарту Single UNIX Specification. Подробнее структура `termios` будет рассматриваться в ходе обсуждения терминального ввода/вывода, в главе 18.

Следующие два флага также относятся к разряду необязательных. Они предназначены для поддержки синхронизированных операций ввода/вывода, определяемых стандартом Single UNIX Specification (а также POSIX.1):

O_DSYNC Каждый вызов функции `write` ожидает завершения физической операции ввода/вывода, но не ожидает, пока будут обновлены атрибуты файла, если они не влияют на доступность для чтения только что записанных данных.

Флаги O_DSYNC и O_SYNC очень похожи друг на друга, но все-таки чуть-чуть отличаются. Флаг O_DSYNC влияет на атрибуты файла, только если их необходимо обновить, чтобы отразить изменения в данных (например, обновить размер файла, если в файл были записаны дополнительные данные). При использовании флага O_SYNC данные и атрибуты всегда обновляются синхронно. При перезаписи существующей части файла, открытого с флагом O_DSYNC, атрибуты времени файла не будут обновляться синхронно с данными. Напротив, если файл открывается с флагом O_SYNC, каждое обращение к функции write будет приводить к изменению атрибутов времени файла независимо от того, были ли перезаписаны существующие данные или в конец добавлены новые.

O_RSYNC Каждый вызов функции `read` приостанавливается до тех пор, пока не будут закончены ждущие завершения операции записи в ту же часть файла.

Solaris 10 поддерживает все три флага. Исторически FreeBSD и Mac OS X поддерживают флаг O_FSYNC, действующий так же, как O_SYNC. Поскольку оба эти флага полностью эквивалентны, они определены с одним и тем же значением. FreeBSD 8.0 не поддерживает флаги O_DSYNC и O_RSYNC. Mac OS X не поддерживает флаг O_RSYNC, но поддерживает флаг D_SYNC, который трактуется так же, как O_SYNC. Linux 3.2.0 поддерживает флаг D_SYNC, а флаг R_SYNC трактуется так же, как O_SYNC.

Функции `open` и `openat` гарантируют, что возвращаемый ими дескриптор файла будет иметь наименьшее неиспользуемое положительное числовое значение. Это обстоятельство используется в некоторых приложениях для открытия нового файла вместо стандартного ввода, стандартного вывода или стандартного вывода сообщений об ошибках. Например, приложение может закрыть файл стандартного вывода (обычно это дескриптор 1) и затем открыть другой файл, зная, что он будет открыт с дескриптором 1. В разделе 3.12 мы продемонстрируем более надежный способ открытия файла на конкретном дескрипторе при помощи функции `dup2`.

Параметр `fd` в функции `openat` отличается от одноименного параметра в функции `open`. Возможны три разных варианта.

1. Параметр `path` содержит строку абсолютного пути. В этом случае параметр `fd` игнорируется и `openat` действует подобно функции `open`.
2. Параметр `path` содержит строку относительного пути, а параметр `fd` содержит дескриптор файла, определяющего местоположение в файловой системе, отку-

да будет откладываться относительный путь. Значение для параметра *fd* можно получить, открыв каталог, относительно которого должен откладываться относительный путь.

3. Параметр *path* содержит строку относительного пути, а параметр *fd* содержит специальное значение `AT_FDCWD`. В этом случае путь начинает откладываться от текущего рабочего каталога и функция `openat` действует подобно функции `open`.

Функция `openat` принадлежит к группе функций, добавленных в последней версии POSIX.1 для решения двух проблем. Во-первых, она дает потокам выполнения возможность использовать относительные пути для открытия файлов в каталогах, отличных от текущего рабочего каталога. Как будет показано в главе 11, все потоки выполнения в одном процессе разделяют один и тот же текущий рабочий каталог, поэтому иногда бывает сложно организовать одновременную работу нескольких потоков выполнения в разных каталогах. Во-вторых, она позволяет избежать ошибок вида «проверка перед использованием» (time-of-check-to-time-of-use, TOCTTOU).

Суть ошибок TOCTTOU в том, что программа оказывается уязвимой, если вызывает две функции обращения к файлам и при этом второй вызов зависит от результатов первого вызова. Поскольку два вызова выполняются неатомарно, файл может измениться между вызовами, сделав результаты первого вызова недействительными, и вызвать ошибку в программе. Ошибки TOCTTOU обычно используются для изменения привилегий доступа к файлам, обманом заставляя привилегированную программу понизить права доступа к привилегированному файлу или изменить его, чтобы пробить брешь в системе безопасности. Вей и Пу [Wei & Pu, 2005] обсуждают возможные ошибки TOCTTOU в интерфейсе файловой системы UNIX.

Усечение имени файла и строки пути

Что произойдет, если конфигурационный параметр `NAME_MAX` имеет значение 14 и мы попытаемся создать новый файл, имя которого состоит из 15 символов? Традиционно ранние версии System V, такие как SVR2, допускали это, просто усекая длину имени файла до 14 символов. BSD-системы возвращали признак ошибки с кодом `ENAMETOOLONG` в переменной `errno`. Простое усечение имени файла создает проблему, которая проявляет себя не только при создании нового файла. Так, если параметр `NAME_MAX` имеет значение 14 и существует файл с именем ровно из 14 символов, ни одна из функций, принимающих имя пути к файлу, например `open` или `stat`, не сможет определить первоначальное имя файла, которое, возможно, было обрезано.

Конфигурационный параметр `_POSIX_NO_TRUNC`, предусматриваемый стандартом POSIX.1, определяет, усекаются ли слишком длинные имена файлов и строки пути или возвращается признак ошибки. Как уже говорилось в главе 2, значение этого параметра может варьироваться в зависимости от типа файловой системы, и мы можем применить функцию `fpathconf` или `pathconf` к каталогу, чтобы узнатъ, какое поведение поддерживается.

Возвращается признак ошибки или нет, во многом обусловлено историческими причинами. Например, операционные системы, основанные на SVR4, не генерируют ошибку для традиционной файловой системы S5. Для BSD-подобной файловой системы (известной также как UFS) те же самые системы возвращают признак ошибки. Другой пример (см. табл. 2.16): Solaris сгенерирует ошибку для файловой системы UFS, но не для PCFS, совместимой с файловой системой DOS, поскольку она «молча» усекает имена файлов, не соответствующие формату 8.3. BSD-системы и Linux всегда возвращают признак ошибки.

Когда параметр `_POSIX_NO_TRUNC` определен и полный путь к файлу превышает значение `PATH_MAX` или какой-либо компонент имени файла или строки пути превышает значение `NAME_MAX`, возвращается признак ошибки и в переменную `errno` записывается код ошибки `ENAMETOOLONG`.

Большинство современных систем поддерживают имена файлов длиной до 255 символов. Поскольку имена файлов обычно короче этого предела, данное ограничение не является большой проблемой для большинства приложений.

3.4. Функция `creat`

Новый файл можно также создать с помощью функции `creat`.

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```

В случае успеха возвращает файловый дескриптор, доступный только для записи, `-1` — в случае ошибки

Обратите внимание: эта функция эквивалентна вызову

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

В ранних версиях UNIX второй аргумент функции `open` мог принимать только три значения: 0, 1 или 2. Открыть несуществующий файл не было никакой возможности. Поэтому для создания нового файла был необходим отдельный системный вызов. В настоящее время флаги `O_CREAT` и `O_TRUNC` обеспечивают функцию `open` необходимыми средствами для создания файлов, и потребность в функции `creat` отпала.

Порядок определения аргумента `mode` мы покажем в разделе 4.5, когда во всех подробностях будем описывать права доступа к файлам.

Функция `creat` имеет один недостаток: файл открывается только на запись. До появления обновленной версии функции `open`, чтобы создать временный файл, записать в него некоторые данные и потом прочитать их, требовалось вызывать `creat`, `close` и затем `open`. Гораздо удобнее использовать в таких случаях функцию `open`, как показано ниже:

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5. Функция close

Закрытие открытого файла производится вызовом функции `close`.

```
#include <unistd.h>
int close(int fd);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Закрытие файла приводит также к снятию любых блокировок, которые могли быть наложены процессом. Мы обсудим этот вопрос в разделе 14.3.

При завершении процесса все открытые им файлы автоматически закрываются ядром. Многие приложения используют это обстоятельство и не закрывают файлы явно. Примером тому служит программа в листинге 1.2.

3.6. Функция lseek

С любым открытым файлом связано такое понятие, как текущая позиция в файле. Как правило, это неотрицательное целое число, которым выражается смещение в байтах от начала файла. (Некоторые исключения, касающиеся слова «неотрицательное», будут упомянуты чуть позже.) Обычно операции чтения и записи начинают выполняться с текущей позиции в файле и увеличивают ее значение на количество прочитанных или записанных байтов. По умолчанию при открытии файла текущая позиция инициализируется числом 0, если не был установлен флаг `O_APPEND`.

Явное изменение текущей позиции в файле выполняется с помощью функции `lseek`.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Возвращает новую текущую позицию файла в случае успеха,
-1 — в случае ошибки

Интерпретация аргумента `offset` зависит от значения аргумента `whence`.

- Если аргумент `whence` имеет значение `SEEK_SET`, то `offset` интерпретируется как смещение от начала файла.
- Если аргумент `whence` имеет значение `SEEK_CUR`, то `offset` интерпретируется как смещение от текущей позиции в файле. В этом случае `offset` может принимать и положительные, и отрицательные значения.
- Если аргумент `whence` имеет значение `SEEK_END`, то `offset` интерпретируется как смещение от конца файла. В этом случае `offset` может принимать и положительные, и отрицательные значения.

Поскольку в случае успеха функция `lseek` возвращает новую текущую позицию в файле, в аргументе `offset` можно передать 0, чтобы узнать текущую позицию:

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

Таким же способом можно определить, поддерживается ли свободное перемещение текущей позиции файла. Если файловый дескриптор относится к именованному или неименованному каналу либо к сокету, функция `lseek` вернет значение `-1` и запишет в переменную `errno` код ошибки `ESPIPE`.

Символьные константы SEEK_SET, SEEK_CUR и SEEK_END изначально появились в System V. До этого аргумент whence мог принимать значения 0 (смещение от начала файла), 1 (смещение от текущей позиции) или 2 (смещение от конца файла). Многие программы до сих пор используют эти предопределенные числовые значения.

Буква `l` в имени функции `lseek` означает «long integer» (длинное целое). До введения типа данных `off_t` аргумент `offset` и возвращаемое значение имели тип `long`. Сама функция `lseek` впервые появилась в Version 7, когда в язык С был добавлен тип длинных целых чисел. (В Version 6 была похожая функциональность, которая обеспечивалась функциями `seek` и `tell`.)

Пример

Программа в листинге 3.1 проверяет возможность свободного перемещения текущей позиции в файле стандартного ввода.

Листинг 3.1. Проверка возможности свободного перемещения текущей позиции в файле стандартного ввода

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("перемещение невозможно\n");
    else
        printf("перемещение выполнено\n");
    exit(0);
}
```

Запустив эту программу, мы получим следующее:

```
$ ./a.out < /etc/passwd
перемещение выполнено
$ cat < /etc/passwd | ./a.out
перемещение невозможно
$ ./a.out < /var/spool/cron/FIFO
перемещение невозможно
```

Обычно смещение относительно текущей позиции должно быть неотрицательным целым числом. Однако некоторые устройства поддерживают отрицательные смещения. Но для обычных файлов смещение должно быть неотрицательным. Поскольку отрицательные смещения все-таки возможны, возвращаемое функци-

ей lseek значение следует сравнивать именно с числом -1 , а не проверять, не является ли оно отрицательным.

В OC FreeBSD на платформе Intel x86 устройство /dev/kmem поддерживает отрицательные смещения.

Поскольку тип off_t является целым числом со знаком (табл. 2.17), теряется половина возможного максимального размера файла. Так, если off_t представляет 32-разрядное целое со знаком, максимальный размер файла будет равен $2^{31}-1$ байт.

Функция lseek изменяет значение текущей позиции в файле лишь в области данных ядра — фактически она не выполняет никаких операций ввода/вывода. Новое значение текущей позиции будет использовано ближайшей операцией чтения или записи.

Текущая позиция в файле может превышать его текущий размер. В этом случае следующая операция записи увеличит размер файла. Это вполне допустимо и может рассматриваться как создание «дырки» в файле. Байты, которые фактически не были записаны, считаются как нули.

«Дырка» в файле не обязательно должна занимать место на диске. В некоторых файловых системах в случае переноса текущей позиции за пределы файла на диске могут выделяться новые блоки для данных, но это совершенно необязательно.

Пример

Программа в листинге 3.2 создает файл с «дыркой».

Листинг 3.2. Создание файла с «дыркой»

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghijkl";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int    fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("ошибка вызова creat");

    if (write(fd, buf1, 10) != 10)
        err_sys("ошибка записи buf1");
    /* теперь текущая позиция = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("ошибка вызова lseek");
    /* теперь текущая позиция = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("ошибка записи buf2");
    /* теперь текущая позиция = 16394 */

    exit(0);
}
```

Запустив эту программу, мы получим следующее:

```
$ ./a.out
$ ls -l file.hole          проверим размер файла
-rw-r--r-- 1 sar 16394 Nov 25 01:01 file.hole
$ od -c file.hole          посмотрим фактическое содержимое

0000000  a b c d e f g h i j \0 \0 \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000  A B C D E F G H I J
0040012
```

Чтобы увидеть содержимое файла, мы воспользовались командой `od(1)`. Флаг `-c` сообщает ей, что содержимое следует выводить в виде символов. Как видите, байты, которые не были фактически записаны, читаются как нули. Семизначные числа в начале каждой строки — это смещение от начала файла в восьмеричном виде.

Чтобы убедиться, что в файле действительно имеется «дырка», сравним только что созданный файл с файлом того же размера, но без «дырки»:

```
$ ls -ls file.hole file.nohole  сравним размеры
8 -rw-r--r-- 1 sar 16394 Nov 25 01:01 file.hole
20 -rw-r--r-- 1 sar 16394 Nov 25 01:03 file.nohole
```

Несмотря на то что файлы имеют одинаковый размер, файл без «дырки» занимает 20 дисковых блоков, а файл с «дыркой» — всего 8.

В этом примере мы вызывали функцию `write` (раздел 3.8). О файлах с «дырками» мы еще поговорим в разделе 4.12.

Поскольку для представления смещения функция `lseek` использует тип `off_t`, реализации поддерживают тот размер, который определен для конкретной платформы. Большинство современных платформ предоставляет два набора интерфейсов для работы со смещением в файле: 32- и 64-разрядный.

Стандарт Single UNIX Specification дает приложениям возможность с помощью функции `sysconf` (раздел 2.5.4) узнать, какие интерфейсы поддерживаются системой. В табл. 3.1 приводится список констант для функции `sysconf`.

Таблица 3.1. Размеры типов данных и имена констант, передаваемые функции `sysconf`

Имя конфигурационного параметра	Описание	Значение аргумента <code>name</code>
<code>_POSIX_V7_ILP32_OFF32</code>	Типы <code>int</code> , <code>long</code> , указатели и <code>off_t</code> представлены 32 разрядами	<code>_SC_V7_ILP32_OFF32</code>
<code>_POSIX_V7_ILP32_OFFBIG</code>	Типы <code>int</code> , <code>long</code> и указатели представлены 32 разрядами, тип <code>off_t</code> имеет размер не менее 64 разрядов	<code>_SC_V7_ILP32_OFFBIG</code>
<code>_POSIX_V7_LP64_OFF64</code>	Тип <code>int</code> представлен 32 разрядами, типы <code>long</code> , указатели и <code>off_t</code> имеют размер 64 разряда	<code>_SC_V7_LP64_OFF64</code>
<code>_POSIX_V7_LP64_OFFBIG</code>	Тип <code>int</code> представлен 32 разрядами, типы <code>long</code> , указатели и <code>off_t</code> имеют размер не менее 64 разрядов	<code>_SC_V7_LP64_OFFBIG</code>

Компилятор `c99` требует использовать команду `getconf(1)` для отображения желаемой модели размерностей во флаги компиляции и связывания. В зависимости от конкретной реализации могут потребоваться различные флаги и библиотеки.

К сожалению, это одна из тех областей, в которых реализации отстают от стандартов. Если ваша система не соответствует последней версии стандарта, возможно, она поддерживает имена параметров из предыдущей версии Single UNIX Specification: `_POSIX_V6_LP32_OFFSET32`, `_POSIX_V6_LP32_OFFSETBIG`, `_POSIX_V6_LP64_OFFSET64` и `_POSIX_V6_LP64_OFFSETBIG`.

Чтобы обойти эти препятствия, приложение может присвоить константе `_FILE_OFFSET_BITS` значение 64 и обеспечить поддержку 64-разрядных смещений. Тогда тип `off_t` будет определен как 64-разрядное целое со знаком. Установив константу `_FILE_OFFSET_BITS` равной 32, мы сможем работать с 32-разрядными смещениями. Все четыре платформы, обсуждаемые в данной книге, поддерживают 32- и 64-разрядные смещения, но помните, что определение константы `_FILE_OFFSET_BITS` не гарантирует переносимости приложений и может не давать желаемого эффекта.

В табл. 3.2 приводятся размеры типа `off_t` в байтах для платформ, охватываемых этой книгой, когда приложение не определяет константу `_FILE_OFFSET_BITS`, а также когда приложение присваивает константе `_FILE_OFFSET_BITS` значение 32 или 64.

Таблица 3.2. Размер типа `off_t` в байтах для разных платформ

Операционная система	Аппаратная архитектура	Значение <code>_FILE_OFFSET_BITS</code>		
		Не определено	32	64
FreeBSD 8.0	x86 32-разрядная	8	8	8
Linux 3.2.0	x86 64-разрядная	8	8	8
Mac OS X 10.6.8	x86 64-разрядная	8	8	8
Solaris 10	SPARC 64-разрядная	8	4	8

Обратите внимание: даже если у вас установлены 64-разрядные смещения, возможность создания файлов размером более 2 Гбайт ($2^{31}-1$ байт) зависит от реализации файловой системы.

3.7. Функция `read`

Чтение данных из открытого файла производится функцией `read`.

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

Возвращает количество прочитанных байтов,
0 — если достигнут конец файла, -1 — в случае ошибки

В случае успеха функция `read` возвращает количество прочитанных байтов. Если достигнут конец файла, возвращается 0.

Существует несколько ситуаций, когда количество фактически прочитанных байтов меньше, чем было запрошено:

- При чтении из обычного файла, когда конец файла встретился до того, как было прочитано требуемое количество байтов. Например, если до конца файла осталось 30 байт, а запрошено 100, функция `read` вернет число 30. При следующем вызове она вернет 0 (конец файла).
- При чтении из терминального устройства. Обычно за одно обращение читается одна строка. (В главе 18 мы увидим, как это можно изменить.)
- При чтении данных из Сети. Промежуточная буферизация может стать причиной получения меньшего количества байтов, чем было запрошено.
- При чтении из именованных или неименованных каналов. Если в канале содержится меньше байтов, чем было запрошено, функция `read` вернет только то, что ей будет доступно.
- При чтении из устройства, ориентированного на доступ к отдельным записям. Примером такого устройства является накопитель на магнитной ленте, который может вернуть только одну запись за одно обращение.
- При прерывании операции чтения сигналом, когда часть данных уже была прочитана. Эту ситуацию мы обсудим подробнее в разделе 10.5.

Операция чтения начинается с текущей позиции в файле. В случае успеха текущая позиция увеличивается на число фактически прочитанных байтов.

Стандарт POSIX.1 изменил прототип функции `read`. Вот как выглядит классическое определение этой функции:

```
int read(int fd, char *buf, unsigned nbytes);
```

- Во-первых, тип второго аргумента (`char *`) был изменен на `void *` для совместимости со стандартом ISO C: тип `void *` используется для определения нетипизированных указателей.
- Далее, возвращаемое значение должно быть целым числом со знаком (`ssize_t`), чтобы была возможность вернуть положительное число (количество прочитанных байтов), 0 (признак конца файла) или `-1` (признак ошибки).
- И наконец, третий аргумент исторически был целым числом без знака, что позволяло в 16-разрядных реализациях читать и записывать до 65 534 байт за одно обращение. В редакции стандарта POSIX.1 1990 года введены новые типы данных: `ssize_t` для представления возвращаемого значения как целого со знаком и `size_t`, целое без знака, для представления третьего аргумента. (Вспомните константу `SSIZE_MAX` из раздела 2.5.2.)

3.8. Функция `write`

Запись данных в открытый файл производится функцией `write`.

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Возвращает количество записанных байтов в случае успеха,
-1 — в случае ошибки

Возвращаемое значение обычно совпадает со значением аргумента *nbytes*, в противном случае возвращается признак ошибки. Наиболее распространенные случаи, когда возникает ошибка записи, — переполнение диска или превышение ограничения на размер файла для заданного процесса (раздел 7.11 и упражнение 10.11).

Для обычных файлов запись начинается с текущей позиции в файле. Если файл открыт с флагом *O_APPEND*, перед началом каждой операции записи текущая позиция устанавливается в конец файла. По окончании записи значение текущей позиции увеличивается на количество фактически записанных байтов.

3.9. Эффективность операций ввода/вывода

Программа в листинге 3.3 выполняет копирование файлов, используя функции *read* и *write*.

Листинг 3.3. Копирование из стандартного ввода в стандартный вывод

```
#include "apue.h"

#define BUFFSIZE 4096

int
main(void)
{
    int     n;
    char   buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка записи");

    if (n < 0)
        err_sys("ошибка чтения");

    exit(0);
}
```

Вот несколько пояснений к этой программе.

- Если чтение производится из стандартного ввода, а запись — в стандартный вывод, предполагается, что они должным образом открыты командной оболочкой до запуска программы. В действительности все командные оболочки UNIX, как правило, позволяют открыть файл для чтения на устройстве стандартного ввода и создания (или перезаписи) на устройстве стандартного вывода. Это освобождает программы от необходимости открывать входной

и выходной файлы и дает пользователю возможность использовать механизм перенаправления ввода/вывода, поддерживаемый командной оболочкой.

- Программа не закрывает входной и выходной файлы. Все открытые дескрипторы закрываются ядром UNIX по завершении процесса, и программа использует это свойство.
- Этот пример одинаково хорошо работает с текстовыми и с двоичными файлами, поскольку ядро не делает никаких различий между этими двумя форматами.

Еще один вопрос, на который предстоит ответить: как выбрать правильное значение константы `BUFFSIZE`. Прежде чем дать на него ответ, попробуем запустить программу с различными значениями `BUFFSIZE`. В табл. 3.3 приведены результаты чтения файла размером 516 581 760 байт с использованием 20 различных размеров буфера.

Таблица 3.3. Производительность операции чтения с различными размерами буфера в ОС Linux

<code>BUFFSIZE</code>	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)	Количество циклов
1	20,03	117,50	138,73	516 581 760
2	9,69	58,76	68,60	258 290 880
4	4,60	36,47	41,27	129 145 440
8	2,47	15,44	18,38	64 572 720
16	1,07	7,93	9,38	32 286 360
32	0,56	4,51	8,82	16 143 180
64	0,34	2,72	8,66	8 071 590
128	0,34	1,84	8,69	4 035 795
256	0,15	1,30	8,69	2 017 898
512	0,09	0,95	8,63	1 008 949
1024	0,02	0,78	8,58	504 475
2048	0,04	0,66	8,68	252 238
4096	0,03	0,58	8,62	126 119
8192	0,00	0,54	8,52	63 060
16 384	0,01	0,56	8,69	31 530
32 768	0,00	0,56	8,51	15 765
65 536	0,01	0,56	9,12	7883
131 072	0,00	0,58	9,08	3942
262 144	0,00	0,60	8,70	1971
524 288	0,01	0,58	8,58	986

Файл читался программой из листинга 3.3 с перенаправлением стандартного вывода в устройство `/dev/null`. Эксперимент проводился в файловой системе Linux `ext4` с размером дискового блока 4096 байт. (Значение `st_blksize`, которое мы рассмотрим в разделе 4.12, составляет 4096.) Это объясняет, почему наименьшее системное время приходится именно на этот размер `BUFFSIZE`. Дальнейшее увеличение буфера дает лишь незначительный положительный эффект.

Для повышения производительности большинство файловых систем поддерживают опережающее чтение. Обнаружив ряд последовательных операций чтения, система пытается прочитать больший объем данных, чем было запрошено приложением, предполагая, что программа вскоре продолжит чтение. Эффект влияния опережающего чтения можно наблюдать в табл. 3.3, где время чтения для буфера размером 32 байта почти совпадает со временем для буферов большего размера.

Позднее мы еще вернемся к этой таблице. В разделе 3.14 мы покажем результат выполнения операции синхронной записи, в разделе 5.8 сравним время выполнения операций небуферизованного ввода/вывода и функций стандартной библиотеки ввода/вывода.

*Будьте внимательны, измеряя производительность программ, работающих с файлами. Операционная система пытается кэшировать файл в оперативной памяти (*incore*), поэтому при проведении серии экспериментов с одним и тем же файлом каждый последующий результат обычно лучше самого первого. Это происходит потому, что первая операция ввода/вывода поместит файл в системный кэш и каждый последующий прогон программы будет получать данные из кэша, а не с диска. (Термин *incore* означает оперативную память. Много лет назад оперативная память компьютеров строилась на магнитных ферритовых сердечниках (по-английски *core*). Отсюда же взялся и термин «*core dump*» (дамп памяти) – образ оперативной памяти программы, сохраненный в файле на диске для последующего анализа.)*

В эксперименте, результаты которого показаны в табл. 3.3, участвовали различные копии файла, поэтому использование системного кэша было сведено к минимуму. Размер этих файлов достаточно велик, поэтому они не могут находиться в кэше одновременно (тестовая система имела 6 Гбайт ОЗУ).

3.10. Совместное использование файлов

ОС UNIX поддерживает совместное использование открытых файлов несколькими процессами. Нам необходимо разобраться с этой возможностью, прежде чем мы перейдем к описанию функции `dup`. Для этого рассмотрим структуры данных, которые используются ядром при выполнении всех операций ввода/вывода.

Далее следует лишь концептуальное описание, которое может не совпадать с конкретной реализацией. За описанием структур в System V следует обращаться к [Bach, 1986]. В [McKusick et al., 1996] описаны те же структуры применительно к 4.4BSD. В [McKusick and NevilleNeil, 2005] рассматривается FreeBSD 5.2. Аналогичное описание для Solaris вы найдете в [Mauro and McDougall, 2001]. Обсуждение архитектуры ядра Linux 2.6 можно найти в [Bovet and Cesati, 2006].

Ядро использует три структуры данных для представления открытого файла, а отношения между ними определяют взаимовлияние процессов при совместном использовании файлов.

1. Каждому процессу соответствует запись в таблице процессов. С каждой записью в таблице процессов связана таблица открытых файловых дескрипторов, которую можно представить как таблицу, в которой каждая строка соответствует одному файловому дескриптору. Для каждого дескриптора хранится следующая информация:
 - а) флаги дескриптора (флаг close-on-exes — закрыть-при-вызове-exes, см. рис. 3.1 и раздел 3.14);
 - б) указатель на запись в таблице файлов.
2. Все открытые файлы представлены в ядре таблицей файлов. Каждая запись в таблице содержит:
 - а) флаги состояния файла, такие как чтение, запись, добавление в конец, синхронный режим операций ввода/вывода, неблокирующий режим (подробнее о них рассказывается в разделе 3.14);
 - б) текущая позиция в файле;
 - в) указатель на запись в таблице виртуальных узлов (v-node).
3. Каждому открытому файлу (или устройству) соответствует структура виртуального узла (v-node) с информацией о типе файла и указатели для функций, работающих с файлом. Для большинства файлов структура v-node также содержит индексный узел (i-node) файла. Эта информация считывается с диска при открытии файла, поэтому вся информация о файле сразу же становится доступной. Индексный узел (i-node) содержит, например, сведения о владельце файла, размере файла, указатели на блоки данных на диске и т. п. (Более подробно об индексных узлах мы поговорим в разделе 4.14 при описании типичной файловой системы UNIX.)

В Linux отсутствует понятие виртуальных узлов (vnode). Вместо него используются структуры индексных узлов (inode). Хотя реализация их различна, концептуально они представляют собой одно и то же. В обоих случаях индексный узел хранит информацию, специфичную для конкретной файловой системы.

Мы опустим некоторые особенности отдельных реализаций, не имеющие для нас большого значения. Например, таблица открытых дескрипторов может храниться не в таблице процессов, а в пространстве пользователя. Самы таблицы могут быть реализованы по-разному: они не обязательно должны быть массивами, вместо этого они могут быть оформлены в виде связанных списков структур. Все эти подробности несущественны для нашего обсуждения совместного доступа к файлам.

На рис. 3.1 показаны все три таблицы для одного процесса, открывшего два файла — файл стандартного ввода (дескриптор 0) и файл стандартного вывода (дескриптор 1).

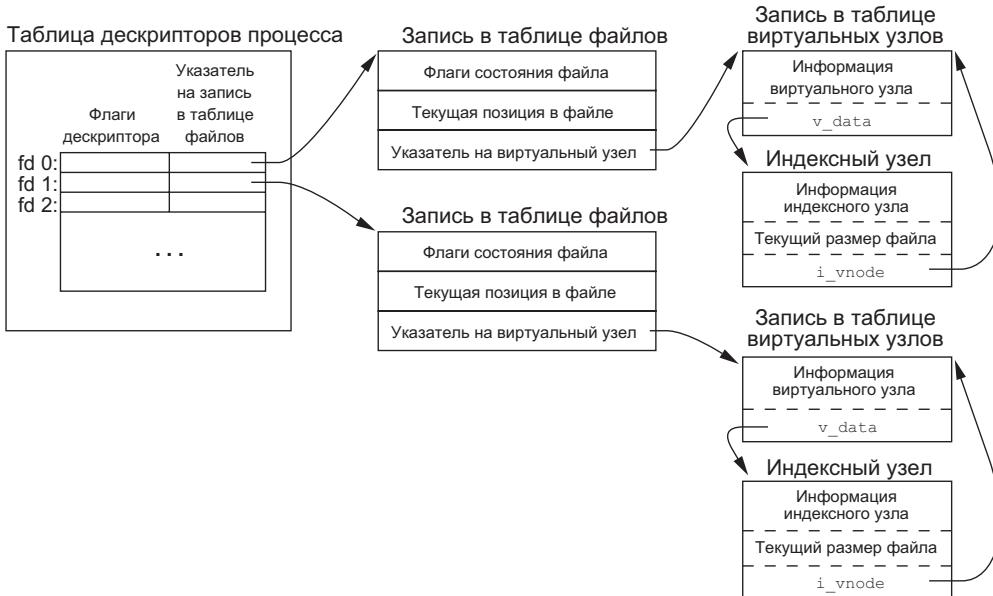


Рис. 3.1. Структуры данных ядра для открытых файлов

Взаимоотношения между таблицами определились, начиная с ранних версий UNIX [Thompson, 1978], и они оказывают весьма существенное влияние на способ совместного использования одного файла несколькими процессами. Мы еще вернемся к этому рисунку в последующих главах, когда будем обсуждать другие способы совместного использования файлов.

Идея виртуального узла (vnode) зародилась в попытках обеспечить поддержку нескольких типов файловых систем в рамках одной операционной системы. Эта работа была проделана независимо Питером Вайнбергером (Peter Weinberger) из Bell Laboratories и Биллом Джоем (Bill Joy) из Sun Microsystems. В Sun эта концепция получила название Virtual File System (виртуальная файловая система), а часть индексного узла (inode), не зависящая от типа файловой системы, была названа виртуальным узлом (vnode) [Kleiman, 1986]. Концепция виртуальных узлов распространилась на различные реализации UNIX вместе с поддержкой Network File System (NFS – сетевая файловая система) компании Sun. Первой версией из Беркли, поддерживающей виртуальные узлы, стала 4.3BSD Reno, в которую была добавлена поддержка NFS.

В SVR4 виртуальные узлы заменили индексные узлы версии SVR3. Solaris, как наследник SVR4, также использует концепцию виртуальных узлов.

Вместо разделения структур данных на виртуальные и индексные узлы в Linux используются понятия индексных узлов, независимых от типа файловой системы, и индексных узлов, зависящих от типа файловой системы.

Ситуация, когда два независимых процесса открывают один и тот же файл, показана на рис. 3.2.

Таблица дескрипторов процесса



Запись в таблице файлов

Флаги состояния файла
Текущая позиция в файле
Указатель на виртуальный узел

Таблица дескрипторов процесса



Запись в таблице файлов

Флаги состояния файла
Текущая позиция в файле
Указатель на виртуальный узел

Запись в таблице виртуальных узлов

Информация виртуального узла
v_data

Индексный узел
Информация индексного узла
Текущий размер файла
i_vnode

Рис. 3.2. Два независимых процесса открыли один и тот же файл

Здесь мы предполагаем, что первый процесс открывает этот файл с дескриптором 3, а второй открывает тот же самый файл с дескриптором 4. Каждый процесс, открывающий файл, создает собственную запись в таблице файлов, но двум этим записям соответствует единственная запись в таблице виртуальных узлов. Одна из причин создания отдельной записи в таблице файлов для каждого процесса состоит в том, что у каждого процесса должна быть собственная текущая позиция в файле.

Теперь, разобравшись со структурами данных, рассмотрим подробнее, что происходит в процессе описанных выше операций ввода/вывода.

- В таблице файлов после завершения каждой операции записи текущая позиция в файле увеличивается на количество записанных байтов. Если текущая позиция оказывается больше текущего размера файла, в таблице индексных узлов изменяется размер файла в соответствии с текущей позицией (это происходит, например, при добавлении новых данных в конец файла).
- Если файл открыт с флагом `O_APPEND`, в таблице файлов устанавливается соответствующий флаг. Каждый раз при выполнении операции записи в качестве текущей позиции принимается значение размера файла из таблицы индексных узлов. В результате запись всегда производится в конец файла.
- Если текущая позиция переносится в конец файла с помощью функции `lseek`, выполняется только перезапись значения текущего размера файла из таблицы индексных узлов в поле текущей позиции в таблице файлов. (Обратите вни-

мание: это не то же самое, что открытие файла с флагом `O_APPEND`, о чём будет говориться в разделе 3.11.)

- Функция `lseek` изменяет в таблице файлов только значение текущей позиции в файле. Никаких операций ввода/вывода при этом не производится.

Существует возможность открыть несколько дескрипторов, ссылающихся на одну запись в таблице файлов; мы увидим это в разделе 3.12 при обсуждении функции `dup`. То же происходит в результате вызова функции `fork`, когда родительский и дочерний процессы совместно используют одни и те же записи в таблице файлов для каждого из открытых дескрипторов (раздел 8.3).

Обратите внимание на различия между флагами дескриптора и флагами состояния файла. Флаги дескриптора уникальны для каждого отдельно взятого дескриптора, открытого процессом, тогда как флаги состояния файла имеют отношение ко всем дескрипторам в любом процессе, ссылающемся на одну и ту же запись в таблице файлов. Рассматривая функцию `fcntl` в разделе 3.14, мы узнаем, как получить и изменить значения флагов дескриптора и флагов состояния файла.

Все описанное выше в этом разделе прекрасно работает, когда несколько процессов читают данные из одного и того же файла. Каждый процесс имеет собственную запись в таблице файлов со своим собственным значением текущей позиции в файле. Однако можно получить совершенно неожиданные результаты, если несколько процессов пытаются выполнить запись данных в один и тот же файл. Чтобы избежать неприятных сюрпризов в будущем, необходимо разобраться с понятием атомарности операций.

3.11. Атомарные операции

Добавление данных в конец файла

Рассмотрим процесс, который дописывает данные в конец файла. Старые версии UNIX не поддерживали флаг `O_APPEND` для функции `open`, в результате приходилось писать нечто вроде:

```
if (lseek(fd, 0L, 2) < 0)          /* переместить текущую позицию в конец файла */  
    err_sys("ошибка вызова функции lseek");  
if (write(fd, buf, 100) != 100)    /* и выполнить запись */  
    err_sys("ошибка вызова функции write");
```

Такой код будет прекрасно работать в случае единственного процесса, но могут возникнуть проблемы, если добавление данных в конец файла производится сразу несколькими процессами. (Подобная ситуация возможна, например, когда несколько процессов добавляют сообщения в файл журнала.)

Допустим, что существуют два независимых процесса A и B, которые записывают данные в конец одного и того же файла. Оба процесса открыли файл без флага `O_APPEND`. Эта ситуация изображена на рис. 3.2. Каждый процесс имеет собственную запись в таблице файлов, но при этом они ссылаются на одну и ту же запись в таблице виртуальных узлов. Предположим, что процесс A вызывает функцию `lseek` и устанавливает текущую позицию файла в значение 1500 (текущий раз-

мер файла). Затем ядро приостанавливает работу процесса А и передает управление процессу В, который, в свою очередь, также вызывает функцию `lseek` и также устанавливает текущую позицию файла в значение 1500 (текущий размер файла). После этого процесс В вызывает функцию `write`, которая увеличивает текущую позицию файла до 1600. Поскольку размер файла увеличился, ядро записывает новое значение размера файла (1600) в таблицу виртуальных узлов. После этого ядро опять переключает процессы и передает управление процессу А. Когда процесс А вызывает функцию `write`, запись выполняется с места, на которое указывает значение текущей позиции в файле для процесса А, то есть 1500. В результате данные окажутся записанными поверх тех, что записаны процессом В.

Проблема в том, что операция «перейти в конец файла и записать данные» требует обращения к двум отдельным функциям (как мы только что показали). Решением проблемы было бы атомарное (относительно других процессов) выполнение операции позиционирования и записи. Никакая операция, требующая обращения более чем к одной функции, не может быть атомарной, поскольку всегда существует вероятность, что ядро временно приостановит процесс между двумя последовательными вызовами функций (как это было показано выше).

ОС UNIX позволяет выполнить эту операцию атомарно, если указать флаг `O_APPEND` при открытии файла. Как уже говорилось в предыдущем разделе, этот флаг заставит ядро выполнять перенос текущей позиции в конец файла непосредственно перед операцией записи. А кроме того, отпадает необходимость вызывать функцию `lseek` перед каждым вызовом функции `write`.

Функции `pread` и `pwrite`

Стандарт Single UNIX Specification включает расширения XSI, которые позволяют процессам атомарно выполнять операции перемещения текущей позиции и ввода/вывода. Эти расширения представлены функциями `pread` и `pwrite`.

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
```

Возвращает количество прочитанных байтов, 0 — по достижении
конца файла и -1 — в случае ошибки

```
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```

Возвращает количество записанных байтов или -1 — в случае ошибки

Вызов функции `pread` эквивалентен двум последовательным вызовам функций `lseek` и `read` со следующими различиями.

- При использовании `pread` нет возможности прервать выполнение этих двух операций.
- Значение текущей позиции в файле не изменяется.

Вызов функции `pwrite` эквивалентен двум последовательным вызовам функций `lseek` и `write` с аналогичными различиями.

Создание файла

Еще один пример атомарной операции мы видели при описании флагов `O_CREAT` и `O_EXCL` функции `open`. При одновременном указании обоих флагов функция `open` будет завершаться ошибкой, если файл уже существует. Мы также говорили, что проверка существования файла и создание файла будут выполняться атомарно. Если бы не было такой атомарной операции, мы могли бы попробовать написать нечто вроде

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("ошибка вызова функции creat");
    } else {
        err_sys("ошибка вызова функции open");
    }
}
```

Эта ситуация чревата проблемами, если файл с тем же именем будет создан другим процессом между обращениями к функциям `open` и `creat`. Если другой процесс создаст файл между вызовами этих функций и успеет туда что-либо записать, эти данные будут утеряны, когда первый процесс вызовет функцию `creat`. Объединение проверки существования файла и его создания в единую атомарную операцию решает проблему.

Вообще говоря, термин *атомарная операция* относится к таким операциям, которые могут состоять из нескольких действий. Если операция атомарна, то либо будут выполнены все необходимые действия до конца, либо не будет выполнено ни одно из них. Атомарность не допускает выполнения лишь некоторой части действий. К теме атомарных операций мы еще вернемся, когда будем рассматривать функцию `link` (раздел 4.15) и блокировку отдельных записей в файле (раздел 14.3).

3.12. Функции dup и dup2

Дубликат дескриптора существующего файла можно создать с помощью одной из следующих функций:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Возвращают новый дескриптор файла или `-1` — в случае ошибки

Функция `dup` гарантирует, что возвращаемый ею новый файловый дескриптор будет иметь наименьшее возможное значение. Вызывая функцию `dup2`, мы указываем значение нового дескриптора в аргументе `fd2`. Если дескриптор `fd2` перед вызовом функции уже был открыт, он предварительно закрывается. Если значения

аргументов *fd* и *fd2* равны, функция `dup2` вернет дескриптор *fd2*, не закрывая его. В противном случае для дескриптора *fd2* сбрасывается флаг `FD_CLOEXEC`, чтобы он оставался открытым после вызова `exec` процессом.

Новый файловый дескриптор, возвращаемый функциями, будет ссылаться на ту же запись в таблице файлов, что и дескриптор *fd*. Продемонстрируем это на рис. 3.3.

Таблица дескрипторов процесса

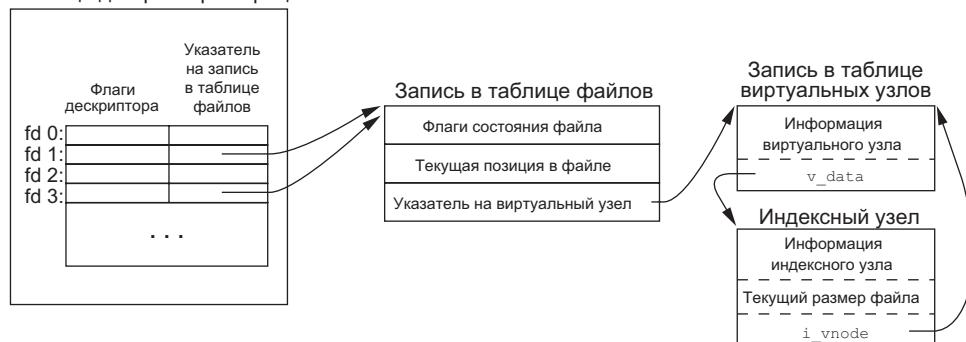


Рис. 3.3. Структуры в ядре после вызова функции `dup(1)`

На рис. 3.3 предполагается, что процесс при запуске выполняет код

```
newfd = dup(1);
```

Допустим, что следующий доступный дескриптор — число 3 (что наиболее вероятно, потому что командная оболочка уже открыла для процесса дескрипторы 0, 1 и 2). Поскольку оба дескриптора указывают на одну и ту же запись в таблице файлов, они совместно будут использовать флаги состояния файла — чтение, запись, добавление в конец и пр. и текущая позиция в файле в них будет совпадать. Каждый дескриптор имеет собственный набор флагов. Как описывается в разделе 3.14, функция `dup` всегда сбрасывает флаг `close-on-exes` («закрыть-при-вызове `exec`») в новом дескрипторе.

Дубликат дескриптора можно также создать с помощью функции `fctl`, которая описывается в разделе 3.14. На самом деле вызов

```
dup(fd);
```

эквивалентен вызову

```
fctl(fd, F_DUPFD, 0);
```

Аналогично вызов

```
dup2(fd, fd2);
```

эквивалентен паре вызовов

```
close(fd2);
fctl(fd, F_DUPFD, fd2);
```

В последнем случае вызов `dup2` не является точным эквивалентом двух последовательных вызовов `close` и `fcntl`. Имеются следующие различия.

1. Функция `dup2` действует атомарно, тогда как альтернативная форма состоит из вызовов двух функций. В результате возникает вероятность, что между обращениями к функциям `close` и `fcntl` будет вызван обработчик сигнала, который изменит дескриптор файла. (Сигналы будут обсуждаться в главе 10.) Та же проблема может возникнуть, если другой поток выполнения изменит дескриптор. (Потоки выполнения будут обсуждаться в главе 10.)
2. Существуют отличия в кодах ошибок, возвращаемых через `errno` функциями `dup2` и `fcntl`.

Системный вызов dup2 впервые появился в Version 7 и затем перекочевал в BSD. Возможность создания дубликатов дескрипторов с помощью fcntl появилась в System III и перешла в System V. В SVR3.2 была включена функция dup2, а в 4.2BSD — функция fcntl и параметр F_DUPFD. Стандарт POSIX.1 требует как наличия функции dup2, так и поддержки функцией fcntl параметра F_DUPFD.

3.13. Функции sync, fsync и fdatasync

Традиционные реализации UNIX имеют в своем распоряжении буферный кэш, или кэш страниц, через который выполняется большинство дисковых операций ввода/вывода. При записи данных в файл они, как правило, сначала помещаются ядром в один из буферов, а затем ставятся в очередь для записи на диск в более позднее время. Этот прием называется *отложенной записью*. (В главе 3 [Bach, 1986] детально рассматривается работа буферного кэша.)

Ядро обычно записывает отложенные данные на диск, когда возникает необходимость в повторном использовании буфера. Для синхронизации файловой системы на диске и содержимого буферного кэша существуют функции `sync`, `fsync` и `fdatasync`.

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);

void sync(void);
```

Возвращают 0 в случае успеха, -1 — в случае ошибки

Функция `sync` просто ставит все измененные блоки буферов в очередь для записи и возвращает управление — она не ждет, пока данные фактически запишутся на диск.

Функция `sync`, как правило, вызывается периодически (обычно каждые 30 секунд) из системного демона, часто называемого `update`. Это обеспечивает регуляр-

ную очистку буферного кэша ядра. Команда `sync(1)` также обращается к функции `sync`.

Функция `fsync` применяется только к одному файлу, который определяется файловым дескриптором `fd`, кроме того, она ожидает завершения физической записи данных на диск, прежде чем вернуть управление. В основном функция `fsync` предназначена для таких приложений, как базы данных, чтобы гарантировать запись измененных блоков с данными на диск.

Функция `fdatasync` похожа на функцию `fsync`, но действует только на содержимое файла. (При использовании `fsync` также синхронно обновляются атрибуты файла.)

Все четыре платформы, обсуждаемые в книге, поддерживают функции `sync` и `fsync`. Однако функция `fdatasync` не поддерживается в FreeBSD 8.0.

3.14. Функция `fcntl`

С помощью функции `fcntl` можно изменять свойства уже открытого файла.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */ );
```

Возвращаемое значение зависит от аргумента `cmd` (см. ниже)
в случае успеха, `-1` — в случае ошибки

В последующих примерах третий аргумент всегда будет представлен целым числом — в соответствии с комментарием в прототипе функции. Однако при обсуждении блокировки записей в разделе 14.3 третий аргумент будет представлять указатель на структуру.

Функция `fcntl` используется в пяти различных случаях.

1. Создание дубликата существующего дескриптора (`cmd = F_DUPFD` или `F_DUPFD_CLOEXEC`).
2. Получение/установка флагов дескриптора (`cmd = F_GETFD` или `F_SETFD`).
3. Получение/установка флагов состояния файла (`cmd = F_GETFL` или `F_SETFL`).
4. Проверка/установка владельца для асинхронных операций ввода/вывода (`cmd = F_GETOWN` или `F_SETOWN`).
5. Получение/установка блокировки на отдельную запись в файле (`cmd = F_GETLK`, `F_SETLK` или `F_SETLKW`).

Теперь рассмотрим первые восемь значений аргумента `cmd` из одиннадцати возможных. (Описание остальных трех, связанных с блокировкой записей, отложим до раздела 14.3.) Взгляните еще раз на рис. 3.1, так как мы будем ссылаться на флаги дескрипторов файлов, связанные с каждым дескриптором в таблице дескрипторов процесса, и на флаги состояния файла, связанные с каждым файлом в таблице файлов.

F_DUPFD Создает дубликат дескриптора *fd*. Новый файловый дескриптор передается в вызывающую программу в виде возвращаемого значения. Это будет наименьший неиспользованный дескриптор, значение которого больше или равно третьему аргументу (заданному в виде целого числа). Новый дескриптор будет ссылаться на ту же запись в таблице файлов, что и *fd* (рис. 3.3). Но при этом новый дескриптор будет иметь собственный набор флагов, а флаг **FD_CLOEXEC** будет сброшен. (Это означает, что дескриптор останется открытм после вызова функции *exec*, которая обсуждается в главе 8.)

F_DUPFD_CLOEXEC Создает дубликат дескриптора и устанавливает флаг **FD_CLOEXEC** для нового дескриптора. Возвращает новый файловый дескриптор.

F_GETFD Возвращает флаги дескриптора *fd*. В настоящее время определен только один флаг — **FD_CLOEXEC**.

F_SETFD Устанавливает флаги дескриптора *fd*. Новые значения флагов берутся из третьего аргумента (заданного в виде целого числа).

Вы должны знать, что существуют программы, которые работают с флагами дескрипторов, но не используют константу FD_CLOEXEC. Вместо этого они используют значение 0 (сбросить флаг FD_CLOEXEC) или 1 (установить флаг FD_CLOEXEC).

F_GETFL Возвращает флаги состояния файла *fd*. Мы уже описывали флаги состояния файла, когда обсуждали функцию *open*. Они перечислены в табл. 3.4.

К сожалению, пять флагов, **O_RDONLY**, **O_WRONLY**, **O_RDWR**, **O_EXEC** и **O_SEARCH**, представлены числовыми значениями, а не отдельными битами, которые можно было бы проверить. (Как уже говорилось, первые три в силу исторических причин обычно имеют значения 0, 1 и 2 соответственно. Кроме того, эти значения являются взаимоисключающими — для файла может быть установлен только один из этих флагов.) Поэтому следует сначала применить маску **O_ACCMODE**, чтобы выделить режимы доступа, и лишь потом сравнивать полученный результат с любым из пяти значений.

F_SETFL Устанавливает флаги состояния файла. Новые значения флагов берутся из третьего аргумента (заданного в виде целого числа). Изменить можно только флаги **O_APPEND**, **O_NONBLOCK**, **O_SYNC**, **O_DSYNC**, **O_RSYNC**, **O_FSYNC** и **O_ASYNC**.

F_GETOWN Возвращает идентификатор процесса или группы процессов, которые в настоящее время получают сигналы **SIGIO** и **SIGURG**. Эти сигналы асинхронного ввода/вывода рассматриваются в разделе 14.5.2.

F_SETOWN Назначает идентификатор процесса или группы процессов, которые будут получать сигналы **SIGIO** и **SIGURG**. Положительное значение аргумента *arg* интерпретируется как идентификатор процесса, отрицательное — как идентификатор группы процессов, эквивалентный абсолютному значению аргумента *arg*.

Значение, возвращаемое функцией *fcntl*, зависит от конкретной команды. Все команды возвращают **-1** в случае ошибки и другие значения — в случае успеха. Команды **F_DUPFD**, **F_GETFD**, **F_GETFL** и **F_GETOWN** возвращают специальные значения. Первые две — дескриптор файла, другие две — соответствующие флаги и последняя — идентификатор процесса (положительное значение) или группы процессов (отрицательное значение).

Таблица 3.4. Флаги состояния файла, используемые функцией fcntl

Флаг состояния файла	Описание
O_RDONLY	Файл открыт только для чтения
O_WRONLY	Файл открыт только для записи
O_RDWR	Файл открыт для чтения и записи
O_EXEC	Файл открыт только для выполнения
O_SEARCH	Каталог открыт только для поиска
O_APPEND	Файл открыт для добавления в конец
O_NONBLOCK	Неблокирующий режим
O_SYNC	Ожидать завершения операции записи (данных и атрибутов)
O_DSYNC	Ожидать завершения операции записи (только данных)
O_RSYNC	Синхронизировать операции чтения и записи
O_FSYNC	Ожидать завершения операции записи (только FreeBSD и Mac OS X)
O_ASYNC	Асинхронный режим ввода/вывода (только FreeBSD и Mac OS X)

Пример

Программа в листинге 3.4 принимает из командной строки один аргумент, который определяет дескриптор файла, и выводит значения флагов состояния файла для этого дескриптора.

Листинг 3.4. Вывод флагов состояния файла для заданного дескриптора

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int      val;

    if (argc != 2)
        err_quit("Использование: a.out <номер_дескриптора>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("ошибка fcntl для дескриптора %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("только для чтения");
        break;

    case O_WRONLY:
```

```

printf("только для записи");
break;

case O_RDWR:
printf("для чтения и для записи");
break;

default:
err_dump("неизвестный режим доступа");
}

if (val & O_APPEND)
printf(", добавление в конец");
if (val & O_NONBLOCK)
printf(", неблокирующий режим");
if (val & O_SYNC)
printf(", синхронный режим записи");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) && (O_FSYNC != O_SYNC)
if (val & O_FSYNC)
printf(", синхронный режим записи");
#endif

putchar('\n');
exit(0);
}

```

Обратите внимание, что мы использовали макроопределение проверки функциональных особенностей `_POSIX_C_SOURCE` и условную компиляцию для флагов, которые не являются частью стандарта POSIX.1. Следующий сценарий демонстрирует работу программы, когда она запускается из `bash` (Bourne-again shell). В зависимости от используемой командной оболочки полученные результаты могут несколько отличаться от приведенных здесь.

```

$ ./a.out 0 < /dev/tty
Только для чтения
$ ./a.out 1 > temp.foo
$ cat temp.foo
Только для записи
$ ./a.out 2 2>>temp.foo
Только для записи, добавление в конец
$ ./a.out 5 5<>temp.foo
Для чтения и для записи

```

Выражение `5<>temp.foo` открывает файл `temp.foo` для чтения и записи на дескрипторе 5.

Пример

Изменяя флаги дескриптора или флаги состояния файла, необходимо сначала получить все имеющиеся значения флагов, изменить желаемые и затем записать полученное значение обратно. Нельзя просто изменить отдельные флаги с помощью команд `F_SETFD` или `F_SETFL`, поскольку так можно сбросить другие флаги, которые были установлены.

В листинге 3.5 приводится функция, которая устанавливает один или более флагов состояния файла.

Листинг 3.5. Включает один или более флагов состояния файла

```
#include "apue.h"
#include <fcntl.h>

void
set_f1(int fd, int flags) /* flags – флаги, которые нужно включить */
{
    int      val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("ошибка выполнения команды F_GETFL функции fcntl");

    val |= flags; /* включить флаги */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("ошибка выполнения команды F_SETFL функции fcntl");
}
```

Если изменить код в середине на

```
val &= .flags; /* выключить флаги */
```

мы получим функцию `clr_f1`, которая будет использоваться в примерах ниже. В этой строке производится объединение по И (AND) текущего значения переменной `val` с логическим дополнением до единицы значения аргумента `flags`.

Если в начало программы в листинге 3.3 добавить строку

```
set_f1(STDOUT_FILENO, O_SYNC);
```

она включит режим синхронной записи. В результате каждый вызов функции `write` будет ждать завершения физической записи данных на диск, прежде чем вернуть управление. Обычно в UNIX функция `write` лишь ставит записываемые данные в очередь, а собственно запись на диск производится несколько позднее. Флаг `O_SYNC` часто используется в системах управления базами данных, так как дает дополнительные гарантии своевременной записи данных на диск и их сохранности в случае отказа системы.

Предполагается, что использование флага `O_SYNC` увеличивает общее время работы программы. Для проверки воспользуемся программой из листинга 3.3. Скопируем с ее помощью 492,6 Мбайт данных из одного файла на диске в другой и сравним результаты с версией программы, которая устанавливает флаг `O_SYNC`. Результаты, полученные нами в Linux с файловой системой `ext4`, приводятся в табл. 3.5.

Во всех шести случаях замеры производились со значением `BUFFSIZE`, равным 4096. Результаты в табл. 3.3 были получены при чтении файла с диска и записи в устройство `/dev/null`, то есть запись на диск не производилась. Вторая строка в табл. 3.5 соответствует чтению файла с диска и записи в другой файл на диске. По этой причине значения времени в первой и во второй строках отличаются. При записи в файл на диске системное время увеличивается, потому что в этом случае ядро должно скопировать данные, полученные от процесса, и поставить их в очередь на запись, которая будет выполняться драйвером диска. Мы ожидали, что общее время также увеличится при записи файла на диск.

Таблица 3.5. Результаты проверки производительности различных режимов синхронизации в Linux с файловой системой ext4

Операция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)
Время чтения, взятое из табл. 3.3, для <code>BUFFSIZE</code> = 4096	0,03	0,58	8,62
Нормальный режим записи файла на диск	0,00	1,05	9,70
Запись на диск с флагом <code>O_SYNC</code>	0,02	1,09	10,28
Запись на диск с последующим вызовом <code>fdatasync</code>	0,02	1,14	17,93
Запись на диск с последующим вызовом <code>fsync</code>	0,00	1,19	18,17
Запись на диск с флагом <code>O_SYNC</code> и последующим вызовом <code>fsync</code>	0,02	1,15	17,88

Включая синхронную запись, мы ожидали существенного увеличения системного и общего времени. Но, как видно в третьей строке, синхронная запись выполняется практически за то же время, что и отложенная. Это означает, что либо Linux проделывает одинаковый объем работы при отложенной и синхронной записи (что маловероятно), либо флаг `O_SYNC` не оказывает должного эффекта в ней. В данном случае Linux не позволяет устанавливать флаг `O_SYNC` с помощью `fcntl` и не возвращает признак ошибки (но этот флаг будет учитываться, если установить его при открытии файла).

Общее время в последних трех строках отражает дополнительные затраты времени на ожидание фактической записи на диск. После выполнения синхронной записи ожидается, что вызов `fsync` не будет иметь никакого эффекта. Данное предположение должна была бы подтвердить последняя строка, но поскольку флаг `O_SYNC` не оказывает ожидаемого эффекта, последняя строка содержит значение, практически равное значению в пятой строке.

В табл. 3.6 приводятся результаты тех же экспериментов в Mac OS X 10.6.8 с файловой системой `HFS`. Обратите внимание, что полученные значения времени полностью соответствуют нашим ожиданиям: синхронная запись оказывается более дорогостоящей по сравнению с отложенной, а вызов функции `fsync` не сказывается на времени при использовании синхронного режима записи. Отметьте также, что добавление вызова функции `fsync` после выполнения обычной отложенной записи не дает существенного прироста времени. Это, скорее всего, говорит о том, что данные из кэша переписывались операционной системой на диск по мере поступления новых данных, так что к моменту вызова функции `fsync` в кэше их осталось не так много.

Сравните эффект от использования функций `fsync` и `fdatasync`, которые обновляют содержимое файла при обращении к ним, и действие флага `O_SYNC`, который обновляет содержимое файла при каждой операции записи. Производительность каждой из альтернатив зависит от множества факторов, включая реализацию самой операционной системы, производительность дискового устройства и тип файловой системы.

Таблица 3.6. Результаты проверки производительности различных режимов синхронизации файловой системы HFS в Mac OS X

Операция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)
Запись в устройство <code>/dev/null</code>	0,14	1,02	5,28
Нормальный режим записи файла на диск	0,14	3,21	17,04
Запись на диск с флагом <code>O_SYNC</code>	0,39	16,89	60,82
Запись на диск с последующим вызовом <code>fsync</code>	0,13	3,07	17,10
Запись на диск с флагом <code>O_SYNC</code> и последующим вызовом <code>fsync</code>	0,39	18,18	62,39

Этот пример наглядно демонстрирует, для чего нужна функция `fcntl`. Наша программа работает с дескриптором (стандартный вывод), не зная названия файла, открытого командной оболочкой на этом дескрипторе. Мы лишиены возможности установить флаг `O_SYNC` при открытии файла, так как его открывает командная оболочка. С помощью функции `fcntl` можно изменить свойства дескриптора, зная только дескриптор открытого файла. Позже мы рассмотрим еще одну область применения функции `fcntl`, когда будем рассказывать о неблокирующих операциях ввода/вывода для неименованных каналов (раздел 15.2), поскольку при работе с ними нам доступен только дескриптор.

3.15. Функция `ioctl`

Функция `ioctl` всегда была универсальным инструментом ввода/вывода. Все, что невозможно выразить с помощью функций, описанных в этой главе, как правило, делается с помощью `ioctl`. Возможности этой функции чаще всего использовались в операциях терминального ввода/вывода. (Когда мы доберемся до главы 18, то увидим, что стандарт POSIX.1 заменил операции терминального ввода/вывода отдельными функциями.)

```
#include <unistd.h>    /* System V */
#include <sys/ioctl.h> /* BSD и Linux */
int ioctl(int fd, int request, ...);
```

Возвращает `-1` в случае ошибки, другие значения — в случае успеха

Функция `ioctl` включена в стандарт Single UNIX Specification только как расширение для работы с устройствами STREAMS [Rago, 1993], но в SUSv4 ей присвоен статус устаревшей. Различные версии UNIX используют `ioctl` для выполнения самых разнообразных операций с устройствами. Некоторые реализации даже расширили ее функциональность для использования с обычными файлами.

Приведенный выше прототип функции определяется стандартом POSIX.1. В операционных системах FreeBSD 8.0 и Mac OS X 10.6.8 второй аргумент определен как `unsigned long`. Это не имеет большого значения, так как в качестве второго аргумента всегда передается имя константы, определяемой в заголовочном файле.

Согласно стандарту ISO C, необязательные аргументы обозначены многоточием. Однако в большинстве случаев передается только один дополнительный аргумент — указатель на переменную или структуру.

В этом прототипе мы указали только те заголовочные файлы, которые требуются для самой функции `ioctl`. Но, как правило, при работе с ней необходимо подключать дополнительные заголовочные файлы для конкретных устройств. Например, все команды `ioctl` для операций терминального ввода/вывода, определяемые стандартом POSIX.1, требуют подключения заголовочного файла `<termios.h>`.

Каждый драйвер устройства может определять собственный набор команд `ioctl`. Тем не менее операционная система предоставляет набор универсальных команд `ioctl` для различных классов устройств. Примеры некоторых категорий универсальных команд `ioctl`, поддерживаемых FreeBSD, приводятся в табл. 3.7.

Таблица 3.7. Команды `ioctl` в OC FreeBSD

Категория	Имена констант	Заголовочный файл	Количество команд
Метки диска	DIOxxx	<code><sys/disklabel.h></code>	4
Файловый ввод/вывод	FIOxxx	<code><sys/filio.h></code>	14
Ввод/вывод для накопителей на магнитной ленте	MTIOxxx	<code><sys/mtio.h></code>	11
Ввод/вывод для сокетов	SIOxxx	<code><sys/sockio.h></code>	73
Терминальный ввод/вывод	TIOxxx	<code><sys/ttycom.h></code>	43

Операции с накопителями на магнитной ленте позволяют записывать на ленту признак конца файла, перематывать ленту в начало, перемещаться вперед через заданное число файлов или записей и т. п. Ни одну из этих операций нельзя достаточно просто выразить в терминах других функций, описанных в данной главе (`read`, `write`, `lseek` и т. д.). Поэтому простейший способ взаимодействия с такими устройствами всегда заключался в управлении ими через функцию `ioctl`.

Мы еще вернемся к функции `ioctl` в разделе 18.12, где с ее помощью будем получать и изменять размер окна терминала, и в разделе 19.7, когда будем исследовать расширенные возможности псевдотерминалов.

3.16. /dev/fd

В современных операционных системах имеется каталог `/dev/fd`, в котором находятся файлы с именами 0, 1, 2 и т. д. Открытие файла `/dev/fd/n` эквивалентно созданию дубликата дескриптора с номером *n*, при условии, что дескриптор *n* открыт.

Поддержка каталога `/dev/fd` была реализована Томом Даффом (*Tom Duff*) и впервые появилась в 8-й редакции *Research UNIX System*. Эта особенность поддерживается всеми четырьмя операционными системами, о которых идет речь в данной книге: *FreeBSD 8.0*, *Linux 3.2.0*, *Mac OS X 10.6.8* и *Solaris 10*. Она не является частью стандарта *POSIX.1*.

Большинство систем игнорируют аргумент `mode` в вызове

```
fd = open("/dev/fd/0", mode);
```

но есть и такие, которые требуют, чтобы он представлял подмножество флагов, которые использовались при открытии оригинального файла (в данном случае файл стандартного ввода). Поскольку данный вызов эквивалентен вызову

```
fd = dup(0);
```

дескрипторы 0 и `fd` будут совместно использовать одну и ту же запись в таблице файлов (см. рис. 3.3). Например, если дескриптор 0 открыт только для чтения, дескриптор `fd` также будет доступен только для чтения. Даже если система игнорирует режим открытия дескриптора и вызов

```
fd = open("/dev/fd/0", O_RDWR);
```

не завершится ошибкой, мы все равно не сможем ничего записать в файл с дескриптором `fd`.

Реализация `/dev/fd` в Linux является исключением. Она отображает файловые дескрипторы в символьские ссылки, указывающие на фактические файлы. Например, при попытке открыть файл `/dev/fd/0` в действительности открывается файл, связанный со стандартным вводом. Поэтому возвращаемый режим нового файлового дескриптора никак не связан с режимом файлового дескриптора в `/dev/fd`.

Кроме того, имя каталога `/dev/fd` можно использовать в аргументе `path` функции `creat`, равно как и в функции `open` с флагом `O_CREAT`. Это позволяет программам, обращающимся к `creat`, продолжать работу, даже если аргумент `path`, например, содержит строку `/dev/fd/1`.

Не используйте этот прием в Linux. Так как в Linux используются символьские ссылки на фактические файлы, вызов `creat` с файлом в `/dev/fd` приведет к усечению файла, на который указывает ссылка.

В некоторых системах имеются файлы `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, эквивалентные файлам `/dev/fd/0`, `/dev/fd/1` и `/dev/fd/2` соответственно.

Файлы из каталога `/dev/fd` в основном используются командными оболочками. Это позволяет программам, требующим указания имени файла, работать со стандартными устройствами ввода и вывода так же, как с обычными файлами. Например, в следующем примере программа `cat(1)` использует в качестве входного файла стандартный ввод, обозначаемый символом «-»:

```
filter file2 | cat file1 - file3 | lpr
```

Сначала `cat` читает содержимое файла `file1`, затем файл стандартного ввода (результат работы утилиты `filter`, обрабатывающей файл `file2`) и, наконец, файл

`file3`. Если система поддерживает `/dev/fd`, можно опустить символ `<->` и переписать команду так:

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

Символ `<->` в качестве аргумента командной строки для обозначения файла стандартного ввода или стандартного вывода — своего рода ляп, который присутствует во многих программах. Например, на месте первого файла он будет очень похож на начало другого аргумента командной строки. Использование `/dev/fd` — это шаг к единообразию и порядку.

3.17. Подведение итогов

В этой главе были описаны базовые функции ввода/вывода в системе UNIX. Их часто называют функциями небуферизованного ввода/вывода, потому что каждый вызов `read` или `write` обращается к системному вызову ядра. Мы увидели, как влияет изменение размера буфера ввода/вывода на время чтения файла. Мы также рассмотрели несколько способов записи данных на диск и их влияние на производительность приложения.

Мы познакомились с атомарными операциями, когда рассматривали дописывание данных в один файл несколькими процессами и создание одного и того же файла несколькими процессами. Мы также увидели структуры данных, используемые ядром для организации совместного доступа к информации об открытых файлах. В дальнейшем мы еще вернемся к этим структурам.

Также были описаны функции `ioctl` и `fcntl`. Мы еще поговорим о них в главе 14, где `fcntl` будет использоваться для организации блокировки отдельных записей в файле. В главах 18 и 19 мы используем функцию `ioctl` для работы с терминальными устройствами.

Упражнения

- 3.1** Действительно ли функции чтения и записи файлов, описанные в данной главе, являются небуферизованными? Объясните почему.
- 3.2** Напишите свою версию функции `dup2`, которая реализует функциональность, описанную в разделе 3.12, но без использования функции `fcntl`. Предусмотрите обработку ошибок.
- 3.3** Предположим, что некоторый процесс вызывает следующие функции:

```
fd1 = open(path, oflags);
fd2 = dup(fd1);
fd3 = open(path, oflags);
```

Нарисуйте диаграмму, подобную той, что приведена на рис. 3.3. На какой дескриптор окажет влияние функция `fcntl`, если ей передать в качестве аргументов `fd1` и `F_SETFD`? На какой дескриптор окажет влияние функция `fcntl`, если ей передать в качестве аргументов `fd1` и `F_SETFL`?

- 3.4 Во многих программах можно наблюдать следующую последовательность операций:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

Чтобы понять, для чего нужен условный оператор `if`, предположите, что `fd` изначально имеет значение 1, и нарисуйте картинку, отображающую, что происходит со всеми тремя дескрипторами и соответствующими записями в таблице файлов после каждого вызова функции `dup2`. Затем нарисуйте аналогичную картинку, исходя из предположения, что изначально `fd` имел значение 3.

- 3.5 Командные оболочки Bourne shell, Bourne-again shell и Korn shell предусматривают такую нотацию:

digit1>&*digit2*

Она говорит о том, что дескриптор *digit1* должен быть перенаправлен в тот же файл, что и дескриптор *digit2*. Чем отличаются следующие две команды (подсказка: командные оболочки обрабатывают командную строку слева направо):

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

- 3.6 Если файл открыт для чтения и записи с флагом `O_APPEND`, можно ли читать данные из произвольного места в файле с помощью функции `lseek`? Можно ли воспользоваться функцией `lseek` для изменения данных в произвольном месте в файле? Напишите программу, чтобы получить ответы на эти вопросы.

4

Файлы и каталоги

4.1. Введение

В предыдущей главе обсуждались базовые функции, выполняющие операции ввода/вывода. Основное внимание уделялось операциям ввода/вывода с обычными файлами: открытие, чтение файла или запись в файл. Теперь мы рассмотрим дополнительные особенности файловой системы и свойства файла. Сначала познакомимся с функцией `stat`, а затем исследуем каждый элемент структуры `stat` и все существующие атрибуты файлов. Попутно рассмотрим все функции, изменяющие эти атрибуты: владельца, права доступа и др. Также более подробно рассмотрим структуру файловой системы UNIX и символические ссылки. А в завершение перейдем к функциям для работы с каталогами и напишем функцию, выполняющую обход дерева каталогов.

4.2. Функции `stat`, `fstat` и `lstat`

В этой главе основное внимание уделяется четырем функциям семейства `stat` и информации, которую они возвращают.

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

int stat(const char *restrict pathname, struct stat *restrict buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *restrict pathname, struct stat restrict buf);
int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

Все четыре возвращают 0 в случае успеха, -1 — в случае ошибки

Функция `stat` возвращает структуру с информацией о файле, указанном в аргументе `pathname`. Функция `fstat` возвращает информацию об открытом файле по его дескриптору `fd`. Функция `lstat` похожа на функцию `stat`, но когда ей передается имя символьской ссылки, она возвращает сведения о самой символьской ссылке, а не о файле, на который она ссылается. (Эта функция понадобится

нам в разделе 4.21, когда мы будем спускаться вниз по дереву каталогов. Более подробно символические ссылки описываются в разделе 4.17.)

Функция `fstatat` возвращает информацию о файле, относительный путь *pathname* к которому начинается в открытом каталоге, представленном дескриптором *fd*. Аргумент *flag* определяет правила следования по символическим ссылкам: если установлен флаг `AT_SYMLINK_NOFOLLOW`, функция `fstatat` не будет следовать по символическим ссылкам, а вернет информацию о самой ссылке. Иначе она будет выполнять переходы и возвращать информацию о файлах, на которые эти ссылки указывают. Если в аргументе *fd* передать значение `AT_FDCWD`, а в аргументе *pathname* — строку относительного пути, путь к файлу *pathname* будет откладываться относительно текущего каталога. Если в аргументе *pathname* передать строку абсолютного пути, аргумент *fd* будет игнорироваться. В этих двух случаях `fstatat` действует подобно `stat` или `lstat`, в зависимости от значения аргумента *flag*.

Второй аргумент, *buf*, является указателем на структуру, которую функция заполнит информацией. Определение структуры может отличаться в разных реализациях, но основная ее часть выглядит так:

```
struct stat {
    mode_t          st_mode;      /* тип файла и режим (права доступа) */
    ino_t           st_ino;       /* номер индексного узла */
    dev_t           st_dev;       /* номер устройства (файловой системы) */
    dev_t           st_rdev;      /* номер устройства для специальных файлов */
    nlink_t         st_nlink;     /* количество ссылок */
    uid_t           st_uid;       /* идентификатор пользователя владельца */
    gid_t           st_gid;       /* идентификатор группы владельца */
    off_t           st_size;      /* размер в байтах, для обычных файлов */
    struct timespec st_atim;     /* время последнего обращения к файлу */
    struct timespec st_mtим;     /* время последнего изменения файла */
    struct timespec st_ctim;     /* время последнего изменения состояния файла */
    blksize_t        st_blksize;   /* оптимальный размер блока ввода/вывода */
    blkcnt_t        st_blocks;    /* количество занятых дисковых блоков */
};
```

Стандарт POSIX.1 не требует наличия полей `st_rdev`, `st_blksize` и `st_blocks`. Они определены как расширения XSI в стандарте Single UNIX Specification.

Структура `timespec` определяет время в секундах и наносекундах и содержит по меньшей мере два поля:

```
time_t  tv_sec;
long    tv_nsec;
```

До появления редакции стандарта 2008 года поля со значениями времени назывались `st_atime`, `st_mtime` и `st_ctime` и имели тип `time_t` (время в секундах). Структура `timespec` позволяет хранить отметки времени (timestamps) с более высокой точностью. Старые имена можно определить в терминах членов `tv_sec` структуры для совместимости. Например, `st_atime` можно определить как `st_atim.tv_sec`.

Обратите внимание, что большинство членов структуры `stat` имеют тот или иной элементарный системный тип данных (раздел 2.8). А теперь исследуем атрибуты файла и познакомимся ближе с каждым членом структуры.

Вероятно, наиболее часто функцию `stat` использует команда `ls -l`, которая выводит полную информацию о файле.

4.3. Типы файлов

Мы уже упоминали файлы двух типов: обычные файлы и каталоги. Большинство файлов в UNIX являются либо обычными файлами, либо каталогами, но есть и другие типы файлов. Перечислим их.

1. Обычный файл — наиболее распространенный тип файлов, хранящих данные в некотором виде. Ядро UNIX не делает различий между текстовыми и двоичными файлами. Интерпретация содержимого файла полностью зависит от прикладной программы, обрабатывающей файл.

Одно из наиболее известных исключений из этого правила — выполняемые файлы. Чтобы запустить программу, ядро должно опознать формат файла. Все двоичные выполняемые файлы следуют конкретному формату, который позволяет ядру определять, куда загрузить выполняемый код и данные программы.

2. Файл каталога. Файлы этого типа содержат имена других файлов и ссылки на информацию о них. Любой процесс, обладающий правом на чтение каталога, может прочитать его содержимое, но только ядро обладает правом на запись непосредственно в файл каталога. Чтобы внести изменения в каталог, процессы должны пользоваться функциями, обсуждаемыми в данной главе.
3. Специальный файл блочного устройства. Этот тип файлов обеспечивает буферизованный ввод/вывод фиксированными блоками для таких устройств, как жесткие диски.

Обратите внимание, что FreeBSD больше не поддерживает специальные файлы блочных устройств. Доступ ко всем устройствам осуществляется через специальный символьный интерфейс.

4. Специальный файл символьного устройства. Этот тип файлов обеспечивает небуферизованный ввод/вывод для устройств с переменным размером блока. Все устройства в системе являются либо специальными файлами блочных устройств, либо специальными файлами символьных устройств.
5. FIFO, или именованный канал. Этот тип файлов используется для организации обмена информацией между процессами. Именованные каналы описываются в разделе 15.5.
6. Сокет. Этот тип файлов используется для обмена информацией между процессами через сетевые соединения. Сокеты можно также применять для обмена информацией между процессами на одной и той же машине. Мы будем использовать сокеты для взаимодействий между процессами в главе 16.
7. Символическая ссылка. Файлы этого типа являются ссылками на другие файлы. Более подробно о символических ссылках мы поговорим в разделе 4.17.

Тип файла хранится в поле `st_mode` структуры `stat`. Определить тип файла можно с помощью макроопределений, перечисленных в табл. 4.1. В качестве аргумента каждое из них принимает значение поля `st_mode` структуры `stat`.

Таблица 4.1. Макросы для определения типа файла из <sys/stat.h>

Макроопределение	Тип файла
<code>S_ISREG()</code>	Обычный файл
<code>S_ISDIR()</code>	Каталог
<code>S_ISCHR()</code>	Специальный файл символьного устройства
<code>S_ISBLK()</code>	Специальный файл блочного устройства
<code>S_ISFIFO()</code>	Канал (именованный или неименованный)
<code>S_ISLNK()</code>	Символическая ссылка
<code>S_ISSOCK()</code>	Сокет

Стандарт POSIX.1 допускает реализацию и представление объектов межпроцессных взаимодействий (IPC), таких как очереди сообщений и семафоры, в виде файлов. Макроопределения из табл. 4.2 позволяют определить тип объекта IPC из структуры `stat`. Главное их отличие от макросов, перечисленных в табл. 4.1, в том, что аргументом для них является указатель на структуру `stat`, а не значение поля `st_mode`.

Таблица 4.2. Макросы для определения типа объекта IPC из <sys/stat.h>

Макроопределение	Тип файла
<code>S_TYPEISMQ()</code>	Очередь сообщений
<code>S_TYPEISSEM()</code>	Семафор
<code>S_TYPEISSHM()</code>	Объект разделяемой памяти

Очереди сообщений, семафоры и объекты разделяемой памяти будут рассматриваться в главе 15. Однако ни одна из реализаций, обсуждаемых в данной книге, не представляет эти объекты в виде файлов.

Пример

Программа в листинге 4.1 выводит тип файла для каждого аргумента командной строки.

Листинг 4.1. Вывод типа файла для каждого аргумента командной строки

```
#include "apue.h"
```

```
int
main(int argc, char *argv[])
{
    int          i;
    struct stat buf;
```

```

char      *ptr;

for (i = 1; i < argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &buf) < 0) {
        err_ret("ошибка вызова функции lstat");
        continue;
    }
    if (S_ISREG(buf.st_mode))
        ptr = "обычный файл";
    else if (S_ISDIR(buf.st_mode))
        ptr = "каталог";
    else if (S_ISCHR(buf.st_mode))
        ptr = "файл символьного устройства";
    else if (S_ISBLK(buf.st_mode))
        ptr = "файл блочного устройства";
    else if (S_ISFIFO(buf.st_mode))
        ptr = "fifo";
    else if (S_ISLNK(buf.st_mode))
        ptr = "символическая ссылка";
    else if (S_ISSOCK(buf.st_mode))
        ptr = "сокет";
    else
        ptr = "** неизвестный тип файла **";
    printf("%s\n", ptr);
}
exit(0);
}

```

Пример вывода программы из листинга 4.1:

```

$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: обычный файл
/etc: каталог
/dev/log: сокет
/dev/tty: файл символьного устройства
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: файл блочного устройства
/dev/cdrom: символическая ссылка

```

(Символ обратного слеша в конце первой строки сообщает командной оболочки, что ввод команды не закончен. В таких случаях командная оболочка выводит на следующей строке вторичное приглашение к вводу — символ >.) Мы нарочно использовали функцию `lstat` вместо `stat`, чтобы обнаружить символические ссылки. Используя функцию `stat`, мы никогда не увидели бы их.

Ранние версии UNIX не имели макроопределений `S_ISxxx`. Вместо этого необходимо было выполнять объединение по И (AND) значения `st_mode` с маской `S_IFMT` и затем сравнивать результат с константами `S_IFxxx`. Определение этой маски и связанных с ней констант в большинстве систем находится в файле `<sys/stat.h>`. Заглянув в этот файл, мы обнаружим, что макрокоманда `S_ISDIR` определена примерно так:

```
#define S_ISDIR(mode) (((mode) & S_IFMT) == S_IFDIR)
```

Мы уже говорили, что обычные файлы являются самыми распространенными, но было бы интересно узнать, какой процент от всех файлов в данной системе занимают файлы каждого типа. В табл. 4.3 приводится количество файлов каждого типа и его процентное выражение в ОС Linux, используемой в качестве однопользовательской рабочей станции. Эти данные получены с помощью программы, которую мы продемонстрируем в разделе 4.22.

Таблица 4.3. Количество файлов различных типов и его процентное выражение

Тип файла	Количество	Процент от общего числа
Обычные файлы	415 803	79,77
Каталоги	62 197	11,93
Символические ссылки	40 018	8,25
Файлы символьных устройств	155	0,03
Файлы блочных устройств	47	0,01
Сокеты	45	0,01
FIFO	0	0,00

4.4. set-user-ID и set-group-ID

С каждым процессом связаны шесть или более идентификаторов. Все они перечислены в табл. 4.4.

Таблица 4.4. Идентификаторы пользователя и группы, связанные с каждым процессом

Реальный идентификатор пользователя Реальный идентификатор группы	Определяет, кто мы на самом деле
Эффективный идентификатор пользователя Эффективный идентификатор группы Идентификаторы дополнительных групп	Используются при проверке прав доступа к файлам
Сохраненный идентификатор пользователя Сохраненный идентификатор группы	Идентификаторы, сохраняемые функциями exec

- Реальные идентификаторы пользователя и группы определяют, кто мы на самом деле. Эти идентификаторы извлекаются из файла паролей во время входа в систему. Обычно в течение сеанса значения этих идентификаторов не меняются, хотя процессы, обладающие правами суперпользователя, имеют возможность изменять их, о чем мы поговорим в разделе 8.11.
- Эффективные идентификаторы пользователя и группы и идентификаторы дополнительных групп определяют права доступа к файлам, о чем мы поговорим в следующем разделе. (Определение дополнительных групп было дано в разделе 1.8.)

- Сохраненные идентификаторы пользователя и группы — это копии эффективных идентификаторов, которые создаются в момент запуска программы. Мы расскажем о назначении этих двух идентификаторов, когда будем описывать функцию `setuid` в разделе 8.11.

Сохраненные идентификаторы перешли в разряд обязательных для реализации в соответствии с версией POSIX.1 2001 года. В более ранних версиях POSIX они находились в категории необязательных. Приложение может проверить наличие константы `_POSIX_SAVED_IDS` на этапе компиляции или вызвать функцию `sysconf` с аргументом `_SC_SAVED_IDS` на этапе выполнения, чтобы определить, поддерживает ли реализация эту функциональную возможность.

Обычно эффективные идентификаторы пользователя и группы совпадают с соответствующими реальными идентификаторами.

У каждого файла в системе есть владелец и группа-владелец. Идентификатор владельца файла хранится в поле `st_uid` структуры `stat`, а идентификатор группы владельца — в поле `st_gid`.

В момент запуска файла программы эффективными идентификаторами пользователя и группы процесса обычно становятся соответствующие реальные идентификаторы. Но можно также установить специальный флаг в поле `st_mode`, который как бы говорит: «При запуске этого файла в качестве эффективного идентификатора процесса взять идентификатор пользователя владельца файла (`st_uid`)». Точно так же в поле `st_mode` можно установить другой флаг, который назначит в качестве эффективного идентификатора группы идентификатор группы владельца файла (`st_gid`). Эти два флага в поле `st_mode` называются битами *set-user-ID* и *set-group-ID* соответственно.

Например, если владельцем файла является суперпользователь и у файла установлен бит *set-user-ID*, во время работы программы соответствующий процесс будет обладать правами суперпользователя. Это происходит независимо от реального идентификатора пользователя процесса, запустившего файл. Так, системная утилита UNIX, позволяющая любому пользователю изменять свой пароль, `passwd(1)`, является программой с установленным битом *set-user-ID*. Это необходимо, чтобы утилита могла записать новый пароль в файл паролей (обычно это файл `/etc/passwd` или `/etc/shadow`), который должен быть доступен на запись только суперпользователю. Поскольку в подобных случаях программы, запускаемые рядовыми пользователями, обычно расширяют их привилегии, при их написании следует проявлять особую осторожность. Такие программы мы обсудим более подробно в главе 8.

Биты *set-user-ID* и *set-group-ID* хранятся в поле `st_mode` структуры `stat`, ассоциированной с файлом. Проверить их можно с помощью констант `S_ISUID` и `S_ISGID`.

4.5. Права доступа к файлу

Поле `st_mode` содержит также биты прав доступа к файлу. Под *файлом* мы подразумеваем файл любого типа из описанных выше. Любые файлы — каталоги, специальные файлы устройств и пр. — обладают правами доступа. Многие полагают, что понятие прав доступа присуще только обычным файлам.

Права доступа к файлу определяются девятью битами, которые подразделяются на три категории. Все они перечислены в табл. 4.5.

Таблица 4.5. Биты прав доступа из файла `<sys/stat.h>`

Маска для поля <code>st_mode</code>	Назначение
<code>S_IRUSR</code>	user-read — доступно пользователю для чтения
<code>S_IWUSR</code>	user-write — доступно пользователю для записи
<code>S_IXUSR</code>	user-execute — доступно пользователю для выполнения
<code>S_IRGRP</code>	group-read — доступно группе для чтения
<code>S_IWGRP</code>	group-write — доступно группе для записи
<code>S_IXGRP</code>	group-execute — доступно группе для выполнения
<code>S_IROTH</code>	other-read — доступно остальным для чтения
<code>S_IWOTH</code>	other-write — доступно остальным для записи
<code>S_IXOTH</code>	other-execute — доступно остальным для выполнения

Под термином *пользователь* (*user*) в табл. 4.5 подразумевается владелец файла. Команда `chmod(1)`, которая обычно используется для изменения прав доступа к файлам, позволяет указать имя категории посредством символов: *u* — *user* (пользователь, или владелец), *g* — *group* (группа) и *o* — *other* (остальные). В некоторых книгах эти три категории обозначаются как *owner* (владелец), *group* (группа) и *world* (весь остальной мир), что может привести к путанице, так как команда `chmod` использует символ *o* не в смысле *owner* (владелец), а в смысле *other* (остальные). Мы будем использовать термины *user* (пользователь), *group* (группа) и *other* (остальные), чтобы сохранить совместимость с командой `chmod`.

Три категории из табл. 4.5 — чтение, запись и выполнение — используются разными функциями по-разному. Сейчас мы коротко опишем их, а затем будем к ним возвращаться при обсуждении конкретных функций.

- Первое правило: чтобы открыть файл любого типа по его полному имени, необходимо иметь право на выполнение для всех каталогов, указанных в имени файла, включая текущий. По этой причине бит права на выполнение для каталогов часто называют битом права на поиск.

Например, чтобы открыть файл `/usr/include/stdio.h`, мы должны иметь право на выполнение для каталогов `/`, `/usr` и `/usr/include`. Далее мы должны обладать соответствующими правами на доступ к открываемому файлу в зависимости от режима — только для чтения, для чтения и записи и т. д.

Если текущим является каталог `/usr/include`, тогда, чтобы открыть файл `stdio.h`, мы должны обладать правом на выполнение для текущего каталога. В этом примере текущий каталог не указан явно, но подразумевается. С тем же успехом можно было бы обозначить имя файла как `./stdio.h`.

Обратите внимание, что право на чтение и право на выполнение для каталогов имеют иной смысл. Право на чтение дает возможность прочитать файл

каталога и получить полный список файлов, находящихся в нем. Право на выполнение дает возможность войти в каталог, когда он является одним из компонентов пути к файлу, к которому требуется получить доступ. (Чтобы найти нужный файл, необходимо выполнить поиск по каталогу).

Еще один пример неявной ссылки на каталог — переменная окружения PATH (обсуждается в разделе 8.10). Если она определяет каталог, для которого у нас нет права на выполнение, командная оболочка никогда не будет просматривать его при поиске выполняемых файлов.

- Право на чтение для файла определяет возможность открыть существующий файл для чтения (флаги O_RDONLY и O_RDWR функции open).
- Право на запись для файла определяет возможность открыть существующий файл для записи (флаги O_WRONLY и O_RDWR функции open).
- Чтобы указать флаг O_TRUNC в вызове функции open, нужно обладать правом на запись.
- Нельзя создать новый файл в каталоге при отсутствии права на запись и права на выполнение для этого каталога.
- Чтобы удалить существующий файл, необходимо обладать правом на запись и правом на выполнение для каталога, который содержит этот файл. Не нужно обладать правом на чтение или на запись для самого файла.
- Чтобы запустить файл на выполнение с помощью одной из шести функций семейства exec (раздел 8.10), нужно обладать правом на выполнение. Кроме того, файл должен быть обычным файлом.

Решение о выдаче полномочий на доступ к файлу, которое принимается ядром всякий раз, когда процесс открывает, создает или удаляет файл, зависит от принадлежности файла (`st_uid` и `st_gid`), от значений эффективных идентификаторов процесса (эффективный идентификатор пользователя и эффективный идентификатор группы) и от идентификаторов дополнительных групп процесса, если таковые поддерживаются. Оба идентификатора владельца являются свойствами самого файла, тогда как эффективные идентификаторы и идентификаторы дополнительных групп — это свойства процесса. Решение принимается ядром по следующему алгоритму.

1. Если процесс имеет эффективный идентификатор пользователя, равный 0 (суперпользователь), доступ разрешается. Это дает суперпользователю абсолютную свободу действий во всей файловой системе.
2. Если процесс имеет эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла (то есть процесс является владельцем файла), доступ разрешается, если установлен соответствующий бит права доступа для владельца. Иначе доступ к файлу запрещается. Под выражением *соответствующий бит права доступа* понимается следующее: если процесс открывает файл для чтения, должен быть установлен бит user-read, если файл открывается для записи, должен быть установлен бит user-write, если процесс собирается запустить файл на выполнение, должен быть установлен бит user-execute.

3. Если эффективный идентификатор группы или один из идентификаторов дополнительных групп процесса совпадает с идентификатором группы файла, доступ разрешается, если установлен соответствующий бит права доступа. Иначе доступ к файлу запрещается.
4. Если установлен соответствующий бит права доступа для остальных, доступ разрешается, иначе доступ запрещается.

Эти четыре шага выполняются в указанной последовательности. Обратите внимание: если процесс является владельцем файла (шаг 2), решение о предоставлении доступа или об отказе в доступе к файлу принимается только на основании прав доступа владельца, права группы уже не проверяются. Аналогично, если процесс не является владельцем файла, но принадлежит к соответствующей группе, решение принимается на основе анализа прав доступа для группы — права остальных не принимаются во внимание.

4.6. Принадлежность новых файлов и каталогов

Рассматривая в главе 3 процедуру создания новых файлов с помощью функций `open` и `creat`, мы не упоминали, какие значения принимаются в качестве идентификатора пользователя и группы для нового файла. Как создаются каталоги, мы покажем в разделе 4.21 при описании функции `mkdir`. Правила выбора владельца для нового каталога аналогичны приводимым здесь правилам выбора владельца для нового файла.

В качестве идентификатора пользователя (владельца) для нового файла принимается значение эффективного идентификатора пользователя процесса. При определении идентификатора группы для нового файла стандарт POSIX.1 допускает выбор одного из двух вариантов.

1. В качестве идентификатора группы для нового файла может быть принят эффективный идентификатор группы процесса.
2. В качестве идентификатора группы для нового файла может быть принят идентификатор группы каталога, в котором создается файл.

Операционные системы FreeBSD 8.0 и Mac OS X 10.6.8 всегда используют идентификатор группы каталога в качестве идентификатора группы для создаваемого файла. Некоторые файловые системы в Linux допускают возможность выбора любого из этих двух вариантов с помощью специального флага команды `mount(1)`. В операционных системах Linux 3.2.0 и Solaris 10 выбор идентификатора группы для нового файла зависит от значения бита `set-group-ID` у каталога, в котором создается файл. Если этот бит установлен, идентификатором группы для нового файла назначается идентификатор группы каталога, иначе — эффективный идентификатор группы процесса.

Второй вариант — наследование идентификатора группы от каталога — дает гарантию, что все файлы и подкаталоги, создаваемые в заданном каталоге, будут принадлежать той же группе, что и родительский каталог. Порядок назначения

группы владельца для файлов и каталогов будет распространяться вниз по всем вложенным каталогам. Например, именно так организована структура каталога `/var/mail` в Linux.

Как упоминалось выше, этот вариант назначения идентификатора группы принят по умолчанию в системах FreeBSD 8.0 и Mac OS X 10.6.8, но в Linux и Solaris он является одним из возможных. Чтобы описанная схема работала в Linux 3.2.0 и Solaris 10, для каталога необходимо установить бит set-group-ID, а функция `mkdir` должна устанавливать его автоматически для всех вложенных каталогов. (Это будет описано в разделе 4.21.)

4.7. Функции access и faccessat

Как говорилось выше, при открытии файла ядро выполняет серию проверок прав доступа, основываясь на эффективных идентификаторах пользователя и группы процесса. Однако в некоторых случаях необходимо проверить права доступа на основе реальных идентификаторов пользователя и группы. Это бывает удобно, когда процесс запущен с правами другого пользователя с помощью set-user-ID или set-group-ID. Даже когда установка бита set-user-ID предоставляет процессу права суперпользователя, все еще может быть необходимость проверить права реального пользователя на доступ к тому или иному файлу. Функции `access` и `faccessat` выполняют проверку прав доступа, основываясь на реальных идентификаторах пользователя и группы процесса. (Замените слово *эффективный* на слово *реальный* в алгоритме принятия решения, что приводится в конце раздела 4.5.)

```
#include <unistd.h>

int access(const char *pathname, int mode);

int faccessat(int fd, const char *pathname, int mode, int flag);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Аргумент `mode` — это набор констант из табл. 4.6, объединяемых по ИЛИ (OR).

Таблица 4.6. Константы, используемые в аргументе `mode` функции `access`

<code>mode</code>	Описание
R_OK	Проверка права на чтение
W_OK	Проверка права на запись
X_OK	Проверка права на выполнение

Функция `faccessat` действует подобно `access`, когда в аргументе `pathname` передается строка абсолютного пути или когда в аргументе `fd` передается значение AT_FDCWD и в аргументе `pathname` — строка относительного пути. В противном случае `faccessat` откладывает путь `pathname` относительно открытого каталога, определяемого дескриптором `fd`.

Аргумент *flag* можно использовать для изменения поведения `faccessat`. Если установлен флаг `AT_EACCESS`, проверка доступа выполняется не для реальных идентификаторов пользователя и группы вызывающего процесса, а для эффективных.

Пример

В листинге 4.2 приводится пример использования функции `access`.

Листинг 4.2. Пример использования функции access

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("Использование: a.out <имя_файла>");
    if (access(argv[1], R_OK) < 0)
        err_ret("ошибка вызова функции access для файла %s", argv[1]);
    else
        printf("доступ для чтения разрешен\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("ошибка вызова функции open для файла %s", argv[1]);
    else
        printf("файл благополучно открыт для чтения\n");
    exit(0);
}
```

Вот пример сеанса работы с этой программой:

```
$ ls -l a.out
-rwxrwxr-x 1 sar          15945 Nov 30 12:10 a.out
$ ./a.out a.out
доступ для чтения разрешен
Файл благополучно открыт для чтения
$ ls -l /etc/shadow
----- 1 root          1315 Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
ошибка вызова функции access для файла /etc/shadow: Permission denied
ошибка вызова функции open для файла /etc/shadow: Permission denied
$ su
получим права суперпользователя
Password:
вводим пароль суперпользователя
# chown root a.out
делаем суперпользователя владельцем файла
# chmod u+s a.out
и устанавливаем бит set-user-ID
# ls -l a.out
проверяем владельца файла и состояние бита SUID
-rwsrwxr-x 1 root          15945 Nov 30 12:10 a.out
# exit
возвращаемся к правам обычного пользователя
$ ./a.out /etc/shadow
ошибка вызова функции access для файла /etc/shadow: Permission denied
файл благополучно открыт для чтения
```

В этом примере программа с установленным битом `set-user-ID` смогла определить, что реальный пользователь не имеет права на чтение для указанного файла, хотя вызов функции `open` завершается успехом.

В предыдущем примере и в главе 8 мы иногда переходим в режим суперпользователя, чтобы продемонстрировать некоторые приемы. Если вы работаете в многопользовательской системе и не обладаете правами суперпользователя, вы не сможете полностью протестировать такие примеры.

4.8. Функция umask

Теперь, когда мы рассмотрели девять бит прав доступа, свойственных всем файлам, перейдем к маске режима создания файла, которой обладает каждый процесс. Функция `umask` устанавливает маску режима создания файлов для процесса и возвращает предыдущее значение. (Это одна из немногих функций, которые не возвращают признак ошибки.)

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Возвращает предыдущее значение маски

Аргумент `cmask` — это набор констант из табл. 4.5 (`S_IRUSR`, `S_IWUSR` и т. д.), объединяемых по ИЛИ (OR).

Маска режима создания файлов используется при создании процессом новых файлов или новых каталогов. (Загляните в разделы 3.3 и 3.4, где были описаны функции `open` и `creat`. Обе функции принимают аргумент `mode`, в котором указываются биты прав доступа к создаваемому файлу.) Процедуру создания новых каталогов мы рассмотрим в разделе 4.21. Любые биты, которые включены в маске, выключают соответствующие биты прав доступа к файлу.

Пример

Программа в листинге 4.3 создает два файла: один со значением маски, равным нулю, и второй — с маской, которая выключает все биты прав доступа для группы и остальных.

Листинг 4.3. Пример использования функции umask

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("ошибка вызова функции creat для файла foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("ошибка вызова функции creat для файла bar");
    exit(0);
}
```

Запустив эту программу, можно увидеть, как устанавливаются биты прав доступа.

```
$ umask                               сначала выведем текущее значение маски
002
$ ./a.out
$ ls -l foo bar
-rw----- 1                           sar 0 Dec 7 21:20 bar
-rw-rw-rw- 1                           sar 0 Dec 7 21:20 foo
$ umask                               проверим, изменилось ли значение маски
002
```

Большинство пользователей UNIX никогда не имеют дела с этой маской. Она обычно устанавливается командной оболочкой единожды, в момент входа в систему, и никогда не изменяется. Тем не менее при разработке программ, создающих новые файлы, необходимо модифицировать значение маски на время работы процесса, чтобы обеспечить установку конкретных битов прав доступа. Например, чтобы дать любому пользователю право на чтение создаваемого файла, нужно установить значение маски в 0. Иначе, вследствие применения действующей маски, может получиться, что необходимые биты прав доступа окажутся сброшенными.

В предыдущем примере мы использовали команду `umask` для вывода значения маски режимов создания файлов до и после запуска программы. Тем самым мы показали, что изменение маски в процессе не влияет на маску родительского процесса (которым часто является командная оболочка). Все командные оболочки имеют встроенную команду `umask`, которая используется для вывода и изменения значения маски режима создания новых файлов.

Пользователи могут установить значение `umask` для управления правами доступа к создаваемым файлам по умолчанию. Значение маски задается в восьмеричной системе счисления, где каждый бит маски соответствует биту прав доступа, который он отключает, как показано в табл. 4.7. Права доступа отключаются установкой соответствующих битов. Наиболее распространенные значения маски — 002 (запрещает запись в файл всем пользователям, кроме владельца), 022 (запрещает запись в файл членам группы и остальным пользователям) и 027 (запрещает членам группы запись в файл, а всем остальным — чтение, запись и исполнение).

Таблица 4.7. Биты прав доступа для маски

Бит маски	Значение
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Стандарт Single UNIX Specification требует, чтобы командная оболочка поддерживала возможность определения маски в символьической форме. В отличие от восьмеричного формата, символьский формат определяет набор прав, которые разрешаются (то есть сброшены в маске), а не тех, которые запрещаются (то есть установлены в маске). Сравните два варианта вызова команды `umask`:

```
$ umask                                выведем текущее значение маски режима создания
002                                     новых файлов
$ umask -S                               выведем значение маски в символьическом представлении
u=rwx,g=rwx,o=rx
$ umask 027                             изменим значение маски режима создания файлов
$ umask -S                               выведем значение маски в символьическом представлении
u=rwx,g=rx,o=
```

4.9. Функции chmod, fchmod и fchmodat

Функции `chmod`, `fchmod` и `fchmodat` позволяют изменять права доступа к существующим файлам.

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

Все три возвращают 0 в случае успеха, -1 — в случае ошибки

Функция `chmod` работает с файлом, заданным его именем, а функция `fchmod` — с уже открытым файлом, заданным дескриптором. Функция `fchmodat` действует подобно `chmod`, когда в аргументе `pathname` передается строка абсолютного пути или когда в аргументе `fd` передается значение `AT_FDCWD` и в аргументе `pathname` — строка относительного пути. Иначе `fchmodat` откладывает путь `pathname` относительно открытого каталога, определяемого дескриптором `fd`. Аргумент `flag` можно использовать для изменения поведения `fchmodat`: когда установлен флаг `AT_SYMLINK_NOFOLLOW`, функция `fchmodat` не выполняет переходы по символьическим ссылкам.

Чтобы можно было изменить права доступа к файлу, эффективный идентификатор процесса должен совпадать с идентификатором владельца файла либо процесс должен обладать привилегиями суперпользователя.

Аргумент `mode` — это набор констант из табл. 4.8, объединяемых по ИЛИ (OR).

Обратите внимание, что имена девяти констант из табл. 4.8 совпадают с именами констант из табл. 4.5. Здесь добавились две константы set-ID (`S_ISUID` и `S_ISGID`), константа saved-text (`S_ISVTX`) и три комбинированные константы (`S_IRWXU`, `S_IRWXG` и `S_IRWXO`).

Таблица 4.8. Константы режимов для функции chmod, определенные в файле <sys/stat.h>

<i>mode</i>	Описание
S_ISUID	set-user-ID при запуске на выполнение
S_ISGID	set-group-ID при запуске на выполнение
S_ISVTX	saved-text (бит sticky)
S_IRWXU	Право на чтение, запись и выполнение для пользователя (владельца)
S_IRUSR	Право на чтение для пользователя (владельца)
S_IWUSR	Право на запись для пользователя (владельца)
S_IXUSR	Право на выполнение для пользователя (владельца)
S_IRWXG	Право на чтение, запись и выполнение для группы
S_IRGRP	Право на чтение для группы
S_IWGRP	Право на запись для группы
S_IXGRP	Право на выполнение для группы
S_IRWXO	Право на чтение, запись и выполнение для остальных
S_IROTH	Право на чтение для остальных
S_IWOTH	Право на запись для остальных
S_IXOTH	Право на выполнение для остальных

Бит *saved-text* (*S_ISVTX*) не является частью стандарта POSIX.1. Он определен как расширение XSI в стандарте Single UNIX Specification. Его назначение будет описано в следующем разделе.

Пример

Вспомните состояние файлов *foo* и *bar*, созданных программой из листинга 4.3, которая демонстрирует работу функции *umask*:

```
$ ls -l foo bar
-rw----- 1 sar          0 Dec 7 21:20 bar
-rw-rw-rw- 1 sar          0 Dec 7 21:20 foo
```

Программа в листинге 4.4 изменяет режимы доступа к этим файлам.

Листинг 4.4. Пример использования функции chmod

```
#include "apue.h"

int
main(void)
{
    struct stat      statbuf;
    /* включить бит set-group-ID и выключить group-execute */
    if (stat("foo", &statbuf) < 0)
```

```

err_sys("ошибка вызова функции stat для файла foo");
if (chmod("foo", (statbuf.st_mode & .S_IXGRP) | S_ISIGID) < 0)
    err_sys("ошибка вызова функции chmod для файла foo");

/* установить режим в значение "rw-r--r--" */

if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
    err_sys("ошибка вызова функции chmod для файла bar");

exit(0);
}

```

После запуска программы из листинга 4.4 мы увидим, что режимы доступа к файлам изменились:

```
$ ls -l foo bar
-rw-r--r-- 1                      sar 0 Dec 7 21:20 bar
-rw-rwSr-- 1                      sar 0 Dec 7 21:20 foo
```

В этом примере для файла `bar` мы установили абсолютные значения прав доступа, не обращая внимания на текущие. Для файла `foo`, наоборот, мы установили права доступа относительно их текущего состояния. Для этого мы сначала получили набор прав доступа с помощью функции `stat` и затем изменили их. Мы явно включили бит set-group-ID и выключили бит group-execute. Обратите внимание, что команда `ls` вывела значение бита group-execute в виде символа `S`, подчеркивая тем самым, что бит set-group-ID установлен при сброшенном бите group-execute.

В OC Solaris вместо символа `S` команда `ls` выводит символ `L`, чтобы подчеркнуть, что для файла включен режим обязательных блокировок файла и отдельных записей. Это справедливо только для обычных файлов, и мы еще обсудим эту тему в разделе 14.3.

И наконец, отметьте, что дата и время, отображаемые командой `ls`, не изменились после запуска программы из листинга 4.4. Позже, в разделе 4.19, мы увидим, что функция `chmod` обновляет только время последнего изменения индексного узла (i-node). Команда `ls -l` по умолчанию выводит время последнего изменения содержимого файла.

Функция `chmod` автоматически сбрасывает два бита прав доступа при следующих условиях:

- В некоторых системах, таких как Solaris, бит sticky имеет особое значение для обычных файлов. Если попытаться установить бит sticky (`S_ISVTX`) для обычного файла, не обладая привилегиями суперпользователя, этот бит в аргументе `mode` будет автоматически сброшен. (Бит sticky рассматривается в следующем разделе.) Отсюда следует, что его может установить только суперпользователь. Сделано это, чтобы предотвратить установку бита `S_ISVTX` злоумышленником и тем самым избежать нанесения ущерба работоспособности системы в целом.

В системах FreeBSD 8.0 и Solaris 10 только суперпользователь может устанавливать бит sticky на обычные файлы. В Linux 3.2.0 и Mac OS X 10.6.8 это ограничение отсутствует, поскольку для обычных файлов в этих системах данный бит не имеет никакого значения. Несмотря на то что в FreeBSD этот бит также не имеет значения для обычных файлов, его установка для обычных файлов разрешена только суперпользователю.

- При создании файла ему можно назначить идентификатор группы, отличный от идентификатора группы процесса, создающего файл. В разделе 4.6 говорилось, что возможна ситуация, когда файл наследует идентификатор группы от каталога, в котором он размещается. Поэтому если идентификатор группы создаваемого файла не является эффективным идентификатором группы процесса или одним из идентификаторов дополнительных групп и процесс не обладает привилегиями суперпользователя, бит set-group-ID автоматически сбрасывается. Это предотвращает возможность создания файла с идентификатором группы, с которой пользователь никак не связан.

В OC FreeBSD 8.0 попытка установить бит set-group-ID в такой ситуации терпит неудачу. Другие системы просто сбрасывают бит, не сообщая об ошибке при попытке изменить права доступа.

Системы FreeBSD 8.0, Mac OS X 10.6.8, Linux 3.2.0 и Solaris 10 имеют еще одну особенность, предотвращающую злонамеренное использование некоторых битов. Если процесс, не обладающий привилегиями суперпользователя, производит запись в файл, биты set-user-ID и set-group-ID автоматически сбрасываются. Даже если злоумышленнику удастся отыскать доступные на запись файлы с установленными битами set-user-ID или set-group-ID, в момент модификации эти файлы утратят особые привилегии.

4.10. Бит sticky

Интересна история появления бита `S_ISVTH`. В версиях UNIX, поддерживавших предварительную подкачку страниц, этот бит был известен под названием *sticky bit* (липкий бит). Если этот бит устанавливался на выполняемый файл, при первом запуске программы ее сегмент кода записывался в файл подкачки и сохранялся там после ее завершения. (Сегмент кода программы состоит из машинных инструкций.) Это приводило к тому, что при следующем вызове программы она запускалась намного быстрее, поскольку файл подкачки представлял собой непрерывную область на диске в отличие от обычных файлов, которые могут размещаться в разрозненных дисковых блоках. Бит sticky обычно устанавливался на наиболее часто используемые программы, такие как текстовые редакторы или компилятор языка С. Естественно, существовало ограничение на количество таких «липких» файлов, которые могли одновременно разместиться в файле подкачки, тем не менее этот прием был очень удобен. Название *sticky* (липкий) объясняется тем, что сегмент кода программы как бы вклеивался в пространство файла подкачки и оставался там до перезагрузки системы. В более поздних версиях UNIX этот бит стал называться *saved-text bit* (закрепляемый сегмент кода), отсюда и название константы — `S_ISVTH`. Большинство современных версий UNIX обладают системой виртуальной памяти и более быстродействующими файловыми системами, поэтому надобность в подобном «закреплении» отпала.

В современных системах назначение бита sticky было расширено. Стандарт Single UNIX Specification допускает установку этого бита на каталоги. Если бит sticky установлен на каталог, удалять или изменять файлы в таком каталоге сможет только пользователь, обладающий правом на запись в каталог и являющийся:

- владельцем файла;
- или владельцем каталога;
- или суперпользователем.

Типичные примеры каталогов, на которые, как правило, устанавливается бит sticky, — `/tmp` и `/var/tmp`. Обычно любой пользователь может создавать файлы в этих каталогах. Часто эти каталоги доступны для записи, чтения и выполнения всем пользователям, но удалять или изменять файлы могут только их владельцы.

Бит saved-text не является частью стандарта POSIX.1. Он входит в состав определяемых стандартом Single UNIX Specification расширений XSI и поддерживается операционными системами FreeBSD 8.0, Linus 3.2.0, Mac OS X 10.6.8 и Solaris 10.

В Solaris 10 бит sticky для обычных файлов имеет специальное значение. Если для файла установлен бит sticky и сброшены биты прав на выполнение, операционная система не будет кэшировать содержимое этого файла.

4.11. Функции chown, fchown, fchownat и lchown

Функции семейства `chown` позволяют изменять идентификаторы пользователя и группы файла, но если в аргументе `owner` или `group` передается `-1`, соответствующий идентификатор не изменяется.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t group, int flag);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Все четыре возвращают `0` в случае успеха и `-1` — в случае ошибки

Эти четыре функции практически идентичны, за исключением случая, когда они применяются к символьским ссылкам. Функции `lchown` и `fchownat` (с установленным флагом `AT_SYMLINK_NOFOLLOW`) изменяют владельца самой символьской ссылки, а не файла, на который она указывает.

Функция `fchown` изменяет владельца открытого файла, на который ссылается дескриптор `fd`. Поскольку операция выполняется над уже открытым файлом, `fchown` не может использоваться для изменения владельца символьской ссылки.

Функция `fchownat` может действовать подобно `chown` или `lchown`, когда в аргументе `pathname` передается строка абсолютного пути или когда в аргументе `fd` передается значение `AT_FDCWD` и в аргументе `pathname` — строка относительного пути. В этих случаях `fchownat` действует подобно `lchown`, если в аргументе `flag`

установлен флаг `AT_SYMLINK_NOFOLLOW`, а если флаг `AT_SYMLINK_NOFOLLOW` в аргументе `flag` сброшен, она действует подобно `chown`. Если в аргументе `fd` передается файловый дескриптор открытого каталога и в аргументе `pathname` — строка относительного пути, `fchownat` откладывает путь `pathname` относительно открытого каталога.

Исторически в BSD-системах существует ограничение на эту операцию — только суперпользователь может изменить владельца файла. Это предотвращает несанкционированную передачу права на владение файлами другим пользователям и тем самым возможность превышения установленных дисковых квот. Однако в System V всем пользователям позволено изменять владельцев для любых файлов, которыми они владеют.

Стандарт POSIX.1 допускает любой из двух режимов в зависимости от значения константы `_POSIX_CHOWN_RESTRICTED`.

В Solaris 10 режим работы зависит от значения параметра конфигурации, значение по умолчанию — ограниченный режим изменения владельца файла. В системах FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 ограниченный режим изменения владельца файла действует всегда.

В разделе 2.6 упоминалось, что константа `_POSIX_CHOWN_RESTRICTED` не всегда определяется в заголовочном файле `<unistd.h>`, но ее значение всегда можно получить с помощью функции `pathconf` или `fpathconf`. Значение этого параметра может зависеть от конкретного файла; кроме того, это ограничение может поддерживаться или не поддерживаться самой файловой системой. Мы будем употреблять выражение «если действует ограничение `_POSIX_CHOWN_RESTRICTED`» в отношении конкретных файлов, о которых идет речь, вне зависимости от наличия определения константы в заголовочном файле `<unistd.h>`.

Если ограничение `_POSIX_CHOWN_RESTRICTED` действует, тогда:

1. Только процесс, обладающий правами суперпользователя, сможет изменить идентификатор пользователя файла.
2. Процесс, не обладающий правами суперпользователя, сможет изменить идентификатор группы файла, если является владельцем этого файла (эффективный идентификатор пользователя процесса совпадает с идентификатором пользователя файла), аргумент `owner` имеет значение `-1` или совпадает с идентификатором пользователя файла и аргумент `group` совпадает с эффективным идентификатором группы или с одним из идентификаторов дополнительных групп процесса.

Это означает, что если ограничение `_POSIX_CHOWN_RESTRICTED` действует, вы не сможете изменить идентификатор пользователя (владельца) файла. Изменить идентификатор группы файла может только владелец этого файла и только при условии, что присваивает идентификатор одной из групп, к которой принадлежит сам.

Если эти функции вызываются из процесса, не обладающего привилегиями суперпользователя, в случае успешного завершения они сбрасывают биты `set-user-ID` и `set-group-ID`.

4.12. Размер файла

Поле `st_size` структуры `stat` содержит размер файла в байтах. Это поле имеет смысл только для обычных файлов, каталогов и символьических ссылок.

FreeBSD 8.0, Mac OS X 10.6.8 и Solaris 10 поддерживают размер файла также для каналов; он обозначает доступное для чтения количество байтов в канале. О каналах речь пойдет в разделе 15.2.

Обычные файлы могут иметь размер, равный нулю. В этом случае первая же операция чтения вернет признак конца файла. Для каталогов размер файла обычно кратен некоторому числу, такому как 16 или 512. О чтении каталогов мы поговорим в разделе 4.22.

Размер файла для символьических ссылок обозначает длину имени файла в байтах. Например, в следующем случае число 7 обозначает длину пути к каталогу `usr/lib`:

```
1rwxrwxrwx 1 root          7 Sep 25 07:14 lib -> usr/lib
```

(Обратите внимание, что символьические ссылки не имеют завершающего нулевого символа в конце имени, типичного для строк в языке C, поэтому поле `st_size` всегда определяет длину строки имени файла.)

В большинстве современных версий UNIX есть поля `st_blksize` и `st_blocks`. Первое определяет оптимальный размер блока для операций ввода/вывода, а второй — фактическое количество 512-байтных блоков, занимаемых файлом. В разделе 3.9 мы определили, что наименьшее время на операции чтения затрачивается, если используется буфер с размером `st_blksize`. Стандартная библиотека ввода/вывода, которую мы рассмотрим в главе 5, также старается производить операции ввода/вывода блоками по `st_blksize` байт, что повышает производительность.

Существуют такие версии UNIX, которые измеряют величину `st_blocks` не в 512-байтных блоках. Использование этого значения снижает переносимость программ.

Дырки в файлах

В разделе 3.6 говорилось, что обычные файлы могут содержать «дырки». Мы продемонстрировали это на примере программы из листинга 3.2. Дырки создаются в результате переноса текущей позиции за пределы файла и последующей записи некоторых данных. Рассмотрим следующий пример:

```
$ ls -l core
-rw-r--r-- 1 sar      8483248 Nov 18 12:18 core
$ du -s core
272      core
```

Размер файла `core` превышает 8 Мбайт, хотя команда `du` сообщает, что он занимает всего 272 блока по 512 байт (139 264 байт). Очевидно, что этот файл имеет дырки.

Команда `du` в большинстве систем, происходящих от BSD, выводит количество 1024-байтных блоков, тогда как в Solaris — количество 512-байтных блоков. В Linux единица измере-

ния зависит от переменной окружения `POSIXLY_CORRECT`. Если она установлена, команда `du` выводит размеры в блоках по 1024 байт, иначе – в блоках по 512 байт.

Как упоминалось в разделе 3.6, функция `read` возвращает 0 для байтов, которые фактически не были записаны. Запустив следующую команду, мы увидим, что обычные операции ввода/вывода считывают полное количество байтов, соответствующее размеру файла:

```
$ wc -c core
8483248 core
```

Команда `wc(1)` с ключом `-c` подсчитывает количество символов (байтов) в файле.

Если скопировать этот файл, например, с помощью утилиты `cat(1)`, дырки будут скопированы как обычные байты данных со значением 0:

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r-- 1      8483248 Nov 18 12:18 core
-rw-rw-r-- 1 sar  8483248 Nov 18 12:27 core.copy
$ du -s core*
272      core
16592   core.copy
```

Здесь фактический размер нового файла составил 8 495 104 байт (то есть $512 \times 16\,592$). Различие между этим числом и размером, выведенным командой `ls`, обусловлено использованием файловой системой некоторого количества блоков для хранения указателей на блоки с фактическими данными.

Кому интересны вопросы, связанные с физическим размещением файлов, могут обратиться к разделу 4.2 [Bach, 1986], разделам 7.2 и 7.3 [McKusick, 1996] (или к разделам 8.2 и 8.3 в [McKusick and Neville-Neil, 2005]), к разделу 15.2 [McDougall and Mauro, 2007] и к главе 12 [Singh, 2006].

4.13. Усечение файлов

Иногда возникает необходимость отсечь некоторые данные, расположенные в конце файла. Усечение размера файла до нуля, которое осуществляется при использовании флага `O_TRUNC` в вызове функции `open`, есть частный случай усечения файла.

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

Обе возвращают 0 в случае успеха, -1 – в случае ошибки

Эти две функции усекают существующий файл до размера, определяемого аргументом `length`. Если первоначальный размер файла превышал значение `length`,

данные за этим пределом будут недоступны. Если первоначальный размер файла меньше значения *length*, размер файла будет увеличен, а данные, расположенные между старым и новым концом файла, будут читаться как нули (то есть в файле, вероятно, будет создана дырка).

В версиях BSD, предшествовавших 4.4BSD, функция `truncate` может только уменьшать размер файла.

Solaris включает расширение `fcntl (F_FREESP)`, позволяющее вырезать любую часть файла, а не только ту, что находится в конце.

Мы будем использовать функцию `ftruncate` в программе, представленной в листинге 13.2, где требуется очистить содержимое файла после получения блокировки.

4.14. Файловые системы

Чтобы понять идею ссылок на файлы, сначала нужно в общих чертах разобраться в устройстве файловой системы UNIX. Также пригодится понимание разницы между индексным узлом (i-node) и записью в файле каталога, которая указывает на индексный узел.

В настоящее время используются самые разные реализации файловых систем UNIX. Например, Solaris поддерживает несколько типов дисковых файловых систем: традиционную для BSD-систем UNIX File System (UFS), DOS-совместимую файловую систему PCFS и файловую систему HSFS, предназначенную для компакт-дисков. Мы уже видели одно из различий между разными типами файловых систем в табл. 2.16. UFS основана на системе Berkeley fast file system, которая рассматривается в этом разделе.

Каждая файловая система имеет свои специфические особенности, и некоторые из этих особенностей могут сбить с толку. Например, большинство файловых систем для UNIX поддерживают имена файлов, чувствительные к регистру символов. То есть если попытаться создать один файл с именем `file.txt` и другой файл с именем `file.TXT`, будет создано два разных файла. Однако в Mac OS X поддерживается файловая система HFS, сохраняющая регистр символов в именах файлов, но выполняющая сравнение без учета регистра символов. То есть если попытаться создать файл с именем `file.txt` и затем `file.TXT`, файл `file.txt` будет затерп. Однако в файловой системе будет сохранено имя, использованное при создании файла (с сохранением регистра символов). Фактически любые комбинации букв верхнего и нижнего регистров в последовательности `f, i, l, e, ., t, x, t` будут соответствовать данному имени файла (сравнение выполняется без учета регистра символов). Как следствие, помимо имен `file.txt` и `file.TXT` для доступа к файлу можно также использовать имена `File.txt`, `FILE.tXt` и `FiLe.TxT`.

Представим диск, поделенный на несколько разделов. Каждый раздел может содержать файловую систему, как показано на рис. 4.1. Индексные узлы — это записи фиксированной длины, которые содержат большую часть сведений о файлах.

Присмотревшись к той части группы цилиндров, где находятся индексные узлы и блоки данных, мы увидим картину, изображенную на рис. 4.2.

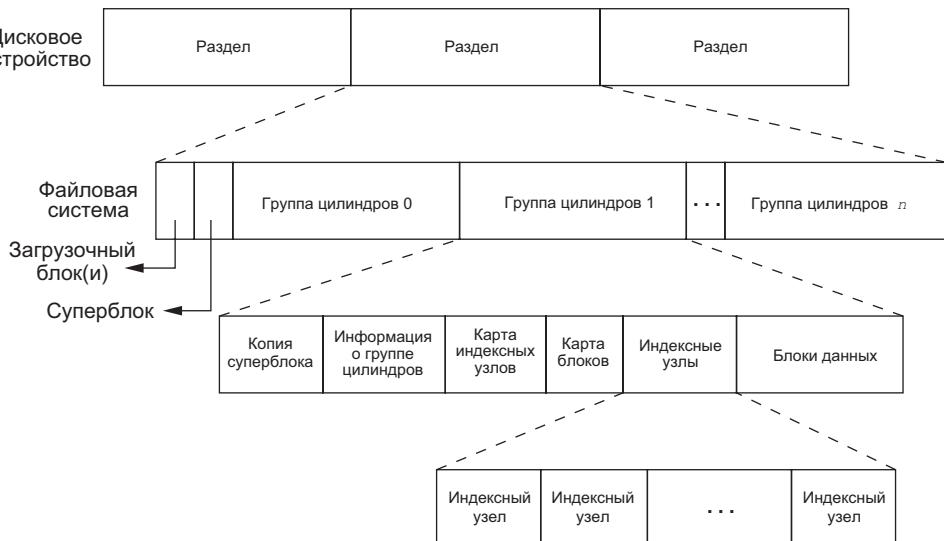


Рис. 4.1. Дисковое устройство, разделы и файловая система

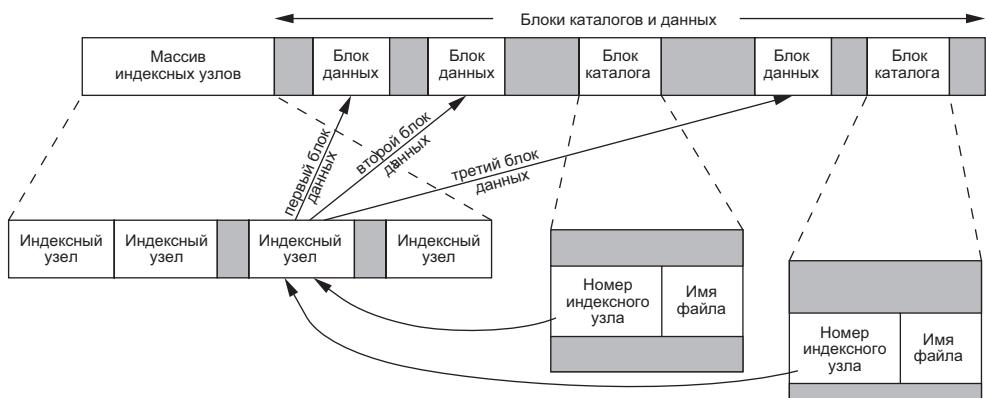


Рис. 4.2. Часть группы цилиндров с индексными узлами и блоками данных более детально

Отметим на рис. 4.2 следующие моменты:

- Здесь мы видим две записи в файле каталога, которые ссылаются на один и тот же индексный узел. Каждый индексный узел имеет счетчик ссылок; в нем хранится число записей в файле каталога, которые ссылаются на данный индексный узел. Только когда этот счетчик достигнет значения 0, файл будет удален (то есть блоки данных, связанные с файлом, перейдут в список свободных блоков). Поэтому операция «отсоединения файла» (unlink) не всегда приводит к «удалению блоков, связанных с файлом» (delete). Именно по этой при-

чине функция удаления записей из каталога носит имя `unlink` (отцепить), а не `delete` (удалить). В структуре `stat` счетчик ссылок находится в поле `st_nlink`. Оно имеет элементарный системный тип `nlink_t`. Этот тип ссылок называют *жесткими ссылками*. В разделе 2.5.2 мы говорили о константе `LINK_MAX`, которая задает максимальное значение для счетчика ссылок.

- Другую разновидность ссылок называют *символическими ссылками*. В этом случае в блоках данных файла-ссылки хранится имя файла, на который указывает эта символьическая ссылка. В следующем примере имя файла в каталожной записи представлено трехсимвольной строкой `lib`, сам же файл содержит 7 символов — `usr/lib`.

```
lrwxrwxrwx 1 root      7 Sep 25 07:14 lib -> usr/lib
```

Тип файла в индексном узле должен быть определен как `S_IFLNK`, чтобы файловая система корректно распознавала символические ссылки.

- Индексный узел содержит полную информацию о файле: тип файла, биты прав доступа, размер файла, указатели на блоки данных файла и т. п. Большая часть информации для структуры `stat` берется из индексного узла. Только два элемента, которые могут представлять для нас интерес, берутся из записи в файле каталога: имя файла и номер индексного узла. Другие элементы, такие как длина имени файла и размер записи в файле каталога, пока для нас особого интереса не представляют. Для хранения номера индексного узла используется тип данных `ino_t`.
- Поскольку номер индексного узла в каталожной записи ссылается на индексный узел, находящийся в той же файловой системе, нельзя создать запись, которая указывала бы на индексный узел в другой файловой системе. По этой причине команда `ln(1)` (создающая новую запись в каталоге, которая указывает на индексный узел существующего файла) не может создавать ссылки на файлы, расположенные в других файловых системах. Функция `link` рассматривается в следующем разделе.
- При переименовании/перемещении файла в пределах одной файловой системы фактическое содержимое файла никуда не перемещается. Все, что нужно сделать, — это добавить в каталог новую запись, которая будет указывать на существующий индексный узел, а затем отцепить старую запись. При этом значение счетчика ссылок не изменится. Например, чтобы переименовать файл `/usr/lib/foo` в `/usr/foo`, нет необходимости перемещать содержимое файла `foo`, если каталоги `/usr/lib` и `/usr` находятся в одной файловой системе. Обычно именно так работает команда `mv(1)`.

Мы только что обсудили смысл счетчика ссылок для обычных файлов, а что означает понятие счетчика ссылок для каталога? Предположим, что мы создаем новый каталог в текущем каталоге:

```
$ mkdir testdir
```

На рис. 4.3 показан результат выполнения этой команды. Обратите внимание: мы явно показали наличие записей о каталогах «точка» (текущий каталог) и «точка-точка» (родительский каталог).

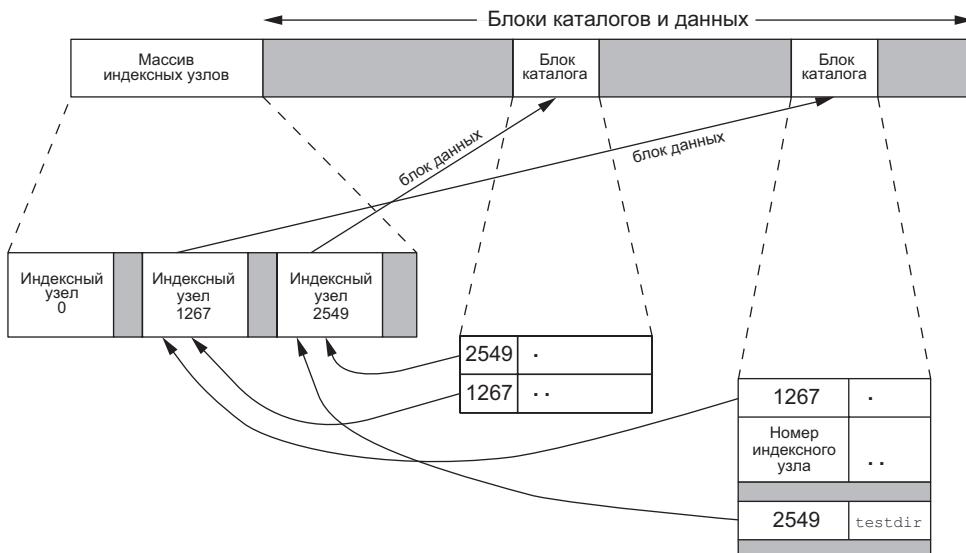


Рис. 4.3. Группа цилиндров после создания каталога testdir

Индексный узел с номером 2549 в поле «тип» хранит значение «каталог», а в счетчике ссылок — значение 2. Любой оконечный каталог (который не содержит других каталогов) в счетчике ссылок всегда хранит число 2, потому что на индексный узел ссылаются две каталожные записи: запись, указывающая на каталог `testdir`, и запись в этом же каталоге, указывающая на каталог «точка». Индексный узел с номером 1267 в поле «тип» хранит значение «каталог», а в счетчике ссылок — значение 3 или выше. Причина, по которой значение счетчика ссылок больше или равно 3, теперь должна быть нам понятна. Число 3 — это минимальное значение, учитывающее записи в каталоге верхнего уровня (который на рисунке не показан), в самом каталоге («точка») и в каталоге `testdir` («точка-точка»). Обратите внимание, что появление каждого нового подкаталога увеличивает счетчик ссылок в родительском каталоге на единицу. Этот формат похож на классический формат файловой системы UNIX, описанный в главе 4 [Bach, 1986]. За дополнительной информацией по изменениям, которые появились в Berkeley fast file system, обращайтесь к главе 7 [McKusick, 1996] или к главе 8 [McKusick and Neville-Neil, 2005]. Подробную информацию о файловой системе UFS версии Berkeley fast file system для Solaris вы найдете в главе 15 [Mauro and McDougall, 2007]. Информацию о файловой системе HFS, используемой в Mac OS X, см. в главе 12 [Singh, 2006].

4.15. Функции `link`, `linkat`, `unlink`, `unlinkat` и `remove`

Как мы уже говорили в предыдущем разделе, на индексный узел любого файла могут указывать несколько каталожных записей. Такие ссылки создаются с помощью функции `link` или `linkat`.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);

int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Эти функции создают в каталоге новую запись с именем *newpath*, которая будет указывать на существующий файл *existingpath*. Если запись с именем *newpath* уже существует, функция вернет признак ошибки. Создается только последний компонент полного пути *newpath*, все промежуточные компоненты должны существовать к моменту вызова функции.

В функции `linkat` существующий файл определяется обоими аргументами, *efd* и *existingpath*, а новая запись определяется аргументами *nfd* и *newpath*. По умолчанию, если какая-нибудь из строк путей определяет относительный путь, этот путь откладывается относительно каталога, представленного соответствующим файловым дескриптором. Если в каком-либо аргументе файлового дескриптора передается значение `AT_FDCWD`, соответствующий путь, если он относительный, откладывается от текущего каталога. Если в какой-либо из строк пути передается абсолютный путь, соответствующий аргумент с файловым дескриптором игнорируется.

Если существующий файл является символической ссылкой, с помощью аргумента *flag* можно указать, создавать ли ссылку на символическую ссылку или на файл, на который указывает эта символическая ссылка. Если в аргументе *flag* установить флаг `AT_SYMLINK_FOLLOW`, будет создана ссылка на файл, на который указывает символическая ссылка. Если передать пустой аргумент *flag*, будет создана ссылка на саму символическую ссылку.

Операции создания новой записи в каталоге и увеличения счетчика ссылок должны выполняться атомарно. (Вспомните обсуждение атомарных операций в разделе 3.11.)

Большинство реализаций требуют, чтобы оба пути находились в пределах одной файловой системы, хотя стандарт POSIX.1 допускает возможность создания ссылок на файлы, расположенные в других файловых системах. Если поддерживается создание жестких ссылок на каталоги, то эта операция может выполняться только суперпользователем. Причина такого ограничения в том, что создание жесткой ссылки на каталог может привести к появлению замкнутых «петель» в файловой системе и большинство обслуживающих ее утилит не смогут обработать их надлежащим образом. (В разделе 4.17 мы покажем пример замкнутой петли, образованной с помощью символической ссылки.) По этой же причине многие реализации файловых систем вообще не допускают создания жестких ссылок на каталоги.

Удаление записей из каталога производится с помощью функции `unlink` или `unlinkat`.

```
#include <unistd.h>
int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname, int flag);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Эти функции удаляют запись из файла каталога и уменьшают значение счетчика ссылок на файл *pathname*. Если на файл указывает несколько ссылок, его содержимое будет по-прежнему доступно через них. В случае ошибки файл не изменяется.

Как мы уже говорили, чтобы удалить жесткую ссылку на файл, необходимо обладать правом на запись и на выполнение для каталога, в котором находится удаляемая запись. Кроме того, в разделе 4.10 говорилось, что если для каталога установлен бит sticky, мы должны обладать правом на запись в каталог и являться либо владельцем файла, либо владельцем каталога, либо суперпользователем.

Содержимое файла может быть удалено, только если счетчик ссылок достиг 0. Кроме того, содержимое файла нельзя удалить, если он открыт каким-либо процессом. Во время закрытия файла ядро в первую очередь проверяет счетчик процессов, открывших его. Если значение счетчика достигло нуля, ядро проверяет счетчик ссылок, и только если его значение достигло нуля, содержимое файла будет удалено.

Если в аргументе *pathname* передается строка относительного пути, функция *unlinkat* откладывает путь относительно открытого каталога, представленного файловым дескриптором *fd*. Если в аргументе *fd* передается значение *AT_FDCWD*, относительный путь *pathname* откладывается относительно текущего рабочего каталога вызывающего процесса. Если в аргументе *pathname* передается строка абсолютного пути, аргумент *fd* игнорируется.

Аргумент *flag* позволяет вызывающей программе изменять поведение по умолчанию функции *unlinkat*. Если установлен флаг *AT_REMOVEDIR*, функция *unlinkat* может использоваться для удаления каталога подобно функции *rmdir*. Если передать пустой аргумент *flag*, *unlinkat* действует подобно функции *unlink*.

Пример

Программа в листинге 4.5 открывает файл и отцепляет его (то есть удаляет с помощью функции *unlink*). Затем программа приостанавливается на 15 секунд и завершает работу.

Листинг 4.5. Открывает файл, а затем удаляет его

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
```

```

    err_sys("ошибка вызова функции open");
if (unlink("tempfile") < 0)
    err_sys("ошибка вызова функции unlink");
printf("файл удален\n");
sleep(15);
printf("конец\n");
exit(0);
}
}

```

Запуск программы дает следующие результаты:

```

$ ls -l tempfile          посмотрим размер файла
-rw-r----- 1 sar 413265408 Jan 21 07:14 tempfile
$ df /home               проверим объем доступного пространства
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/hda4 11021440 1956332 9065108 18% /home
$ ./a.out &             запустим программу из листинга 4.5 как фоновый процесс
1364                   командная оболочка вывела идентификатор процесса
$ файл удален           файл отцеплен
ls -l tempfile           проверим, остался ли файл на месте
ls: tempfile: No such file or directory  запись из каталога была удалена
$ df /home               проверим, освободилось ли дисковое пространство
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/hda4 11021440 1956332 9065108 18% /home
$ конец                 программа завершилась, все файлы закрыты
df /home                теперь должно освободиться пространство на диске
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/hda4 11021440 1552352 9469088 15% /home
на диске освободилось 394,1 Мбайт

```

Эта характерная особенность функции `unlink` часто используется программами, чтобы обеспечить удаление временных файлов в случае аварийного завершения. Процесс создает файл вызовом `open` или `creat` и сразу же вызывает функцию `unlink`. Однако файл не будет удален, поскольку остается открытым. Только когда процесс закроет файл или завершит работу, что, в свою очередь, заставит ядро закрыть все файлы, открытые процессом, файл удалится.

Если аргумент `pathname` является символьской ссылкой, удалится сама символьская ссылка, а не файл, на который она ссылается, — не существует функции, которая удаляла бы файл по символьской ссылке.

Процесс, обладающий привилегиями суперпользователя, может вызвать функцию `unlink` для удаления каталога, если такая возможность поддерживается системой, но вообще в таких случаях следует использовать функцию `rmdir`, которую мы рассмотрим в разделе 4.21.

Удалить жесткую ссылку на файл или каталог можно также с помощью функции `remove`. Для файлов функция `remove` абсолютно идентична функции `unlink`, для каталогов — функции `rmdir`.

```

#include <stdio.h>

int remove(const char *pathname);

```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Стандарт ISO C определяет *remove* как функцию для удаления файлов. Исторически сложившееся в UNIX название *unLink* было заменено, потому что в то время в большинстве других операционных систем, которые следовали стандарту ISO C, понятие ссылок на файлы не поддерживалось.

4.16. Функции *rename* и *renameat*

Для переименования файла или каталога используется функция *rename* или *renameat*.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd, const char *newname);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Стандарт ISO C определяет *rename* как функцию для переименования файлов. (ISO C вообще не касается каталогов.) Стандарт POSIX.1 расширил это определение, включив в него каталоги и символьские ссылки.

Следует отдельно рассмотреть случаи, когда аргумент *oldname* представляет файл, символьскую ссылку или каталог. Также необходимо упомянуть о том, что произойдет, если *newname* уже существует.

- Если аргумент *oldname* указывает на файл, который не является каталогом, происходит переименование файла или символьской ссылки. Если файл *newname* уже существует, он не должен быть каталогом. Если файл *newname* существует и не является каталогом, он будет удален, а файл *oldname* будет переименован в *newname*. Процесс должен обладать правом записи в каталоги, где находятся файлы *newname* и *oldname*, поскольку предполагается внесение изменений в оба каталога.
- Если аргумент *oldname* указывает на каталог, выполняется переименование каталога. Если *newname* существует, он также должен быть каталогом, и этот каталог должен быть пустым. (Под «пустым каталогом» имеется в виду каталог, содержащий только две записи: «точка» и «точка-точка».) Если *newname* существует и является пустым каталогом, он будет удален, а каталог *oldname* будет переименован в *newname*. Кроме того, при переименовании каталога аргумент *newname* не должен начинаться с имени каталога *oldname*. Так, например, нельзя переименовать каталог /usr/foo в /usr/foo/testdir, поскольку прежнее имя каталога (/usr/foo) содержится в начале нового имени и он не может быть удален.
- Если аргументы *oldname* или *newname* содержат имя символьской ссылки, будет переименована сама символьская ссылка, а не файл, на который она ссылается.

4. Каталоги «точка» и «точка-точка» нельзя переименовывать. Точнее, ни одно из этих имен не может передаваться в аргументах *oldname* и *newname*.
5. В особом случае, когда аргументы *oldname* и *newname* указывают на один и тот же файл, функция завершается без признака ошибки, но и не производит никаких изменений.

Если файл *newname* уже существует, мы должны обладать теми же правами, что и для удаления файла. Кроме того, поскольку мы удаляем запись из каталога, содержащего файл *oldname*, и создаем новую запись в файле каталога, где будет находиться *newname*, мы должны обладать правом на запись и выполнение для обоих каталогов.

Функция `renameat` действует так же, как функция `rename`, кроме случая, когда какой-либо из аргументов, *oldname* или *newname*, содержит строку относительного пути. Если *oldname* определяет относительный путь, он откладывается относительно каталога, представленного файловым дескриптором *oldfd*. Аналогично, если *newname* определяет относительный путь, он откладывается относительно каталога, представленного файловым дескриптором *newfd*. В любом из аргументов *oldfd* и *newfd* (или в обоих) можно передать значение `AT_FDCWD`. В этом случае соответствующие относительные пути будут откладываться относительно текущего каталога.

4.17. Символические ссылки

Символическая ссылка — это косвенная ссылка на файл в отличие от жесткой ссылки, которая является прямым указателем на индексный узел файла. Символические ссылки придуманы с целью обойти ограничения, присущие жестким ссылкам.

- Жесткие ссылки обычно требуют, чтобы ссылка и файл размещались в одной файловой системе.
- Только суперпользователь имеет право создавать жесткие ссылки на каталоги (если это поддерживается файловой системой).

Символические ссылки не имеют ограничений, связанных с файловой системой, и любой пользователь сможет создать символическую ссылку на каталог. Символические ссылки обычно используются для «перемещения» файлов или даже целой иерархии каталогов в другое местоположение в системе.

При использовании функций, которые обращаются к файлам по именам, всегда нужно знать, как они обрабатывают символические ссылки. Если функция следует по символической ссылке, она будет воздействовать на файл, на который указывает символическая ссылка. В противном случае операция будет производиться над самой символической ссылкой, а не над файлом, на который она указывает. В табл. 4.9 приводится перечень описываемых в этой главе функций, которые следуют по символическим ссылкам. В этом списке отсутствуют функции `mkdir`, `mkdir`, `mknod` и `rmdir` — они возвращают признак ошибки, если им в качестве аргумента передается символическая ссылка. Кроме того, функции, которые принимают

ют дескриптор файла, такие как `fstat` и `fchmod`, также не были включены в список, поскольку в этом случае обработка символьских ссылок производится функциями, возвращающими файловые дескрипторы (как правило, `open`). Исторически в разных реализациях функция `chown` по-разному обрабатывала символьские ссылки. Однако во всех современных системах `chown` следует по символьским ссылкам.

Впервые символические ссылки появились в 4.2BSD. Первоначально функция `chown` не следовала по символьским ссылкам, но это ее поведение было изменено в версии 4.4BSD. Поддержка символьских ссылок в System V появилась в версии SVR4, но, в отличие от оригинальной версии, она следовала по символьским ссылкам. В старых версиях Linux (до 2.1.81) функция `chown` не следовала по символьским ссылкам. Начиная с версии 2.1.81 функция `chown` следует по символьским ссылкам. В системах FreeBSD 8.0, Mac OS X 10.6.8 и Solaris 10 функция `chown` также следует по символьским ссылкам. Все эти платформы предоставляют функцию `lchown` для изменения владельца самих символьских ссылок.

Таблица 4.9. Интерпретация символьских ссылок различными функциями

Функция	Не следует по символьской ссылке	Следует по символьской ссылке
<code>access</code>		✓
<code>chdir</code>		✓
<code>chmod</code>		✓
<code>chown</code>		✓
<code>creat</code>		✓
<code>exec</code>		✓
<code>lchown</code>	✓	
<code>link</code>		✓
<code>lstat</code>	✓	
<code>open</code>		✓
<code>opendir</code>		✓
<code>pathconf</code>		✓
<code>readlink</code>	✓	
<code>remove</code>	✓	
<code>rename</code>	✓	
<code>stat</code>		✓
<code>truncate</code>		✓
<code>unlink</code>	✓	

Существует одно исключение, не отмеченное в табл. 4.9: когда функция `open` вызывается с установленными одновременно флагами `O_CREAT` и `O_EXCL`. Если в этом случае аргумент `pathname` содержит имя символьской ссылки, функция будет завершаться ошибкой с кодом `EEXIST`. Сделано это с целью закрыть брешь в си-

системе безопасности и предотвратить возможность «обмана» привилегированных процессов путем подмены файлов символическими ссылками.

Пример

С помощью символьской ссылки можно создать замкнутую петлю в файловой системе. Большинство функций, анализирующих путь к файлу, обнаружив такую петлю, возвращают в `errno` код ошибки `ELOOP`. Рассмотрим следующую последовательность команд:

```
$ mkdir foo          создать новый каталог
$ touch foo/a       создать пустой файл
$ ln -s ./foo foo/testdir   создать символьскую ссылку
$ ls -l foo
total 0
-rw-r----- 1 sar      0 Jan 22 00:16 a
1rwxrwxrwx 1 sar     6 Jan 22 00:16 testdir -> ../foo
```

Эта последовательность команд создает каталог `foo`, файл `a` в нем и символьскую ссылку на каталог `foo`. На рис. 4.4 приводится схема, на которой каталоги представлены в виде окружностей, а файл — в виде квадрата.

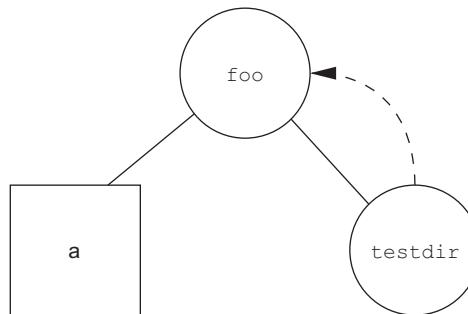


Рис. 4.4. Символьская ссылка `testdir` создает замкнутую петлю

Написав простую программу, которая использует стандартную функцию `ftw(3)` для обхода дерева каталогов и вывода имён всех встретившихся файлов, и запустив её в Solaris, мы получим

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
```

(и еще много строк, пока не произойдет ошибка с кодом `ELOOP`).

В разделе 4.22 будет представлена версия функции `ftw`, которая использует `lstat` вместо `stat`, чтобы предотвратить следование по символическим ссылкам.

Обратите внимание, что в Linux функция `ftruncate` использует функцию `lstat`, поэтому вы не сможете наблюдать подобный эффект.

Разорвать такую замкнутую петлю не составляет труда. Для этого можно вызовом `unlink` удалить файл `foo/testdir`, так как `unlink` не следует по символическим ссылкам. Однако если аналогичную петлю создать с помощью жесткой ссылки, разорвать ее будет намного сложнее. По этой причине функция `link` создает жесткие ссылки на каталоги только при наличии привилегий суперпользователя.

При написании оригинального текста к этому разделу Ричард Стивенс ради эксперимента действительно создал такую петлю у себя. В результате файловая система была повреждена, и не помогла даже утилита `fsck(1)`. Для восстановления файловой системы пришлось прибегнуть к помощи утилит `cltre(8)` и `dcheck(8)`.

Потребность в жестких ссылках на каталоги давно прошла. Пользователи больше не нуждаются в них благодаря функции `mkdir` и символическим ссылкам.

Когда мы открываем файл и передаем функции `open` имя символьической ссылки, она следует по ссылке и открывает файл, на который эта ссылка указывает. Если файл отсутствует, функция `open` возвращает признак ошибки, сообщая о невозможности открытия файла. Это может ввести в заблуждение пользователей, которые не знакомы с символьическими ссылками, например:

```
$ ln -s /no/such/file myfile    создать символьическую ссылку
$ ls myfile
myfile                           команда ls говорит, что файл существует
$ cat myfile                      попробуем заглянуть внутрь файла
cat: myfile: No such file or directory
$ ls -l myfile                     попробуем с ключом -l
lrwxrwxrwx 1 sar     13 Jan 22 00:26 myfile -> /no/such/file
```

Файл `myfile` существует, однако утилита `cat` утверждает обратное, потому что `myfile` — это символьическая ссылка, а сам файл, на который она указывает, отсутствует. Запуск команды `ls` с ключом `-l` дает нам две подсказки: во-первых, строка вывода `ls` начинается с символа `l`, который обозначает символьическую ссылку (`link`), а во-вторых, последовательность `->` говорит о том же. У команды `ls` есть еще один ключ (`-F`), который добавляет к именам символьических ссылок символ `<@>`, благодаря чему можно без труда распознать их даже без ключа `-l`.

4.18. Создание и чтение символьических ссылок

Символьические ссылки создаются с помощью функции `symlink` или `symlinkat`.

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

В файле каталога создается новая запись *sympath*, которая указывает на файл *actualpath*. Функция не требует существования файла *actualpath* на момент создания символьской ссылки. (Мы продемонстрировали эту возможность на примере в предыдущем разделе.) Кроме того, не требуется, чтобы файлы *actualpath* и *sympath* находились в одной и той же файловой системе.

Функция *symlinkat* действует подобно *symlink*, но когда в аргументе *sympath* передается строка относительного пути, этот путь откладывается относительно каталога, представленного файловым дескриптором *fd*. Если в аргументе *sympath* передается строка абсолютного пути или если в аргументе *fd* передается специальное значение *AT_FDCWD*, функция *symlinkat* действует точно как функция *symlink*.

Поскольку функция *open* следует по символьским ссылкам, нам необходим инструмент, с помощью которого можно было бы открыть саму символьскую ссылку, чтобы прочитать имя файла, на который она ссылается. Эти действия выполняют функции *readlink* и *readlinkat*.

```
#include <unistd.h>

ssize_t readlink(const char *restrict pathname, char *restrict buf,
                 size_t bufsize);

ssize_t readlinkat(int fd, const char *restrict pathname,
                  char *restrict buf, size_t bufsize);
```

Обе возвращают количество прочитанных байтов в случае успеха,
-1 — в случае ошибки

Эти функции совмещают в себе операции *open*, *read* и *close*. В случае успеха они возвращают количество прочитанных байтов, размещенных в *buf*. Содержимое символьской ссылки, помещаемое в буфер *buf*, не завершается нулевым символом.

Функция *readlinkat* действует подобно *readlink*, когда в аргументе *pathname* передается строка абсолютного пути или когда аргумент *fd* установлен в специальное значение *AT_FDCWD*. Однако когда в аргументе *fd* указывается действительный файловый дескриптор открытого каталога и в аргументе *pathname* передается строка относительного пути, функция *readlinkat* откладывает путь *pathname* относительно каталога, представленного дескриптором *fd*.

4.19. Временные характеристики файлов

В разделе 4.2 говорилось, что в редакции стандарта Single UNIX Specification 2008 года было увеличено разрешение полей структуры *stat*, представляющих время, и теперь они представляют время не как прежде — в секундах, а в секундах и наносекундах. Фактическое разрешение значений времени, сохраняемых в атрибутах файла, зависит от реализации файловой системы. В файловых системах, сохраняющих время с точностью до секунды, поля с наносекундами будут получать нулевое значение. В файловых системах, сохраняющих время с более

высокой точностью, доли секунд будут преобразовываться в наносекунды и возвращаться в полях с наносекундами.

Каждый файл характеризуется тремя атрибутами времени. Их назначение приводится в табл. 4.10.

Таблица 4.10. Три атрибута времени

Поле	Описание	Пример	Ключи команды ls(1)
<code>st_atim</code>	Время последнего доступа к содержимому файла	<code>read</code>	<code>-u</code>
<code>st_mtim</code>	Время последнего изменения содержимого файла	<code>write</code>	по умолчанию
<code>st_ctim</code>	Время последнего изменения статуса индексного узла	<code>chmod, chown</code>	<code>-c</code>

Обратите внимание на различие между временем последнего изменения содержимого файла (`st_mtim`) и временем последнего изменения статуса индексного узла (`st_ctim`). Время последнего изменения содержимого файла показывает, когда в последний раз вносились изменения в файл. Время последнего изменения статуса индексного узла определяет время последней модификации индексного узла файла. В этой главе мы упоминали множество операций, которые изменяют индексный узел, не затрагивая содержимого файла: изменение прав доступа, идентификатора пользователя (владельца), количества ссылок на файл и др. Поскольку информация индексного узла хранится отдельно от содержимого файла, то кроме времени последнего изменения содержимого файла существует и такая характеристика, как время последнего изменения его статуса.

Отметьте также, что система не отслеживает время последнего доступа к индексному узлу. По этой причине функции `access` и `stat`, например, не изменяют ни одной из трех величин.

Время последнего доступа к файлу часто используется системными администраторами для удаления ненужных файлов, к которым давно никто не обращался. Классический пример — удаление файлов с именами `a.out` и `core`, к которым не обращались более одной недели. Для выполнения подобного рода действий часто применяется утилита `find(1)`.

Время последнего изменения содержимого файла и время последнего изменения статуса индексного узла могут использоваться, чтобы отобрать для архивирования только те файлы, содержимое которых претерпело изменения или у которых был изменен статус индексного узла.

Команда `ls` может отображать или сортировать только по одному из трех значений. По умолчанию при запуске с ключом `-l` или `-t` она использует время последнего изменения содержимого файла. Ключ `-u` заставляет ее использовать время последнего доступа, а ключ `-c` — время последнего изменения статуса индексного узла.

В табл. 4.11 приводится перечень функций и их действие на эти три величины. В разделе 4.14 мы уже говорили, что каталог — это на самом деле файл, состоящий из серии записей, каждая из которых содержит имя файла и номер индексного узла файла. Добавление, удаление или изменение этих записей приводит к из-

менению значений времени, связанных с каталогом. По этой причине табл. 4.11 содержит одну колонку для атрибутов времени, связанных с самим файлом или каталогом, и отдельную колонку для атрибутов времени родительского каталога. Создание нового файла, например, воздействует на временные характеристики не только самого файла, но и каталога, в котором этот файл размещается. Однако операции чтения и записи оказывают влияние только на индексный узел файла и никак не сказываются на содержащем этот файл каталоге.

Таблица 4.11. Влияние различных функций на время последнего доступа к файлу, последнего изменения содержимого файла и последнего изменения статуса индексного узла

Функция	Файл или каталог			Родительский каталог			Раздел	Примечание
	а	м	с	а	м	с		
chmod, chmod			✓				4.9	
chown, fchown			✓				4.11	
creat	✓	✓	✓		✓	✓	3.4	Создание нового файла (O_CREAT)
creat		✓	✓				3.4	Усечение существующего файла (O_TRUNC)
exec	✓						8.10	
lchown			✓				4.11	
link			✓		✓	✓	4.15	Родительский каталог для второго аргумента
mkdir	✓	✓	✓		✓	✓	4.21	
mkfifo	✓	✓	✓		✓	✓	15.5	
open	✓	✓	✓		✓	✓	3.3	Создание нового файла (O_CREAT)
open		✓	✓				3.3	Усечение существующего файла (O_TRUNC)
pipe	✓	✓	✓				15.2	
read	✓						3.7	
remove			✓		✓	✓	4.15	Когда remove = unlink
remove					✓	✓	4.15	Когда remove = rmdir
rename			✓		✓	✓	4.16	Для обоих аргументов
rmdir					✓	✓	4.21	
truncate, ftruncate		✓	✓				4.13	
unlink			✓		✓	✓	4.15	
utimes, utimensat, futimens	✓	✓	✓				4.20	
write		✓	✓				3.8	

(Функции `mkdir` и `rmdir` обсуждаются в разделе 4.21. Функции `utimes`, `utimensat`, `futimens` описаны в следующем разделе. Шесть функций семейства `exec` рассматриваются в разделе 8.10, а функции `mkfifo` и `pipe` — в главе 15.)

4.20. Функции `futimens`, `utimensat` и `utimes`

Существует несколько функций, позволяющих изменять значения времени последнего доступа и последнего изменения файла. Функции `futimens` и `utimensat` поддерживают значения времени с точностью до наносекунд, принимая структуру `timespec` (ту же самую, что используется семейством функций `stat`, как описывается в разделе 4.2).

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

int futimens(int fd, const struct timespec times[2]);
int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

В первом элементе массива `times` обе функции принимают время последнего доступа к файлу, а во втором — время последнего изменения содержимого файла. Оба значения интерпретируются как календарное время — количество секунд, прошедших с начала Эпохи, как описывается в разделе 1.10. Доли секунд выражаются в наносекундах.

В аргументе `times` можно передать четыре значения.

1. Пустой указатель. В этом случае время последнего доступа к файлу и время последнего изменения файла устанавливаются равными текущему времени.
2. Указатель на массив с двумя структурами `timespec`. Если в какой-либо из структур поле `tv_nsec` содержит специальное значение `UTIME_NOW`, в качестве соответствующего значения времени файла принимается текущее время, при этом значение поля `tv_sec` игнорируется.
3. Указатель на массив с двумя структурами `timespec`. Если в какой-либо из структур поле `tv_nsec` содержит специальное значение `UTIME_OMIT`, соответствующее значение времени файла не изменяется, при этом значение поля `tv_sec` игнорируется.
4. Указатель на массив с двумя структурами `timespec`. Если в какой-либо из структур поле `tv_nsec` содержит значение, отличное от `UTIME_NOW` и `UTIME_OMIT`, соответствующее значение времени файла определяется полями `tv_sec` и `tv_nsec`.

Привилегии, необходимые для вызова функций, зависят от значения аргумента `times`.

- Если в аргументе `times` передается пустой указатель или поле `tv_nsec` какой-либо из структур установлено в значение `UTIME_NOW`, процесс должен обладать

правом записи в файл, или иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла, или обладать привилегиями суперпользователя.

- Если в аргументе `times` передается непустой указатель и поле `tv_nsec` в обеих структурах имеет значение, отличное от `UTIME_NOW` и `UTIME_OMIT`, процесс должен иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла, или обладать привилегиями суперпользователя. Простого права на запись будет недостаточно.
- Если в аргументе `times` передается непустой указатель и поле `tv_nsec` в обеих структурах имеет значение `UTIME_OMIT`, проверка прав доступа не производится.

Изменить значения времени с помощью `futimens` можно только для открытого файла. Функция `utimensat` позволяет изменить время последнего доступа и время последнего изменения файла по его имени. Значение аргумента `pathname` интерпретируется как строка относительного пути от каталога, представленного аргументом `fd`, в котором можно передать файловый дескриптор открытого каталога или специальное значение `AT_FDCWD`, соответствующее текущему каталогу вызывающего процесса. Если в аргументе `pathname` указана строка абсолютного пути, аргумент `fd` игнорируется.

Аргумент `flag` функции `utimensat` можно использовать для изменения поведения по умолчанию этой функции. Если в этом аргументе передать флаг `AT_SYMLINK_NOFOLLOW`, будет изменяться время самой символической ссылки (если аргумент `pathname` ссылается на символическую ссылку). По умолчанию функция следует по символическим ссылкам и изменяет значения времени целевого файла, на который указывает ссылка.

Обе функции, `futimens` и `utimensat`, включены в стандарт POSIX.1. Третья функция, `utimes`, включена в стандарт Single UNIX Specification как часть расширений XSI.

```
#include <sys/time.h>
int utimes(const char *pathname, const struct timeval times[2]);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Функция `utimes` изменяет значения времени в файле с именем `pathname`. Аргумент `times` — указатель на массив двух значений времени — последнего доступа и последнего изменения содержимого, но выраженных в секундах и микросекундах:

```
struct timeval {
    time_t tv_sec; /* секунды */
    long   tv_usec; /* микросекунды */
};
```

Обратите внимание: невозможно задать время последнего изменения статуса индексного узла (`st_ctim`), так как оно автоматически изменяется в результате вызова функции `utimes`.

В некоторых версиях UNIX команда `touch(1)` использует эту функцию. Кроме того, стандартные архиваторы `tar(1)` и `cpio(1)` могут вызывать эти функции для установки сохраненных при архивации временных характеристик распакованных файлов.

Пример

Программа в листинге 4.6 усекает размер файла до нуля, используя функцию `open` с флагом `O_TRUNC`, но не изменяет при этом ни время последнего доступа к файлу, ни время последнего изменения файла. Чтобы добиться такого эффекта, программа сначала получает значения временных характеристик файла с помощью функции `stat`, затем усекает размер файла до нуля и в заключение переустанавливает значения времени с помощью функции `futimens`.

Листинг 4.6. Пример использования функции `futimens`

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* получить значения времени */
            err_ret("%s: ошибка вызова функции stat", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* усечение */
            err_ret("%s: ошибка вызова функции open", argv[i]);
            continue;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0) { /* установить значения времени */
            err_ret("%s: ошибка вызова функции futimens", argv[i]);
            close(fd);
        }
    }
    exit(0);
}
```

Продемонстрируем работу программы из листинга 4.6 на примере.

```
$ ls -l changemode times      определим размер и время последнего изменения файлов
-rwxr-xr-x  1 sar   13792 Jan 22 01:26 changemode
-rwxr-xr-x  1 sar   13824 Jan 22 01:26 times
$ ls -lu changemode times    определим время последнего доступа
-rw xr-xr-x  1 sar   13792 Jan 22 22:22 changemode
-rw xr-xr-x  1 sar   13824 Jan 22 22:22 times
$ date          выведем текущее время и дату
Fri Jan 27 20:53:46 EST 2012
$ ./a.out changemode times  запустим программу из листинга 4.6
$ ls -l changemode times    и проверим результаты
-rwxr-xr-x  1 sar       0 Jan 22 01:26 changemode
```

```
-rwxr-xr-x 1 sar      0 Jan 22 01:26 times
$ ls -lu changemode times  проверим также время последнего доступа
-rwxr-xr-x 1 sar      0 Jan 22 22:22 changemode
-rwxr-xr-x 1 sar      0 Jan 22 22:22 times
$ ls -lc changemode times  и время последнего изменения статуса индексного узла
-rwxr-xr-x 1 sar      0 Jan 27 20:53 changemode
-rwxr-xr-x 1 sar      0 Jan 27 20:53 times
```

Как и ожидалось, время последнего доступа к файлу и время последнего изменения его содержимого не изменились. Однако время последнего изменения статуса индексного узла было установлено равным времени запуска программы.

4.21. Функции mkdir, mkdirat и rmdir

Создание каталогов производится с помощью функций `mkdir` и `mkdirat`, а удаление — с помощью функции `rmdir`.

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
int mkdirat(int fd, const char *pathname, mode_t mode);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Эти функции создают новый пустой каталог. Записи «точка» и «точка-точка» создаются автоматически. Права доступа к каталогу, задаваемые аргументом `mode`, модифицируются маской режима создания файлов процесса.

Часто встречается ошибка, когда аргумент `mode` назначается по аналогии с файлами: выдаются только права на запись и на чтение. Но для каталогов, как правило, необходимо устанавливать хотя бы один бит, дающий право на выполнение, чтобы разрешить доступ к файлам, находящимся в каталоге, по их именам (упражнение 4.16).

Идентификаторы пользователя и группы устанавливаются в соответствии с правилами, приведенными в разделе 4.6.

В системах Solaris 10 и Linux 3.2.0 новый каталог наследует бит set-group-ID от родительского каталога. Файлы, созданные в новом каталоге, наследуют от каталога идентификатор группы. В Linux это поведение определяется реализацией файловой системы. Например, файловые системы ext2, ext3 и ext4 предоставляют такую возможность при использовании определенных ключей команды mount(1). Однако реализация файловой системы UFS для Linux не предполагает возможности выбора: бит set-group-ID наследуется всегда, чтобы имитировать исторически сложившуюся реализацию BSD, где идентификатор группы каталога наследуется от родительского каталога.

Реализации, основанные на BSD, не передают бит set-group-ID по наследству — в них просто наследуется идентификатор группы. Поскольку операционные системы FreeBSD 8.0 и Mac OS X 10.6.8 основаны на 4.4BSD, они не требуют наследования бита set-group-ID. На этих платформах вновь создаваемые файлы и каталоги всегда наследуют идентификатор группы родительского каталога независимо от состояния бита set-group-ID.

В ранних версиях UNIX не было функции `mkdir`. Она впервые появилась в 4.2BSD и SVR3. Чтобы создать новый каталог в этих версиях, процесс должен был вызывать функцию `mknode`. Однако использовать эту функцию мог только процесс, обладающий привилегиями суперпользователя. Чтобы как-то обойти это ограничение, обычная команда создания каталога `mkdir(1)` должна была иметь установленный бит `set-user-ID` и принадлежать пользователю `root`. Чтобы создать каталог из процесса, необходимо было вызывать команду `mkdir(1)` с помощью функции `system(3)`.

Функция `mkdirat` действует подобно функции `mkdir`. Если в аргументе `fd` передается специальное значение `AT_FDCWD` или в аргументе `pathname` указана строка абсолютного пути, `mkdirat` действует точно как `mkdir`. Иначе аргумент `fd` интерпретируется как файловый дескриптор открытого каталога, относительно которого откладывается путь `pathname`.

Удаление пустого каталога производится с помощью функции `rmdir`. Напоминаем, что пустым называется каталог, который содержит только две записи: «точка» и «точка-точка».

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Если в результате вызова этой функции счетчик ссылок на каталог становится равным нулю и при этом никакой процесс не удерживает каталог открытым, пространство, занимаемое каталогом, освобождается. Если один или более процессов держат каталог открытым в момент, когда счетчик ссылок достигает значения 0, функция удаляет последнюю ссылку и перед возвратом управления удаляет записи «точка» и «точка-точка». Кроме того, в таком каталоге не могут быть созданы новые файлы. Однако файл каталога не удаляется, пока последний процесс не закроет его. (Даже если другой процесс держит каталог открытым, вряд ли он там может делать что-то особенное, так как для успешного завершения функции `rmdir` каталог должен был быть пуст.)

4.22. Чтение каталогов

Прочитать информацию из файла каталога может любой, кто имеет право на чтение этого каталога. Но только ядро может выполнять запись в каталоги, благодаря чему обеспечивается сохранность файловой системы. В разделе 4.5 мы утверждаем, что возможность создания и удаления файлов в каталоге определяется битами прав на запись и на выполнение, но это не относится к непосредственной записи в файл каталога.

Фактический формат файлов каталогов зависит от реализации UNIX и архитектуры файловой системы. В ранних версиях UNIX, таких как Version 7, структура каталогов была очень простой — каждая запись имела фиксированную длину 16 байт: 14 байт отводилось для имени файла и 2 байта — для номера индексно-

го узла. Когда в 4.2BSD была добавлена поддержка более длинных имен файлов, записи стали иметь переменную длину. Это означало, что любая программа, выполняющая прямое чтение данных из файла каталога, попадала в зависимость от конкретной реализации. Чтобы упростить положение дел, был разработан набор функций для работы с каталогами, который стал частью стандарта POSIX.1. Многие реализации не допускают чтения содержимого файлов каталогов с помощью функции `read`, тем самым препятствуя зависимости приложений от особенностей, присущих конкретной реализации.

```
#include <dirent.h>

DIR *opendir(const char *pathname);

DIR *fdopendir(int fd);
```

Возвращает указатель в случае успеха или
NULL – в случае ошибки

```
struct dirent *readdir(DIR *dp);
```

Возвращает указатель в случае успеха,
NULL – по достижении конца каталога или в случае ошибки

```
void rewinddir(DIR *dp);
```

```
int closedir(DIR *dp);
```

Возвращает 0 в случае успеха или –1 – в случае ошибки

```
long telldir(DIR *dp);
```

Возвращает значение текущей позиции в каталоге, ассоциированном с *dp*

```
void seekdir(DIR *dp, long Loc);
```

Функция `fdopendir` впервые появилась в версии 4 стандарта Single UNIX Specification. Она преобразует дескриптор открытого файла в структуру `DIR` для использования в других функциях обслуживания каталогов.

Функции `telldir` и `seekdir` не являются частью стандарта POSIX.1. Это расширения XSI стандарта Single UNIX Specification, поэтому предполагается, что они должны быть реализованы во всех версиях UNIX, следующих этой спецификации.

Как вы помните, некоторые из этих функций использовались в программе из листинга 1.1, которая воспроизводила ограниченную функциональность команды `ls`.

Структура `dirent` определена в файле `<dirent.h>` и зависит от конкретной реализации. Однако в любой версии UNIX эта структура содержит как минимум следующие два поля:

```
ino_t d_ino; /* номер индексного узла */
char d_name[]; /* строка с именем файла, завершающаяся нулевым символом */
```

Поле d_ino не определено в стандарте POSIX.1, поскольку эта характеристика зависит от конкретной реализации, но оно определяется в расширении XSI базового стандарта POSIX.1. Сам же стандарт POSIX.1 определяет в этой структуре только поле d_name.

Обратите внимание, что размер поля `d_name` не определен, но гарантируется, что он будет не меньше `NAME_MAX`, исключая нулевой символ (вспомните табл. 2.12). Так как строка имени файла заканчивается нулевым символом, не имеет значения, как определен массив `d_name` в заголовочном файле, поскольку размер массива не соответствует длине имени файла.

`DIR` — это внутренняя структура, которая используется этими семью функциями для хранения информации о каталоге. Она похожа на структуру `FILE`, используемую функциями из стандартной библиотеки ввода/вывода, о которых рассказывается в главе 5.

Указатель на структуру `DIR`, возвращаемый функциями `opendir` и `fdopendir`, используется в качестве аргумента остальных пяти функций. Функция `opendir` выполняет первичную инициализацию так, чтобы последующий вызов `readdir` прочитал первую запись из файла каталога. Когда структура `DIR` создается функцией `fdopendir`, выбор первой записи, возвращаемой функцией `readdir`, зависит от смещения в файле, связанном с дескриптором, переданным функции `fdopendir`. Порядок следования записей в каталоге, как правило, зависит от реализации и обычно не совпадает с алфавитным.

Пример

Воспользуемся этими функциями и напишем программу, выполняющую обход дерева каталогов. Цель программы — подсчитать количество файлов каждого типа из перечисленных в табл. 4.3. Программа в листинге 4.7 принимает единственный параметр — имя начального каталога — и рекурсивно спускается от этой точки вниз по дереву каталогов. В Solaris имеется функция `ftw(3)`, которая обходит дерево каталогов, вызывая пользовательскую функцию для каждого встреченного файла. Но с ней связана одна проблема: она вызывает функцию `stat` для каждого файла, в результате чего программа следует по символическим ссылкам. Например, если мы начнем просмотр каталогов от корня файловой системы, в котором имеется символьская ссылка с именем `/lib`, указывающая на каталог `/usr/lib`, все файлы в каталоге `/usr/lib` будут посчитаны дважды. Чтобы устранить эту проблему, Solaris предоставляет дополнительную функцию `nftw(3)`, позволяющую отключить следование по символическим ссылкам. Мы могли бы использовать функцию `nftw`, но давайте напишем собственную версию функции обхода дерева каталогов, чтобы показать принципы работы с каталогами.

В SUSv4 функция nftw включена в стандарт как часть расширения XSI. Реализации этой функции имеются в операционных системах FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10. Системы, основанные на BSD, предоставляют функцию fts(3) с аналогичной функциональностью. Она реализована в операционных системах FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8.

Листинг 4.7. Рекурсивный обход дерева каталогов с подсчетом количества файлов по типам

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* тип функции, которая будет вызываться для каждого встреченного файла */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc    myfunc;
static int       myftw(char *, Myfunc *);
static int       dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int ret;

    if (argc != 2)
        err_quit("Использование: ftw <начальный_каталог>");

    ret = myftw(argv[1], myfunc); /* выполняет всю работу */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1; /* во избежание деления на 0 вывести 0 для всех счетчиков */
    printf("обычные файлы = %7ld, %5.2f %%\n", nreg,
           nreg*100.0/ntot);
    printf("каталоги = %7ld, %5.2f %%\n", ndir,
           ndir*100.0/ntot);
    printf("специальные файлы блочных устройств = %7ld, %5.2f %%\n", nblk,
           nblk*100.0/ntot);
    printf("специальные файлы символьных устройств = %7ld, %5.2f %%\n", nchr,
           nchr*100.0/ntot);
    printf("FIFO = %7ld, %5.2f %%\n", nfifo,
           nfifo*100.0/ntot);
    printf("символические ссылки = %7ld, %5.2f %%\n", nslink,
           nslink*100.0/ntot);
    printf("сокеты = %7ld, %5.2f %%\n", nsock,
           nsock*100.0/ntot);
    exit(ret);
}

/*
 * Выполняет обход дерева каталогов, начиная с каталога "pathname".
 * Для каждого встреченного файла вызывает пользовательскую функцию func().
 */

#define FTW_F 1 /* файл, не являющийся каталогом */
#define FTW_D 2 /* каталог */
#define FTW_DNR 3 /* каталог, который недоступен для чтения */
#define FTW_NS 4 /* файл, информацию о котором */
               /* невозможно получить с помощью stat */

static char *fullpath; /* полный путь к каждому из файлов */
static size_t pathlen;
```

```

static int /* возвращает то, что вернула функция func() */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(&len); /* выделить память для PATH_MAX+1 байт */
                                    /* (листинг 2.3) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("ошибка вызова realloc");
    }
    strcpy(fullpath, pathname);
    return(dopath(func));
}

/*
 * Выполняет обход дерева каталогов, начиная с "fullpath".
 * Если "fullpath" не является каталогом, для него вызывается lstat(),
 * func() и затем выполняется возврат.
 * Для каталогов производится рекурсивный вызов функции.
 */
static int /* возвращает то, что вернула функция func() */
dopath(Myfunc* func)
{
    struct stat      statbuf;
    struct dirent   *dirp;
    DIR              *dp;
    int               ret, n;

    if (lstat(fullpath, &statbuf) < 0) /* ошибка вызова функции stat */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0) /* не каталог */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * Это каталог. Сначала вызвать функцию func(),
     * а затем обработать все файлы в этом каталоге.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    n = strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) { /* увеличить размер буфера */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("ошибка вызова realloc");
    }
    fullpath[n++] = '/';
    fullpath[n] = 0

    if ((dp = opendir(fullpath)) == NULL) /* каталог недоступен */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 ||
            strcmp(dirp->d_name, "..") == 0)
            continue; /* пропустить каталоги "." и ".." */
        strcpy(&fullpath[n], dirp->d_name); /* добавить имя после слеша */
        if ((ret = dopath(func)) != 0)          /* рекурсия */
            break; /* выход по ошибке */
    }
}

```

```

fullpath[n-1] = 0; /* стереть часть строки от слеша и до конца */

if (closedir(dp) < 0)
    err_ret("невозможно закрыть каталог %s", fullpath);
return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;      break;
        case S_IFBLK:   nblk++;      break;
        case S_IFCHR:   nchr++;      break;
        case S_IFIFO:   nfifo++;     break;
        case S_IFLNK:   nslink++;    break;
        case S_IFSOCK:  nsock++;     break;
        case S_IFDIR:   /* каталоги должны иметь type = FTW_D*/
            err_dump("признак S_IFDIR для %s", pathname);
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("закрыт доступ к каталогу %s", pathname);
        break;
    case FTW_NS:
        err_ret("ошибка вызова функции stat для %s", pathname);
        break;
    default:
        err_dump("неизвестный тип %d для файла %s", type, pathname);
    }
    return(0);
}

```

Эта программа получилась даже более универсальной, чем необходимо для демонстрации возможностей функций `ftw` и `nftw`. Например, функция `myfunc` всегда возвращает 0, хотя функция, которая ее вызывает, готова обработать и ненулевое значение.

За дополнительной информацией о технике обхода дерева каталогов и использовании ее в стандартных командах UNIX — `find`, `ls`, `tar` и др. — обращайтесь к [Fowler, Korn and Vo, 1989].

4.23. Функции chdir, fchdir и getcwd

Для каждого процесса определен текущий рабочий каталог. Относительно этого каталога вычисляются все относительные пути (то есть пути, которые не начинаются с символа слеша). Когда пользователь входит в систему, текущим рабочим каталогом обычно становится каталог в шестом поле записи из файла `/etc/passwd` — домашний каталог пользователя. Текущий рабочий каталог — это атрибут процесса, домашний каталог — атрибут пользователя.

Процесс может изменить текущий рабочий каталог вызовом функции `chdir` или `fchdir`.

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
```

Возвращают 0 в случае успеха, -1 — в случае ошибки

Новый рабочий каталог можно задать строкой *pathname* или файловым дескриптором.

Пример

Поскольку текущий рабочий каталог является атрибутом процесса, вызов функции `chdir` в дочернем процессе никак не влияет на текущий рабочий каталог родительского процесса. (Отношения между процессами подробно рассматриваются в главе 8.) Это значит, что программа в листинге 4.8 работает не так, как мы ожидаем.

Листинг 4.8. Пример использования функции `chdir`

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("ошибка вызова функции chdir");
    printf("каталог /tmp стал текущим рабочим каталогом\n");
    exit(0);
}
```

После компиляции и запуска этой программы мы получим следующие результаты (`mycd` — выполняемый файл программы):

```
$ pwd
/usr/lib
$ mycd
каталог /tmp стал текущим рабочим каталогом
$ pwd
/usr/lib
```

Текущий рабочий каталог командной оболочки, запустившей программу `mycd`, не изменился. Это побочный эффект способа, каким командная оболочка запускает программы. Каждая программа выполняется как отдельный процесс, благодаря чему текущий рабочий каталог самой командной оболочки нельзя изменить вызовом функции `chdir` из программы. По этой причине функция `chdir` должна вызываться самой командной оболочкой, для чего командные оболочки предоставляют встроенную команду `cd`.

Поскольку ядро хранит сведения о текущем рабочем каталоге, должен быть способ получить его текущее значение. К сожалению, ядро хранит не полный путь

к каталогу, а некоторую иную информацию, такую как указатель на виртуальный узел (*v-node*) каталога.

Ядро Linux может определить полный путь к каталогу. Компоненты этого пути разбросаны по таблицам смонтированных файловых систем и кэша каталогов и повторно собираются воедино, например, когда выполняется попытка прочитать символьическую ссылку /proc/self/cwd.

Чтобы определить абсолютный путь к текущему рабочему каталогу, нужна функция, перемещающаяся вверх по дереву каталогов, начиная с текущего («точка») и далее через специальные каталоги «точка-точка», пока не достигнет корневого каталога. В каждом из промежуточных каталогов функция должна читать записи из файла каталога, пока не найдет название, соответствующее индексному узлу предыдущего каталога. Повторяя эту процедуру до достижения корневого каталога, мы в результате получим абсолютный путь к текущему рабочему каталогу. К счастью, такая функция уже существует.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Возвращает указатель на *buf* в случае успеха, NULL – в случае ошибки

Эта функция требует передать ей адрес буфера *buf* и его размер в байтах. Буфер должен быть достаточно большим, чтобы вместить строку абсолютного пути к каталогу плюс завершающий нулевой символ. (Проблему выделения памяти для строки абсолютного пути к файлу мы уже обсуждали в разделе 2.5.5.)

Некоторые старые версии UNIX допускают в качестве указателя на буфер передавать значение NULL. В этом случае функция сама выделяет для буфера память размером size байтов с помощью функции malloc. Такое поведение не предусматривается стандартами POSIX.1 или Single UNIX Specification, и его не следует использовать в программах.

Пример

Программа в листинге 4.9 переходит в указанный каталог, после чего вызывает *getcwd* и выводит строку пути к текущему рабочему каталогу. Запустив программу, мы получили следующее:

```
$ ./a.out
 cwd = /var/spool/uucppublic
 $ ls -l /usr/spool
 lrwxrwxrwx 1 root 12 Jan 31 07:57 /usr/spool -> ../../var/spool
```

Листинг 4.9. Пример использования функции getcwd

```
#include "apue.h"
```

```
int
main(void)
{
    char    *ptr;
    size_t  size;
```

```

if (chdir("/usr/spool/uucppublic") < 0)
    err_sys("ошибка вызова функции chdir");

ptr = path_alloc(&size); /* наша собственная функция */
if (getcwd(ptr, size) == NULL)
    err_sys("ошибка вызова getcwd");

printf("cwd = %s\n", ptr);
exit(0);
}

```

Обратите внимание, что функция `chdir` следует по символическим ссылкам, как это и должно быть в соответствии с табл. 4.9, но когда происходит подъем вверх по дереву каталогов, `getcwd` понятия не имеет, что попала в каталог `/var/spool` по символической ссылке `/usr/spool`. Это одна из особенностей символьических ссылок.

Функция `getcwd` очень удобна для приложений, в которых возникает необходимость возврата к первоначальному текущему каталогу. Для этого перед сменой текущего рабочего каталога нужно вызвать `getcwd` и сохранить полученное значение. По окончании работы можно передать сохраненную строку функции `chdir` и вернуться в первоначальный рабочий каталог.

Функция `fchdir` позволяет решить задачу еще проще. Вместо вызова `getcwd` можно открыть текущий каталог, сохранить файловый дескриптор и затем сменить текущий каталог. Когда возникнет необходимость вернуться к первоначальному местоположению, остается просто передать дескриптор функции `fchdir`.

4.24. Специальные файлы устройств

Очень часто возникает путаница с полями `st_dev` и `st_rdev`. Нам они потребуются в разделе 18.9 при написании функции `ttyname`. Правила их использования просты.

- Каждая файловая система характеризуется старшим и младшим номерами устройства, представленными элементарным системным типом `dev_t`. Старший номер устройства идентифицирует драйвер устройства и иногда указывает, с какой платой периферийного устройства следует взаимодействовать. Младший номер идентифицирует конкретное подустройство. Как показано на рис. 4.1, на одном и том же дисковом устройстве может размещаться несколько файловых систем. Все файловые системы на одном и том же дисковом устройстве, как правило, имеют одинаковые старшие номера, но различные младшие номера устройства.
- Обычно старший и младший номера устройства можно получить с помощью макросов, определенных в большинстве реализаций: `major` и `minor`. То есть нам не нужно задумываться, как хранятся два номера в одной переменной типа `dev_t`.

В ранних версиях UNIX старший и младший номера устройств хранились в виде 16-разрядного целого числа, в котором 8 разрядов отводилось для старшего и 8 разрядов – для младшего номера устройства. FreeBSD 8.0 и Mac OS X 10.6.8 используют для этих целей

32-разрядные целые числа, где для хранения старшего номера устройства отводится 8 разрядов, а для младшего – 24 разряда. На 32-разрядных платформах Solaris 10 использует 32-разрядные целые числа, в которых для старшего номера отводится 14 разрядов, а для младшего – 18 разрядов. На 64-разрядных платформах Solaris 10 используются 64-разрядные целые числа, в которых каждому номеру отводится по 32 разряда. В Linux 3.2.0, несмотря на то что тип `dev_t` определен как 64-разрядное целое, для старшего номера отводится только 12 разрядов, а для младшего – 20 разрядов.

Стандарт POSIX.1 оговаривает существование типа `dev_t`, но не определяет формат хранения и способ интерпретации его содержимого. В большинстве систем для этих целей существуют макросы `major` и `minor`, но имя заголовочного файла, в котором они определены, зависит от конкретной системы. В BSD-системах их определения находятся в файле `<sys/types.h>`; в Solaris – в файле `<sys/mkdev.h>`, потому что макросы в файле `<sys/sysmacros.h>` в Solaris считаются устаревшими; в Linux – в файле `<sys/sysmacros.h>`, который подключается в файле `<sys/types.h>`.

- В поле `st_dev` для каждого файла хранится номер устройства файловой системы, где располагается файл и соответствующий ему индексный узел.
- Поле `st_rdev` имеет определенное значение только для специальных файлов символьных или блочных устройств. В этом поле хранится номер фактического устройства.

Пример

Программа в листинге 4.10 выводит номера устройств для каждого аргумента командной строки. Кроме того, если аргумент представляет специальный файл блочного или символьного устройства, дополнительно выводится содержимое поля `st_rdev`.

Листинг 4.10. Вывод содержимого полей `st_dev` и `st_rdev`

```
#include "apue.h"
#ifndef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int             i;
    struct stat    buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("ошибка вызова функции stat");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev), minor(buf.st_dev));

        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                   (S_ISCHR(buf.st_mode)) ? "симв. устр." : "блочное устр.",
                   major(buf.st_rdev), minor(buf.st_rdev));
        }
    }
}
```

```

        }
        printf("\n");
    }

    exit(0);
}

```

Запуск этой программы дает следующие результаты:

```

$ ./a.out / /home/sar /dev/tty[01]
/: dev = 8/3
/home/sar: dev = 8/4
/dev/tty0: dev = 0/5 (симв. устр.) rdev = 4/0
/dev/tty1: dev = 0/5 (симв. устр.) rdev = 4/1
$ mount                                     какие устройства в какие каталоги смонтированы?
/dev/sda3 on / type ext3 (rw,errors=remount-ro,commit=0)
/dev/sda4 on /home type ext3 (rw,commit=0)
$ ls -l /dev/tty[01] /dev/sda[34]
brw-rw---- 1 root      8,  3 2011-07-01 11:08 /dev/sda3
brw-rw---- 1 root      8,  4 2011-07-01 11:08 /dev/sda4
crw--w----
```

Первые два аргумента программы — это каталоги (`/` и `/home/sar`), другие два — специальные файлы устройств `/dev/tty[01]`. (Мы воспользовались поддержкой регулярных выражений в языке командной оболочки, чтобы сократить объем вводимого с клавиатуры текста. Командная оболочка преобразует строку `/dev/tty[01]` в `/dev/tty0 /dev/tty1`.)

Мы предполагаем, что специальные файлы представляют символьные устройства. Наша программа показала, что номера устройств для каталогов `/` и `/home/sar` различны, следовательно, они находятся в разных файловых системах. Это подтверждается командой `mount(1)`.

Затем мы воспользовались командой `ls`, чтобы отыскать дисковые устройства, о которых сообщила команда `mount`, и терминальные устройства. Два дисковых устройства представлены специальными файлами блочных устройств, терминальные устройства — специальными файлами символьных устройств. (Обычно файлы блочных устройств представляют устройства, которые могут содержать файловые системы с произвольным доступом к данным, — жесткие диски, накопители на гибких магнитных дисках, CD-ROM. Некоторые старые версии UNIX поддерживали накопители на магнитных лентах, но они не получили широкого распространения.)

Обратите внимание, что имена файлов и индексные узлы терминальных устройств (`st_dev`) находятся на устройстве `0/5`, в псевдофайловой системе `devtmpfs`, которая реализована в виде каталога `/dev`, но их фактические номера устройств: `4/0` и `4/1`.

4.25. Коротко о битах прав доступа к файлам

Мы рассмотрели все биты прав доступа к файлам, некоторые из которых могут иметь множество интерпретаций. В табл. 4.12 приводится полный перечень битов прав доступа и их интерпретация для обычных файлов и для каталогов.

Таблица 4.12. Перечень битов прав доступа к файлам

Константа	Описание	Назначение для обычных файлов	Назначение для каталогов
S_ISUID	set-user-ID	Устанавливает эффективный идентификатор пользователя при выполнении	Не используется
S_ISGID	set-group-ID	Если установлен бит group-execute, устанавливает эффективный идентификатор группы при выполнении, иначе включает режим обязательной блокировки файла или отдельных записей (если поддерживается)	Устанавливает идентификатор группы для файлов, создаваемых в этом каталоге, в соответствии с идентификатором группы самого каталога
S_ISVTX	бит sticky	Управляет кэшированием содержимого файлов (если поддерживается)	Ограничивает возможность удаления и переименования файлов в каталоге
S_IRUSR	user-read	Разрешает пользователю читать файл	Разрешает пользователю читать записи в файле каталога
S_IWUSR	user-write	Разрешает пользователю писать в файл	Разрешает пользователю удалять и создавать файлы в каталоге
S_IXUSR	user-execute	Разрешает пользователю выполнять файл	Разрешает пользователю производить поиск по каталогу
S_IRGRP	group-read	Разрешает группе читать файл	Разрешает группе читать записи в файле каталога
S_IWGRP	group-write	Разрешает группе писать в файл	Разрешает группе удалять и создавать файлы в каталоге
S_IXGRP	group-execute	Разрешает группе выполнять файл	Разрешает группе производить поиск по каталогу
S_IROTH	other-read	Разрешает всем остальным читать файл	Разрешает всем остальным читать записи в файле каталога
S_IWOTH	other-write	Разрешает всем остальным писать в файл	Разрешает всем остальным удалять и создавать файлы в каталоге
S_IXOTH	other-execute	Разрешает всем остальным выполнять файл	Разрешает всем остальным производить поиск по каталогу

И наконец, девять констант, которые могут быть сгруппированы по три:

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

4.26. Подведение итогов

Основным предметом обсуждения в этой главе была функция `stat`. Мы детально рассмотрели каждое поле структуры `stat`. Это, в свою очередь, заставило нас исследовать все существующие в UNIX атрибуты файлов. Мы увидели, как файлы и каталоги располагаются в файловой системе и как осуществлять обход файлов и каталогов. Уверенное знание всех свойств файла и всех функций, которые работают с файлами, составляет основу программирования в системе UNIX.

Упражнения

- 4.1 Измените программу в листинге 4.1, чтобы вместо функции `lstat` она вызывала `stat`. Что изменится, если в аргументе командной строки передать программе символьическую ссылку?
- 4.2 Что произойдет, если маску режима создания задать равной 777 (в восьмеричном представлении)? Проверьте результаты с помощью команды `umask`.
- 4.3 Убедитесь, что при сброшенном бите user-read вы не сможете прочитать свои собственные файлы.
- 4.4 Запустите программу в листинге 4.3 после создания файлов `foo` и `bar`. Что произойдет в этом случае?
- 4.5 В разделе 4.12 мы говорили, что нулевой размер для обычных файлов вполне допустим. Мы также говорили, что поле `st_size` имеет определенный смысл для каталогов и символьических ссылок. Могут ли существовать каталоги или символьические ссылки с нулевым размером?
- 4.6 Напишите утилиту, аналогичную `cp(1)`, которая копировала бы файлы с дырками, не записывая байты со значением 0 в выходной файл.
- 4.7 Взгляните на вывод команды `ls` в разделе 4.12, который показывает, что файлы `core` и `core.corey` имеют различные права доступа. Объясните, как могли появиться такие различия, если исходить из предположения, что в промежутке времени между созданием этих файлов значение `umask` не изменилось.
- 4.8 При запуске программы из листинга 4.5 мы проверяли доступный объем дискового пространства с помощью команды `df(1)`. Почему нельзя было воспользоваться командой `du(1)`?
- 4.9 Таблица 4.11 утверждает, что функция `unlink` воздействует на время последнего изменения статуса индексного узла. Как это может быть?
- 4.10 Как влияет системный предел числа одновременно открытых файлов на функцию `myftw` из раздела 4.22?
- 4.11 Наша версия функции `ftw` никогда не покидает текущий каталог. Измените эту функцию так, чтобы каждый раз, встречая каталог, она вызывала функцию `chdir` для перехода в этот каталог и передавала функции `lstat` не полный путь к файлу, а только его имя. После обработки всех файлов в ка-

тологе произведите вызов `chdir("..")`. Сравните время работы этих двух версий.

- 4.12 Для каждого процесса определен также корневой каталог, который используется в качестве отправной точки при разрешении абсолютных путей к файлам. Корневой каталог процесса можно изменить вызовом функции `chroot`. Найдите описание этой функции в своем справочном руководстве. В каких случаях можно использовать эту функцию?
- 4.13 Как с помощью функции `utimes` можно изменить только один атрибут времени из двух?
- 4.14 Некоторые версии команды `finger(1)` выводят сообщения «New mail received ...» (Получена новая почта ...) и «unread since ...» (не прочитано после ...), где многоточием обозначены соответствующее время и дата. Как программа может определить эти время и дату?
- 4.15 Изучите различные форматы архивов, создаваемых командами `cpio(1)` и `tar(1)`. (Их описание обычно можно найти в разделе 5 «UNIX Programmer's Manual».) Какие временные характеристики файлов могут быть сохранены в архиве? Какое значение времени последнего доступа к файлу будет установлено при его разархивировании и почему?
- 4.16 Существует ли в UNIX фундаментальное ограничение на количество вложенных каталогов? Чтобы узнать это, напишите программу, которая в цикле будет создавать новый каталог и сразу же выполнять переход в него. Убедитесь, что длина строки абсолютного пути к последнему каталогу превышает системный предел `PATH_MAX`. Есть ли возможность вызвать функцию `getcwd` из последнего каталога, чтобы получить абсолютный путь к нему? Как стандартные утилиты UNIX работают с такими длинными путями? Можно ли заархивировать такое дерево каталогов с помощью `tar` или `cpio`?
- 4.17 В разделе 3.16 мы описали специальный каталог `/dev/fd/`. Чтобы любой пользователь смог обращаться к файлам в этом каталоге, для них должны быть установлены права доступа `rw-rw-rw-`. Некоторые программы перед созданием нового файла сначала удаляют его, если он уже существует, игнорируя при этом возвращаемое значение функции. Вот как это делается:

```
unlink(path);
if ((fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

Что произойдет, если в аргументе `path` передать строку `/dev/fd/1`?

5

Стандартная библиотека ввода/вывода

5.1. Введение

В этой главе мы исследуем стандартную библиотеку ввода/вывода. Эта библиотека определена стандартом ISO C, потому что реализована во многих операционных системах, не относящихся к семейству UNIX. Стандарт Single UNIX Specification определяет для нее дополнительные интерфейсы в качестве расширений стандарта ISO C.

Стандартная библиотека ввода/вывода сама производит размещение буферов и выполняет операции ввода/вывода блоками оптимального размера, что избавляет от необходимости задумываться о правильности выбора (раздел 3.9). Это упрощает использование библиотеки, но в то же время неумелое обращение с ней может стать источником других проблем.

Стандартная библиотека ввода/вывода написана Деннисом Ритчи примерно в 1975 году. Это была генеральная ревизия библиотеки Portable I/O Майка Леска. Удивительно, насколько несущественно изменилась библиотека за последние 30 лет.

5.2. Потоки и объекты FILE

Все функции, описанные в главе 3, работали с файлами посредством дескрипторов. Функция, открывающая файл, возвращает дескриптор, который затем используется во всех последующих операциях ввода/вывода. При обсуждении стандартной библиотеки ввода/вывода мы будем отталкиваться от термина *поток ввода/вывода*, или просто *поток*. (Не путайте стандартный термин *поток ввода/вывода* (stream) с системой ввода/вывода STREAMS, которая является частью System V и стандартизирована в расширении XSI STREAMS к стандарту Single UNIX Specification.) Открывая или создавая файл средствами стандартной библиотеки ввода/вывода, мы говорим, что связали поток с файлом.

В наборе символов ASCII каждый символ представлен одним байтом. В национальных наборах символов один символ может быть представлен несколькими байтами. Стандартные файловые потоки ввода/вывода могут использоваться как с однобайтными, так и с многобайтными («wide» — «широкими») наборами символов. Ориентация потока определяет, являются ли читаемые и записываемые символы однобайтными или многобайтными. Изначально, в момент создания,

поток не имеет ориентации. Если с неориентированным потоком ввода/вывода используется функция, оперирующая многобайтными символами (см. `<wchar.h>`), для потока устанавливается ориентация на «широкие» символы. Если с неориентированным потоком ввода/вывода используется функция, оперирующая однобайтными символами, устанавливается ориентация на однобайтные символы. Изменить установленную ориентацию могут только две функции. Функция `freopen` (рассматривается чуть ниже) сбрасывает ориентацию потока, а функция `fwide` устанавливает ориентацию потока.

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

Возвращает положительное число, если поток ориентирован на многобайтные символы, отрицательное число, если ориентирован на однобайтные символы, или 0, если поток не имеет ориентации

Функция `fwide` решает разные задачи в зависимости от значения аргумента `mode`.

- Если аргумент `mode` – отрицательное число, функция `fwide` попытается назначить потоку ориентацию на однобайтные символы.
- Если аргумент `mode` – положительное число, функция `fwide` попытается назначить потоку ориентацию на многобайтные символы.
- Если аргумент `mode` равен 0, функция `fwide` не будет менять ориентацию потока, а просто вернет значение, соответствующее текущей ориентации.

Обратите внимание: функция `fwide` не может изменить ориентацию уже ориентированного потока. Кроме того, она не возвращает признак ошибки. Единственное, что можно сделать в этом случае, – очистить переменную `errno` перед вызовом функции `fwide` и затем проверить ее значение после вызова. На протяжении оставшейся части книги мы будем иметь дело только с потоками, ориентированными на однобайтные символы.

При открытии потока стандартная функция `fopen` (раздел 5.5) возвращает указатель на объект `FILE`. Этот объект, как правило, является структурой со всей информацией, необходимой для управления потоком средствами стандартной библиотеки ввода/вывода: дескриптор файла, используемый в операциях ввода/вывода, указатель на буфер потока, размер буфера, счетчик символов, находящихся в настоящий момент в буфере, флаг ошибки и т. д.

Прикладные программы никогда не работают с объектом `FILE` напрямую. Чтобы сослаться на поток, достаточно просто передать указатель на объект `FILE` в аргументе любой стандартной функции ввода/вывода. Далее в тексте книги указатель на объект `FILE`, тип `FILE *` мы будем называть указателем на файл.

В этой главе стандартная библиотека ввода/вывода обсуждается в контексте ОС UNIX. Как мы уже упоминали, эта библиотека перенесена на самые разные платформы. Но чтобы вы получили некоторое представление, как эта библиотека мо-

жет быть реализована, мы будем отталкиваться от типичной ее реализации в системе UNIX.

5.3. Стандартные потоки ввода, вывода и сообщений об ошибках

Для любого процесса автоматически создается три предопределенных потока: стандартный поток ввода, стандартный поток вывода и стандартный поток сообщений об ошибках. Эти потоки связаны с теми же файлами, что и дескрипторы `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`, упоминавшиеся в разделе 3.2.

Эти три потока доступны посредством предопределенных указателей на файлы `stdin`, `stdout` и `stderr`. Определения файловых указателей находятся в заголовочном файле `<stdio.h>`.

5.4. Буферизация

Поддержка буферизации в стандартной библиотеке ввода/вывода реализована с целью уменьшить количество обращений к функциям `read` и `write`. (В табл. 3.3 мы приводили зависимость производительности операций ввода/вывода от размера буфера.) Кроме того, библиотека стремится произвести буферизацию потоков ввода/вывода автоматически, чтобы избавить приложения от необходимости беспокоиться о ней. К сожалению, буферизация — это тот самый аспект стандартной библиотеки ввода/вывода, который более всего смущает программистов.

Библиотека поддерживает три типа буферизации.

1. Полная буферизация. В этом случае фактический ввод/вывод осуществляется, только когда будет заполнен стандартный буфер ввода/вывода. Обычно стандартная библиотека ввода/вывода использует полную буферизацию для файлов на диске. Буфер, как правило, создается одной из стандартных функций ввода/вывода вызовом `malloc` (раздел 7.8) во время первой операции ввода/вывода.

Операция записи содержимого стандартного буфера ввода/вывода описывается термином *flush* (сбрасывать). Буфер может сбрасываться на диск автоматически одной из функций, например при его заполнении или с помощью функции `fflush`. К сожалению, в UNIX термин *flush* имеет два разных смысла. В терминах стандартной библиотеки ввода/вывода он означает «запись содержимого буфера на диск». В терминах драйвера терминала, например, для функции `tcfflush` (глава 18), он означает «удаление данных из буфера».

2. Построчная буферизация. В этом случае фактический ввод/вывод осуществляется, когда в потоке встречается символ перевода строки. Это позволяет выводить по одному символу за раз (с помощью стандартной функции `fputc`), зная при этом, что фактическая запись произойдет, только когда строка будет закончена. Построчная буферизация обычно используется для потоков, свя-

занных с терминальными устройствами, например для стандартного ввода и стандартного вывода.

Необходимо упомянуть несколько аспектов, касающихся построчной буферизации. Во-первых, буфер, используемый стандартной библиотекой ввода/вывода для сборки строки, имеет фиксированный размер, поэтому фактическая операция ввода/вывода может быть выполнена еще до того, как встретится символ перевода строки, если буфер заполнится раньше. Во-вторых, всякий раз, когда ввод производится средствами стандартной библиотеки ввода/вывода либо (а) из небуферизованного потока, либо (б) из потока с построчной буферизацией, который требует обращения к ядру за данными, все выходные потоки с построчной буферизацией сбрасываются. Уточнение для случая (б) необходимо, поскольку требуемые данные могут уже находиться в буфере и за ними не обязательно было бы обращаться к ядру. В случае (а) вполне очевидно, что требуемые данные можно получить только от ядра.

3. Отсутствие буферизации. Стандартная библиотека ввода/вывода не буферизует операции с символами. Если мы записываем в поток 15 символов (например, с помощью функции `fputs`), то рассчитываем, что эти 15 символов будут выведены как можно скорее, возможно, с помощью функции `write` (раздел 3.8).

Так, например, стандартный поток сообщений об ошибках обычно не буферизуется. В результате сообщения выводятся максимально быстро, вне зависимости от наличия символа перевода строки.

Стандарт ISO C предъявляет следующие требования к буферизации:

- Стандартные потоки ввода/вывода буферизуются полностью, но только если они не связаны с интерактивными устройствами.
- Стандартный поток сообщений об ошибках никогда не подвергается полной буферизации.

Однако эти требования ничего не говорят о том, могут ли стандартные потоки ввода и вывода быть небуферизованными или построчно буферизованными, если они связаны с интерактивными устройствами, и должен ли стандартный поток сообщений об ошибках быть небуферизованным или построчно буферизованным. В большинстве реализаций по умолчанию используются следующие виды буферизации:

- Стандартный поток сообщений об ошибках никогда не буферизуется.
- Все остальные потоки подвергаются построчной буферизации, если связаны с терминальным устройством, и полной буферизации — в любом другом случае.

Все четыре платформы, обсуждаемые в этой книге, следуют этим соглашениям: стандартный поток сообщений об ошибках не буферизуется, потоки, связанные с терминальными устройствами, подвергаются построчной буферизации, а все остальные потоки буферизуются полностью.

Более детально мы исследуем буферизацию стандартного ввода/вывода в разделе 5.12 и на примере программы в листинге 5.3.

Если нас не устраивают принятые по умолчанию виды буферизации для какого-либо потока, их можно изменить с помощью функции `setbuf` или `setvbuf`.

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);
```

Возвращают 0 в случае успеха, ненулевое значение — в случае ошибки

Эти функции должны вызываться только *после* открытия потока (это очевидно, так как каждая требует передачи действительного указателя на файл в первом аргументе), но перед любой другой операцией, выполняемой над потоком.

С помощью функции `setbuf` можно разрешить или запретить буферизацию. Чтобы разрешить буферизацию, аргумент `buf` должен содержать указатель на буфер размером `BUFSIZ` (константа, значение которой определено в файле `<stdio.h>`). В этом случае поток обычно буферизуется полностью, но некоторые системы могут назначить потоку построчную буферизацию, если он связан с терминальным устройством. Чтобы запретить буферизацию, нужно в аргументе `buf` передать значение `NULL`.

При использовании функции `setvbuf` явно указывается желаемый режим буферизации. Делается это с помощью аргумента `mode`:

- `_IOFBF` полная буферизация.
- `_IOLBF` построчная буферизация.
- `_IONBF` буферизация отсутствует.

Когда выполняется отключение буферизации, значения аргументов `buf` и `size` игнорируются. Когда включается полная или построчная буферизация, через аргументы `buf` и `size` можно передать указатель на буфер и его размер. Если в аргументе `buf` передать значение `NULL`, библиотека автоматически выделит для потока собственный буфер соответствующего размера. (Под «соответствующим размером» здесь подразумевается значение константы `BUFSIZ`.)

Некоторые реализации библиотеки языка C используют значение из поля `st_blksize` структуры `stat` (раздел 4.2), чтобы определить оптимальный размер буфера ввода/вывода. Далее в этой главе мы увидим, что библиотека GNU C использует этот метод.

В табл. 5.1 перечислены действия, выполняемые этими двумя функциями, в зависимости от значения их аргументов.

Следует знать, что если стандартный буфер ввода/вывода размещен как автоматическая переменная внутри функции, перед возвращением из функции необходимо закрыть поток. (Подробнее мы обсудим этот вопрос в разделе 7.8.) Кроме того, некоторые реализации используют часть буфера для своих внутренних целей, поэтому фактический объем данных, которые могут храниться в буфере, меньше указанного размера `size`. Вообще, лучше позволить системе самой выбирать размер буфера и автоматически размещать его в памяти. В этом случае

Таблица 5.1. Функции setbuf и setvbuf

Функция	mode	buf	Буфер и его размер	Тип буферизации
setbuf		Непустой указатель	Пользовательский буфер размером <code>BUFSIZ</code>	Полная или построчная буферизация
		<code>NULL</code>	Нет буфера	Буферизация отсутствует
setvbuf	<code>_IOFBF</code>	Непустой указатель	Пользовательский буфер размером <code>size</code>	Полная буферизация
		<code>NULL</code>	Системный буфер соответствующего размера	
	<code>_IOLBF</code>	Непустой указатель	Пользовательский буфер размером <code>size</code>	Построчная буферизация
		<code>NULL</code>	Системный буфер соответствующего размера	
	<code>_IONBF</code>	(Игнорируется)	Нет буфера	Буферизация отсутствует

стандартная библиотека ввода/вывода сама освободит память, занимаемую буфером в момент закрытия потока.

Содержимое буфера потока можно сбросить в любой момент.

```
#include <stdio.h>
int fflush(FILE *fp);
```

Возвращает 0 в случае успеха, EOF — в случае ошибки

Функция `fflush` передает ядру все незаписанные данные из буфера. В особом случае, когда в аргументе `fp` передается `NULL`, сбрасывается содержимое буферов всех потоков вывода.

5.5. Открытие потока

Функции `fopen`, `freopen` и `fdopen` открывают стандартный поток ввода/вывода.

```
#include <stdio.h>
FILE *fopen(const char *restrict pathname, const char *restrict type);
FILE *freopen(const char *restrict pathname, const char *restrict type,
             FILE *restrict fp);
FILE *fdopen(int fd, const char *type);
```

Все три возвращают указатель на файл в случае успеха,
`NULL` — в случае ошибки

Перечислим различия между этими функциями.

1. Функция `fopen` открывает указанный файл.
2. Функция `freopen` открывает указанный файл и связывает его с указанным потоком, предварительно закрывая поток, если он уже был открыт. Если перед этим поток имел ориентацию, функция сбрасывает ее. Как правило, эта функция используется для связывания открываемого файла с предопределенным стандартным потоком ввода, вывода или сообщений об ошибках.
3. Функция `fdopen` принимает открытый дескриптор файла, полученный вызовом функций `open`, `dup`, `dup2`, `fcntl`, `pipe`, `socket`, `socketpair` или `accept`, и связывает его с потоком ввода/вывода. Часто эта функция вызывается с дескриптором, который получен в результате создания канала или сетевого соединения. Поскольку эти типы файлов нельзя открыть стандартной функцией `fopen`, приходится сначала открывать их специальными функциями, чтобы получить дескриптор файла, а затем связывать дескриптор с потоком ввода/вывода, используя функцию `fdopen`.

Обе функции, `fopen` и `freopen`, являются частью стандарта ISO C. Функция `fdopen` определена стандартом POSIX.1, потому что ISO C не имеет дела с дескрипторами.

Таблица 5.2. Возможные значения аргумента `type` при открытии потока

type	Описание	Флаги <code>open(2)</code>
r или rb	Открыть для чтения	<code>O_RDONLY</code>
w или wb	Усечь размер файла до 0 или создать и открыть на запись	<code>O_WRONLY O_CREAT O_TRUNC</code>
a или ab	Открыть для записи в конец файла или создать для записи	<code>O_WRONLY O_CREAT O_APPEND</code>
r+, или r+b, или rb+	Открыть для чтения и для записи	<code>O_RDWR</code>
w+, или w+b, или wb+	Усечь размер файла до 0 или создать и открыть для чтения и для записи	<code>O_RDWR O_CREAT O_TRUNC</code>
a+, или a+b, или ab+	Открыть или создать для чтения и для записи в конец файла	<code>O_RDWR O_CREAT O_APPEND</code>

Стандарт ISO C определяет 15 возможных значений аргумента `type`. Все они перечислены в табл. 5.2. Использование символа `b` в аргументе `type` позволяет стандартной системе ввода/вывода различать текстовые и двоичные файлы. Так как ядро UNIX не различает эти типы файлов, указание символа `b` не оказывает никакого влияния.

Для функции `fdopen` значение аргумента `type` несколько отличается. Так как дескриптор уже открыт, открытие для записи не приводит к усечению файла. (Если

дескриптор для существующего файла создается, например, функцией `open`, усечение можно выполнить с помощью флага `O_TRUNC`. Функция `fopen` сама не может выполнить усечение файла, который она открывает для записи.) Кроме того, открытие для записи в конец файла не приводит к созданию нового файла (так как дескриптор может быть связан только с существующим файлом).

Когда файл открыт в режиме добавления в конец, все операции записи производятся в конец файла. Если несколько процессов откроют один и тот же файл в этом режиме, стандартная функция ввода/вывода будет корректно записывать данные каждого процесса.

Версии функции `fopen` из Беркли, предшествовавшие 4.4BSD, и простейшая версия, которая приводится на с. 177 в [Kernighan и Ritchie, 1988], работают с режимом добавления в конец файла не совсем корректно. Эти версии вызывают функцию `lseek` для перехода в конец файла при его открытии. В случае, когда с файлом работают несколько процессов, он должен быть открыт с флагом `O_APPEND`, который мы рассматривали в разделе 3.3. Вызов функции `lseek` перед каждой операцией записи не даст желаемого эффекта; эту проблему мы также обсуждали в разделе 3.11.

Когда файл открывается для чтения и записи (символ «+» в аргументе `type`), применяются следующие ограничения.

- Вывод не может сразу же следовать за вводом без промежуточного вызова функций `fflush`, `fseek`, `fsetpos` или `rewind`.
- Ввод не может сразу следовать за выводом без вызова функций `fseek`, `fsetpos` или `rewind` или операции ввода, которая встречает конец файла.

Различные характеристики шести способов открытия потока, перечисленных в табл. 5.2, можно обобщить, как показано в табл. 5.3.

Таблица 5.3. Шесть способов открытия потоков ввода/вывода

Ограничение	r	w	a	r+	w+	a+
Файл должен существовать	✓			✓		
Предыдущее содержимое файла будет утеряно		✓			✓	
Поток доступен для чтения	✓			✓	✓	✓
Поток доступен для записи		✓	✓	✓	✓	✓
Запись возможна только в конец потока			✓			✓

Обратите внимание, что, используя в аргументе `type` символы `a` или `w`, можно создать новый файл, но при этом нельзя определить биты прав доступа к файлу, как мы делали это с помощью функции `open` или `creat` в главе 3. Стандарт POSIX.1 требует, чтобы реализации создавали файлы со следующим набором битов прав доступа:

`S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH`

Однако в разделе 4.8 говорилось, что эти права могут ограничиваться значением маски `umask`.

По умолчанию поток открывается в режиме полной буферизации, если, конечно, он не связан с терминальным устройством, так как в этом случае используется режим построчной буферизации. Сразу же после открытия потока, но до выполнения первой операции ввода/вывода режим буферизации можно изменить с помощью функций `setbuf` и `setvbuf`, описанных в предыдущем разделе.

Закрывается открытый поток с помощью функции `fclose`.

```
#include <stdio.h>
int fclose(FILE *fp);
```

Возвращает 0 в случае успеха, EOF — в случае ошибки

Перед закрытием потока все данные, находящиеся в буфере вывода, сбрасываются. Все данные, находящиеся в буфере ввода, будут потеряны. Если память под буфер была выделена самой библиотекой ввода/вывода, она освобождается автоматически.

При нормальном завершении процесса, то есть когда непосредственно вызывается функция `exit` или происходит возврат из функции `main`, все незаписанные данные в буферах вывода сбрасываются на диск, после чего все потоки закрываются.

5.6. Чтение из потока и запись в поток

После открытия потока можно выбрать один из трех типов неформатированного ввода/вывода.

1. Посимвольный ввод/вывод. Можно читать или писать по одному символу за раз с помощью стандартных функций ввода/вывода, которые буферизуют данные, если поток буферизован.
2. Построчный ввод/вывод. Если необходимо читать или писать данные построчно, используются функции `fgets` и `fputs`. Каждая строка заканчивается символом перевода строки, а при использовании функции `fgets` нужно указать максимальную длину строки, которую мы можем принять. Эти две функции рассматриваются в разделе 5.7.
3. Прямой ввод/вывод. Этот тип ввода/вывода поддерживается функциями `fread` и `fwrite`. Каждая операция выполняет чтение или запись определенного количества объектов, имеющих заданный размер. Эти функции часто используются для работы с двоичными файлами, когда в каждой операции чтения или записи участвует одна структура данных. Эти функции рассматриваются в разделе 5.9.

Термин «прямой ввод/вывод» из стандарта ISO C имеет также несколько синонимов: двоичный ввод/вывод, ввод/вывод объектами, ввод/вывод записями или ввод/вывод структурами. Не путайте эту особенность с флагом `O_DIRECT` функции `open`, поддерживаемым в FreeBSD и Linux, — они никак не связаны между собой.

(Функции форматированного ввода/вывода, такие как `printf` и `scanf`, описываются в разделе 5.11.)

Функции ввода

Три функции позволяют читать по одному символу за одно обращение.

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

Все три возвращают очередной символ в случае успеха,
EOF — в случае ошибки

Функция `getchar` определена как эквивалент `getc(stdin)`. Разница между первыми двумя функциями заключается в том, что `getc` может быть реализована в виде макроса, а `fgetc` — нет. Это означает следующее.

1. Аргумент функции `getc` не должен быть выражением с побочными эффектами.
2. Поскольку `fgetc` обязательно будет функцией, мы всегда можем узнать ее адрес. Это позволит передать адрес функции `fgetc` в виде аргумента другой функции.
3. Вызов функции `fgetc`, скорее всего, будет более длительным, чем вызов `getc`, так как обычно вызов функции занимает больше времени, чем обращение к макросу.

Все три функции возвращают очередной символ как `unsigned char`, преобразованный в `int`. Причина в том, что функции должны возвращать положительное значение, даже когда старший бит символа установлен в 1. Преобразование в `int` связано с тем, что наряду с обычными символами функции могут возвращать признак ошибки или признак конца файла. Константа `EOF`, определяемая в файле `<stdio.h>`, должна иметь отрицательное значение. Чаще она имеет значение `-1`. Это также означает, что мы не сможем сохранить возвращаемое значение любой из этих трех функций в переменной символьного типа и затем сравнить его с константой `EOF`. Обратите внимание, что эти функции возвращают одно и то же значение и в случае ошибки, и в случае достижения конца файла. Чтобы отличить один случай от другого, следует использовать функцию `ferror` или `feof`.

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
```

Обе возвращают ненулевое значение (истина), если условие истинно,
или 0 (ложь), если условие ложно

```
void clearerr(FILE *fp);
```

В большинстве реализаций в объекте FILE для каждого потока предусматриваются два флага:

- флаг ошибки;
- флаг конца файла.

Оба флага сбрасываются вызовом функции `clearerr`.

После чтения символа из потока его можно вернуть обратно в поток вызовом функции `ungetc`.

```
#include <stdio.h>
int ungetc(int c, FILE *fp);
```

Возвращает значение аргумента `c` в случае успеха, `EOF` — в случае ошибки

Символы, возвращенные в поток, будут заново прочитаны следующими операциями чтения в порядке, обратном порядку их возврата. Надо отметить, что хотя стандарт ISO C позволяет возвращать в поток произвольное количество символов, реализации обязаны предоставлять возможность возврата только одного символа. Поэтому не следует рассчитывать более чем на один символ.

Возвращаемый символ не обязательно должен быть последним прочитанным символом. Нельзя вернуть в поток признак конца файла (`EOF`). Однако по достижении конца файла можно вернуть в поток один символ. Следующая операция чтения вернет этот символ, а следующая за ней вернет `EOF`. Этот прием работает, потому что функция `ungetc` сбрасывает флаг конца файла у потока.

Возврат символов в поток часто используется, когда необходимо прервать чтение на границе слова или лексемы определенного вида. Иногда нужно увидеть следующий символ, чтобы решить, как обрабатывать текущий. В этом случае мы просто возвращаем прочитанный символ в поток, и он будет получен следующим вызовом функции `getc`. Если бы стандартная библиотека ввода/вывода не давала такой возможности, нам приходилось бы сохранять его в переменной, равно как и флаг, указывающий, что следующий символ следует взять из переменной, а не из потока.

Когда символ возвращается в поток вызовом `ungetc`, он на самом деле не записывается обратно в файл или в устройство. Возвращаемые символы просто сохраняются библиотекой ввода/вывода во внутреннем буфере потока.

Функции вывода

Каждой из описанных выше функций ввода соответствует функция вывода.

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

Все три возвращают значение аргумента `c` в случае успеха,

`EOF` — в случае ошибки

По аналогии с функциями ввода, вызов `putchar(c)` эквивалентен вызову `putc(c, stdout)`, и `putc` также может быть реализована в виде макроса, тогда как `fputc` — нет.

5.7. Построчный ввод/вывод

Построчный ввод выполняется двумя функциями, `fgets` и `gets`.

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);
```

Обе возвращают `buf` в случае успеха, `NULL` — в случае ошибки или по достижении конца файла

Обеим функциям передается адрес буфера для размещения прочитанной строки. Функция `gets` читает из стандартного потока ввода, функция `fgets` — из указанного потока.

Для функции `fgets` указывается размер приемного буфера, *n*. Эта функция будет читать входные данные в буфер до тех пор, пока не встретит символ перевода строки, но не более *n* – 1 символов. В конец прочитанной строки добавляется нулевой символ. Если длина строки, включая символ перевода строки, составляет более *n* – 1 символов, функция вернет только часть строки, но в конец буфера все равно будет добавлен завершающий нулевой символ. Последующий вызов `fgets` вернет остаток строки.

Функцию `gets` использовать не следует. Проблема в том, что она не позволяет определить размер приемного буфера. Если входная строка окажется длиннее буфера, это приведет к его переполнению и порче данных, которые находятся в памяти сразу после буфера. Описание, как эта брешь в безопасности использовалась программой-червем в 1988 году, вы найдете в июньском номере «Communications of the ACM» за 1989 год (vol. 32 no. 6). Еще одно отличие `gets` от `fgets` заключается в том, что функция `gets` не сохраняет символ перевода строки в буфере, как это делает функция `fgets`.

Это различие в обработке символа перевода строки уходит корнями в историю UNIX. Еще в руководстве к Version 7 (1979) было отмечено, что: «`gets` удаляет символ перевода строки, `fgets` оставляет его, и все это ради сохранения обратной совместимости».

Хотя стандарт ISO C требует, чтобы реализация предоставляла функцию `gets`, используйте вместо нее функцию `fgets`. В версии стандарта SUSv4 функция `gets` объявлена устаревшей и была исключена из последней версии стандарта ISO C (ISO/IEC 9899:2011).

Операции построчного вывода обеспечиваются функциями `fputs` и `puts`.

```
#include <stdio.h>
int fputs(const char *restrict str, FILE *restrict fp);
int puts(const char *str);
```

Обе возвращают неотрицательное значение в случае успеха,
EOF — в случае ошибки

Функция `fputs` записывает строку, завершающуюся нулевым символом, в указанный поток. Нулевой символ в поток не записывается. Примечательно, что это не построчный вывод в строгом смысле слова, поскольку строка может не содержать символ перевода строки в качестве последнего ненулевого символа. Обычно завершающему нулевому символу действительно предшествует символ перевода строки, но это совершенно не обязательно.

Функция `puts` записывает строку, завершающуюся нулевым символом, в поток стандартного вывода. Она не выводит завершающий нулевой символ, но добавляет символ перевода строки.

Функция `puts` достаточно безопасна в отличие от парной ей `gets`. Однако мы также рекомендуем не пользоваться ею, чтобы не задумываться постоянно о том, добавляет ли она символ перевода строки. Пользуясь только функциями `fgets` и `fputs`, мы всегда будем точно знать, что должны обрабатывать символ перевода строки.

5.8. Эффективность стандартных функций ввода/вывода

Используя функции из предыдущего раздела, мы можем оценить эффективность стандартной библиотеки ввода/вывода. Программа в листинге 5.1 очень похожа на ту, что приводилась в листинге 3.1, — она просто копирует данные из стандартного ввода в стандартный вывод с помощью функций `getc` и `putc`. Эти две функции могут быть реализованы в виде макросов.

Листинг 5.1. Копирование данных со стандартного ввода на стандартный вывод с помощью функций `getc` и `putc`

```
#include "apue.h"

int
main(void)
{
    int      c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Можно также написать версию этой программы с использованием функций `fgetc` и `fputc`, которые всегда реализованы как функции, а не как макросы. (Не будем здесь приводить изменения в исходном коде, поскольку они достаточно тривиальны.)

В заключение приведем еще одну версию программы, которая выполняет чтение и запись построчно (листинг 5.2).

Листинг 5.2. Копирование данных со стандартного ввода на стандартный вывод с помощью функций `fgets` и `fputs`

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("ошибка вывода");

    if (ferror(stdin))
        err_sys("ошибка ввода");

    exit(0);
}
```

Обратите внимание, что программы в листингах 5.1 и 5.2 не закрывают потоки ввода/вывода явно. Мы уже знаем, что функция `exit`бросит все незаписанные данные из буферов и закроет все открытые потоки. (Этот вопрос еще будет обсуждаться в разделе 8.5.) Интересно сравнить время, затраченное этими программами, с результатами из табл. 3.3. Данные для сравнения приводятся в табл. 5.4 (все операции производились с одним и тем же файлом размером 98,5 Мбайт, содержащим 3 миллиона строк).

Таблица 5.4. Время выполнения операций стандартными функциями ввода/вывода

Функция	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)	Размер программы (байты)
Лучшее время из табл. 3.3	0,01	0,54	8,51	
<code>fgets, fputs</code>	2,27	0,30	3,49	143
<code>getc, putc</code>	8,45	0,29	10,33	114
<code>fgetc, fputc</code>	8,16	0,40	10,18	114
Время из табл. 3.3, с размером буфера 1 байт	20,03	117,50	138,73	

Для каждой из трех версий, использующих стандартные функции ввода/вывода, пользовательское время получилось больше, чем наилучший результат из табл. 3.3, потому что две версии выполняют 100 миллионов циклов для переда-

чи данных по одному байту, а версия с построчным вводом/выводом выполняет 3 144 984 циклов. Версия программы, основанная на функции `read`, выполняет всего 25 244 цикла (для размера буфера 4096 байт). Различия общего времени выполнения обусловлены различиями в пользовательском времени и во времени ожидания завершения ввода/вывода, тогда как значения системного времени вполне сопоставимы.

Системное время примерно то же, что и прежде, потому что производится примерно одинаковое количество системных вызовов. Одно из преимуществ стандартной библиотеки ввода/вывода состоит в том, что она избавляет нас от беспокойства по поводу буферизации или оптимальности выбранного размера буфера. Конечно, мы все-таки должны определиться с максимальным размером строки для версии программы, которая использует функцию `fgets`, но это гораздо проще, чем выбирать оптимальный размер буфера.

В последней колонке табл. 5.4 приводится размер сегмента кода программы, сгенерированного компилятором языка C. Здесь мы видим, что версия с функциями `fgetc/fputc` имеет тот же размер, что и версия с функциями `getc/putc`. Обычно функции `getc` и `putc` реализованы в виде макросов, но в библиотеке GNU C эти макросы просто разворачиваются в вызовы функций.

Версия программы с построчным вводом/выводом выполняется почти в два раза быстрее, чем версии с посимвольным вводом/выводом. Если бы `fgets` и `fputs` были реализованы через функции `getc` и `putc` (раздел 7.7 [Kernighan and Ritchie, 1988]), результаты совпали бы с результатами версии, основанной на функции `getc`. На самом деле версия с построчным вводом/выводом выполнялась бы даже значительно дольше, так как в этом случае к существующим 6 миллионам вызовов функций добавились бы еще 200 миллионов. Из полученных результатов можно сделать вывод, что функции построчного ввода/вывода реализованы с помощью функции `memccpy(3)`. Часто для повышения эффективности функция `memccpy` пишется не на C, а на языке ассемблера.

И последнее, что представляет для нас интерес в этих результатах, — версия на основе функции `fgetc` выполняется намного быстрее, чем версия из листинга 3.1 с размером буфера `BUFFSIZE=1`. Обе версии производят одно и то же количество вызовов функций — приблизительно 200 миллионов, и все же на выполнение версии на основе функции `fgetc` потребовалось почти в 16 раз меньше пользовательского времени, а общее время выполнения получилось более чем в 39 раз меньше. Эта разница обусловлена тем, что версия на основе функции `open` выполняет 200 миллионов вызовов функций, которые, в свою очередь, производят 200 миллионов системных вызовов. Версия на основе функции `fgetc` также выполняет 200 миллионов вызовов функций, но производит всего 25 224 обращений к системным вызовам. Обращение к системному вызову обычно намного дороже, чем обращение к обычной функции.

Тут мы должны оговориться: результаты испытаний справедливы только для той системы, в которой они получены. Результаты зависят от многих особенностей конкретных реализаций UNIX. Тем не менее приводимые здесь числа и объяснения различий между версиями одной программы помогут нам лучше понять саму операционную систему. Сравнивая результаты из этого раздела и из раздела 3.9,

мы узнали, что стандартная библиотека ввода/вывода не намного медленнее, чем прямое обращение к функциям `read` и `write`. В большинстве сложных приложений наибольшее количество пользовательского времени уходит на выполнение самого приложения, а не на обращения к стандартным функциям ввода/вывода.

5.9. Ввод/вывод двоичных данных

Функции из раздела 5.6 оперируют одним символом, функции из раздела 5.7 — одной строкой. При выполнении операций ввода/вывода двоичных данных предпочтительнее читать или записывать сразу целые структуры. Чтобы сделать это с помощью `getc` или `putc`, нам пришлось выполнить обход полей структуры, выполняя чтение или запись по одному байту. Мы не можем воспользоваться функциями построчного ввода/вывода, поскольку функция `fputs` прекращает запись, встретив нулевой байт, а внутри структуры вполне могут содержаться нулевые байты. Точно так же и функция `fgets` не сможет корректно читать данные с нулевыми байтами или символами перевода строки. Поэтому для ввода/вывода двоичных данных предоставляются следующие две функции.

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);
```

Обе возвращают количество прочитанных или записанных объектов

Два наиболее распространенных случая использования этих функций:

1. Чтение или запись массивов двоичных данных. Например, вот как можно записать со 2-го по 5-й элементы массива чисел с плавающей точкой:

```
float data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("ошибка вызова функции fwrite");
```

Здесь мы передали в аргументе `size` размер одного элемента массива, а в аргументе `nobj` — количество элементов.

2. Чтение или запись структур данных. Вот как это делается:

```
struct {
    short count;
    long total;
    char name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("ошибка вызова функции fwrite");
```

Здесь в аргументе `size` указан размер структуры, а в аргументе `nobj` — количество объектов для записи (один).

Очевидное обобщение этих двух случаев — чтение или запись массива структур. Для этого аргумент `size` должен содержать размер структуры (определяемый с помощью оператора `sizeof`), а аргумент `nobj` — количество элементов массива.

Обе функции, `fread` и `fwrite`, возвращают количество прочитанных или записанных объектов. Для функции `fread` это число может быть меньше значения `nobj`, если произошла ошибка или достигнут конец файла. В этой ситуации нужно вызвать функцию `perror` или `feof`. Если функция `fwrite` вернула число меньше значения аргумента `nobj`, это свидетельствует об ошибке.

Фундаментальная проблема, связанная с вводом/выводом двоичных данных, заключается в том, что они корректно читаются только в той же системе, в какой были записаны. Много лет назад, когда все версии UNIX работали на PDP-11, этой проблемы не существовало, но сегодня стало нормой объединение в сети разнородных систем. И нередко возникает желание записать данные в одной системе и обработать их в другой. В такой ситуации эти две функции не будут работать по двум причинам.

1. Смещение полей структур может отличаться для разных компиляторов и операционных систем из-за различных требований выравнивания. Некоторые компиляторы могут упаковывать структуры в целях экономии занимаемого пространства и, возможно, в ущерб производительности либо, наоборот, выполнять выравнивание полей для повышения скорости доступа во время выполнения. Это означает, что даже для одной и той же системы раскладка структуры может варьироваться в зависимости от параметров компиляции.
2. Форматы представления многобайтных целых чисел или чисел с плавающей точкой могут различаться на разных аппаратных платформах.

Мы коснемся некоторых из этих проблем, когда будем говорить о сокетах в главе 16. Решение проблемы обмена двоичными данными между различными системами заключается в использовании высокогоревневого протокола. За описанием приемов, используемых сетевыми протоколами для обмена двоичными данными, обращайтесь к разделу 8.2 [Rago, 1993] или к разделу 5.18 [Stevens, Fenner, & Rudoff, 2004].

Мы еще вернемся к функции `fread` в разделе 8.14, когда с ее помощью будем читать двоичные структуры данных учетной информации о процессах.

5.10. Позиционирование в потоке

Существует три способа позиционирования в потоке ввода/вывода.

1. С помощью функций `ftell` и `fseek`, которые впервые появились в Version 7. Они предполагают, что позиция в файле может быть представлена в виде длинного целого.
2. С помощью функций `ftello` и `fseeko`. Они определены стандартом Single UNIX Specification для случаев, когда длинного целого недостаточно для представления позиции в файле. Вместо типа данных `long int` они используют `off_t`.

3. С помощью функций `fgetpos` и `fsetpos`. Они определены стандартом ISO C.

Для представления позиции в файле эти функции используют абстрактный тип данных `fpos_t`. Этот тип может быть увеличен настолько, насколько это необходимо для представления позиции в файле.

Переносимые приложения, которые предполагается портировать в операционные системы, отличные от UNIX, должны использовать функции `fgetpos` и `fsetpos`.

```
#include <stdio.h>
long ftell(FILE *fp);
```

Возвращает текущую позицию файла в случае успеха,
-1L — в случае ошибки

```
int fseek(FILE *fp, long offset, int whence);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

```
void rewind(FILE *fp);
```

Для двоичных файлов текущая позиция измеряется в байтах относительно начала файла. Функция `ftell` возвращает для двоичных файлов позицию данного байта. Чтобы установить позицию в двоичном файле с помощью функции `fseek`, нужно указать в аргументе `offset` смещение байта и как это смещение интерпретируется. Значение аргумента `whence` выбирается точно так же, как для функции `lseek` (раздел 3.6): `SEEK_SET` означает смещение от начала файла, `SEEK_CUR` — от текущей позиции файла и `SEEK_END` — от конца файла. Стандарт ISO C не требует, чтобы реализации поддерживали константу `SEEK_END` для двоичных файлов, поскольку некоторые системы требуют дополнения двоичных файлов в конце нулями, чтобы сделать размер файла кратным некоторому числу. Однако UNIX поддерживает использование константы `SEEK_END` для двоичных файлов.

В случае текстовых файлов текущая позиция может не соответствовать простому смещению байта. Опять же это главным образом относится к системам, отличным от UNIX, которые могут хранить текстовые данные в другом формате. Чтобы установить текущую позицию в текстовом файле, аргумент `whence` должен иметь значение `SEEK_SET`, а для аргумента `offset` допускаются только два значения — 0, что означает возврат к началу файла, или значение, полученное вызовом функции `ftell` для этого файла. Кроме того, вернуться в начало файла можно с помощью функции `rewind`.

Функция `ftello` практически идентична функции `ftell`, а функция `fseeko` — функции `fseek`, за исключением того, что тип смещения у них не `long`, а `off_t`.

```
#include <stdio.h>
off_t ftello(FILE *fp);
```

Возвращает текущую позицию файла в случае успеха,
(off_t)-1 — в случае ошибки

```
int fseeko(FILE *fp, off_t offset, int whence);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

О типе `off_t` мы говорили в разделе 3.6. Реализации могут определять тип `off_t` большего размера, чем 32-разрядное целое.

Как уже говорилось ранее, функции `fgetpos` и `fsetpos` определены стандартом ISO C.

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

Обе возвращают 0 в случае успеха, ненулевое значение — в случае ошибки

Функция `fgetpos` записывает значение текущей позиции в объект, на который указывает аргумент `pos`. Это значение может использоваться в последующих вызовах `fsetpos` для переустановки текущей позиции в файле.

5.11. Форматированный ввод/вывод

Форматированный вывод

Форматированный вывод производится с помощью пяти функций из семейства `printf`.

```
#include <stdio.h>

int printf(const char *restrict format, ...);
int fprintf(FILE *restrict fp, const char *restrict format, ...);
int dprintf(int fd, const char *restrict format, ...);
```

Все три возвращают количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int sprintf(char *restrict buf, const char *restrict format, ...);
```

Возвращает количество символов, записанных в массив, в случае успеха, отрицательное значение — в случае ошибки

```
int snprintf(char *restrict buf, size_t n, const char *restrict format, ...);
```

Возвращает количество символов, записанных в массив, если буфер имеет достаточный размер, отрицательное значение — в случае ошибки

Функция `printf` осуществляет запись в стандартный поток вывода, `fprintf` — в заданный поток, `dprintf` — в заданный файловый дескриптор, а `sprintf` помещает форматированную строку в массив `buf`. Функция `sprintf` автоматически добавляет нулевой байт в конец массива `buf`, но он не включается в возвращаемое значение.

Обратите внимание, что функция `sprintf` может вызвать переполнение буфера, на который указывает аргумент `buf`. Поэтомузывающая программа должна передавать ей буфер достаточного размера. Во избежание проблем, связанных с переполнением буфера, была добавлена функция `snprintf`. Эта функция принимает размер буфера в виде дополнительного аргумента и игнорирует символы, которые могли бы быть записаны за пределы буфера. Функция `snprintf` возвращает количество символов, которое было бы записано в буфер, если бы он имел достаточно большой размер. Как и в случае с функцией `sprintf`, возвращаемое значение не включает завершающий нулевой байт. Если `snprintf` возвращает положительное значение, меньшее, чем размер буфера `n`, это означает, что строка полностью записана в буфер и не была усечена. Если возникает ошибка, функция `snprintf` возвращает отрицательное значение.

Хотя функция `dprintf` не принимает указатель на файл, мы включили ее в раздел с описанием остальных, родственных ей функций форматированного вывода. Обратите внимание, что функция `dprintf` избавляет от необходимости вызывать `fdopen`, чтобы преобразовать дескриптор файла в файловый указатель для использования в вызове функции `fprintf`.

Спецификация формата управляет порядком интерпретации и в конечном счете отображением остальных аргументов. Каждый аргумент интерпретируется согласно спецификатору формата, который начинается с символа процента (%). Все символы строки формата, за исключением спецификаторов, копируются без изменений. Спецификатор формата включает четыре необязательных компонента, которые ниже показаны в квадратных скобках:

`%[flags][fldwidth][precision][lenmodifier]convtype`

Перечень возможных значений компонента `flags` приводится в табл. 5.5.

Таблица 5.5. Значения компонента `flags` строки спецификации формата

Флаг	Описание
'	Апостроф, вывод целых чисел с выделением групп десятичных разрядов
-	Выравнивание по левому краю поля
+	Всегда отображать знак числа
(пробел)	Выводить пробел, если отсутствует знак числа
#	Преобразовать в альтернативную форму (например, включить префикс 0x при выводе чисел в шестнадцатеричном формате)
0	Дополнять нулями слева вместо пробелов при выравнивании по правому краю

Компонент `fldwidth` определяет минимальную ширину поля для преобразования. Если в результате преобразования получено меньшее количество символов, они будут дополнены пробелами. Ширина поля выражается положительным целым числом или звездочкой (*).

Компонент `precision` определяет минимальное количество цифр для отображения целых чисел, минимальное количество цифр, расположенных правее десятичной точки, для чисел с плавающей точкой или максимальное количество символов для отображения строк. Компонент `precision` представляется в виде точки (.), за которой следует неотрицательное целое число или символ звездочки.

В полях `fldwidth` и `precision` можно указать звездочку. В этом случае значение компонента определяется целочисленным аргументом функции. Этот аргумент должен стоять непосредственно перед аргументом, значение которого будет подвергнуто преобразованию.

Компонент `lenmodifier` определяет размер аргумента. Возможные значения приведены в табл. 5.6.

Таблица 5.6. Значения компонента `lenmodifier` строки спецификации формата

Модификатор длины	Описание
<code>hh</code>	<code>signed char</code> или <code>unsigned char</code>
<code>h</code>	<code>signed short</code> или <code>unsigned short</code>
<code>l</code>	<code>signed long</code> , <code>unsigned long</code> или многобайтный символ
<code>ll</code>	<code>signed long long</code> или <code>unsigned long long</code>
<code>j</code>	<code>intmax_t</code> или <code>uintmax_t</code>
<code>z</code>	<code>size_t</code>
<code>t</code>	<code>ptrdiff_t</code>
<code>L</code>	<code>long double</code>

Компонент `convtype` является обязательным. Он управляет интерпретацией аргумента. Различные виды преобразований приведены в табл. 5.7.

Обычно спецификаторы применяются к аргументам в порядке их следования за аргументом `format`. Однако существует альтернативный синтаксис спецификаторов, позволяющий явно определять аргументы, к которым они относятся: последовательность `%n$` представляет *n*-й аргумент. Но имейте в виду, что эти две формы записи спецификаторов нельзя смешивать в одной строке формата. Нумерация аргументов при использовании альтернативного синтаксиса определения формата начинается с единицы. Если значение для какого-либо из полей, `fldwidth` или `precision`, определяется дополнительным аргументом, форма записи со звездочкой должна иметь вид: `*m$`, где *m* — порядковый номер аргумента с требуемым значением.

Следующие пять разновидностей `printf` очень похожи на предыдущие, но в них список аргументов переменной длины (...) заменен аргументом *arg*.

Таблица 5.7. Значения компонента convtype строки спецификации формата

Спецификатор	Описание
d, i	Десятичное число со знаком
o	Восьмеричное число без знака
u	Десятичное число без знака
x, X	Шестнадцатеричное число без знака
f, F	Число с плавающей точкой двойной точности
e, E	Число с плавающей точкой двойной точности, в экспоненциальной форме
g, G	Интерпретируется как f, F, e или E, в зависимости от значения интерпретируемого аргумента
a, A	Число с плавающей точкой двойной точности в шестнадцатеричной экспоненциальной форме
c	Символ (с модификатором длины 1 — многобайтный символ)
s	Строка (с модификатором длины 1 — строка многобайтных символов)
p	Указатель типа void
n	Указатель на целое со знаком, куда записывается количество уже выведенных символов
%	Символ %
C	Многобайтный символ (расширение XSI, эквивалент 1c)
S	Строка многобайтных символов (расширение XSI, эквивалент 1s)

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list arg);
int vfprintf(FILE *restrict fp, const char *restrict format,
             va_list arg);
int vdprintf(int fd, const char *restrict format, va_list arg);
```

Все три возвращают количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int vsprintf(char *restrict buf, const char *restrict format,
             va_list arg);
```

Возвращает количество символов, записанных в массив, в случае успеха, отрицательное значение — в случае ошибки

```
int vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);
```

Возвращает количество символов, записанных в массив, если буфер имеет достаточный размер, отрицательное значение — в случае ошибки

Мы использовали функцию `vsnprintf` в функциях вывода сообщений об ошибках (приложение В).

Описание особенностей обработки списков аргументов переменной длины в соответствии со стандартом ISO C вы найдете в разделе 7.3 [Kernighan and Ritchie, 1988]. Следует помнить, что средства обработки списков аргументов переменной длины, определяемые стандартом ISO C, — заголовочный файл `<stdarg.h>` и связанные с ним функции — отличаются от функций `<varargs.h>`, которые представлялись старыми версиями UNIX.

Форматированный ввод

Форматированный ввод выполняется с помощью трех функций из семейства `scanf`.

```
#include <stdio.h>

int scanf(const char *restrict format, ...);
int fscanf(FILE *restrict fp, const char *restrict format, ...);
int sscanf(const char *restrict buf, const char *restrict format, ...);
```

Все три возвращают количество введенных элементов или EOF, если
возникла ошибка ввода или перед каким-либо преобразованием
достигнут конец файла

Функции семейства `scanf` используются для анализа входной строки и преобразования последовательностей символов в переменные указанных типов. Аргументы, следующие за строкой формата, содержат адреса переменных, куда будут записаны результаты преобразований.

Спецификация формата управляет порядком преобразования. Символ процента (%) обозначает начало спецификатора формата. Все символы в строке формата, за исключением спецификаторов формата и пробелов, должны совпадать с вводимыми символами. Если обнаружится какое-либо несоответствие, обработка ввода остановится и остальная часть входной строки остается непрочитанной.

Спецификатор формата включает четыре необязательных компонента, которые ниже показаны в квадратных скобках:

`%[*][fldwidth][m][lenmodifier]convtype`

Необязательный первый компонент с символом звездочки (*) используется для подавления преобразования. Вводимая строка преобразуется согласно указанному формату, но результат преобразования не сохраняется.

Компонент `fldwidth` определяет максимальную ширину поля в символах. Компонент `lenmodifier` — размер аргумента, куда будет записан результат преобразования. Семейство функций `scanf` поддерживает те же модификаторы длины (`lenmodifier`), что и семейство функций `printf` (табл. 5.6).

Компонент `convtype` подобен соответствующему компоненту спецификатора формата функции `printf`, но между ними есть и некоторые отличия. Одно из них состоит в том, что результат преобразования, который сохраняется как беззнаковый тип, может на входе быть числом со знаком. Например, число `-1` будет преобразовано в `4 294 967 295` и записано в переменную беззнакового типа. В табл. 5.8 перечисляются типы преобразований, которые поддерживаются функциями семейства `scanf`.

Таблица 5.8. Значения компонента `convtype` строки спецификации формата

Спецификатор	Описание
<code>d</code>	Десятичное число со знаком, с основанием 10
<code>i</code>	Десятичное число со знаком, основание определяется форматом ввода
<code>o</code>	Восьмеричное число без знака (на входе может быть со знаком)
<code>u</code>	Десятичное число без знака, с основанием 10 (на входе может быть со знаком)
<code>x</code>	Шестнадцатеричное число без знака (на входе может быть со знаком)
<code>a, A, e, E, f, F, g, G</code>	Число с плавающей точкой
<code>c</code>	Символ (с модификатором длины 1 — многобайтный символ)
<code>s</code>	Строка (с модификатором длины 1 — строка многобайтных символов)
<code>[</code>	Начинает последовательность, состоящую только из указанных символов, ограниченную символом <code>]</code>
<code>[^</code>	Начинает последовательность любых символов, кроме указанных, ограниченную символом <code>]</code>
<code>p</code>	Указатель типа <code>void</code>
<code>n</code>	Указатель на целое со знаком, куда записывается количество уже выведенных символов
<code>%</code>	Символ <code>%</code>
<code>C</code>	Многобайтный символ (расширение XSI, эквивалент <code>1c</code>)
<code>S</code>	Строка многобайтных символов (расширение XSI, эквивалент <code>1s</code>)

Необязательный символ `m` между полями `fldwidth` и `lenmodifier` называется *символом выделения памяти при присваивании*. Его можно использовать со спецификаторами `%c`, `%s` и `%[`, чтобы обеспечить выделение памяти для строки, полученной в результате преобразования. В этом случае соответствующий аргумент должен быть адресом указателя, куда будет скопирован адрес выделенного буфера. В случае успешного вызова вызывающий код должен по окончании использования буфера освободить выделенную таким способом память вызовом функции `free`.

Семейство функций `scanf` также поддерживает альтернативный синтаксис спецификаторов формата, позволяющий явно определять аргументы, к которым они относятся: последовательность `%n$` представляет *n-й аргумент*. Как и в функции

ях семейства `printf`, на один и тот же аргумент можно неоднократно ссылаться в строке формата. Однако стандарт Single UNIX Specification отмечает, что поведение функций семейства `scanf` в этом случае не определено.

Аналогично семейству `printf` семейство `scanf` также включает функции, поддерживающие передачу списка аргументов в виде переменной, как это определено в заголовочном файле `<stdarg.h>`.

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);
int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);
int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

Все три возвращают количество введенных элементов или EOF, если возникла ошибка ввода или перед каким-либо преобразованием достигнут конец файла

За дополнительной информацией о функциях семейства `scanf` обращайтесь к справочному руководству вашей системы UNIX.

5.12. Подробности реализации

Как уже упоминалось, функции стандартной библиотеки ввода/вывода в конечном итоге обращаются к функциям, описанным в главе 3. Каждому потоку ввода/вывода соответствует дескриптор файла, который можно получить, обратившись к функции `fileno`.

Обратите внимание, что функция `fileno` не определена стандартом ISO C – это расширение, поддерживаемое стандартом POSIX.1.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Возвращает дескриптор файла, ассоциированный с потоком

Эта функция необходима, например, для вызова функции `dup` или `fcntl`.

Чтобы увидеть, как реализована стандартная библиотека ввода/вывода в вашей системе, начните с заголовочного файла `<stdio.h>`. Здесь вы найдете определение объекта `FILE`, флагов потока и всех стандартных функций ввода/вывода, таких как `getc`, которые определены как макросы. В разделе 8.5 [Kernighan and Ritchie, 1988] приводится пример, демонстрирующий особенности большинства реализаций в UNIX. В главе 12 [Plauger, 1992] вы найдете полные исходные тексты одной

из реализаций стандартной библиотеки ввода/вывода. Кроме того, в свободном доступе имеется реализация стандартной библиотеки ввода/вывода GNU.

Пример

Программа в листинге 5.3 демонстрирует особенности буферизации для всех трех стандартных потоков ввода/вывода и для потока, ассоциированного с обычным файлом.

Листинг 5.3. Вывод сведений о буферизации для различных потоков ввода/вывода

```
#include "apue.h"

void    pr_stdio(const char *, FILE *);
int     is_unbuffered(FILE *);
int     is_linebuffered(FILE *);
int     buffer_size(FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("введите любой символ\n", stdout);
    if (getchar() == EOF)
        err_sys("ошибка вызова функции getchar");
    fputs("эта строка выводится в стандартный вывод сообщений об ошибках\n",
         stderr);

    pr_stdio("stdin", stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ((fp = fopen("/etc/passwd", "r")) == NULL)
        err_sys("ошибка вызова функции fopen");
    if (getc(fp) == EOF)
        err_sys("ошибка вызова функции getc");
    pr_stdio("/etc/passwd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("поток = %s, ", name);
    if (is_unbuffered(fp))
        printf("буферизация отсутствует");
    else if (is_linebuffered(fp))
        printf("построчная буферизация");
    else /* ни один из вышеперечисленных режимов */
        printf("полная буферизация");
    printf(", размер буфера = %d\n", buffer_size(fp));
}

/*
 * Следующий код непереносим.
 */
#endif defined(_IO_UNBUFFERED)
```

```
int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & _IO_UNBUFFERED);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & _IO_LINE_BUF);
}

int
buffer_size(FILE *fp)
{
    return(fp->_IO_buf_end - fp->_IO_buf_base);
}

#if defined(__SNBF)

int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & __SNBF);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & __SLBF);
}

int
buffer_size(FILE *fp)
{
    return(fp->bf._size);
}

#elif defined(_IONBF)

#endif _LP64
#define _flag __pad[4]
#define _ptr __pad[1]
#define _base __pad[2]
#endif

int
is_unbuffered(FILE *fp)
{
    return(fp->_flag & _IONBF);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flag & _IOLBF);
}

int
```

```
buffer_size(FILE *fp)
{
#ifndef _LP64
    return(fp->_base - fp->_ptr);
#else
    return(BUFSIZ); /* простая попытка угадать */
#endif
}

#else

#error неизвестная реализация stdio!

#endif
```

Обратите внимание: для каждого потока, перед выводом сведений о буферизации, мы выполняем операцию ввода/вывода, поскольку размещение буферов в памяти производится обычно во время первой операции ввода/вывода. Поля структуры и константы, используемые в этом примере, определены в реализациях стандартной библиотеки ввода/вывода во всех четырех plataформах, описываемых в этой книге. Помните, что реализации стандартной библиотеки ввода/вывода могут различаться в разных версиях UNIX, и программы, такие как в данном примере, являются непереносимыми, потому что в них используются особенности конкретных реализаций.

Запустив программу дважды: один раз, когда все три стандартных потока ввода/вывода были связаны с терминалом, и второй раз, когда они были перенаправлены в файлы, — мы получили следующие результаты:

```
$ ./a.out          stdin, stdout и stderr связаны с терминалом
введите любой символ
здесь мы ввели символ перевода строки
эта строка выводится в стандартный вывод сообщений об ошибках
поток = stdin, построчная буферизация, размер буфера = 1024
поток = stdout, построчная буферизация, размер буфера = 1024
поток = stderr, буферизация отсутствует, размер буфера = 1
поток = /etc/motd, полная буферизация, размер буфера = 4096
$ ./a.out < /etc/group > std.out 2> std.err
запустим еще раз с перенаправлением
трех стандартных потоков в файлы
$ cat std.err
эта строка выводится в стандартный вывод сообщений об ошибках
$ cat std.out
введите любой символ
поток = stdin, полная буферизация, размер буфера = 4096
поток = stdout, полная буферизация, размер буфера = 4096
поток = stderr, буферизация отсутствует, размер буфера = 1
поток = /etc/motd, полная буферизация, размер буфера = 4096
```

Как видите, в данной системе стандартные потоки ввода и вывода буферизуются построчно, когда они связаны с терминалом. Размер буфера в этом случае составляет 1024 байта. Обратите внимание, что это не ограничивает размер вводимых и выводимых строк 1024 байтами — это лишь размер буфера. Для записи строки длиной 2048 байт в стандартный поток вывода потребуется два обращения к системному вызову `write`. Когда эти два потока перенаправляются в обычные файлы, они при-

обретают режим полной буферизации с размером буфера, равным предпочтительному размеру блока ввода/вывода (значение `st_blksize` структуры `stat`) для данной файловой системы. Также можно видеть, что стандартный поток сообщений об ошибках не буферизуется ни в одном из случаев, как это и должно быть, и что по умолчанию для обычных файлов назначается режим полной буферизации.

5.13. Временные файлы

Стандарт ISO C определяет две вспомогательные функции для создания временных файлов.

```
#include <stdio.h>
char *tmpnam(char *ptr);
```

Возвращает указатель на строку с уникальным именем файла

```
FILE *tmpfile(void);
```

Возвращает указатель на файл в случае успеха,
NULL — в случае ошибки

Функция `tmpnam` генерирует строку с уникальным полным именем файла, не существующего в данный момент в системе. При каждом вызове эта функция генерирует неповторяющиеся имена файлов до `TMP_MAX` раз. Константа `TMP_MAX` определена в файле `<stdio.h>`.

Стандарт ISO C определяет эту константу и требует, чтобы ее значение было не меньше 25. Стандарт Single UNIX Specification требует, чтобы XSI-совместимые системы поддерживали константу `TMP_MAX` со значением не меньше 10 000. Это минимальное значение позволяет использовать четыре цифры (0000–9999) для создания уникальных имен файлов, но большинство реализаций UNIX используют для этих целей алфавитно-цифровые символы.

В SUSv4 функция `tmpnam` отмечена как устаревшая, но стандарт ISO C продолжает поддерживать ее.

Если аргумент `ptr` содержит значение `NULL`, генерируемое имя файла сохраняется в статической области памяти и функция возвращает указатель на эту область. Последующие вызовы функции `tmpnam` могут затереть эту область памяти. (То есть если требуется вызвать функцию `tmpnam` более одного раза и каждый раз сохранять имя временного файла, мы должны копировать сами строки, а не указатели на них.) Если в аргументе `ptr` передается непустой указатель, предполагается, что он содержит адрес буфера размером не менее `L_tmpnam` символов. (Константа `L_tmpnam` определена в файле `<stdio.h>`.) Сгенерированное имя файла сохраняется в этом буфере, и функция возвращает значение указателя `ptr`.

Функция `tmpfile` создает временный двоичный файл (`wb+`), который автоматически удаляется при его закрытии или по завершении процесса. В UNIX не имеет никакого значения тот факт, что файл двоичный.

Пример

Программа в листинге 5.4 демонстрирует работу с обеими функциями.

Листинг 5.4. Демонстрация функций tmpnam и tmpfile

```
#include "apue.h"

int
main(void)
{
    char      name[L_tmpnam], line[MAXLINE];
    FILE     *fp;

    printf("%s\n", tmpnam(NULL));      /* первое имя временного файла */
    tmpnam(name);                    /* второе имя временного файла */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)      /* создать временный файл */
        err_sys("ошибка вызова функции tmpfile");
    fputs("записанная строка\n", fp); /* записать во временный файл */
    rewind(fp);                      /* затем прочитать */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("ошибка вызова функции fgets");
    fputs(line, stdout);             /* и вывести прочитанную строку */

    exit(0);
}
```

Запустив программу из листинга 5.4, мы получили следующие результаты:

```
$ ./a.out
/tmp/fileT0Hsu6
/tmp/filekmAsYQ
записанная строка
```

Как правило, функция `tmpfile` использует следующий алгоритм: сначала вызовом `tmpnam` создается уникальное имя файла, затем создается сам файл, для которого сразу же вызывается функция `unlink`. В разделе 4.15 упоминалось, что вызов функции `unlink` не приводит к немедленному удалению файла. Он будет удален системой автоматически в момент закрытия или по завершении процесса.

Стандарт Single UNIX Specification определяет в качестве расширений XSI две дополнительные функции для работы с временными файлами, `mkdtemp` и `mkstemp`.

Прежние версии стандарта Single UNIX Specification определяли функцию `tmpnam` как способ создания временных файлов в каталоге, указанном вызывающей программой. В версии SUSv4 она отмечена как устаревшая.

```
#include <stdlib.h>
char *mkdtemp(char *template);
```

Возвращает указатель на имя каталога в случае успеха,
NULL – в случае ошибки

```
int mkstemp(char *template);
```

Возвращает дескриптор файла в случае успеха, -1 – в случае ошибки

Функция `mkdtemp` создает каталог с уникальным именем, а функция `mkstemp` — обычный файл с уникальным именем. Имя файла генерируется с использованием строки шаблона *template*. Эта строка должна быть строкой пути, в которой последние шесть символов должны быть `xxxxxx`. Функция заменит эти символы другими, чтобы обеспечить уникальность имени. В случае успеха эти функции изменяют строку *template*, сохраняя в ней имя временного файла.

Функция `mkdtemp` создает каталог со следующими установленными битами прав доступа: `S_IRUSR | S_IWUSR | S_IXUSR`. Имейте в виду, что эти права могут быть ограничены маской `umask` вызывающего процесса. В случае успеха `mkdtemp` возвращает имя нового каталога.

Функция `mkstemp` создает обычный файл с уникальным именем и открывает его. Файловый дескриптор, возвращаемый функцией `mkstemp`, открыт для чтения и для записи. Файлы создаются со следующими установленными битами прав доступа: `S_IRUSR | S_IWUSR`.

В отличие от `tmpfile`, функция `mkstemp` создает временные файлы, которые не удаляются автоматически. Если это необходимо, следует вручную вызвать функцию `unlink`.

Применение функций `tmpnam` и `tempnam` имеет как минимум один недостаток: между моментом, когда приложение получит уникальное имя файла, и моментом, когда оно создаст файл с этим именем, проходит некоторое время. В этот промежуток другой процесс может создать файл с тем же именем. Вместо них следует использовать функции `tmpfile` и `mkstemp`, так как они не порождают такую ошибку.

Пример

Программа в листинге 5.5 демонстрирует, как следует (и как не следует) использовать функцию `mkstemp`.

Листинг 5.5. Использование функции `mkstemp`

```
#include "apue.h"
#include <errno.h>

void make_temp(char *template);

int
main()
{
    char    good_template[] = "/tmp/dirXXXXXX"; /* правильно */
    char    *bad_template = "/tmp/dirXXXXXX";    /* неправильно */

    printf("попытка создать первый временный файл...\n");
    make_temp(good_template);
    printf("попытка создать второй временный файл...\n");
    make_temp(bad_template);
    exit(0);
}

void
make_temp(char *template)
```

```
{  
    int          fd;  
    struct stat sbuf;  
  
    if ((fd = mkstemp(template)) < 0)  
        err_sys("ошибка создания временного файла");  
    printf("temp name = %s\n", template);  
    close(fd);  
    if (stat(template, &sbuf) < 0) {  
        if (errno == ENOENT)  
            printf("файл отсутствует\n");  
        else  
            err_sys("ошибка вызова stat");  
    } else {  
        printf("файл существует\n");  
        unlink(template);  
    }  
}
```

Запустив эту программу, мы получили следующие результаты:

```
$ ./a.out  
попытка создать первый временный файл...  
temp name = /tmp/dirUmBT7h  
файл существует  
попытка создать второй временный файл...  
Segmentation fault
```

Разница в поведении обусловлена способом объявления двух строк шаблонов. Первая строка шаблона размещается в стеке, потому что используется переменная-массив. Во втором случае используется указатель, и только этот указатель размещается в стеке, сама же строка размещается компилятором в сегменте кода, доступном только для чтения. Когда `mkstemp` пытается изменить строку, возникает ошибка доступа к памяти.

5.14. Потоки ввода/вывода в памяти

Как мы уже знаем, стандартная библиотека ввода/вывода буферизует данные в памяти, что увеличивает эффективность операций посимвольного и построчного ввода/вывода. Мы также знаем, что с помощью функций `setbuf` и `setvbuf` можно передать библиотеке собственные буфера. В версии 4 стандарта Single UNIX Specification добавлена поддержка *потоков ввода/вывода в памяти* (*memory streams*). Это стандартные потоки ввода/вывода, которые не связаны с файлами, хотя доступ к ним также осуществляется с применением файловых указателей `FILE`. Все операции ввода/вывода с такими потоками заключаются в передаче байтов в буфера, находящиеся в памяти, и обратно. Как будет показано ниже, даже при том, что эти потоки напоминают файлы, они имеют некоторые особенности, которые делают их более подходящими для работы со строками символов.

Создать поток ввода/вывода в памяти можно с помощью одной из трех функций. Первая из них — `fmemopen`.

```
#include <stdio.h>

FILE *fmemopen(void *restrict buf, size_t size,
               const char *restrict type);
```

Возвращает указатель потока в случае успеха,
NULL — в случае ошибки

Функция `fmemopen` позволяет вызывающей программе передать буфер для использования в качестве потока ввода/вывода в памяти: аргумент `buf` указывает на начало буфера, а аргумент `size` определяет размер буфера в байтах. Если в `buf` передать пустой указатель, функция `fmemopen` выделит буфер размером `size` байтов. В этом случае буфер будет автоматически освобожден при закрытии потока.

Аргумент `type` определяет способ использования потока. Возможные значения аргумента `type` перечислены в табл. 5.9.

Таблица 5.9. Возможные значения аргумента `type` при открытии потока в памяти

type	Описание
r или rb	Открыть для чтения
w или wb	Открыть для записи
a или ab	Открыть для записи в конец, с первого нулевого байта
r+, или r+b, или rb+	Открыть для чтения и для записи
w+, или w+b, или wb+	Усечь размер до 0 или создать и открыть для чтения и для записи
a+, или a+b, или ab+	Открыть для записи в конец, открыть для чтения и для записи с первого нулевого байта

Обратите внимание, что эти значения аналогичны значениям аргумента `type` в функциях открытия стандартных файловых потоков ввода/вывода, но имеют некоторые тонкие отличия. Во-первых, когда поток в памяти открывается для записи в конец, текущая позиция в файле устанавливается в позицию первого нулевого байта в буфере. Если буфер не содержит нулевых байтов, текущей становится позиция предпоследнего байта в буфере. Если поток в памяти открывается не для записи в конец, текущая позиция устанавливается в начало буфера. Поскольку в режиме открытия для записи в конец текущая позиция определяется позицией первого нулевого байта, потоки ввода/вывода в памяти не очень хорошо подходят для хранения двоичных данных (которые могут содержать нулевые байты внутри фактических данных).

Во-вторых, если в аргументе `buf` передается пустой указатель, бессмысленно открывать поток только для чтения или только для записи. Поскольку в этом случае буфер создается самой функцией `fmemopen`, нет никакой возможности получить адрес буфера, поэтому, открывая таким способом поток только для записи, вы никогда не сможете прочитать из него то, что было записано. Аналогично, открывая поток только для чтения, вы сможете только читать содержимое буфера, в который никогда не сможете ничего записать.

В-третьих, когда после записи в поток вызывается функция `fclose`, `fflush`, `fseek`, `fseeko` или `fsetpos`, в текущую позицию всегда записывается нулевой байт.

Пример

Будет очень поучительно взглянуть, как операция записи в поток в памяти воздействует на буфер. Программа в листинге 5.6 заполняет буфер определенным шаблоном, позволяя посмотреть, как действует операция записи в поток.

Листинг 5.6. Исследование особенностей поведения операции записи в поток

```
#include "apue.h"

#define BSZ 48

int
main()
{
    FILE *fp;
    char buf[BSZ];

    memset(buf, 'a', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    if ((fp = fmemopen(buf, BSZ, "w+")) == NULL)
        err_sys("ошибка вызова fmemopen");
    printf("первоначально буфер содержит: %s\n", buf);
    fprintf(fp, "привет, мир");
    printf("перед вызовом flush: %s\n", buf);
    fflush(fp);
    printf("после вызова fflush: %s\n", buf);
    printf("длина строки в буфере = %ld\n", (long)strlen(buf));

    memset(buf, 'b', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "привет, мир");
    fseek(fp, 0, SEEK_SET);
    printf("после вызова fseek: %s\n", buf);
    printf("длина строки в буфере = %ld\n", (long)strlen(buf));

    memset(buf, 'c', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "привет, мир");
    fclose(fp);
    printf("после вызова fclose: %s\n", buf);
    printf("длина строки в буфере = %ld\n", (long)strlen(buf));

    return(0);
}
```

Запустив программу в Linux, мы получили следующее:

```
$ ./a.out
буфер заполняется символами 'a'
первоначально буфер содержит:   fmemopen поместила нулевой байт в начало буфера
перед вызовом flush:           буфер не изменится, пока поток не будет сброшен
```

```
после вызова fflush: привет, мир
длина строки в буфере = 11      в конец строки добавлен нулевой байт
                                  теперь буфер заполнен символами 'b'
после вызова fseek: bbbbbbbbbbprivet, мир      fseek вызывает сброс
длина строки в буфере = 22      снова добавлен нулевой байт
                                  теперь буфер заполнен символами 'c'
после вызова fclose: привет, мирccccccccccccccccccccccccccc
длина строки в буфере = 46      нулевой байт не добавляется
```

Этот пример наглядно демонстрирует, как при сбросе потока добавляются нулевые байты. Нулевой байт добавляется автоматически всякий раз, когда производится запись в поток, для поддержки представления о размере содержимого потока (в противоположность размеру буфера, который имеет фиксированное значение). Размер содержимого потока позволяет определить, как много данных записано в него.

Из всех четырех платформ, описываемых в книге, только Linux 3.2.0 поддерживает потоки в памяти. Это показывает, что не все реализации успевают за последними стандартами, но должны будут изменяться со временем.

Двумя другими функциями, с помощью которых можно создавать потоки в памяти, являются `open_memstream` и `open_wmemstream`.

```
#include <stdio.h>

FILE *open_memstream(char **bufp, size_t *sizep);

#include <wchar.h>

FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

Обе возвращают указатель потока в случае успеха,
NULL — в случае ошибки

Функция `open_memstream` создает поток для работы с данными на уровне байтов, а `open_wmemstream` — для работы с данными на уровне многобайтных символов (обсуждение многобайтных символов можно найти в разделе 5.2). Эти две функции имеют следующие отличия от `fmemopen`:

- Создаваемый поток открывается только для записи.
- Функциям нельзя передать свой буфер, но можно получить доступ к буферу и его размеру через аргументы `bufp` и `sizep` соответственно.
- После закрытия потока буфер должен освобождаться вручную.
- Буфер будет расти в размерах по мере добавления новых байтов в поток.

При использовании значений, определяющих адрес и размер буфера, необходимо следовать определенным правилам. Во-первых, адрес и длина буфера могут использоваться только после вызова `fclose` или `fflush`. Во-вторых, эти значения остаются действительными только до следующей операции записи в поток или вызова `fclose` или `fflush`. Поскольку буфер автоматически увеличивается в размерах, память для него может перераспределиться. В этом случае можно обнаружить, что после вызова `fclose` или `fflush` значение адреса буфера в памяти изменилось.

Потоки в памяти хорошо подходят для создания строк, потому что не подвержены ошибке переполнения буфера. Они могут также способствовать увеличению производительности функций, принимающих стандартные потоки ввода/вывода для хранения временных данных, потому что хранятся в оперативной памяти, а не в файлах на диске.

5.15. Альтернативы стандартной библиотеке ввода/вывода

Стандартная библиотека ввода/вывода несовершенна. В книге [Korn and Vo, 1991] перечисляются ее многочисленные недостатки, часть которых присуща базовой архитектуре, но большая часть связана с различными аспектами реализации.

Один из врожденных недостатков стандартной библиотеки ввода/вывода, снижающий ее эффективность, заключается в большом количестве операций копирования данных. При использовании функций построчного ввода/вывода, `fgets` и `fputs`, данные обычно копируются дважды: один раз — между ядром и буфером ввода/вывода (то есть при вызове функций `read` и `write`) и второй раз — между буфером ввода/вывода и строкой. Библиотека скоростного ввода/вывода (`fio(3)` в [AT&T, 1990a]) обходит этот недостаток за счет того, что функция, которая читает строку, возвращает указатель на нее, а не копирует строку в другой буфер. В [Hume, 1988] говорится, что таким образом можно в три раза увеличить скорость работы утилиты `grep(1)`.

В [Korn and Vo, 1991] описывается другая альтернатива стандартной библиотеке ввода/вывода — `sfio`. Этот пакет почти не уступает в скорости библиотеке `fio` и обычно гораздо эффективнее стандартной библиотеки ввода/вывода. Кроме того, пакет `sfio` реализует некоторые функциональные возможности, недоступные в других библиотеках: потоки ввода/вывода более универсальны и могут представлять как файлы, так и области памяти; операции с потоками ввода/вывода можно изменять за счет подключения дополнительных модулей; улучшена обработка исключительных ситуаций.

В [Krieger, Stumm, and Unrau, 1992] рассматривается другая альтернатива — с использованием файлов, отображаемых в память с помощью функции `mmap` (она описана в разделе 14.8). Этот пакет называется ASI, Alloc Stream Interface (интерфейс размещения потоков). Программный интерфейс очень напоминает функции распределения памяти, используемые в UNIX (`malloc`, `realloc` и `free`, которые мы рассмотрим в разделе 7.8). Как и пакет `sfio`, ASI старается минимизировать количество операций копирования за счет использования указателей.

Существуют реализации стандартной библиотеки ввода/вывода, спроектированные для систем с небольшими объемами памяти, таких как встраиваемые устройства. Скромное потребление памяти — их сильная сторона, в то время как переносимости, скорости или функциональным возможностям в них уделяется меньше внимания. В качестве примеров таких реализаций можно назвать библиотеку `uClibc` (за подробной информацией обращайтесь по адресу <http://www.uclibc.org>) и библиотеку `Newlib` (<http://sources.redhat.com/newlib>).

5.16. Подведение итогов

Стандартная библиотека ввода/вывода используется в большинстве приложений UNIX. Мы рассмотрели все функции, предоставляемые этой библиотекой, а также некоторые особенности ее реализации и вопросы эффективности. Самое главное — никогда не забывайте о буферизации, поскольку именно в связи с ней возникает больше всего проблем и недопонимания.

Упражнения

- 5.1 Напишите реализацию функции `setbuf` с использованием функции `setvbuf`.
- 5.2 Измените программу в листинге 5.2, которая копирует файл с помощью функций построчного ввода/вывода (`fgets` и `fputs`), так, чтобы вместо константы `MAXLINE` использовалось значение 4. Что произойдет, если попытаться скопировать файл, в котором длина строк превышает 4 байта? Объясните почему.
- 5.3 О чём говорит значение 0, возвращаемое функцией `printf`?
- 5.4 Следующий код корректно работает в одних системах и некорректно в других. В чём причина такого поведения?

```
#include <stdio.h>

int
main(void)
{
    char      c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

- 5.5 Как можно использовать функцию `fsync` (раздел 3.13) с потоками ввода/вывода?
- 5.6 В программах из листингов 1.5 и 1.8 строка приглашения не содержит символа перевода строки, и мы не вызываем функцию `fflush`. Почему тогда она выводится на экран?
- 5.7 В BSD-системах имеется функция `funopen`, позволяющая перехватывать операции чтения, записи, позиционирования и закрытия потоков. На основе этой функции реализуйте функцию `fmemopen` для FreeBSD и Mac OS X.

6

Информация о системе и файлы данных

6.1. Введение

Нормальное функционирование любой UNIX-системы требует наличия различных файлов данных. Файл паролей `/etc/passwd` и файл групп `/etc/group` очень часто используются в самых разных приложениях. Например, обращение к файлу паролей происходит всякий раз, когда пользователь входит в систему или когда кто-либо выполняет команду `ls -l`.

Исторически эти файлы содержат простой текст в формате ASCII и их можно прочитать с помощью стандартной библиотеки ввода/вывода. Но в больших системах последовательный просмотр записей в файле паролей может требовать довольно много времени. Имело бы смысл хранить эти данные в формате, отличном от ASCII, но при этом необходим интерфейс для прикладных программ, который мог бы работать с любым форматом. Переносимые интерфейсы для доступа к этим файлам — тема данной главы. Кроме того, мы рассмотрим функции для получения информации о системе и функции даты и времени.

6.2. Файл паролей

Файл паролей в UNIX, который стандарт POSIX.1 называет базой данных пользователей, содержит поля, перечисленные в табл. 6.1. Эти поля также имеются в структуре `passwd`, которая определена в заголовочном файле `<pwd.h>`.

Обратите внимание, что стандарт POSIX.1 определяет только пять из десяти полей структуры `passwd`. Большинство платформ поддерживают как минимум семь полей. Системы, производные от BSD, поддерживают все десять.

Традиционно файл паролей хранится под именем `/etc/passwd` и представляет собой обычный текстовый файл в формате ASCII. Каждая строка файла состоит из полей (см. табл. 6.1), разделенных двоеточием. Для примера приведем четыре строки из файла `/etc/passwd` в Linux:

```
root:x:0:0:root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Таблица 6.1. Поля в файле /etc/passwd

Описание	Поле структуры passwd	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Имя пользователя	<code>char *pw_name</code>	✓	✓	✓	✓	✓
Зашифрованный пароль	<code>char *pw_passwd</code>		✓	✓	✓	✓
Числовой идентификатор пользователя	<code>uid_t pw_uid</code>	✓	✓	✓	✓	✓
Числовой идентификатор группы	<code>gid_t pw_gid</code>	✓	✓	✓	✓	✓
Поле комментария	<code>char *pw_gecos</code>		✓	✓	✓	✓
Начальный рабочий каталог	<code>char *pw_dir</code>	✓	✓	✓	✓	✓
Командный интерпретатор	<code>char *pw_shell</code>	✓	✓	✓	✓	✓
Класс доступа пользователя	<code>char *pw_class</code>		✓		✓	
Время следующего изменения пароля	<code>time_t pw_change</code>		✓		✓	
Время истечения действия учетной записи	<code>time_t pw_expire</code>		✓		✓	

Обратите внимание на следующие обстоятельства.

- Как правило, в файле паролей имеется запись для пользователя с именем `root`. Этот пользователь имеет числовой идентификатор 0 (суперпользователь).
- Поле зашифрованного пароля содержит единственный символ-заполнитель. Старые версии UNIX хранили в этом поле зашифрованный пароль. Поскольку хранение даже зашифрованных паролей в файле, доступном для чтения всем пользователям, представляет угрозу безопасности, современные системы хранят зашифрованные пароли в другом месте. Мы подробнее рассмотрим этот вопрос в следующем разделе, когда будем обсуждать пароли.
- Некоторые поля могут быть пустыми. Так, если поле зашифрованного пароля не заполнено, это означает, что у пользователя нет пароля (что не рекомендуется). Запись для пользователя `squid` имеет одно пустое поле — поле комментария. Пустое поле комментария не оказывает никакого эффекта.
- Поле командного интерпретатора содержит имя выполняемого файла программы, которая используется в качестве командной оболочки при входе пользователя в систему. Если поле пустое, для него выбирается значение по умолчанию `/bin/sh`. Однако отметьте, что для пользователя `squid` в качестве командного интерпретатора используется устройство `/dev/null`. Очевидно, что оно не является выполняемым файлом, — это предотвращает возможность входа в систему под именем `squid` для кого бы то ни было.

Многие процессы-демоны, предоставляющие различные службы, имеют собственные идентификаторы пользователей (глава 13). Так, учетная запись `squid` предназначена для процессов, которые предоставляют услуги кэширующего прокси-сервера `squid`.

- Использование псевдоустройства `/dev/null` — не единственный способ воспрепятствовать конкретному пользователю войти в систему. Не менее часто в роли командного интерпретатора используется программа `/bin/false`. Она просто возвращает признак ошибки (ненулевое значение); оболочка расценивает этот код завершения как ложь. Также нередко для отключения учетной записи используется команда `/bin>true`. Она всегда возвращает признак успешного завершения (нулевое значение). В некоторых системах имеется команда `nologin`. Она выводит заданное сообщение об ошибке и возвращает код завершения, отличный от нуля.
- Имя пользователя `nobody` применяется с целью дать возможность кому-либо войти в систему, но с идентификатором пользователя 65534 и идентификатором группы 65534, которые не дают никаких привилегий. Такой пользователь сможет получить доступ лишь к тем файлам, которые доступны на чтение или на запись для всех. (Это предполагает отсутствие в системе файлов, принадлежащих пользователю с идентификатором 65534 или группе с идентификатором 65534.)
- Некоторые системы, имеющие команду `finger(1)`, поддерживают включение дополнительной информации в поле комментария. Дополнительные поля в поле комментария отделяются друг от друга запятой и содержат реальное имя пользователя, место работы, номера рабочего и домашнего телефонов. Кроме того, некоторые утилиты заменяют амперсанд (&) в поле комментария именем пользователя (в верхнем регистре). Например, представьте такую учетную запись:

```
sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
```

В этом случае утилита `finger` выведет следующую информацию о пользователе Steve Rago:

```
$ finger -p sar
Login: sar Name: Steve Rago
Directory: /home/sar Shell: /bin/sh
Office: SF 5-121, 555-1111 Home Phone: 555-2222
On since Mon Jan 19 03:57 (EST) on ttv0 (messages off)
No Mail.
```

Даже если система не поддерживает команду `finger`, ничто не мешает включать дополнительную информацию в поле комментария, поскольку это поле никак не интерпретируется системными утилитами.

В некоторых системах администратору доступна команда `vipw`, предназначенная для редактирования файла паролей. Команда `vipw` производит сериализацию изменений в файле паролей и обеспечивает согласование внесенных изменений с данными, хранящимися в дополнительных файлах. Многие системы предоставляют подобные возможности через графический интерфейс.

Стандарт POSIX.1 определяет всего две функции для доступа к отдельным учетным записям в файле паролей. Эти функции позволяют найти учетную запись по числовому идентификатору или имени пользователя.

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

Обе возвращают указатель в случае успеха, NULL — в случае ошибки

Функция `getpwuid` используется утилитой `ls(1)` для преобразования числового идентификатора, хранящегося в индексном узле, в имя пользователя. Функция `getpwnam` используется утилитой `login(1)`, когда пользователь вводит свое имя при входе в систему.

Обе функции возвращают указатель на заполненную структуру `passwd`. Эта структура обычно является статической переменной, расположенной в области памяти функции; в результате ее содержимое перезаписывается при каждом обращении к функции.

Эти две функции удобно использовать для получения сведений о конкретном пользователе по его имени или числовому идентификатору, однако некоторым программам может потребоваться просмотреть весь файл паролей. Для этого предназначены следующие три функции.

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Возвращает указатель в случае успеха,
NULL — в случае ошибки или по достижении конца файла

```
void setpwent(void);

void endpwent(void);
```

Эти три функции не включены в базовый стандарт POSIX.1, а определены как расширения XSI в Single UNIX Specification. Поэтому предполагается, что все версии UNIX должны их поддерживать.

Функция `getpwent` возвращает следующую запись из файла паролей. Как и предыдущие две, `getpwent` возвращает указатель на заполненную структуру `passwd`. Содержимое структуры обычно перезаписывается при каждом вызове функции. Первый вызов функции открывает все необходимые файлы. Порядок, в котором возвращаются записи, заранее не определен — они могут следовать в любом порядке, потому что некоторые системы используют хешированную версию файла `/etc/passwd`.

Функция `setpwent` переходит в начало каждого из используемых файлов, а функция `endpwent` закрывает файлы. По окончании работы с `getpwent` всегда следует закрывать файлы вызовом `endpwent`. Функция `getpwent` в состоянии определить, когда нужно открывать файлы (при первом обращении к ней), но она не может определить момент окончания работы с файлом паролей.

Пример

В листинге 6.1 показана реализация функции `getpwnam`.

Листинг 6.1. Функция `getpwnam`

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break; /* совпадение найдено */
    endpwent();
    return(ptr); /* вернет NULL в ptr, если совпадение не найдено */
}
```

Обращение к функции `setpwent` в самом начале — это мера предосторожности; мы просто выполняем переход в начало файла на тот случай, если вызывающая программа уже открыла его вызовом функции `getpwent`. Функция `endpwent` вызывается по окончании работы потому, что ни `getpwnam`, ни `getpwuid` не должны оставлять файлы открытыми.

6.3. Теневые пароли

Зашифрованный пароль — это пароль пользователя, пропущенный через односторонний алгоритм шифрования. Поскольку алгоритм является односторонним, невозможно получить оригинальный пароль, имея его зашифрованную версию.

Традиционно этот алгоритм (см. [Morris and Thompson, 1979]) генерирует 13 печатных символов из 64-символьного набора [a-zA-Z0-9./]. Некоторые современные системы используют для шифрования паролей алгоритм MD5 или SHA-1, который генерирует более длинную последовательность символов. (Чем больше символов используется для хранения зашифрованного пароля, тем больше возможных комбинаций и тем сложнее будет подобрать истинный пароль простым перебором вариантов.) Если в поле зашифрованного пароля поместить единственный символ, зашифрованный пароль гарантированно никогда не будет совпадать с этим значением.

Нет алгоритма, который можно было бы применить к зашифрованному паролю и получить простой текст. (Пароль в виде простого текста — это тот набор символов, который мы вводим в ответ на приглашение `Password:.`) Однако пароль можно подобрать, перебирая различные комбинации символов, пропуская их через односторонний алгоритм шифрования и сравнивая полученные результаты с зашифрованной версией пароля. Если бы пароли пользователей представляли собой набор случайных символов, такой метод был бы не очень удачным решением. Но пользователи стремятся выбирать пароли не случайным образом, нередко в этом качестве они используют имена супружеских пар, названия улиц или клички домашних животных. Это облегчает задачу любому, кто получил в свои руки копию файла паролей и пытается вычислить их. (Глава 4 книги [Garfinkel et al., 2003] содержит дополнительные сведения о паролях и о схеме шифрования, используемой в UNIX.)

Чтобы затруднить доступ к зашифрованным паролям, современные системы хранят их в другом файле, который часто называют *теневым файлом паролей* (*shadow password file*). Этот файл должен содержать как минимум имена пользователей и зашифрованные пароли. Здесь также хранится другая информация, имеющая отношение к паролям (табл. 6.2).

Таблица 6.2. Поля записей в файле /etc/shadow

Описание	Поле структуры spwd
Имя пользователя	<code>char *sp_namp</code>
Зашифрованный пароль	<code>char *sp_pwdp</code>
Время последнего изменения пароля в днях от начала Эпохи	<code>int sp_lstchg</code>
Минимальный период в днях между изменениями пароля	<code>int sp_min</code>
Максимальный период в днях между изменениями пароля	<code>int sp_max</code>
Количество дней до истечения срока действия пароля, в течение которых пользователь будет предупреждаться о необходимости его изменения	<code>int sp_warn</code>
Количество дней, когда учетная запись не использовалась	<code>int sp_inact</code>
Время отключения учетной записи в днях от начала Эпохи	<code>int sp_expire</code>
Резерв	<code>unsigned int sp_flag</code>

Обязательными являются только два поля — имя пользователя и зашифрованный пароль. Другие поля хранят информацию о том, как часто должен изменяться пароль и как долго будет оставаться активной учетная запись.

Теневой файл паролей не должен быть доступен на чтение для всех. Доступ к нему должны иметь лишь несколько программ, например `login(1)` и `passwd(1)`. Часто владельцем таких программ является суперпользователь и для них устанавливается бит set-user-ID. При использовании теневых паролей обычный файл паролей, `/etc/passwd`, может быть доступен для чтения любому.

В Linux 3.2.0 и Solaris 10 для доступа к теневому файлу паролей предусмотрен отдельный набор функций, очень похожих на те, что используются для работы с обычным файлом паролей.

```
#include <shadow.h>
struct spwd *getspnam(const char *name);
struct spwd *getspent(void);
```

Обе возвращают указатель в случае успеха, NULL – в случае ошибки

```
void setspent(void);
void endspent(void);
```

В FreeBSD 8.0 и Mac OS X 10.6.8 теневой файл паролей отсутствует¹, а вся дополнительная информация хранится в обычном файле паролей (см. табл. 6.1).

6.4. Файл групп

Файл групп UNIX, называемый в стандарте POSIX.1 базой данных групп, содержит поля, перечисленные в табл. 6.3. Значения этих полей хранятся в структуре `group`, определенной в файле `<grp.h>`.

Таблица 6.3. Поля в файле `/etc/group`

Описание	Поле структуры <code>group</code>	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Имя группы	<code>char *gr_name</code>	✓	✓	✓	✓	✓
Зашифрованный пароль	<code>char *gr_passwd</code>		✓	✓	✓	✓
Числовой идентификатор группы	<code>int gr_gid</code>	✓	✓	✓	✓	✓
Массив указателей на отдельные имена пользователей	<code>char **gr_mem</code>	✓	✓	✓	✓	✓

Поле `gr_mem` – это массив указателей на имена пользователей, входящих в состав группы. Этот массив завершается пустым указателем.

Для поиска имени группы или ее числового идентификатора в файле групп стандарт POSIX.1 определяет следующие две функции.

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

Обе возвращают указатель в случае успеха, NULL – в случае ошибки

¹ В BSD-системах функцию файла теневых паролей выполняет файл `/etc/master.passwd`. По формату он аналогичен файлу `passwd`, но содержит зашифрованный пароль. Подробнее см. в разделе 6.6. – *Примеч. пер.*

Как и в случае с файлом паролей, обе функции обычно возвращают указатель на статическую переменную, которая перезаписывается при каждом обращении к функциям.

Чтобы произвести поиск по всему файлу групп, потребуются некоторые дополнительные функции. Следующие три функции напоминают те, что используются для работы с файлом паролей.

```
#include <grp.h>
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
```

Возвращает указатель в случае успеха,
NULL – в случае ошибки или по достижении конца файла

Эти три функции не включены в базовый стандарт POSIX.1. Они определены как расширения XSI в Single UNIX Specification и предоставляются всеми версиями UNIX.

Функция **setgrent** открывает файл групп, если он еще не был открыт, и переходит в его начало. Функция **getgrent** читает очередную запись из файла групп, предварительно открыв его, если он еще не был открыт. Функция **endgrent** закрывает файл групп.

6.5. Идентификаторы дополнительных групп

Правила использования групп в UNIX существенно изменились за прошедшее время. Так, в Version 7 каждый пользователь в конкретный момент времени мог принадлежать только к одной группе. После входа в систему пользователю назначался реальный идентификатор группы, соответствующий числовому идентификатору группы из записи в файле паролей. Пользователь в любой момент мог изменить свою принадлежность к группе с помощью утилиты **newgrp(1)**. Если команда **newgrp** завершалась успехом (за информацией о правилах назначения прав доступа обращайтесь к справочному руководству), реальный идентификатор группы заменялся идентификатором новой группы, который затем использовался во всех последующих проверках прав доступа к файлам. Пользователь всегда мог вернуться к первоначальной группе, запустив команду **newgrp** без параметров.

Такая практика сохранялась до тех пор, пока не была изменена в версии 4.2BSD (около 1983 года). Начиная с версии 4.2BSD, появилось понятие идентификаторов дополнительных групп. Теперь пользователь мог принадлежать к группе, идентификатор которой указан в учетной записи в файле паролей, и входить в состав до 16 дополнительных групп. Проверки прав доступа к файлам изменились так, чтобы проверялся не только эффективный идентификатор группы, но и все идентификаторы дополнительных групп.

Дополнительные группы – обязательная для реализации функциональная особенность в соответствии со стандартом POSIX.1. (В ранних версиях POSIX.1 она была необязательной.) Константа `NGROUPS_MAX` (см. табл. 2.8) определяет количество идентификаторов дополнительных групп. Наиболее часто встречающееся значение – 16 (см. табл. 2.13).

Преимущество дополнительных групп состоит в том, что пользователю больше не нужно явно изменять свою принадлежность к группе. Членство в нескольких группах одновременно стало самым обычным делом (например, участие в разработке нескольких проектов).

Для получения и изменения идентификаторов дополнительных групп предназначены следующие три функции.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Возвращает количество дополнительных групп в случае успеха,
–1 – в случае ошибки

```
#include <grp.h>      /* в Linux */  
#include <unistd.h> /* в FreeBSD, Mac OS X и Solaris */  
  
int setgroups(int ngroups, const gid_t grouplist[]);  
  
#include <grp.h>      /* в Linux и Solaris */  
#include <unistd.h> /* в FreeBSD и Mac OS X */  
  
int initgroups(const char *username, gid_t basegid);
```

Обе возвращают 0 в случае успеха, –1 – в случае ошибки

Из этих трех функций только `getgroups` определена стандартом POSIX.1. Функции `setgroups` и `initgroups` не вошли в стандарт, поскольку относятся к разряду привилегированных операций. Однако четыре платформы, рассматриваемые в данной книге, поддерживают все три функции. В Mac OS X 10.6.8 аргумент `basegid` объявлен с типом `int`.

Функция `getgroups` заполняет массив `grouplist` идентификаторами дополнительных групп. В массив будет записано до `gidsetsize` элементов. Количество идентификаторов, записанных в массив, передается в вызывающую программу в виде возвращаемого значения.

Если в аргументе `gidsetsize` передается 0, функция возвращает только количество дополнительных групп. Массив `grouplist` не изменяется. (Это позволяет вызывающей программе определить размер массива `grouplist` перед его размещением в динамической памяти.)

Функция `setgroups` может вызываться только суперпользователем для изменения списка идентификаторов дополнительных групп вызывающего процесса: в этом случае `grouplist` содержит массив идентификаторов групп, а `ngroups` – количество элементов в массиве. Значение `ngroups` не должно превышать константу `NGROUPS_MAX`.

Единственное практическое применение функции `setgroups` — вызов из функции `initgroups`, которая читает файл групп с помощью `getgrent`, `setgrent` и `endgrent` и определяет группы, к которым принадлежит *username*. После этого она вызывает `setgroups`, чтобы инициализировать список идентификаторов дополнительных групп пользователя. Чтобы вызвать функцию `initgroups`, процесс должен обладать привилегиями суперпользователя, так как она вызывает функцию `setgroups`. В дополнение ко всем найденным группам, к которым принадлежит *username*, `initgroups` также включает в список идентификаторов дополнительных групп и значение *basegid*, где *basegid* — идентификатор группы, взятый из записи для *username* в файле паролей.

Функция `initgroups` используется немногими программами, в качестве примера можно упомянуть утилиту `login(1)`, которая вызывает `initgroups`, когда пользователь входит в систему.

6.6. Различия реализаций

Мы уже рассмотрели теневой файл паролей, который поддерживается в Linux и Solaris. Системы FreeBSD и Mac OS X хранят зашифрованные пароли иначе. В табл. 6.4 обобщены сведения о том, как четыре платформы, обсуждаемые в данной книге, хранят информацию о пользователях и группах.

Таблица 6.4. Различия в реализации хранения учетных записей

Информация	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Учетные записи	/etc/passwd	/etc/passwd	Служба каталогов	/etc/passwd
Зашифрованные пароли	/etc/master.passwd	/etc/shadow	Служба каталогов	/etc/shadow
Наличие хешированных файлов паролей	Да	Нет	Нет	Нет
Информация о группах	/etc/group	/etc/group	Служба каталогов	/etc/group

В ОС FreeBSD теневой файл паролей хранится под именем `/etc/master.passwd`. Для редактирования этого файла используются специальные команды, которые, в свою очередь, генерируют файл `/etc/passwd` на основе теневого файла паролей. Кроме того, создаются хешированные версии файлов: `/etc/pwd.db` — хешированная версия файла `/etc/passwd`, и `/etc/spwd.db` — хешированная версия файла `/etc/master.passwd`. Они обеспечивают более высокую производительность в крупных системах.

Однако в Mac OS X файлы `/etc/passwd` и `/etc/master.passwd` используются только в однопользовательском режиме (когда администратор проводит обслуживание системы; однопользовательский режим обычно означает, что системные службы не запущены). В нормальном, то есть многопользовательском, режиме доступ к информации о пользователях и группах обеспечивает служба каталогов.

Хотя Linux и Solaris поддерживают схожие интерфейсы теневых паролей, в их реализации все же имеются некоторые отличия. Например, в Solaris целочисленные поля (см. табл. 6.2) определены с типом `int`, а в Linux — с типом `long int`. Другое отличие — поле, где подсчитывается период, в течение которого учетная запись оставалась неактивной. В Solaris это поле обозначает количество дней, прошедших с момента последнего входа пользователя в систему, тогда как в Linux — это количество дней, оставшихся до окончания максимального срока действия существующего пароля.

Во многих системах базы данных пользователей и групп реализованы с использованием сетевой информационной службы (Network Information Service, NIS). Это позволяет администраторам редактировать эталонные копии баз данных и распространять их автоматически по всем серверам в организации. Клиентские системы соединяются с серверами и получают от них необходимые сведения о пользователях и группах. Аналогичные функциональные возможности предоставляют NIS+ и LDAP (Lightweight Directory Access Protocol — облегченный протокол доступа к каталогам). В большинстве систем метод администрирования каждого вида информации контролируется с помощью конфигурационного файла `/etc/nsswitch.conf`.

6.7. Прочие файлы данных

До сих пор обсуждались только два файла системных данных: файл паролей и файл групп. В своей повседневной работе UNIX-системы используют множество других файлов. Например, для поддержки сетевых взаимодействий платформы BSD используют файлы `/etc/services` (перечень служб, предоставляемых серверами сети), `/etc/protocols` (перечень сетевых протоколов) и `/etc/networks` (список сетей). К счастью, интерфейсы для работы с этими файлами похожи на те, что используются для доступа к файлам паролей и групп.

Эти интерфейсы следуют одному общему принципу — для работы с каждым файлом предоставляется по меньшей мере три функции.

1. Функция `get` читает следующую запись, открывая файл при необходимости. Эти функции обычно возвращают указатель на структуру. Если достигнут конец файла, возвращается пустой указатель. Большинство функций `get` возвращают указатель на структуру, размещенную статически, поэтому мы должны скопировать ее содержимое, чтобы сохранить для последующего использования.
2. Функция `set` открывает файл, если он еще не открыт, и переходит в начало. Эта функция используется, если по каким-то причинам необходимо вернуться в начало файла.
3. Функция `end` закрывает файл данных. Как уже упоминалось выше, она всегда должна вызываться по завершении работы с файлом.

Кроме того, если файл данных поддерживает тот или иной вид поиска по ключу, предусматриваются функции, которые выполняют такой поиск. Например,

для файла паролей имеются две функции: `getpwnam` ищет запись с определенным именем пользователя и `getpwuid` ищет запись с определенным идентификатором пользователя.

В табл. 6.5 перечислены некоторые функции для работы с файлами данных, обычные для систем UNIX. В эту таблицу включены функции для работы с файлами паролей и групп, которые рассматривались ранее в этой главе, и некоторые из функций поддержки сети. Перечислены все функции `get`, `set` и `end` для всех файлов данных, упомянутых в таблице.

Таблица 6.5. Функции для работы с системными файлами данных

Описание	Файл	Заголовочный файл	Структура	Дополнительные функции поиска
Пароли	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
Группы	/etc/group	<grp.h>	group	getgrnam, getgrgid
Теневой файл паролей	/etc/shadow	<shadow.h>	spwd	getspnam
Сетевые узлы	/etc/hosts	<netdb.h>	hostent	getnameinfo, getaddrinfo
Сети	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
Протоколы	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
Службы	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

В Solaris последние четыре файла из табл. 6.5 являются символическими ссылками на файлы с теми же именами в каталоге /etc/inet. Большинство реализаций UNIX предоставляют дополнительные функции, подобные перечисленным, но предназначенные для системного администрирования и специфичные для каждой конкретной реализации.

6.8. Учет входов в систему

В большинстве систем UNIX имеется два файла данных: `utmp`, с информацией обо всех работающих в системе пользователях, и `wtmp`, с информацией обо всех попытках входа в систему и выхода из нее. В Version 7 в оба файла информация записывалась в двоичном виде и определялась одной и той же структурой:

```
struct utmp {
    char ut_line[8]; /* имя устройства: "ttyh0", "ttyp0", ... */
    char ut_name[8]; /* пользователь */
    long ut_time; /* количество секунд от начала Эпохи */
};
```

Во время входа пользователя в систему программа `login` заполняла одну такую структуру и записывала в файл `utmp`, и та же самая структура добавлялась в файл `wtmp`. При выходе из системы процесс `init` стирал запись в файле `utmp`, заполняя

ее нулевыми байтами, а в файл `wtmp` добавлялась новая запись. В этой записи, соответствующей выходу из системы, поле `ut_name` очищалось. Во время перезагрузки системы, а также до и после изменения системной даты и времени в файл `wtmp` добавлялись специальные записи. Утилита `who(1)` извлекала данные из файла `utmp` и выводила их в удобочитаемом виде. В более поздних версиях UNIX имеется команда `last(1)`, которая читает данные из файла `wtmp` и выводит выбранные записи.

Большинство версий UNIX все еще поддерживают файлы `utmp` и `wtmp`, но, как и следовало ожидать, объем информации в этих файлах вырос. Так, 20-байтная структура, использовавшаяся в Version 7, выросла до 36 байт в SVR2, а в SVR4 расширенная структура `utmp` занимает уже 350 байт!

Детальное описание формата этих записей в Solaris приводится на странице справочного руководства `utmpx(4)`. В Solaris 10 оба файла находятся в каталоге `/var/adm`. ОС Solaris предоставляет много функций для работы с этими файлами, описание которых можно найти в `getutxent(3)`.

На странице справочного руководства `utmp(5)` в Linux 3.2.0 и FreeBSD 8.0 дается описание формата записей для этих версий. Полные имена этих двух файлов — `/var/run/utmp` и `/var/Log/wtmp`. В Mac OS X 10.6.8 файлы `utmp` и `wtmp` отсутствуют. Начиная с версии Mac OS X 10.5 информацию, прежде находившуюся в файле `wtmp`, можно получить с помощью системного механизма журналирования, а файл `utmpx` содержит информацию об активных сессиях, открытых пользователями.

6.9. Информация о системе

Стандарт POSIX.1 определяет функцию `uname`, которая возвращает сведения о текущем хосте и операционной системе.

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

Возвращает неотрицательное значение в случае успеха,
–1 — в случае ошибки

В качестве аргумента передается адрес структуры `utsname`, и функция заполняет ее. Стандартом POSIX.1 определен лишь минимальный набор полей в этой структуре, каждое из которых является массивом символов, а размеры этих массивов определяются конкретными реализациями. Некоторые реализации добавляют в эту структуру дополнительные поля.

```
struct utsname {
    char sysname[]; /* имя операционной системы */
    char nodename[]; /* имя узла сети */
    char release[]; /* номер выпуска операционной системы */
    char version[]; /* номер версии этого выпуска */
    char machine[]; /* тип аппаратной архитектуры */
};
```

Каждая строка заканчивается нулевым символом. В табл. 6.6 приводятся максимальные размеры массивов для всех четырех платформ, обсуждаемых в книге. Информация, содержащаяся в структуре `utsname`, обычно выводится командой `uname(1)`.

Стандарт POSIX.1 предупреждает, что поле `nodename` может не соответствовать действительному сетевому имени хоста. Это поле пришло из System V и ранее хранило имя хоста в сети, работающей по протоколу UUCP.

Кроме того, помните, что содержимое этой структуры не дает никакой информации о версии POSIX.1. Эти сведения можно получить при помощи константы `_POSIX_VERSION` (как описывалось в разделе 2.6).

Наконец, эта функция дает лишь возможность получить информацию в виде структуры, стандарт POSIX.1 никак не оговаривает порядок инициализации этой информации.

Системы, производные от BSD, традиционно предоставляют функцию `gethostname`, возвращающую только имя хоста. Как правило, это имя соответствует сетевому имени хоста в сети TCP/IP.

```
#include <unistd.h>
int gethostname(char *name, int nameLen);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Аргумент `namelen` определяет размер буфера `name`. Если в буфере достаточно места, возвращаемая в `name` строка будет завершаться нулевым символом. Если места в буфере недостаточно, то не оговаривается, будет ли полученная строка завершаться нулевым символом.

В настоящее время функция `gethostname` является частью стандарта POSIX.1, который указывает, что максимальная длина имени хоста равна `HOST_NAME_MAX`. Значения максимальной длины имен хостов для всех четырех обсуждаемых платформ приводятся в табл. 6.6.

Таблица 6.6. Ограничения на размеры строк идентификации системы

Функция	Максимальная длина аргумента <code>name</code>			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>uname</code>	256	65	256	257
<code>gethostname</code>	256	64	256	256

Если хост подключен к сети TCP/IP, имя хоста обычно представляет собой полное доменное имя.

Существует также команда `hostname(1)`, которая может выводить или изменять имя хоста. (Имя хоста устанавливается суперпользователем с помощью аналогичной функции `sethostname`.) Как правило, имя хоста устанавливается во время загрузки системы в одном из файлов инициализации, вызываемых сценариями из `/etc/rc` или `init`.

6.10. Функции даты и времени

Роль основной службы времени в ядре UNIX играет счетчик секунд, прошедших от начала Эпохи – 00:00:00 1 января 1970 года по всеобщему скоординированному времени (UTC). В разделе 1.10 мы уже говорили, что значение счетчика представлено типом данных `time_t` и называется *календарным временем*. С помощью календарного времени можно представить как дату, так и время суток. UNIX всегда отличалась от других систем тем, что она (а) хранит время UTC, а не местное время, (б) автоматически выполняет преобразования, такие как переход на летнее время, и (в) хранит дату и время как единое целое.

Функция `time` возвращает текущее время и дату.

```
#include <time.h>
time_t time(time_t *calptr);
```

Возвращает значение времени в случае успеха, -1 – в случае ошибки

Функция всегда возвращает значение времени. Если в качестве аргумента `calptr` передается непустой указатель, значение времени дополнительно записывается по указанному адресу.

Расширения реального времени, определяемые стандартом POSIX.1, добавляют поддержку нескольких системных часов. В версии 4 стандарта Single UNIX Specification интерфейсы управления этими часами были перенесены из категории необязательных в категорию базовых. Часы идентифицируются типом `clockid_t`. Стандартные значения перечислены в табл. 6.7.

Таблица 6.7. Идентификаторы типов часов

Идентификатор	Параметр	Описание
<code>CLOCK_REALTIME</code>		Действительное системное время
<code>CLOCK_MONOTONIC</code>	<code>_POSIX_MONOTONIC_CLOCK</code>	Действительное системное время без отрицательных переходов
<code>CLOCK_PROCESS_CPUTIME_ID</code>	<code>_POSIX_CPUTIME</code>	Процессорное время вызывающего процесса
<code>CLOCK_THREAD_CPUTIME_ID</code>	<code>_POSIX_THREAD_CPUTIME</code>	Процессорное время вызывающего потока выполнения

Получить время конкретных часов можно с помощью функции `clock_gettime`. Она возвращает время в виде структуры `timespec`, представленной в разделе 4.2, которая выражает время в секундах и наносекундах.

```
#include <sys/time.h>
int clock_gettime(clockid_t clock_id, struct timespec *tsp);
```

Возвращает 0 в случае успеха, -1 – в случае ошибки

Когда в аргументе *clock_id* передается идентификатор `CLOCK_REALTIME`, функция `clock_gettime` действует подобно функции `time`, за исключением того, что с помощью `clock_gettime` можно получить значение времени с большим разрешением, если система поддерживает такую возможность.

Определить разрешение конкретных системных часов можно с помощью функции `clock_getres`.

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

int clock_getres(clockid_t clock_id, struct timespec *tsp);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Функция `clock_getres` инициализирует структуру `timespec`, на которую указывает аргумент *tsp*, сохраняя величину разрешения часов, соответствующих идентификатору в аргументе *clock_id*. Например, для разрешения в 1 миллисекунду в поле `tv_sec` будет помещено значение 0, а в поле `tv_nsec` — значение 1 000 000. Установить время в определенных часах можно вызовом функции `clock_settime`.

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

int clock_settime(clockid_t clock_id, const struct timespec *tsp);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Для изменения времени в часах необходимо обладать соответствующими привилегиями. При этом некоторые часы недоступны для корректировки.

Исторически в реализациях, производных от System V, для установки нового времени использовалась функция `stime(2)`, а в производных от BSD — функция `settimeofday(2)`.

Версия 4 стандарта Single UNIX Specification объявила функцию `gettimeofday` устаревшей. Однако многие программы все еще используют ее, потому что она дает более высокую точность (до микросекунд), чем функция `time`.

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

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Всегда возвращает 0

Единственное допустимое значение аргумента *tzp* — `NULL`; любые другие значения могут привести к непредсказуемым результатам. Некоторые платформы поддерживают передачу часового пояса через аргумент *tzp*, но это зависит от конкретной реализации и не определено в Single UNIX Specification.

Функция `gettimeofday` сохраняет время, прошедшее от начала Эпохи до настоящего момента, по адресу *tp*. Это время представлено в виде структуры `timeval`, которая хранит секунды и микросекунды.

После получения целочисленного количества секунд, прошедших с начала Эпохи, как правило, вызывается одна из функций преобразования, которая переведет числовое значение в удобочитаемые время и дату. На рис. 6.1 показаны взаимоотношения между различными функциями преобразования времени. (Три функции, изображенные на этом рисунке пунктирными линиями, — `localtime`, `mktime` и `strftime`, — учитывают значение переменной окружения `TZ`, которую мы рассмотрим далее в этом разделе.)

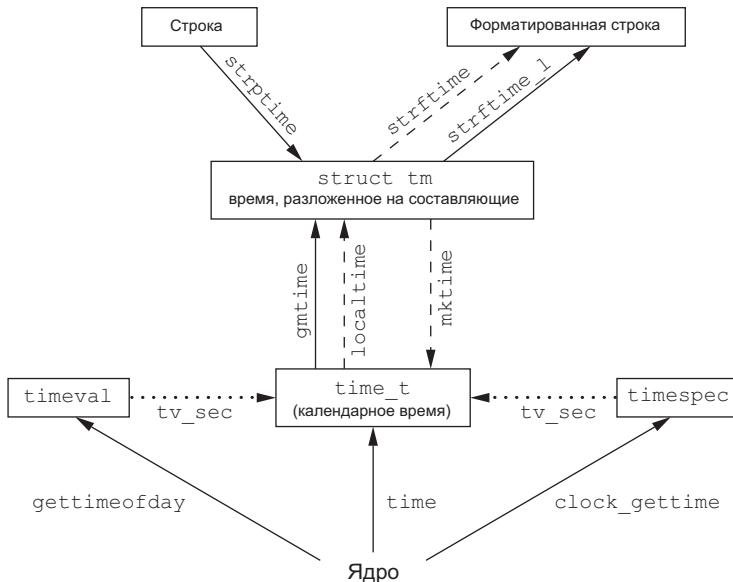


Рис. 6.1. Взаимоотношения между различными функциями представления времени

Две функции, `localtime` и `gmtime`, преобразуют календарное время в структуру `tm`, состоящую из следующих элементов:

```

struct tm { /* время, разложенное на составляющие */
    int tm_sec; /* секунды от начала минуты: [0 - 60] */
    int tm_min; /* минуты от начала часа: [0 - 59] */
    int tm_hour; /* часы от полуночи: [0 - 23] */
    int tm_mday; /* дни от начала месяца: [1 - 31] */
    int tm_mon; /* месяцы с января: [0 - 11] */
    int tm_year; /* годы с 1900 года */
    int tm_wday; /* дни с воскресенья: [0 - 6] */
    int tm_yday; /* дни от начала года (1 января): [0 - 365] */
    int tm_isdst; /* флаг перехода на летнее время: <0, 0, >0 */
};

```

Количество секунд может превышать 59, когда для коррекции времени вставляется дополнительная секунда. Обратите внимание, что отсчет всех компонентов, кроме дня месяца, начинается с 0. Флаг перехода на летнее время представлен положительным числом, если действует летнее время, 0 — если нет, и отрицательным числом, если данная информация недоступна.

В предыдущих версиях Single UNIX Specification допускалась вставка двух дополнительных секунд. Таким образом, диапазон значений поля `tm_sec` составлял 0–61. Формальное определение UTC не допускает вставки двух дополнительных секунд, поэтому сейчас диапазон представления секунд определяется как 0–60.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);
struct tm *localtime(const time_t *calptr);
```

Обе возвращают указатель на структуру `tm`, `NULL` — в случае ошибки

Функции `localtime` и `gmtime` отличаются тем, что первая преобразует календарное время в местное, учитывая при этом часовой пояс и переход на летнее время, а вторая разбивает календарное время UTC на составляющие.

Функция `mktime` принимает местное время в виде структуры `tm` и преобразует его в значение `time_t`.

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

Возвращает календарное время в случае успеха, `-1` — в случае ошибки

Функция `strftime` — это `printf`-подобная функция для представления значений времени. Она отличается разнообразием аргументов, позволяющих настраивать возвращаемую строку.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
               const char *restrict format,
               const struct tm *restrict tmptr);

size_t strftime_l(char *restrict buf, size_t maxsize,
                  const char *restrict format,
                  const struct tm *restrict tmptr, locale_t Locale);
```

Обе возвращают количество символов, записанных в массив, если в нем достаточно места, в противном случае возвращают 0

Для получения 26-байтной строки, напоминающей результат команды `date(1)`, прежде можно было использовать две старые функции, `asctime` и `ctime`. Однако в настоящее время они отмечены как устаревшие, так как не ограждают от проблемы переполнения буфера.

Функции `strftime` и `strftime_l` действуют практически одинаково, за исключением того, что функция `strftime_l` позволяет вызывающей программе передавать

региональные настройки в аргументе *locale*. Функция `strftime` использует региональные настройки в переменной окружения `TZ`.

Аргумент *tmptr* — указатель на структуру `tm`, содержащую время, которое должно быть представлено в виде отформатированной строки. Результат форматирования сохраняется в буфере *buf*, размер которого определяется аргументом *maxsize*. Если полученная строка, включая завершающий нулевой символ, умещается в буфере, эти функции возвращают длину строки без завершающего нулевого символа. Иначе возвращается 0.

Аргумент *format* управляет форматированием значения времени. Как и в случае с функцией `printf`, спецификаторы формата начинаются с символа процента, за которым следуют служебные символы. Все остальные символы в строке *format* выводятся без изменений. Два символа процента, следующие друг за другом, будут отображаться как один символ процента. В отличие от функции `printf`, каждый спецификатор формата генерирует на выходе строки фиксированного размера — спецификаторы ширины поля вывода не предусмотрены. В табл. 6.8 перечислены 37 спецификаторов формата, определяемых стандартом ISO C.

Таблица 6.8. Спецификаторы формата функции `strftime`

Спецификатор формата	Описание	Пример
%a	Краткое название дня недели	Thu
%A	Полное название дня недели	Thursday
%b	Краткое название месяца	Jan
%B	Полное название месяца	January
%c	Дата и время	Thu Jan 19 21:24:52 2012
%C	Две первые цифры года [00–99]	20
%d	День месяца: [01–31]	19
%D	Дата: [MM/DD/YY]	01/19/12
%e	День месяца (начальный 0 замещается пробелом): [1–31]	19
%F	Дата в формате ISO 8601: [YYYY-MM-DD]	2012-01-19
%g	Последние две цифры года в формате ISO 8601 с учетом номера недели: [00–99]	12
%G	Год в формате ISO 8601 с учетом номера недели	2012
%h	То же, что %b	Jan
%H	Час (в 24-часовом формате): [00–23]	21
%I	Час (в 12-часовом формате): [00–12]	09
%j	День года: [001–366]	019
%m	Номер месяца: [01–12]	01
%M	Минуты: [00–59]	24

Таблица 6.8 (окончание)

Спецификатор формата	Описание	Пример
%n	Символ перевода строки	
%p	AM или PM (до или после полудня)	PM
%r	Местное время в 12-часовом формате	09:24:52 PM
%R	То же, что %H:%M	21:24
%S	Секунды: [00–60]	52
%t	Символ горизонтальной табуляции	
%T	То же, что %H:%M:%S	21:24:52
%u	Номер дня недели в формате ISO 8601 [понедельник = 1, 1–7]	4
%U	Номер недели в году, воскресенье — первый день недели: [01–53]	03
%V	Номер недели в году в формате ISO 8601: [01–53]	03
%w	Номер дня недели [воскресенье = 0, 0–6]	4
%W	Номер недели в году, понедельник — первый день недели: [00–53]	03
%x	Дата	01/19/12
%X	Время	21:24:52
%y	Последние две цифры года: [00–99]	12
%Y	Год	2012
%z	Разница между поясным временем и UTC	-0500
%Z	Название часового пояса	EST
%%	Символ процента	%

В третьей колонке таблицы приводится вывод функции `strftime` в Mac OS X, соответствующий времени `Thu Jan 19 21:24:52 EST 2012`.

Единственные спецификаторы, смысл которых неочевиден: `%U`, `%V` и `%w`. Спецификатор `%U` представляет номер недели в году, начиная с недели, на которую выпадает первое воскресенье года. Спецификатор `%w` представляет номер недели в году, начиная с недели, на которую выпадает первый понедельник года. Действие спецификатора `%V` зависит от конкретного года. Если неделя, на которую выпадает 1 января, содержит четыре или более дня нового года, она считается первой неделей года, иначе — последней неделей предыдущего года. В обоих случаях первым днем недели считается понедельник.

Как и `printf`, функция `strftime` поддерживает модификаторы для некоторых спецификаторов формата. Модификаторы `O` и `E` позволяют получить строку в альтернативном формате, если таковой поддерживается региональными настройками системы.

Некоторые системы поддерживают дополнительные, нестандартные спецификаторы формата для функции strftime.

Пример

Программа в листинге 6.2 демонстрирует, как использовать функции для работы со временем, описанные в этой главе. В частности, она показывает, как с помощью strftime выводить строки с текущими датой и временем.

Листинг 6.2. Пример использования функции strftime

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int
main(void)
{
    time_t      t;
    struct tm *tmp;
    char        buf1[16];
    char        buf2[64];

    time(&t);
    tmp = localtime(&t);
    if (strftime(buf1, 16, "время и дата: %r, %a %b %d, %Y", tmp) == 0)
        printf("длина буфера 16 слишком мала\n");
    else
        printf("%s\n", buf1);
    if (strftime(buf2, 64, "время и дата: %r, %a %b %d, %Y", tmp) == 0)
        printf("длина буфера 64 слишком мала\n");
    else
        printf("%s\n", buf2);
    exit(0);
}
```

Взгляните еще раз на рис. 6.1, где изображены взаимоотношения между различными функциями представления времени. Прежде чем вывести время в удобочитаемом представлении, его необходимо получить и преобразовать в структуру tm. Пример в листинге 6.2 выводит:

```
$ ./a.out
длина буфера 16 слишком мала
время и дата: 11:12:35 PM, Thu Jan 19, 2012
```

Функция strptime — обратная по отношению к strftime. Она принимает строку и преобразует ее в структуру tm.

```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
               struct tm *restrict tmprtr);
```

Возвращает указатель на символ, находящийся за последним проанализированным символом, NULL — в противном случае

Аргумент *format* описывает формат строки в буфере, на который указывает аргумент *buf*. Спецификаторы формата напоминают спецификаторы, поддерживающие функцией `strftime`, хотя и с некоторыми различиями. Перечень спецификаторов для функции `strptime` приводится в табл. 6.9.

Таблица 6.9. Спецификаторы формата функции `strptime`

Спецификатор формата	Описание
%a	Краткое или полное название дня недели
%A	То же, что %a
%b	Краткое или полное название месяца
%B	То же, что %b
%c	Дата и время
%C	Две первые цифры года
%d	День месяца: [01–31]
%D	Дата: [MM/DD/YY]
%e	То же, что %d
%h	То же, что %b
%H	Час (в 24-часовом формате): [00–23]
%I	Час (в 12-часовом формате): [00–12]
%j	день года: [001–366]
%m	Номер месяца: [01–12]
%M	Минуты: [00–59]
%n	Любой пробельный символ
%p	AM или PM (до или после полудня)
%r	Местное время в 12-часовом формате
%R	То же, что %H:%M
%S	Секунды: [00–60]
%t	Любой пробельный символ
%T	То же, что %H:%M:%S
%U	Номер недели в году, воскресенье — первый день недели: [01–53]
%w	Номер дня недели [воскресенье = 0, 0–6]
%W	Номер недели в году, понедельник — первый день недели: [00–53]
%x	Местная дата
%X	Местное время
%y	Последние две цифры года: [00–99]
%Y	Год
%%	Символ процента

Мы уже упоминали, что три функции, обозначенные на рис. 6.1 пунктирующими линиями, учитывают значение переменной окружения `TZ` — это функции `localtime`, `mktime` и `strftime`. Если эта переменная определена, ее значение используется вместо значения часового пояса по умолчанию. Если переменная определена как пустая строка, например как `TZ=`, обычно в качестве часового пояса используется UTC. Эта переменная часто содержит нечто вроде `TZ=EST5EDT`, но стандарт POSIX.1 допускает указывать более детальную информацию. За дополнительными сведениями о переменной окружения `TZ` обращайтесь к главе «Environment Variables» стандарта Single UNIX Specification [Open Group, 2010].

Дополнительную информацию о переменной окружения `TZ` можно найти на странице справочного руководства `tzset(3)`.

6.11. Подведение итогов

Все системы UNIX используют файл паролей и файл групп. Мы рассмотрели различные функции для работы с этими файлами, а также обсудили теневые файлы паролей, использование которых повышает уровень безопасности системы. Поддержка дополнительных групп позволяет включать пользователя сразу в несколько групп. Кроме того, мы рассмотрели ряд функций, предусмотренных большинством систем для работы с другими файлами данных. Обсудили определяемые стандартом POSIX.1 функции, которые могут использоваться приложениями для идентификации системы, в которой они запущены. Глава заканчивается обзором функций для работы с датой и временем, определяемых стандартами ISO C и Single UNIX Specification.

Упражнения

- 6.1** Представьте, что система использует теневой файл паролей и вам необходимо получить пароль в зашифрованном виде. Как это можно сделать?
- 6.2** Реализуйте предыдущее упражнение в виде функции, исходя из предположения, что в системе имеется теневой файл паролей и вы обладаете привилегиями суперпользователя.
- 6.3** Напишите программу, которая вызывает функцию `uname` и выводит содержимое всех полей структуры `utsname`. Сравните получившиеся результаты с тем, что выводит команда `uname(1)`.
- 6.4** Вычислите максимально возможное значение времени, которое может быть представлено с помощью типа `time_t`. Что произойдет, когда счетчик времени переполнится?
- 6.5** Напишите программу, которая получает текущее время и выводит его с помощью функции `strftime`, при этом результат должен выглядеть так же, как вывод команды `date(1)` по умолчанию. Присвойте переменной окружения `TZ` другое значение и проверьте, что произойдет.

7

Окружение процесса

7.1. Введение

Прежде чем перейти к функциям управления процессами, которые будут обсуждаться в следующей главе, исследуем окружение отдельного процесса. В этой главе мы рассмотрим, как вызывается функция `main` в момент запуска программы, как программе передаются аргументы командной строки, как выглядит типичная раскладка памяти, как распределяется дополнительная память, как процесс может использовать переменные окружения и как завершается работа процесса. Дополнительно мы исследуем функции `longjmp` и `setjmp` и их взаимодействие со стеком. И наконец, рассмотрим ограничения на ресурсы процесса.

7.2. Функция `main`

Программы на языке С начинают работу с вызова функции `main`. Прототип этой функции:

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

где `argc` — количество аргументов командной строки, а `argv` — массив указателей на аргументы. Мы подробно рассмотрим эти аргументы в разделе 7.4.

Когда ядро запускает программу на языке С (с помощью одной из функций семейства `exec`, которые будут описаны в разделе 8.10), перед вызовом функции `main` выполняется специальная процедура начального запуска. Адрес этой процедуры указывается в выполняемом файле программы как точка входа. Этот адрес определяется редактором связей, который вызывается компилятором языка С. Процедура начального запуска принимает от ядра аргументы командной строки и значения переменных окружения, после чего выполняет вызов функции `main`.

7.3. Завершение работы процесса

Существует восемь способов завершения работы процесса. Нормальными считаются пять из них.

1. Возврат из функции `main`.
2. Вызов функции `exit`.

3. Вызов функции `_exit` или `_Exit`.
4. Возврат из функции запуска последнего потока выполнения (раздел 11.5).
5. Вызов функции `pthread_exit` (раздел 11.5) из последнего потока выполнения. Ненормальное завершение процесса происходит в следующих случаях.
 1. При вызове функции `abort` (раздел 10.17).
 2. При получении сигнала (раздел 10.2).
 3. По запросу на завершение последнего потока выполнения (разделы 11.5 и 12.7).

Мы не будем рассматривать способы завершения, связанные с потоками выполнения, до их обсуждения в главах 11 и 12.

Процедура начального запуска, о которой мы говорили в предыдущем разделе, спроектирована так, что она вызывает функцию `exit`, когда происходит возврат из функции `main`. Если процедура начального запуска написана на С (хотя чаще всего она написана на языке ассемблера), запуск функции `main` выглядит примерно так:

```
exit(main(argc, argv));
```

Функции семейства `exit`

Нормальное завершение программы осуществляется тремя функциями: `_exit` и `_Exit`, сразу же возвращающими управление ядру, и `exit`, выполняющей ряд дополнительных операций и только после этого возвращающей управление ядру.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

В разделе 8.5 мы рассмотрим влияние этих функций на другие процессы, например на родителя и потомков завершаемого процесса.

Прототипы функций объявлены в разных заголовочных файлах потому, что функции `_exit` и `_Exit` определяются стандартом ISO C, а функция `exit` – стандартом POSIX.1.

Функция `exit` всегда выполняла корректное закрытие стандартной библиотеки ввода/вывода, вызывая функцию `fclose` для всех открытых потоков ввода/вывода. В разделе 5.5 мы уже говорили, что это приводит к сбросу всех буферов (то есть к записи их содержимого на диск).

Все три функции завершения принимают единственный аргумент целочисленного типа, который называется *кодом завершения*. Большинство командных оболочек UNIX позволяют узнать код завершения процесса. Код завершения процес-

са считается неопределенным, если (а) любая из этих функций вызвана без кода завершения процесса, (б) функция `main` вызывает оператор `return` без аргумента или (в) функция `main` объявлена как не возвращающая целочисленное значение. Однако если функция `main`, объявленная как возвращающая целое число, «неожиданно завершается» (неявный возврат из функции), код завершения процесса считается равным 0.

Такое поведение было определено стандартом ISO C в 1999 году. Исторически код завершения считался неопределенным, если выход из функции `main` осуществлялся не через явное обращение к оператору `return` или к функции `exit`.

Возврат целочисленного значения из функции `main` эквивалентен вызову функции `exit` с тем же самым значением. То есть

```
exit(0);
```

означает то же самое, что

```
return(0);
```

Пример

В листинге 7.1 представлен пример классической программы «Привет, МИР!».

Листинг 7.1. Классический пример программы на языке С

```
#include <stdio.h>

main()
{
    printf("Привет, МИР!\n");
}
```

Если скомпилировать и запустить эту программу, окажется, что она возвращает случайный код завершения. В разных системах мы почти наверняка получим различные коды завершения в зависимости от содержимого стека и регистров процессора в момент выхода из функции `main`:

```
$ gcc hello.c
$ ./a.out
Привет, МИР!
$ echo $?                                вывести код завершения
13
```

Теперь, если включить режим совместимости компилятора с расширением 1999 ISO C, можно увидеть, что код завершения изменился:

```
$ gcc -std=c99 hello.c      включить расширения 1999 ISO C компилятора gcc
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
Привет, МИР!
$ echo $?                                вывести код завершения
0
```

Обратите внимание на предупреждение компилятора, появившееся после включения режима совместимости с расширением 1999 ISO C. Это обусловлено тем, что функция

`main` не объявлена явно как возвращающая целочисленное значение. Если добавить это объявление, предупреждение выводиться не будет. Однако если включить вывод всех предупреждений компилятора (с помощью флага `-Wall`), появится еще одно предупреждение примерно такого содержания: «*control reaches end of nonvoid function*» («достигнут конец функции, имеющей возвращаемое значение»).

Объявление функции `main` как возвращающей целочисленное значение и использование функции `exit` вместо оператора `return` приводят к появлению бесполезных предупреждений от некоторых компиляторов и от программы `lint(1)`. Эти компиляторы не понимают, что выход из `main` с помощью функции `exit`, по сути, то же самое, что обращение к оператору `return`. Избавиться от таких предупреждений, которые через некоторое время начинают раздражать, можно, использовав оператор `return` вместо вызова `exit`. Но такое решение лишает нас возможности легко и просто находить все точки выхода из программы с помощью утилиты `grep`. Другое возможное решение — объявить функцию `main` с типом возвращаемого значения `void` вместо `int` и продолжать использовать функцию `exit`. Это избавит от надоедливых предупреждений компилятора, но выглядит не очень правильно (особенно в исходных текстах программы) и, кроме того, может вызвать появление других предупреждений, поскольку предполагается, что функция `main` должна возвращать целочисленный результат. В этой книге мы представляем функцию `main` как возвращающую целое число, поскольку это определено стандартами ISO C и POSIX.1.

В зависимости от компилятора выводимые предупреждения могут быть более или менее подробными. Обратите внимание: компилятор GNU C обычно не выдает эти ненужные сообщения, если не используются дополнительные параметры управления выводом предупреждений.

В следующей главе мы увидим, как один процесс может запустить другой, дождаться его завершения и получить код завершения.

Функция `atexit`

В соответствии со стандартом ISO C процесс может зарегистрировать до 32 функций, которые будут автоматически вызываться функцией `exit`. Они называются обработчиками выхода и регистрируются с помощью функции `atexit`.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

Это объявление говорит о том, что функции `atexit` передается аргумент с адресом функции обработчика. Функции-обработчику при вызове не передается никаких аргументов и от нее не ожидается возврата значения. Функция `exit` вызывает обработчики в последовательности, обратной порядку их регистрации. Каждый обработчик вызывается столько раз, сколько раз он был зарегистрирован.

Обработчики выхода впервые появились в стандарте ANSI C в 1989 году. Системы, предшествовавшие этому стандарту, такие как SVR3 и 4.3BSD, не поддерживали обработку выхода.

Стандарт ISO C требует, чтобы системы поддерживали возможность регистрации как минимум 32 обработчиков, но реализации обычно поддерживают большие (см. табл. 2.13). Максимально возможное количество обработчиков для заданной системы можно определить с помощью функции `sysconf` (см. табл. 2.11).

В соответствии со стандартами ISO C и POSIX.1 функция `exit` сначала должна вызвать все зарегистрированные функции-обработчики и затем закрыть все открытые потоки ввода/вывода (с помощью функции `fclose`). Стандарт POSIX.1 расширил положения ISO C, указав, что регистрация всех функций-обработчиков аннулируется, если процесс вызывает одну из функций семейства `exec`. На рис. 7.1 показано, как запускается и завершается программа, написанная на языке C.

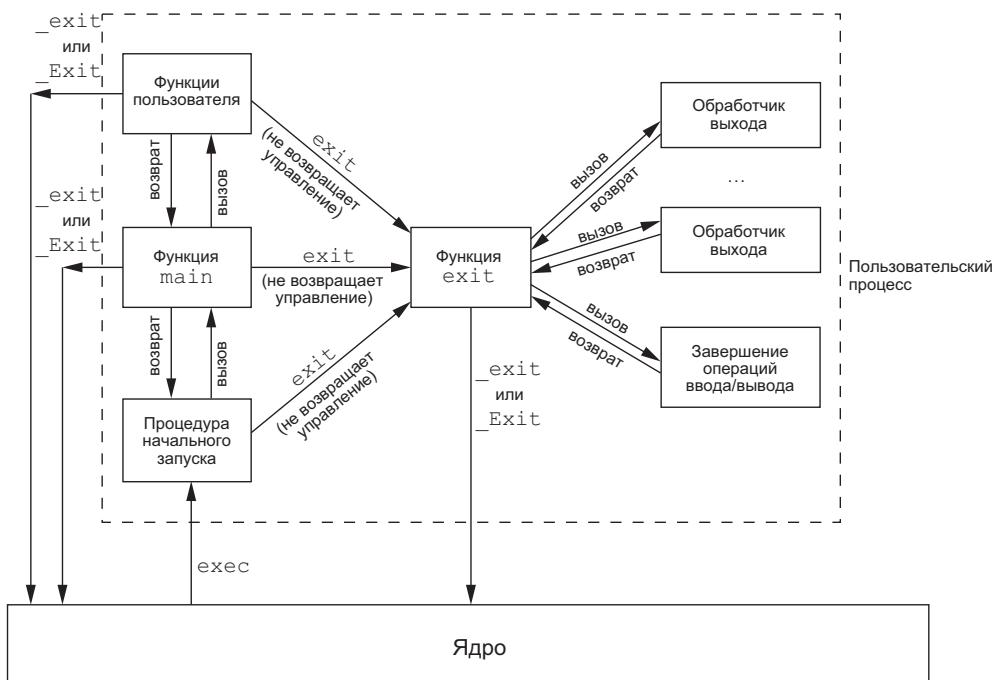


Рис. 7.1. Процесс запуска и завершения программы, написанной на языке C

Ядро может запустить программу единственным способом — через вызов одной из функций семейства `exec`. Процесс может добровольно завершиться только через вызов функции `_exit` или `_Exit`, явно или неявно (с помощью функции `exit`). Процесс может также непреднамеренно прекратить работу по сигналу (что не отражено на рис. 7.1).

Пример

Программа в листинге 7.2 демонстрирует использование функции `atexit`.

Листинг 7.2. Пример использования обработчиков выхода

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("невозможно зарегистрировать my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("невозможно зарегистрировать my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("невозможно зарегистрировать my_exit1");

    printf("функция main завершила работу\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("первый обработчик выхода\n");
}

static void
my_exit2(void)
{
    printf("второй обработчик выхода\n");
}
```

У нас эта программа дала следующие результаты:

```
$ ./a.out
функция main завершила работу
первый обработчик выхода
первый обработчик выхода
второй обработчик выхода
```

Обработчик выхода вызывается столько раз, сколько раз был зарегистрирован. В программе из листинга 7.2 первый обработчик регистрируется дважды, поэтому он был вызван два раза. Обратите внимание, что здесь функция `exit` не вызывается, вместо этого выполняется оператор `return`.

7.4. Аргументы командной строки

Процесс, запускающий программу вызовом функции `exec`, может передать ей аргументы командной строки. Это обычная практика для командных оболочек UNIX, как мы уже видели на многочисленных примерах из предыдущих глав.

Пример

Программа в листинге 7.3 выводит все аргументы командной строки в стандартный вывод. Обратите внимание, что стандартная утилита `echo(1)` не выводит нулевой аргумент.

Листинг 7.3. Вывод всех аргументов командной строки

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     i;

    for (i = 0; i < argc; i++) /* вывести все аргументы командной строки */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Если скомпилировать эту программу и дать выполняемому файлу имя echoarg, ее можно запустить, как показано ниже

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

Согласно стандартам ISO C и POSIX.1, элемент массива `argv[argc]` должен быть представлен пустым указателем. Учитывая это, цикл обработки аргументов командной строки можно оформить иначе:

```
for (i = 0; argv[i] != NULL; i++)
```

7.5. Список переменных окружения

Каждой программе, помимо аргументов командной строки, передается также список переменных окружения. Подобно списку аргументов командной строки, список переменных окружения доступен как массив указателей, каждый из которых указывает на строку, завершающуюся нулевым символом. Адрес массива указателей хранится в глобальной переменной `environ`:

```
extern char **environ;
```

Например, среда окружения, насчитывающая пять переменных, будет похожа на то, что изображено на рис. 7.2. Здесь явно показаны нулевые символы, которыми завершаются строки. Переменная `environ` отмечена как *указатель на среду окружения*, массив указателей — как *список среды окружения*, а строки, на которые они указывают, — как *строки окружения*.

В соответствии с принятыми соглашениями окружение состоит из строк в формате (рис. 7.2):

```
name=value
```

Большинство предопределенных имен *name* состоят из символов верхнего регистра, но это всего лишь традиция.

Исторически в большинстве версий UNIX функции `main` передается третий аргумент — указатель на среду окружения:

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



Рис. 7.2. Процесс запуска и завершения программы, написанной на языке С

Стандарт ISO C определяет только два аргумента функции `main`, а передача среды окружения через третий аргумент не дает никаких преимуществ перед передачей той же информации через глобальную переменную `environ`. Поэтому стандарт POSIX.1 указывает, что передача среды окружения должна осуществляться не через третий аргумент функции `main`, а через глобальную переменную `environ` (если это возможно). Доступ к конкретным переменным окружения обычно осуществляется с помощью функций `getenv` и `putenv`, которые будут описаны в разделе 7.9, а не через глобальную переменную `environ`. Однако для просмотра всех переменных окружения следует использовать указатель `environ`.

7.6. Организация памяти программы на языке С

Традиционно программы на языке С всегда состояли из следующих частей:

- Сегмент кода с машинными инструкциями, которые выполняются центральным процессором. Обычно сегмент кода является разделяемым, чтобы для часто используемых программ, таких как текстовые редакторы, компиляторы языка С, командные оболочки и некоторые другие, в памяти находилась только одна копия сегмента. Кроме того, сегмент кода часто доступен только для чтения, чтобы предотвратить возможность случайного изменения машинных инструкций в нем.
- Сегмент инициализированных данных, который обычно называют просто сегментом данных. Содержит переменные, инициализированные определенными значениями в тексте программы. Например, если где-либо за пределами функции имеется объявление

```
int maxcount = 99;
```

указанная переменная будет сохранена вместе со своим значением в сегменте инициализированных данных.

- Сегмент неинициализированных данных, часто называемый сегментом «bss». Это название происходит от древнего оператора языка ассемблера, который расшифровывается как «block started by symbol» (блок, начинающийся с символа). Перед запуском программы ядро инициализирует данные в этом сегменте арифметическим нулем или нулевыми указателями. Если где-либо за пределами функции имеется объявление

```
long sum[1000];
```

указанная переменная будет сохранена в сегменте неинициализированных данных.

- Сегмент стека (stack), где хранятся переменные с автоматическим классом размещения, а также информация, которая сохраняется при каждом вызове функции. Каждый раз, когда вызывается функция, в стеке сохраняется адрес возврата из нее и определенная информация об окружении вызывающей программы, например регистры процессора. После этого вызванная функция резервирует на стеке дополнительное место для автоматических и временных переменных. Благодаря такой организации в языке C возможны рекурсивные вызовы функций. Всякий раз, когда функция рекурсивно вызывает сама себя, создается новый кадр стека, благодаря чему один набор локальных переменных не накладывается на другой.
- Куча (heap), или область динамической памяти. Традиционно куча располагалась в пространстве между сегментом неинициализированных данных и стеком.

На рис. 7.3 показано типичное размещение этих сегментов. Это логическое представление, как выглядит программа; в конкретной системе организация памяти программы не обязательно будет выглядеть именно так. Тем не менее этот рисунок показывает типичный пример организации памяти, которую мы будем обсуждать. В 32-разрядных версиях Linux для микропроцессоров Intel x86 сегмент кода начинается с адреса `0x8048000`, а дно стека расположено ниже адреса `0xC0000000` (на данной аппаратной архитектуре стек растет вниз — от старших адресов к младшим). Неиспользуемое виртуальное адресное пространство между вершиной кучи и вершиной стека очень велико.

В выполняемом файле `a.out` существует несколько сегментов дополнительных типов с таблицей идентификаторов, информацией для отладчика, таблицей связи с динамическими библиотеками и т. п. Эти дополнительные сегменты не загружаются как часть образа программы, выполняемой процессом.

Обратите внимание, что сегмент неинициализированных данных на рис. 7.3 не хранится в файле программы на диске, так как ядро очищает этот сегмент, прежде чем запустить программу. Единственные сегменты, которые должны быть сохранены в файле программы, — это сегмент кода и сегмент инициализированных данных.

Команда `size(1)` выводит размеры (в байтах) сегментов кода, данных и bss. Например:

```
$ size /usr/bin/cc /bin/sh
   text      data      bss      dec      hex    filename
346919       3576     6680   357175   57337  /usr/bin/cc
102134      1776    11272   115182   1c1ee  /bin/sh
```

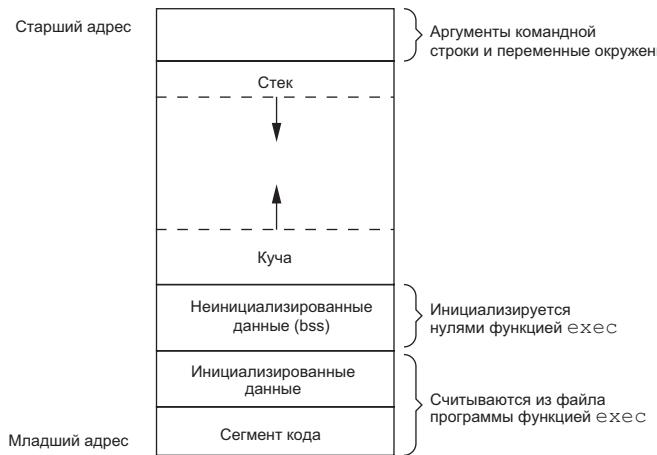


Рис. 7.3. Типичное размещение сегментов в памяти

В четвертой и пятой колонках выводится общий размер всех трех сегментов в десятичном и шестнадцатеричном представлении соответственно.

7.7. Разделяемые библиотеки

Большинство современных версий UNIX поддерживает разделяемые библиотеки. В [Arnold, 1986] описана ранняя реализация таких библиотек в System V, в [Gingel et al., 1987] — реализация в SunOS. Разделяемые библиотеки позволяют изъять из выполняемого файла библиотечные функции; в результате в памяти системы хранится единственная копия библиотеки, к которой обращаются все процессы. Это заметно уменьшает размер выполняемых файлов, но может несколько увеличить нагрузку, когда приложение запускается в первый раз или когда происходит первое обращение к библиотечной функции. Еще одно преимущество разделяемых библиотек в том, что при обновлении библиотеки не требуется исправлять связи с библиотекой в каждой программе, которая использует эту библиотеку. (Здесь мы исходим из предположения, что количество и типы аргументов библиотечных функций не изменились.)

Разные системы предоставляют программам различные способы заявить, что они используют разделяемые библиотеки. Наиболее типичным является передача параметров командам `cc(1)` и `ld(1)`. Для демонстрации различий в размерах попробуем собрать выполняемый файл — классическую программу `hello.c` — сначала без разделяемых библиотек:

```
$ gcc -static hello1.c      укажем явно, что разделяемые библиотеки
                               не должны использоваться
$ ls -l a.out
-rwxr-xr-x 1 sar 879443 Sep 2 10:39 a.out
```

```
$ size a.out
text      data      bss      dec      hex filename
787775     6128    11272   805175   c4937  a.out
```

Если теперь скомпилировать программу с поддержкой разделяемых библиотек, размеры сегмента кода и данных существенно уменьшатся:

```
$ gcc hello1.c           по умолчанию gcc использует разделяемые библиотеки
$ ls -l a.out
-rwxr-xr-x 1 sar        8378 Sep 2 10:39 a.out
$ size a.out
text      data      bss      dec      hex filename
1176      504       16     1696     6a0  a.out
```

7.8. Распределение памяти

Стандарт ISO C определяет три функции распределения памяти.

- Функция `malloc` выделяет заданное количество байтов памяти. Выделенная память не очищается.
- Функция `calloc` выделяет пространство для заданного количества объектов определенного размера. Выделенная память заполняется нулевыми байтами.
- Функция `realloc` перераспределяет выделенную ранее память, увеличивая или уменьшая ее объем. Увеличение выделенного ранее объема может сопровождаться перемещением участка памяти в новое место. Кроме того, участок памяти, который оказывается между концом ранее выделенного блока и новым концом, не инициализируется.

```
#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

Все три возвращают непустой указатель в случае успеха,
NULL — в случае ошибки

```
void free(void *ptr);
```

Эти три функции гарантируют возврат указателей с гарантированным выравниванием, подходящим для сохранения любого объекта данных. Например, если самым сильным ограничением в конкретной системе является требование размещать объекты типа `double` в адресах, кратных 8, все указатели, возвращаемые этими функциями, будут содержать адреса, кратные 8.

Все три функции возвращают нетипизированный указатель `void*`, поэтому, подключая к программе заголовочный файл `<stdlib.h>` (где находятся прототипы функций), мы не должны выполнять явное приведение типов при присваивании

значений функций типизированным указателям. По умолчанию считается, что необъявленные функции возвращают значения типа `int`, поэтому приведение типа без соответствующего объявления функции может привести к ошибкам в системах, где размер типа `int` отличается от размера значения, фактически возвращаемого функцией (в данном случае – указателя).

Функция `free` освобождает выделенную ранее память, на которую указывает аргумент `ptr`. Освобожденное пространство, как правило, помещается в пул свободной памяти и может быть снова распределено при последующем обращении к одной из трех функций `alloc`.

Функция `realloc` позволяет увеличить или уменьшить размер ранее выделенной области памяти. (Чаще производится увеличение.) Например, если выделяется участок памяти для массива из 512 элементов и затем в процессе его заполнения вдруг обнаруживается, что потребуется память для хранения более 512 элементов, можно вызвать функцию `realloc`. В этом случае, если непосредственно после выделенной ранее области имеется блок свободной памяти достаточного объема, функция `realloc` ничего никуда не переместит, а просто добавит область требуемого объема в конец и вернет тот же самый указатель, который был ей передан. Но если после выделенной ранее области нет свободной памяти достаточного объема, функция `realloc` выделит другую область памяти требуемого объема и скопирует существующий массив из 512 элементов в новое место, после чего освободит старую область памяти и вернет указатель на новую область. Поскольку ранее выделенный объем памяти может перемещаться, не следует использовать другие указатели на эту область. Упражнение 4.16 и программа в листинге C2 демонстрируют, как можно использовать функцию `realloc` вместе с `getcwd` для обработки имен файлов любой длины. В листинге 17.28 приводится пример использования функции `realloc` для хранения динамических массивов, что позволяет не указывать их размер во время компиляции.

Обратите внимание, что последний аргумент функции `realloc` определяет новый размер требуемой области, а не разницу между новым и старым размерами. В особом случае, когда в аргументе `ptr` передается пустой указатель, `realloc` действует как функция `malloc` и выделяет область размером `newsize`.

Ранние версии этих функций позволяли снова получить с помощью функции `realloc` блок, освобожденный функцией `free` после последнего обращения к функциям `malloc`, `realloc` или `calloc`. Эта хитрость существовала еще в Version 7 и использовала свойство стратегии поиска, реализованной в функции `malloc` для уплотнения памяти. В Solaris эта особенность сохранилась и поныне, но в других системах – нет. Она не документирована и не должна использоваться.

Функции распределения памяти обычно реализуются на основе системного вызова `sbrk(2)`. Он расширяет (или усекает) область динамической памяти (кучи) процесса (см. рис. 7.3). Пример типичной реализации функций `malloc` и `free` приводится в разделе 8.7 [Kernighan and Ritchie, 1988].

Хотя системный вызов `sbrk(2)` может не только увеличивать, но и уменьшать объем памяти процесса, большинство версий `malloc` и `free` никогда не уменьшают его. Освобождаемое пространство становится доступным для последующего рас-

пределения и, как правило, не возвращается ядру, а помещается в пул свободной памяти функции `malloc`.

Важно понимать, что большинство реализаций выделяют несколько больший объем памяти, чем требуется, и используют дополнительное пространство для хранения служебной информации: размер распределенного блока, указатель на следующий распределенный блок и т. п. Это означает, что запись за пределы выделенной области может уничтожить служебную информацию в следующем блоке. Подобного рода ошибки часто носят катастрофический характер и найти их чрезвычайно трудно, потому что они могут не проявлять себя достаточно длительное время.

Кроме того, существует возможность уничтожить служебную информацию в блоке памяти, если записать данные перед началом распределенной области. Запись за пределы выделенного блока памяти может уничтожить не только служебную информацию. Память до и после такого блока может использоваться для хранения других динамических объектов. Эти объекты могут быть не связаны с разрушающим их участком программы, что еще больше осложняет поиск источника повреждений.

Другие возможные ошибки, которые могут оказаться фатальными, — попытка освобождения блока памяти, уже освобожденного ранее, и передача функции `free` указателя, который не был получен от одной из трех функций распределения памяти. Если процесс вызывает функцию `malloc`, но забывает вызвать функцию `free`, объем используемой памяти начинает непрерывно увеличиваться; это называют утечкой памяти. Если процесс не будет возвращать ставшие ненужными блоки памяти вызовом `free`, объем адресного пространства, занимаемого процессом, будет медленно увеличиваться, пока свободное пространство не закончится. Это может привести к снижению производительности системы из-за лишних обращений к файлу подкачки.

Поскольку ошибки, связанные с распределением памяти, отыскать очень сложно, некоторые системы предоставляют версии функций распределения памяти, выполняющие дополнительные проверки при каждом вызове. Эти версии функций часто характеризуются включением специальной библиотеки редактора связей. Кроме того, существуют общедоступные исходные тексты, которые можно скомпилировать со специальными флагами, разрешающими проведение дополнительных проверок во время выполнения.

Операционные системы FreeBSD, Mac OS X и Linux поддерживают дополнительные возможности отладки через установку переменных среды. Кроме того, библиотеке FreeBSD можно передать дополнительные параметры через символическую ссылку /etc/malloc.conf.

Альтернативные функции распределения памяти

Существует большое количество функций, которые могут служить заменой для `malloc` и `free`. Некоторые системы уже включают библиотеки с альтернативными реализациями функций распределения памяти. Другие системы поддерживают только стандартные функции, оставляя программистам право скачивать и ис-

пользовать альтернативные библиотеки, если они того пожелают. Здесь мы упомянем некоторые из альтернатив.

libmalloc

Системы, основанные на SVR4, такие как Solaris, включают библиотеку **libmalloc**, поддерживающую ряд интерфейсов, соответствующих функциям распределения памяти стандарта ISO C. Библиотека **libmalloc** включает функцию **mallopt**, позволяющую процессу установить специальные переменные, управляющие поведением функций распределения памяти. Кроме того, в библиотеке имеется функция **mallinfo**, с помощью которой можно получить статистику по функциям распределения памяти.

vmalloc

В [Vo, 1996] описывается библиотека функций распределения памяти, которая позволяет использовать различные приемы для различных областей памяти. В дополнение к специфичным функциям библиотека **vmalloc** предоставляет функции, эмулирующие функции распределения памяти стандарта ISO C.

quick-fit

Традиционно в качестве стандартного алгоритма выделения памяти используется либо метод наилучшего приближения (best-fit), либо метод первого подходящего (first-fit). Алгоритм quick-fit (быстрого приближения) превосходит по скорости любой из них, но использует больше памяти. Описание этого алгоритма можно найти в [Weinstock and Wulf, 1988]. В его основе лежит принцип разделения памяти на блоки разных размеров и размещения их в различных списках свободных блоков в зависимости от размера. Большинство современных реализаций функций управления памятью основаны на алгоритме quick-fit.

jemalloc

jemalloc — это реализация семейства библиотечных функций **malloc**, используемая по умолчанию в FreeBSD 8.0. Она хорошо масштабируется при использовании в многопоточных приложениях, при выполнении в многопроцессорных системах. Описание и оценку производительности этой реализации можно найти в [Evans, 2006].

TCMalloc

Библиотека **TCMalloc** проектировалась как замена семейству функций **malloc** с целью обеспечить более высокую производительность, масштабируемость и эффективность использования памяти. Чтобы избежать лишних накладных расходов на использование механизма блокировки, она использует локальные пулы памяти для потоков выполнения, из которых производится выделение памяти и куда возвращается освобождаемая память. Она также включает встроенные инструменты проверки и профилирования динамической памяти с целью помочь в отладке и анализе особенностей использования динамической памяти. Библиотека **TCMalloc** распространяется компанией Google в исходных текстах. Краткое ее описание можно найти в [Ghemawat and Menage, 2005].

Функция `alloc`

Это еще одна функция, которая заслуживает внимания. Функция `alloc` вызывается точно так же, как функция `malloc`, но выделяет память не в куче, а в кадре стека текущей функции. Преимущество такого выделения памяти заключается в отсутствии необходимости освобождать выделенное пространство — это происходит автоматически после выхода из функции. Функция `alloc` увеличивает размер кадра стека. Главный ее недостаток — она не может использоваться в системах, где невозможно увеличить кадр стека после вызова функции. Тем не менее она используется во многих программных пакетах, и существуют ее реализации для большого количества систем.

Все четыре платформы, обсуждаемые в этой книге, поддерживают функцию `alloc`.

7.9. Переменные окружения

Как уже говорилось выше, строка окружения обычно имеет формат

name=value

Ядро UNIX никогда не обращается к этим строкам; их интерпретация полностью зависит от самих приложений. Так, например, командные оболочки используют в своей работе многочисленные переменные окружения. Некоторые из них, такие как `HOME` и `USER`, устанавливаются автоматически при входе в систему, другие определяются пользователем. Обычно инициализация переменных окружения производится в файле начального запуска командной оболочки. Если, например, установить переменную среды окружения `MAILPATH`, она будет сообщать командным оболочкам Bourne shell, GNU Bourne-again shell и Korn shell имя каталога, в котором хранится электронная почта.

Стандарт ISO C определяет функцию для получения значения любой переменной окружения, но оговаривает, что содержимое среды окружения зависит от реализации.

```
#include <stdlib.h>
char *getenv(const char *name);
```

Возвращает указатель на значение переменной с именем *name* или NULL, если переменная не найдена

Обратите внимание, что эта функция возвращает указатель на подстроку *value* в строке *name=value*. Когда нужно получить значение конкретной переменной окружения, всегда следует использовать функцию `getenv` вместо прямого обращения к массиву `environ`.

Некоторые переменные окружения в Single UNIX Specification определяются стандартом POSIX.1, тогда как другие определены только в системах, которые поддерживают расширения XSI. В табл. 7.1 перечислены переменные окружения, определяемые в Single UNIX Specification, а также отмечено, какими реализа-

циями они поддерживаются. Переменные окружения, определяемые стандартом POSIX.1, отмечены галочкой, остальные являются расширениями XSI. В четырех реализациях, обсуждаемых в данной книге, поддерживаются много дополнительных переменных окружения. Обратите внимание, что стандарт ISO C не определяет никаких переменных окружения.

Таблица 7.1. Переменные окружения, определяемые стандартом Single UNIX Specification

Переменная	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
COLUMNS	✓	✓	✓	✓	✓	Ширина строки терминала
DATEMSK	XSI		✓	✓	✓	Полный путь к файлу шаблона для функции <code>getdate(3)</code>
HOME	✓	✓	✓	✓	✓	Домашний каталог
LANG	✓	✓	✓	✓	✓	Название локали (региональных настроек)
LC_ALL	✓	✓	✓	✓	✓	Название локали (региональных настроек)
LC_COLLATE	✓	✓	✓	✓	✓	Название локали (региональных настроек) для выполнения сравнения
LC_CTYPE	✓	✓	✓	✓	✓	Название локали (региональных настроек) для классификации символов языка
LC_MESSAGES	✓	✓	✓	✓	✓	Название локали (региональных настроек) для вывода сообщений
LC_MONETARY	✓	✓	✓	✓	✓	Название локали (региональных настроек) для представления денежных величин
LC_NUMERIC	✓	✓	✓	✓	✓	Название локали (региональных настроек) для представления чисел
LC_TIME	✓	✓	✓	✓	✓	Название локали (региональных настроек) для форматирования даты и времени
LINES	✓	✓	✓	✓	✓	Количество строк терминала
LOGNAME	✓	✓	✓	✓	✓	Имя пользователя
MSGVERB	XSI	✓	✓	✓	✓	Определяет компонент сообщения для вывода функцией <code>fmtmsg(3)</code>
NLSPATH	✓	✓	✓	✓	✓	Шаблон имени каталога с сообщениями
PATH	✓	✓	✓	✓	✓	Список каталогов для поиска выполняемых файлов

Таблица 7.1 (окончание)

Переменная	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
PWD	✓	✓	✓	✓	✓	Абсолютный путь к текущему каталогу
SHELL	✓	✓	✓	✓	✓	Имя командной оболочки, предпочтаемой пользователем
TERM	✓	✓	✓	✓	✓	Тип терминала
TMPDIR	✓	✓	✓	✓	✓	Путь к каталогу для временных файлов
TZ	✓	✓	✓	✓	✓	Информация о часовом поясе

Иногда может потребоваться не только получить, но и изменить значение существующей переменной или даже добавить новую. (В следующей главе мы посмотрим, как оказывать влияние на среду окружения текущего процесса и его потомков. Мы не можем изменить среду окружения родительского процесса, который зачастую является командной оболочкой. Тем не менее было бы удобно иметь возможность изменять среду окружения текущего процесса.) К сожалению, не все системы поддерживают эту возможность. В табл. 7.2 приводится список функций, которые поддерживаются различными стандартами и реализациями.

Таблица 7.2. Различные функции для работы со средой окружения

Функция	ISO C	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
getenv	✓	✓	✓	✓	✓	✓
putenv		XSI	✓	✓	✓	✓
setenv		✓	✓	✓	✓	
unsetenv		✓	✓	✓	✓	
clearenv				✓		

Функция `clearenv` не входит в стандарт Single UNIX Specification. Она используется для удаления всех записей из списка строк окружения.

Ниже приводятся прототипы второй, третьей и четвертой функций из табл. 7.2.

```
#include <stdlib.h>
int putenv(char *str);
```

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
int setenv(const char *name, const char *value, int rewrite);
```

```
int unsetenv(const char *name);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Действуют эти функции следующим образом.

- Функция `putenv` принимает строку в формате `name=value` и помещает ее в список переменных окружения. Если переменная с именем `name` уже существует, она будет удалена перед вставкой новой строки.
- Функция `setenv` присваивает переменной `name` значение `value`. Если переменная `name` уже существует в среде окружения, тогда: (а) если аргумент `rewrite` не равен нулю, существующее определение переменной сначала удаляется из списка; (б) если аргумент `rewrite` равен нулю, существующее определение переменной не удаляется, новое значение `value` не запоминается и функция возвращает управление без признака ошибки.
- Функция `unsetenv` удаляет определение переменной с именем `name`. Если заданной переменной не существует, это не считается ошибкой.

Обратите внимание на различия между функциями `putenv` и `setenv`. Функция `setenv` выделяет память, чтобы создать строку `name=value` из своих аргументов, а `putenv` просто вставляет переданную строку непосредственно в список переменных окружения. Многие реализации действуют именно так, поэтому было бы ошибкой передавать функции `putenv` строки, размещенные в стеке, так как память стека еще не раз будет использована после того, как текущая функция вернет управление в вызывающую программу.

Было бы интересно узнать, какие действия выполняются этими функциями при изменении списка переменных окружения. Вспомните рис. 7.3: список переменных окружения — это массив указателей на строки в формате `name=value`, и эти строки обычно хранятся в верхней части адресного пространства процесса — над стеком. Операция удаления строки выглядит достаточно просто: нужный указатель отыскивается в массиве и все последующие указатели перемещаются на один вниз. Но операция добавления новой строки или изменения существующей оказывается более сложной. Пространство над стеком не может быть раздвинуто, потому что эта область часто находится в старших адресах адресного пространства процесса и не может расти вверх; она также не может расти вниз, потому что ниже находится стек, который не может быть перемещен.

1. Если необходимо изменить значение существующей переменной `name`:
 - а) если длина подстроки `value` меньше или равна существующей, новая подстрока просто копируется поверх существующей;
 - б) если длина подстроки `value` больше существующей, необходимо выделить память для новой строки с помощью функции `malloc`, скопировать туда новую строку и затем заменить в массиве старый указатель на новый.
2. Если добавляется новая переменная, требуются более сложные действия. Прежде всего нужно вызвать функцию `malloc`, выделить память для строки в формате `name=value` и скопировать ее туда:
 - а) затем, если новая переменная добавляется впервые, следует опять вызвать `malloc` для выделения памяти под новый массив указателей, скопировать список указателей в новое место, добавить к нему указатель на новую строку и, разумеется, поместить в конец списка пустой указатель. Наконец, записать в переменную `environ` адрес нового списка. Обратите внимание:

если изначально список переменных окружения размещался выше стека (рис. 7.3), то теперь он переместится в область динамической памяти (в кучу). Однако большинство указателей в этом списке по-прежнему будут указывать на строки, размещенные выше стека;

- 6) если добавление переменной производится не в первый раз, мы уже знаем, что список указателей располагается в куче. Поэтому нужно вызвать `realloc`, чтобы выделить место для еще одного указателя, добавить в список новую строку `name=value` (на место пустого указателя) и затем записать пустой указатель в самый конец списка.

7.10. Функции `setjmp` и `longjmp`

В языке С нельзя выполнить безусловный переход (оператор `goto`) к метке в теле другой функции. В таких случаях необходимо использовать функции `setjmp` и `longjmp`. Позже мы увидим, что эти функции очень удобны для обработки ошибочных ситуаций, когда ошибка происходит в глубоко вложенном вызове.

Рассмотрим программу-заготовку в листинге 7.4. Здесь имеется главный цикл, который читает строки со стандартного ввода и для обработки каждой вызывает функцию `do_line`. Эта функция, в свою очередь, обращается к функции `get_token`, которая выбирает из входной строки очередную лексему. Предполагается, что первая лексема строки является некоторой командой, и оператор `switch` определяет, как обрабатывать каждую из команд. Для единственной показанной здесь команды вызывается функция `cmd_add`.

Программа в листинге 7.4 представляет собой типичную заготовку для программ, которые читают команды, определяют их тип и затем вызывают функции, соответствующие командам. На рис. 7.4 показано, как мог бы выглядеть стек после вызова функции `cmd_add`.

Листинг 7.4. Типичная заготовка программы обработки команд

```
#include "apue.h"

#define TOK_ADD      5

void      do_line(char *);
void      cmd_add(void);
int       get_token(void);

int
main(void)
{
    char    line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char *tok_ptr; /* глобальный указатель для get_token() */
```

```

void
do_line(char *ptr) /* обработка одной строки ввода */
{
    int      cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) { /* для каждой команды свой оператор case */
        case TOK_ADD:
            cmd_add();
            break;
        }
    }
}

void
cmd_add(void)
{
    int      token;

    token = get_token();
    /* остальные действия по обработке этой команды */
}

int
get_token(void)
{
    /* получить очередную лексему из строки, на которую указывает tok_ptr */
}

```

Переменные с автоматическим классом размещения хранятся в пределах кадров стека каждой из функций. Массив `line` хранится в кадре стека функции `main`, целочисленная переменная `cmd` — в кадре стека функции `do_line`, целочисленная переменная `token` — в кадре стека функции `cmd_add`.

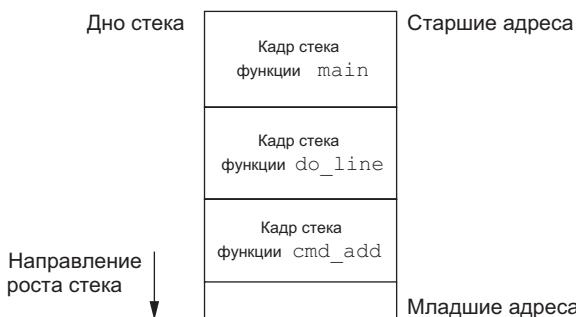


Рис. 7.4. Состояние стека после вызова функции `cmd_add`

Как уже говорилось, такая организация стека достаточно типична, но необязательна. Стеки не обязательно должны расти в направлении младших адресов памяти. В системах, не имеющих аппаратной поддержки механизма стека, его реализация на языке С могла бы использовать связанный список кадров стека.

При разработке программ, подобных представленной в листинге 7.4, часто возникает проблема обработки нефатальных ошибок. Например, если функция `cmd_add` встречает ошибку, скажем недопустимое число, может потребоваться вывести сообщение об ошибке, проигнорировать остальную часть входной строки и вернуться в функцию `main`, чтобы перейти к обработке следующей строки. Но когда ошибка возникает в глубоко вложенной функции, сделать это на С достаточно трудно. (В этом примере функция `cmd_add` находится на втором уровне вложенности относительно функции `main`, но часто точка, из которой требуется вернуться, находится на пятом уровне вложенности и даже глубже.) Если в каждую функцию добавлять код, который будет возвращать признак ошибки на один уровень вверх, исходные тексты станут неудобочитаемыми.

Решение этой проблемы заключается в использовании нелокальных переходов: функций `setjmp` и `longjmp`. Определение «нелокальный» означает, что мы не используем обычный оператор перехода языка С, вместо этого мы выполняем обратный переход через кадры стека к некоторой функции, которая находится в цепочке вызовов на пути к текущей функции.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Возвращает 0, если вызвана непосредственно, или ненулевое значение,
если возврат произошел в результате обращения к функции `longjmp`

```
void longjmp(jmp_buf env, int val);
```

Функция `setjmp` вызывается из точки, куда требуется вернуться; в данном примере она находится в функции `main`. В этом случае `setjmp` возвращает 0, потому что это непосредственный вызов функции. Аргумент `env` функции `setjmp` имеет специальный тип `jmp_buf`. Этот тип данных — своего рода массив, который может хранить информацию, необходимую для восстановления состояния стека, когда будет произведен вызов функции `longjmp`. Обычно переменная `env` является глобальной, так как она должна быть доступна из других функций.

Когда возникает ошибка, например, в функции `cmd_add`, мы вызываем `longjmp` с двумя аргументами. Первый — тот самый `env`, который использовался при обращении к `setjmp`, а второй — `val`, значение, отличное от нуля, которое становится возвращаемым значением функции `setjmp`. Второй аргумент позволяет вызывать `longjmp` более одного раза для каждого `setjmp`. Например, можно выполнить переход с помощью `longjmp` из `cmd_add` со значением аргумента `val`, равным 1, а из `get_token` — со значением `val`, равным 2. В таком случае `setjmp` в функции `main` будет возвращать либо 1, либо 2, что позволит определить, откуда произведен переход — из `cmd_add` или из `get_token`.

Теперь вернемся к нашему примеру. В листинге 7.5 приводятся функции `cmd_add` и `main`. (Две другие функции, `do_line` и `get_token`, остались без изменений.)

Листинг 7.5. Пример использования функций setjmp и longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD      5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("ошибка");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}
...
void
cmd_add(void)
{
    int    token;

    token = get_token();
    if (token < 0) /* проверка наличия ошибки */
        longjmp(jmpbuffer, 1);
    /* остальные действия по обработке этой команды */
}
```

Когда начинается выполнение функции `main`, функция `setjmp` записывает всю необходимую информацию в переменную `jmpbuffer` и возвращает 0. Затем вызывается функция `do_line`, которая, в свою очередь, вызывает функцию `cmd_add`. Теперь предположим, что была обнаружена некая ошибка. Перед вызовом функции `longjmp` из `cmd_add` стек выглядит, как показано на рис. 7.4. Функция `longjmp` «раскручивает» стек в обратную сторону — до кадра функции `main`, выбрасывая кадры, созданные во время вызова функций `cmd_add` и `do_line` (рис. 7.5). В результате вызова функции `longjmp` происходит возврат из функции `setjmp` в `main`, но на этот раз возвращаемое значение равно 1 (второй параметр `longjmp`).

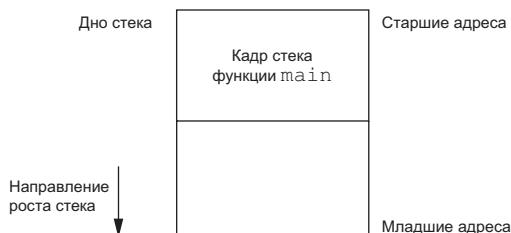


Рис. 7.5. Состояние стека после вызова функции longjmp

Переменные с классами размещения register, automatic и volatile

Мы увидели, как выглядит стек после вызова функции `longjmp`. Следующий вопрос, на который предстоит ответить: каково состояние автоматических и регистраховых переменных в функции `main`? Когда в результате вызова функции `longjmp` управление возвращается в функцию `main`, получают ли эти переменные значения, которые они имели на момент вызова функции `setjmp` (то есть «откручиваются» ли их значения назад) или их значения остаются без изменения с момента вызова функции `do_line` (вызвавшей функцию `cmd_add`, которая, в свою очередь, вызвала функцию `longjmp`)? К сожалению, ответ на этот вопрос: «Зависит от реализации». Большинство реализаций не «откручивают» назад автоматические и регистраховые переменные, а стандарты утверждают, что их значения в этом случае не определены. Если у вас есть автоматические переменные, значения которых не должны «откручиваться» назад, определите их со спецификатором `volatile`. Вызов функции `longjmp` не оказывает влияния на глобальные или статические переменные.

Пример

Программа в листинге 7.6 демонстрирует различия в поведении автоматических, регистраховых, глобальных, статических и `volatile`-переменных, наблюдаемые после вызова функции `longjmp`.

Листинг 7.6. Влияние `longjmp` на переменные с различными классами размещения

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int globval;

int
main(void)
{
    int autoval;
    register int regival;
    volatile int volaval;
    static int statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("после вызова longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
               " volaval = %d, statval = %d\n",
               globval, autoval, regival, volaval, statval);
        exit(0);
    }
    /*
```

```

 * Изменить переменные после вызова setjmp, но до вызова longjmp.
 */
globval = 95; autoval = 96; regival = 97; volaval = 98;
statval = 99;

f1(autoval, regival, volaval, statval); /* управление никогда */
                                         /* не вернется в эту точку */
exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("в функции f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
           " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}

```

Если скомпилировать эту программу с включенной оптимизацией и без оптимизации, мы получим разные результаты:

```

$ gcc testjmp.c          скомпилировать без оптимизации
$ ./a.out
в функции f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
после вызова longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ gcc -O testjmp.c      скомпилировать с полной оптимизацией
$ ./a.out
в функции f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
после вызова longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99

```

Обратите внимание, что оптимизация не оказывает влияния на глобальные, статические переменные и на переменные, объявленные со спецификатором `volatile`; после вызова `longjmp` эти переменные сохраняют последние присвоенные им значения. Страница справочного руководства `setjmp(3)` в одной из систем заявляет, что переменные, хранящиеся в памяти, будут иметь те же значения, что и в момент вызова `longjmp`, тогда как переменные в регистрах центрального процессора и арифметического сопроцессора будут восстановлены в состояние, соответствующее первому вызову функции `setjmp`. Это в точности соответствует тому, что мы наблюдали в экспериментах с программой в листинге 7.6. Когда оптимизация отключена, все пять переменных сохраняются в памяти (спецификатор `register` для переменной `regival` игнорируется). Когда оптимизация включена, переменные `autoval` и `regival` перемещаются в регистры (даже при том, что первая из них не была объявлена как `register`), а переменная, объявленная со спецификатором `volatile`, остается в памяти. Из этого примера

следует вывод: если вы пишете переносимый код, выполняющий нелокальные переходы, используйте спецификатор `volatile`. В зависимости от системы могут обнаружиться и другие отличия.

Некоторые строки в листинге 7.6, содержащие обращения к функции `printf`, не умещаются по ширине экрана, что несколько неудобно. Вместо того чтобы много-кратно повторять вызовы `printf`, мы полагаемся на возможность конкатенации строк, предусмотренную стандартом ISO C, когда последовательность

```
"string1" "string2"
```

эквивалентна последовательности

```
"string1string2"
```

Мы еще вернемся к функциям `setjmp` и `longjmp` в главе 10, когда будем обсуждать обработчики сигналов и версии этих функций для работы с сигналами: `sigsetjmp` и `siglongjmp`.

Возможные проблемы с автоматическими переменными

Рассмотрев порядок работы с кадрами стека, мы должны обратить ваше внимание на одну потенциальную ошибку, связанную с автоматическими переменными. Всегда следует придерживаться основного правила — не обращаться к автоматической переменной после того, как функция, в которой она была объявлена, вернула управление. Многочисленные предупреждения об этом встречаются повсюду в справочном руководстве UNIX.

В листинге 7.7 показана функция `open_data`, которая открывает поток ввода/вывода и выполняет настройку режима его буферизации.

Листинг 7.7. Неправильное использование автоматической переменной

```
#include <stdio.h>

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ]; /*setvbuf сделает этот массив буфером ввода/вывода*/

    if ((fp = fopen("datafile", "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp); /* ошибка */
}
```

Проблема в том, что когда функция `open_data` вернет управление, пространство на стеке, которое она использовала, будут отдано под кадр стека следующей вызываемой функции. Однако стандартная библиотека ввода/вывода по-прежнему будет использовать эту часть памяти под буфер потока ввода/вывода. Хаос неминуем. Чтобы избежать этой проблемы, следует разместить массив `databuf` в глобальной памяти, статически (`static` или `extern`) или динамически (с помощью одной из функций распределения памяти).

7.11. Функции `getrlimit` и `setrlimit`

Любой процесс имеет ряд ограничений на использование ресурсов. Некоторые из этих ограничений можно изменить с помощью функций `getrlimit` и `setrlimit`.

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

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Обе возвращают 0 в случае успеха,
–1 – в случае ошибки

Эти две функции определены стандартом Single UNIX Specification как расширения XSI. Ограничения на ресурсы для процесса обычно устанавливаются процессом с идентификатором 0 во время инициализации системы и затем наследуются остальными процессами. Каждая реализация предлагает собственный способ настройки различных ограничений.

При обращении к этим функциям им передается ресурс (*resource*) и указатель на следующую структуру:

```
struct rlimit {
    rlim_t rlim_cur; /* мягкий предел: текущий предел */
    rlim_t rlim_max; /* жесткий предел: максимальное значение для rlim_cur */
};
```

Изменение пределов ресурсов производится в соответствии со следующими тремя правилами.

1. Процесс может изменять значение мягкого предела при условии, что оно не превышает жесткий предел.
2. Процесс может понизить значение жесткого предела вплоть до значения мягкого предела. Операция понижения жесткого предела необратима для рядовых пользователей.
3. Только процесс, обладающий привилегиями суперпользователя, может поднять значение жесткого предела.

Бесконечность предела определяется константой `RLIM_INFINITY`.

В аргументе *resource* передается одно из следующих значений.

RLIMIT_AS Максимальный размер доступной процессу памяти (в байтах). Этот предел оказывает влияние на функции `sbrk` (раздел 1.11) и `mmap` (раздел 14.8).

RLIMIT_CORE Максимальный размер файла дампа памяти (*core*) в байтах. Значение 0 отключает создание таких файлов.

RLIMIT_CPU Максимальное количество процессорного времени в секундах. По достижении мягкого предела процессу будет послан сигнал `SIGXCPU`.

RLIMIT_DATA Максимальный размер сегмента данных в байтах: сумма размеров сегментов инициализированных данных, неинициализированных данных и кучи (см. рис. 7.3).

RLIMIT_FSIZE Максимальный размер создаваемого файла в байтах. По достижении мягкого предела процессу будет послан сигнал **SIGXFSZ**.

RLIMIT_MEMLOCK Максимальный объем памяти в байтах, которую процесс может заблокировать с помощью функции **mlock(2)**.

RLIMIT_MSGQUEUE Максимальный объем памяти в байтах, которую процесс может выделить для очередей сообщений POSIX.

RLIMIT_NICE Максимальный уровень (раздел 8.16), до которого процесс может поднять свой приоритет.

RLIMIT_NOFILE Максимальное количество одновременно открытых файлов. Изменение этого предела оказывает влияние на значение, возвращаемое функцией **sysconf** для аргумента **_SC_OPEN_MAX** (раздел 2.5.4 и листинг 2.4).

RLIMIT_NPROC Максимальное количество дочерних процессов на реальный идентификатор пользователя. Изменение этого предела оказывает влияние на значение, возвращаемое функцией **sysconf** для аргумента **_SC_CHILD_MAX** (раздел 2.5.4).

RLIMIT_NPTS Максимальное количество псевдотерминалов (глава 19), которые могут быть открыты пользователем одновременно.

RLIMIT_RSS Максимальный объем страниц виртуальной памяти процесса, размещаемых резидентно в оперативной памяти, в байтах. Если физической памяти недостаточно, ядро будет «отнимать» память у процессов, которые превысили этот предел.

RLIMIT_SBSIZE Максимальный объем буферов сокетов в байтах, который можно использовать в конкретный момент времени.

RLIMIT_SIGPENDING Максимальное количество сигналов в очереди процесса. Этот предел устанавливается функцией **sigqueue** (раздел 10.20).

RLIMIT_STACK Максимальный размер стека в байтах (см. рис. 7.3).

RLIMIT_SWAP Максимальный объем пространства подкачки в байтах, которое может быть занято пользователем.

RLIMIT_VMEM Синоним **RLIMIT_AS**.

В табл. 7.3 указано, какие ограничения на ресурсы определены стандартом Single UNIX Specification и какие из них поддерживаются каждой из реализаций.

Таблица 7.3. Поддерживаемые ограничения на ресурсы

Предел	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_AS	✓	✓	✓		✓
RLIMIT_CORE	✓	✓	✓	✓	✓
RLIMIT_CPU	✓	✓	✓	✓	✓
RLIMIT_DATA	✓	✓	✓	✓	✓
RLIMIT_FSIZE	✓	✓	✓	✓	✓

Предел	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_MEMLOCK		✓	✓	✓	
RLIMIT_MSGQUEUE			✓		
RLIMIT_NICE			✓		
RLIMIT_NOFILE	✓	✓	✓	✓	✓
RLIMIT_NPROC		✓	✓	✓	
RLIMIT_NPTS		✓			
RLIMIT_RSS		✓	✓	✓	
RLIMIT_SBSIZE		✓			
RLIMIT_SIGPENDING			✓		
RLIMIT_STACK	✓	✓	✓	✓	✓
RLIMIT_SWAP		✓			
RLIMIT_VMEM					✓

Предельные значения для ресурсов оказывают влияние на вызывающий процесс и наследуются всеми его дочерними процессами. Это означает, что команда изменения ограничений на ресурсы должна быть встроена в командную оболочку, чтобы воздействовать на все процессы, запускаемые из нее. И действительно, в командных оболочках Bourne shell, GNU Bourne-again shell и Korn shell имеется встроенная команда `ulimit`, а в командной оболочке C shell — команда `limit`. (Команды `umask` и `chdir` также должны быть встроенными.)

Пример

Программа в листинге 7.8 выводит текущие мягкие и жесткие значения для всех пределов, поддерживаемых системой. Чтобы скомпилировать эту программу в различных реализациях UNIX, мы использовали директивы условной компиляции для подключения заголовочных файлов и обработки констант, имена которых могут различаться в разных системах. Обратите внимание, что некоторые платформы определяют тип `rlim_t` как `unsigned long long` вместо `unsigned long`. Более того, определение может отличаться в одной и той же системе в зависимости от типа платформы, 32- или 64-разрядной. Некоторые пределы определяют размеры файлов, поэтому тип `rlim_t` должен быть достаточно большим, чтобы с его помощью можно было представлять предельные значения размеров файлов. Чтобы избежать предупреждений компилятора о неверных спецификаторах формата в функциях `printf`, мы сначала копируем значение предела в 64-разрядную переменную, чтобы затем использовать единый формат.

Листинг 7.8. Вывод значений пределов ресурсов

```
#include "apue.h"
#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)
static void pr_limits(char *, int);
```

```
int
main(void)
{
#ifndef RLIMIT_AS
    doit(RLIMIT_AS);
#endif

    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifndef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif

#ifndef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
#endif

#ifndef RLIMIT_NICE
    doit(RLIMIT_NICE);
#endif

    doit(RLIMIT_NOFILE);

#ifndef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif

#ifndef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
#endif

#ifndef RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif

#ifndef RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif

#ifndef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
#endif

    doit(RLIMIT_STACK);

#ifndef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
#endif

#ifndef RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif

    exit(0);
}

static void
```

```

pr_limits(char *name, int resource)
{
    struct rlimit      limit;
    unsigned long long lim;

    if (getrlimit(resource, &limit) < 0)
        err_sys("ошибка вызова функции getrlimit для %s", name);
    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY){
        printf("(бесконечность)");
    } else {
        lim = limit.rlim_cur;
        printf("%15lld ", lim);
    }
    if (limit.rlim_max == RLIM_INFINITY){
        printf("(бесконечность)");
    } else {
        lim = limit.rlim_max;
        printf("%15lld", lim);
    }
    putchar((int)'\\n');
}

```

Обратите внимание, как для получения строки с именем ресурса в макросе `doit` мы использовали оператор (#), который предусматривается стандартом ISO C. Поэтому вызов макроса

`doit(RLIMIT_CORE)`

препроцессор C развернет в строку

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Запустив программу в FreeBSD, мы получили следующие результаты:

```
$ ./a.out
RLIMIT_AS      (бесконечность)  (бесконечность)
RLIMIT_CORE    (бесконечность)  (бесконечность)
RLIMIT_CPU     (бесконечность)  (бесконечность)
RLIMIT_DATA    536870912       536870912
RLIMIT_FSIZE   (бесконечность)  (бесконечность)
RLIMIT_MEMLOCK (бесконечность)  (бесконечность)
RLIMIT_NOFILE  3520            3520
RLIMIT_NPROC   1760            1760
RLIMIT_NPTS    (бесконечность)  (бесконечность)
RLIMIT_RSS     (бесконечность)  (бесконечность)
RLIMIT_SBSIZE  (бесконечность)  (бесконечность)
RLIMIT_STACK   67108864        67108864
RLIMIT_SWAP    (бесконечность)  (бесконечность)
RLIMIT_VMEM    (бесконечность)  (бесконечность)
```

В Solaris:

```
$ ./a.out
RLIMIT_AS      (бесконечность)  (бесконечность)
RLIMIT_CORE    (бесконечность)  (бесконечность)
RLIMIT_CPU     (бесконечность)  (бесконечность)
RLIMIT_DATA    (бесконечность)  (бесконечность)
RLIMIT_FSIZE   (бесконечность)  (бесконечность)
RLIMIT_NOFILE  256             65536
RLIMIT_STACK   8388608         (бесконечность)
RLIMIT_VMEM    (бесконечность)  (бесконечность)
```

После знакомства с сигналами мы продолжим обсуждение пределов ресурсов в упражнении 10.11.

7.12. Подведение итогов

Понимание особенностей окружения программ, написанных на С, совершенно необходимо для понимания особенностей управления процессами в UNIX. Мы узнали, как запускается процесс, как он может завершиться и как процессу передаются списки аргументов и переменных окружения. Несмотря на то что ядро никак не анализирует ни тот ни другой список, именно оно передает их новому процессу от программы, вызвавшей `exec`.

Мы также исследовали типичную организацию памяти программ, написанных на С, и коснулись вопроса динамического распределения и освобождения памяти. Детально рассмотрели функции управления окружением, так как они связаны с распределением памяти. Познакомились с функциями `setjmp` и `longjmp`, выполняющими нелокальные переходы в пределах процесса. И в завершение посмотрели, какие ограничения на ресурсы накладывают различные реализации.

Упражнения

- 7.1 Если в Linux на аппаратной архитектуре x86 запустить программу «Привет, МИР!», которая не вызывает функцию `exit` и не использует оператор `return` для выхода из функции `main`, код завершения программы окажется равным 13 (это легко проверить средствами командной оболочки). Почему?
- 7.2 Когда фактически происходит отображение строк, которые выводятся с помощью функции `printf` в листинге 7.2?
- 7.3 Существует ли способ получить доступ к аргументам командной строки из функций, вызываемых из функции `main`, при условии, что (а) аргументы `argc` и `argv` в вызываемую функцию не передаются и (б) их содержимое не копируется в глобальные переменные?
- 7.4 В некоторых реализациях UNIX нулевой адрес в сегменте данных программы преднамеренно делается недоступным. Почему?
- 7.5 Попробуйте определить с помощью `typedef` новый тип данных `Exitfunc` для функции — обработчика выхода. Измените прототип функции `atexit` с использованием этого типа.
- 7.6 Если разместить массив значений типа `long` с помощью функции `calloc`, будут ли элементы массива инициализированы нулевыми значениями? Если разместить массив указателей с помощью функции `calloc`, будут ли элементы массива инициализированы как пустые указатели?
- 7.7 Почему в конце раздела 7.6 мы не получили размеры стека и кучи от команды `size`?

- 7.8** Почему в разделе 7.7 размеры файлов (879 443 и 8378) не совпадают с суммой размеров их сегментов кода и данных?
- 7.9** Почему в разделе 7.7 при использовании разделяемых библиотек получается такая большая разница в размерах файлов такой простенькой программы?
- 7.10** В конце раздела 7.10 мы показали, что функция не может возвращать указатель на локальную переменную с автоматическим классом размещения. Как вы думаете, будет ли следующий код работать корректно?

```
int
f1(int val)
{
    int      num = 0;
    int      *ptr = &num;

    if (val == 0) {
        int val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Управление процессами

8.1. Введение

Теперь перейдем к обсуждению управления процессами в UNIX. Сюда относится создание новых процессов, запуск программ и их завершение. Мы также рассмотрим различные идентификаторы, определяющие принадлежность процесса, — реальные, эффективные и сохраненные идентификаторы пользователя и группы — и их влияние на элементарные функции управления процессами. Также будут обсуждаться интерпретируемые файлы и функция `system`. В завершение речь пойдет о средствах учета использования ресурсов процессами, предоставляемых большинством UNIX-систем. Это позволит взглянуть на функции управления процессами под другим углом.

8.2. Идентификаторы процесса

Любой процесс обладает уникальным идентификатором процесса, который представляет собой целое положительное число. Поскольку идентификатор процесса — это единственный широко используемый идентификатор, уникальность которого гарантируется системой, он часто присоединяется к другим идентификаторам для придания им уникальности. Например, приложения иногда включают идентификатор процесса в имена файлов, чтобы обеспечить их уникальность. Но, несмотря на уникальность, идентификаторы процесса могут использоваться многократно. По завершении процесса его идентификатор может использоваться повторно для другого процесса. Однако в большинстве версий UNIX реализованы специальные алгоритмы, позволяющие отложить повторное использование идентификатора на более позднее время, чтобы вновь созданный процесс не получил идентификатор процесса, завершившегося совсем недавно. Это помогает избежать ситуации, когда новый процесс по ошибке может быть принят за предыдущий при использовании того же самого идентификатора.

Существует ряд специальных процессов, определяемых конкретной реализацией. Процесс с идентификатором 0 — это, как правило, процесс-планировщик, который часто называют *swapper* (программа подкачки). Этому процессу не соответствует никакая программа на диске, поскольку он является частью ядра и считается системным процессом. Процесс с идентификатором 1 — это обычно процесс `init`,

который запускается ядром в конце процедуры начальной загрузки. В старых версиях UNIX этому процессу соответствует программа `/etc/init`, в более новых версиях — `/sbin/init`. Этот процесс отвечает за запуск операционной системы после загрузки ядра. Обычно `init` читает системные файлы инициализации — `/etc/rc*` или `/etc/inittab`, а также файлы, расположенные в каталоге `/etc/init.d`, и переводит систему в некоторое состояние, например в многопользовательский режим. Процесс `init` никогда не «умирает». Это обычный пользовательский процесс, он не является системным процессом ядра, как `swapper`, хотя и обладает привилегиями суперпользователя. Далее в этой главе мы увидим, как процесс `init` становится родительским процессом любого осиротевшего дочернего процесса.

В Mac OS X 10.4 на смену процессу *init* пришел процесс *Launchd*, выполняющий тот же комплекс задач, но имеющий более широкие функциональные возможности. Обсуждение особенностей работы процесса *Launchd* можно найти в разделе 5.10 [Singh, 2006].

Каждая версия UNIX имеет собственный набор процессов ядра, отвечающих за работу системных служб. Например, в некоторых реализациях виртуальной памяти UNIX идентификатор 2 соответствует процессу *pagedaemon*. Этот процесс отвечает за поддержку страницного обмена системы виртуальной памяти.

В дополнение к идентификатору процесса каждый процесс обладает еще рядом идентификаторов. Вот функции, которые возвращают эти идентификаторы:

#include <unistd.h>	
pid_t getpid(void);	Возвращает идентификатор вызывающего процесса
pid_t getppid(void);	Возвращает идентификатор родительского процесса
uid_t getuid(void);	Возвращает реальный идентификатор пользователя вызывающего процесса
uid_t geteuid(void);	Возвращает эффективный идентификатор пользователя вызывающего процесса
gid_t getgid(void);	Возвращает реальный идентификатор группы вызывающего процесса
gid_t getegid(void);	Возвращает эффективный идентификатор группы вызывающего процесса

Примечательно, что ни одна из этих функций не возвращает признак ошибки. Идентификатор родительского процесса мы рассмотрим в следующем разделе, где обсудим функцию `fork`. С реальным и эффективным идентификаторами пользователя и группы мы уже познакомились в разделе 4.4.

8.3. Функция `fork`

Любой процесс может создать новый процесс, вызвав функцию `fork`.

```
#include <unistd.h>
pid_t fork(void);
```

Возвращает 0 в дочернем процессе, идентификатор дочернего
процесса — в родительском, -1 — в случае ошибки

Новый процесс, созданный функцией `fork`, называется *дочерним процессом*, или процессом-потомком. Эта функция вызывается один раз, а управление возвращает дважды, с единственным отличием: в дочернем процессе она возвращает 0, а в родительском — идентификатор созданного дочернего процесса. Последнее обстоятельство объясняется тем, что процесс может иметь несколько потомков, а система не предусматривает функций, с помощью которых можно было бы получить идентификаторы дочерних процессов. В дочернем процессе функция `fork` возвращает 0, поскольку дочерний процесс имеет только одного родителя и всегда может получить его идентификатор с помощью функции `getppid`. (Идентификатор процесса 0 зарезервирован за ядром, поэтому невозможно получить 0 в качестве идентификатора дочернего процесса.)

И родительский и дочерний процессы продолжают выполнение программы с инструкции, следующей за вызовом функции `fork`. Процесс-потомок является точной копией родительского процесса. Например, потомок получает копии сегмента данных, кучи и стека родителя. Обратите внимание, что это именно копии; родительский и дочерний процессы не используют совместно одни и те же области памяти. Но они совместно используют сегмент кода (раздел 7.6).

Современные версии UNIX не производят немедленного полного копирования сегмента данных, стека и кучи, потому что часто вслед за вызовом `fork` сразу же следует вызов `exec`. Поэтому используется метод, который получил название *копирование при записи* (copy-on-write, COW). Указанные выше области памяти используются совместно обоими процессами, но ядро делает их доступными только для чтения. Если один из процессов попытается изменить данные в этих областях, ядро немедленно создаст копию конкретного участка памяти; обычно это «страница» виртуальной памяти. Более подробно об этом можно прочитать в разделе 9.2 [Bach, 1986] и в разделах 5.6 и 5.7 [McKusick et al., 1996].

Некоторые платформы предоставляют несколько версий функции `fork`. Все четыре платформы, обсуждаемые в данной книге, поддерживают функцию `vfork(2)`, которую мы рассмотрим в следующем разделе.

Кроме того, Linux 3.2.0 дает возможность создавать новые процессы с помощью системного вызова `clone(2)`. Это более универсальный вариант функции `fork`, позволяющий вызывающему процессу определить, что будет совместно использоваться дочерним и родительским процессами.

В FreeBSD 8.0 имеется системный вызов `rfork(2)`, напоминающий системный вызов `clone` в Linux и заимствованный из OC Plan 9 ([Pike et al., 1995]).

В Solaris 10 имеются две библиотеки для работы с потоками выполнения: одна — для потоков POSIX (`pthreads`) и другая — для потоков Solaris. Поведение функции `fork` в этих библиотеках различно. В случае потоков POSIX функция `fork` создает процесс, содержащий только вызывающий поток, а в случае потоков Solaris — процесс, содержащий копии всех потоков вызывающего процесса. В версии Solaris 10 это поведение изменилось; `fork` создает дочерний процесс, содержащий только копию вызывающего потока выполнения, независимо от используемой библиотеки. В Solaris также имеется функция `fork1`, которая создает процесс, включающий только копию вызывающего потока выполнения, и функция `forkall`, которая создает процесс, включающий копии всех потоков выполнения. Более подробно потоки выполнения обсуждаются в главах 11 и 12.

Пример

Программа в листинге 8.1 демонстрирует работу с функцией `fork` и показывает, что изменение переменных в дочернем процессе никак не сказывается на переменных в родительском процессе.

Листинг 8.1. Пример работы с функцией fork

```
#include "apue.h"

int      globvar = 6; /* глобальная переменная в сегменте */
                     /* инициализированных данных */
char buf[] = "запись в stdout\n";

int
main(void)
{
    int      var; /* переменная, размещаемая на стеке */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("ошибка вызова функции write");
    printf("перед вызовом функции fork\n"); /* мы не сбрасываем */
                                              /* буферы stdout */

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* дочерний процесс */
        globvar++;          /* изменить переменные */
        var++;
    } else {
        sleep(2);           /* родительский процесс */
    }

    printf("pid = %ld, globvar = %d, var = %d\n",
           (long)getpid(), globvar,
           var);
    exit(0);
}
```

После запуска программы мы получим:

```
$ ./a.out
запись в stdout
перед вызовом функции fork
pid = 430, globvar = 7, var = 89    переменные в дочернем процессе были изменены
pid = 429, globvar = 6, var = 88    родительская копия осталась без изменений
$ ./a.out > temp.out
$ cat temp.out
запись в stdout
перед вызовом функции fork
pid = 432, globvar = 7, var = 89
перед вызовом функции fork
pid = 431, globvar = 6, var = 88
```

В общем случае никогда нельзя сказать точно, какой из двух процессов первым получит управление после вызова функции `fork` — дочерний или родительский. Это во многом зависит от алгоритма планирования, используемого ядром. При необходимости синхронизировать работу родительского и дочернего процессов можно воспользоваться каким-либо механизмом взаимодействий. В программе из листинга 8.1 мы просто приостановили родительский процесс на 2 секунды, чтобы позволить дочернему процессу выполнятся первым. Но нет никакой гарантии, что этот прием сработает при любых условиях. Об этом и некоторых других видах синхронизации мы поговорим в разделе 8.9, когда будем обсуждать состояние гонки за ресурсами. В разделе 10.16 мы покажем, как использовать сигналы для синхронизации родительского и дочернего процессов после возврата из функции `fork`.

Записывая данные в стандартный вывод, мы вычитаем 1 из размера `buf`, чтобы избежать записи завершающего нулевого байта. Несмотря на то что функция `strlen` вычисляет длину строки без учета завершающего нулевого байта, `sizeof` вычисляет размер буфера, который включает завершающий нулевой байт. Другое отличие: обращение к `strlen` — это полноценный вызов функции, тогда как `sizeof` вычисляет размер буфера на этапе компиляции, поскольку буфер инициализирован известной строкой фиксированной длины.

Обратите внимание, как функция `fork` в программе из листинга 8.1 взаимодействует с функциями ввода/вывода. В главе 3 мы уже говорили, что функция `write` не буферизуется. Так как функция `write` вызывается перед `fork`, она выведет данные в стандартный вывод только один раз. С другой стороны, стандартная библиотека ввода/вывода буферизуется. В разделе 5.12 мы говорили, что стандартному потоку вывода назначается режим построчной буферизации, если он связан с терминалом, и режим полной буферизации — в любом другом случае. Запуская программу в интерактивном режиме, мы получаем только одну копию строки, выводимой функцией `printf`, потому что буфер стандартного вывода сбрасывается автоматически, когда встречается символ перевода строки. Но когда стандартный поток вывода перенаправляется в файл, мы получаем две копии строки. В этом случае перед обращением к `fork` функция `printf` вызывается один раз, но строка в момент вызова функции `fork` еще находится в буфере. В результате этот буфер будет скопирован в адресное пространство дочернего процесса при копировании сегмента данных родителя. Оба процесса, родитель-

ский и дочерний, получают стандартные буферы ввода/вывода, в которых хранится одна и та же строка. Второй вызов функции `printf`, который происходит непосредственно перед вызовом функции `exit`, лишь добавляет свои данные в конец существующего буфера. По завершении каждого из процессов его копия буфера сбрасывается на диск.

Совместное использование файлов

Когда при запуске программы из листинга 8.1 мы перенаправляем стандартный вывод родительского процесса в файл, стандартный вывод дочернего процесса также оказывается перенаправленным. Действительно, одна из особенностей функции `fork` в том, что она передает дочернему процессу дубликаты всех дескрипторов, открытых в родительском процессе. Мы говорим «дубликаты», потому что это действительно копии дескрипторов, подобные тем, что возвращает функция `dup`. Родительский и дочерний процессы совместно используют одни и те же записи в таблице файлов для каждого из открытых дескрипторов (вспомните рис. 3.3).

Представьте процесс, открывший три файла: стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках. По возвращении из функции `fork` дескрипторы будут распределены, как показано на рис. 8.1.

Важно заметить, что родительский и дочерний процессы совместно используют текущую позицию в файле. Представьте процесс, запустивший дочерний процесс и ожидающий его завершения. Допустим, что оба процесса в ходе работы производят запись в стандартный поток вывода. Если стандартный поток вывода родительского процесса будет перенаправлен в файл (например, командной оболочкой), текущая позиция в файле, установленная родительским процессом, неизбежно будет изменена дочерним процессом, когда он выполнит запись в стандартный поток вывода. В этом случае дочерний процесс может записывать данные в стандартный поток вывода, пока родительский процесс ожидает его завершения. По завершении потомка родительский процесс сможет продолжить запись в стандартный поток вывода, зная, что его данные будут записаны после тех, что записал дочерний процесс. Если бы текущая позиция в файле различалась у родительского и дочернего процессов, подобного эффекта достичь было бы гораздо сложнее, и это потребовало бы дополнительных усилий со стороны родительского процесса.

Если и родительский и дочерний процессы пишут в один и тот же дескриптор без какой-либо синхронизации, например, когда родительский процесс не ожидает завершения дочернего процесса, данные их вывода будут перемешаны (если дескриптор был открыт до вызова функции `fork`). Хотя это возможно согласно рис. 8.1, тем не менее такой режим работы не является нормальным.

Существуют два стандартных способа обслуживания дескрипторов после вызова `fork`.

1. Родительский процесс ожидает, когда потомок завершится. В этом случае родительскому процессу ничего не нужно делать со своими дескрипторами. Когда потомок завершится, текущая позиция в файле любого из разделяемых

дескрипторов, которые использовались им для чтения или записи, изменится надлежащим образом.

- Оба процесса, родительский и дочерний, продолжают работу независимо друг от друга. В этом случае после вызова функции `fork` родительский процесс закрывает дескрипторы, которые ему больше не потребуются, дочерний процесс делает то же самое. То есть они прекращают совместно использовать одни и те же дескрипторы. Этот сценарий часто используется в сетевых серверах.

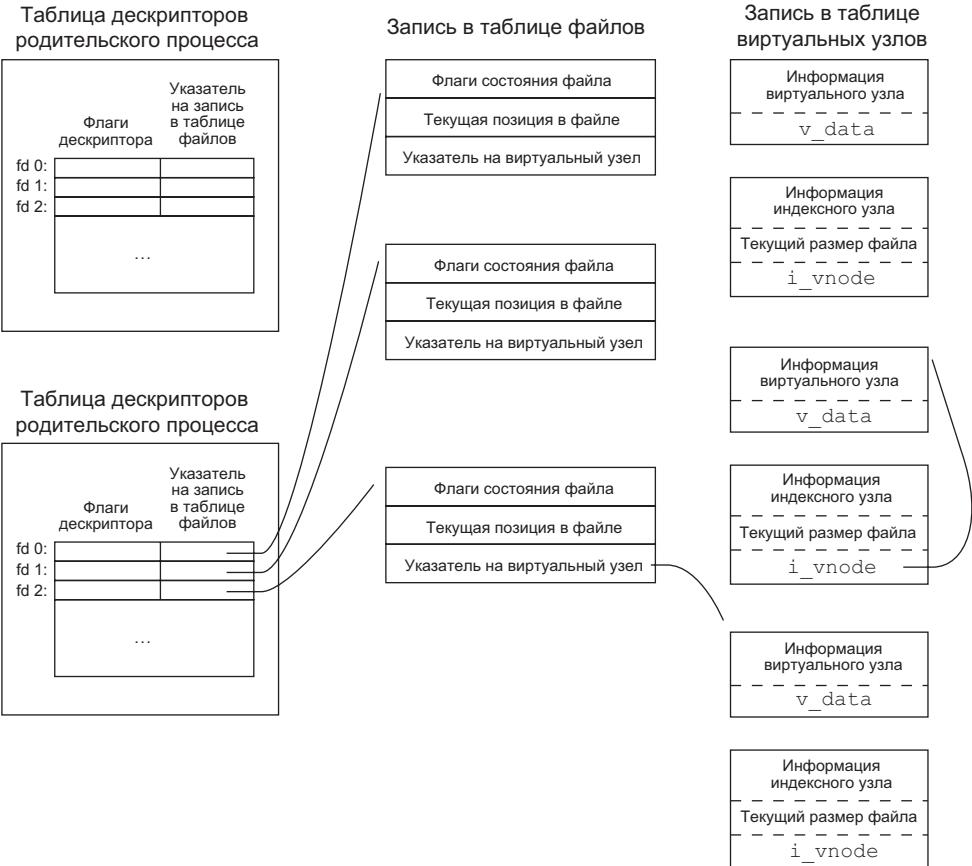


Рис. 8.1. Совместное использование открытых файлов родительским и дочерним процессами после вызова функции `fork`

Помимо открытых файлов, есть много других характеристик родительского процесса, которые наследуются дочерним:

- Реальный идентификатор пользователя, реальный идентификатор группы, эффективный идентификатор пользователя, эффективный идентификатор группы.

- Идентификаторы дополнительных групп.
- Идентификатор группы процессов.
- Идентификатор сеанса.
- Управляющий терминал.
- Флаги set-user-ID и set-group-ID.
- Текущий рабочий каталог.
- Корневой каталог.
- Маска режима создания файлов.
- Маска сигналов и их диспозиция.
- Флаги close-on-exes для открытых дескрипторов.
- Окружение.
- Присоединенные сегменты разделяемой памяти.
- Отображения в память.
- Ограничения на ресурсы.

Существуют следующие отличия между родительским и дочерним процессами:

- Функция `fork` возвращает различные значения.
- Различные идентификаторы процессов.
- Различные идентификаторы родительских процессов: идентификатор родительского процесса в потомке соответствует идентификатору процесса в родительском процессе, идентификатор родительского процесса в родительском процессе остается без изменений.
- Значения `tms_utime`, `tms_stime`, `tms_cutime` и `tms_cstime` в дочернем процессе устанавливаются равными 0.
- Блокировки файлов, установленные в родительском процессе, не наследуются.
- Таймеры, ожидающие срабатывания, в дочернем процессе сбрасываются.
- Набор сигналов, ожидающих обработки, в дочернем процессе очищается.

Многие из этих характеристик еще не обсуждались; мы поговорим о них в следующих главах.

Вызов функции `fork` терпит неудачу обычно в двух случаях: (а) когда в системе слишком много действующих процессов, что обычно свидетельствует о неполадках, и (б) когда общее количество процессов превысило системный предел для заданного реального идентификатора пользователя. В табл. 2.10 мы уже указывали, что максимальное количество одновременно работающих процессов на один реальный идентификатор пользователя определяется константой `CHILD_MAX`.

Два основных случая, когда используется функция `fork`:

1. Когда процесс хочет продублировать себя, чтобы родительский и дочерний процессы могли выполнять различные участки программы одновременно. Это обычно используется в сетевых серверах. Родительский процесс ожидает запроса от клиента и по его получении вызывает `fork` и передает обслуживание запроса дочернему процессу, после чего возвращается к ожиданию следующего запроса.

- Когда процесс хочет запустить другую программу. Эта ситуация характерна для командных оболочек. В этом случае дочерний процесс вызывает функцию `exec` (которую мы рассмотрим в разделе 8.10), как только функция `fork` вернет управление.

Некоторые операционные системы объединяют вызов `fork` и следующий за ним вызов `exec` в одну операцию, которая называется *spawn*. UNIX разделяет две операции по той простой причине, что достаточно часто вызов `fork` не сопровождается вызовом `exec`. Кроме того, такое разделение позволяет дочернему процессу между вызовами `fork` и `exec` изменить некоторые параметры процесса, например перенаправление ввода/вывода, идентификатор пользователя, диспозицию сигналов и т. д. Многочисленные примеры, иллюстрирующие это, мы увидим в главе 15.

Стандарт Single UNIX Specification включает интерфейсы spawn в группу расширений реального времени. Однако эти интерфейсы не служат заменой fork и exec. Они предназначены для систем, в которых имеются определенные сложности с эффективной реализацией функции fork, особенно для тех, в которых отсутствует аппаратная поддержка управления памятью.

8.4. Функция vfork

Порядок вызова и возвращаемые значения функций `vfork` и `fork` одинаковы, но их семантика различается.

Функция vfork впервые появилась в 2.9BSD. Некоторые считают ее пятном на репутации UNIX, однако все платформы, обсуждаемые в этой книге, поддерживают ее. Разработчики удалили vfork из версии 4.4BSD, но все дистрибутивы BSD с открытыми исходными текстами, происходящие от 4.4BSD, восстановили ее поддержку. В третьей версии Single UNIX Specification функция vfork отмечена как устаревший интерфейс и в четвертой версии была полностью удалена. Мы включили ее описание только как дань традиции. Ее не следует использовать в переносимых приложениях.

Функция `vfork` предназначена для создания новых процессов, когда целью нового процесса является запуск новой программы с помощью функции `exec` (пункт 2 в конце предыдущего раздела). Программа из листинга 1.5 также относится к программам этого типа. Функция `vfork` создает новый процесс точно так же, как `fork`, но не копирует адресное пространство родительского процесса в адресное пространство потомка, поскольку потомок не будет работать с этим адресным пространством — он просто вызывает функцию `exec` (или `exit`) сразу, как только `vfork` вернет управление. То есть до вызова `exec` или `exit` дочерний процесс выполняется в адресном пространстве родительского процесса. Такой подход более эффективен для некоторых реализаций UNIX, но может приводить к неожиданным результатам, если дочерний процесс изменит какие-либо данные (исключение составляет переменная, используемая для сохранения значения, возвращаемого функцией `vfork`), вызовет другие функции или вернет управление, не вызывая `exec` или `exit`. (Как уже упоминалось в предыдущем разделе, для повышения эффективности работы связки `fork/exec` многие реализации использу-

зуют технику копирования при записи, но полное отсутствие копирования все же гораздо эффективнее, чем копирование даже небольших объемов данных.)

Еще одно различие между этими функциями — `vfork` гарантирует, что дочерний процесс получит управление первым и будет удерживать его, пока не вызовет функцию `exec` или `exit`. Когда дочерний процесс вызовет любую из этих функций, родительский процесс возобновит работу. (Это может привести к тупиковой ситуации, если процесс-потомок зависит от дальнейших действий родительского процесса, которые должны быть выполнены до вызова любой из этих функций.)

Пример

Программа в листинге 8.2 — это измененная версия программы из листинга 8.1. Мы заменили функцию `fork` на `vfork` и убрали запись в стандартный вывод. Теперь нет необходимости приостанавливать родительский процесс с помощью функции `sleep`, поскольку `vfork` гарантирует, что он будет приостановлен ядром, пока дочерний процесс не вызовет `exec` или `exit`.

Листинг 8.2. Пример работы с функцией vfork

```
#include "apue.h"

int globvar = 6; /* глобальная переменная в сегменте инициализированных данных */

int
main(void)
{
    int      var; /* локальная переменная в стеке */
    pid_t   pid;

    var = 88;
    printf("перед вызовом функции vfork\n"); /* мы не сбрасываем буферы stdout */
    if ((pid = vfork()) < 0) {
        err_sys("ошибка вызова функции vfork");
    } else if (pid == 0) { /* дочерний процесс */
        globvar++;
        var++;
        _exit(0);           /* завершение дочернего процесса */
    }

    /*
     * Родительский процесс продолжит работу отсюда.
     */
    printf("pid = %ld, globvar = %d, var = %d\n", (long)getpid(), globvar, var);
    exit(0);
}
```

Запуск этой программы дает следующие результаты:

```
$ ./a.out
перед вызовом функции vfork
pid = 29039, globvar = 7, var = 89
```

Здесь значения переменных, увеличенные в дочернем процессе, изменились и в родительском процессе. Поскольку известно, что дочерний процесс продолжает работу в адресном пространстве родительского процесса, это не стало для нас

сюрпризом. Однако это поведение отличается от того, что мы видели при работе с функцией `fork`.

Обратите внимание, что в программе из листинга 8.2 вместо функции `exit` используется `_exit`. Как уже говорилось в разделе 7.3, функция `_exit` не производит сброс буферов ввода/вывода. Вызвав функцию `exit`, мы получили бы несколько иные результаты. В зависимости от реализации стандартной библиотеки ввода/вывода мы могли бы и не заметить никаких различий или увидели бы, что пропали данные, выводимые функцией `printf` в родительском процессе.

Когда дочерний процесс завершает работу вызовом `exit`, содержимое всех буферов ввода/вывода сбрасывается. Если это единственное действие, которое производится библиотекой, мы не увидим никаких различий по сравнению с вызовом `_exit`. Однако если реализация дополнительно закрывает потоки ввода/вывода, память, в которой размещается объект `FILE` стандартного потока вывода, будет очищена. Поскольку дочерний процесс заимствует адресное пространство родительского процесса, когда родительский процесс возобновит работу и вызовет функцию `printf`, она ничего не сможет вывести и вернет признак ошибки (-1). Обратите внимание, что дескриптор `STDOUT_FILENO` родительского процесса все еще является допустимым, поскольку дочерний процесс получает копию массива файловых дескрипторов родительского процесса (см. рис. 8.1).

В большинстве современных реализаций функция `exit` не закрывает потоки ввода/вывода. Поскольку процесс собирается завершить работу, ядро все равно закроет все открытые дескрипторы файлов. Закрытие их в библиотеке только увеличивает нагрузку и не несет никакой выгоды.

Дополнительные сведения о реализации функций `fork` и `vfork` можно найти в [McKusick et al., 1996], раздел 5.6. К изучению функции `vfork` мы вернемся в упражнениях 8.1 и 8.2.

8.5. Функции `exit`

В разделе 7.3 упоминалось пять способов нормального завершения работы процесса.

1. Возврат из функции `main`. Как уже говорилось в разделе 7.3, это эквивалентно вызову функции `exit`.
2. Вызов функции `exit`. Эта функция определена стандартом ISO C, она производит вызов всех функций — обработчиков выхода, зарегистрированных функцией `atexit`, и закрывает все стандартные потоки ввода/вывода. Поскольку стандарт ISO C не затрагивает дескрипторы файлов, многозадачность (родительский и дочерний процессы) и управление заданиями, определение этой функции для UNIX является неполным.
3. Вызов функции `_exit` или `_Exit`. Стандарт ISO C определяет функцию `_Exit` как способ завершения процесса без запуска функций — обработчиков выхода или обработчиков сигналов. При этом от конкретной реализации зависит,

будут ли буферы ввода/вывода сбрасываться на диск или нет. В системе UNIX имена `_exit` и `_Exit` являются синонимами, обе функции не сбрасывают буферы ввода/вывода. Функция `_exit` вызывается из `exit` и производит действия, характерные для UNIX. Функция `_exit` определена стандартом POSIX.1.

В большинстве версий UNIX `exit(3)` реализована как библиотечная функция, а `_exit(2)` – как системный вызов.

4. Возврат из процедуры запуска последнего потока выполнения в процессе. Код завершения потока при этом не будет использоваться в качестве кода завершения процесса. Когда последний поток выполнения в процессе вернется из своей процедуры запуска, процесс завершится с кодом 0.
5. Вызов функции `pthread_exit` из последнего потока выполнения в процессе. Как и в предыдущем случае, процесс вернет код завершения 0, аргумент функции `pthread_exit` при этом игнорируется. Более подробно об этой функции мы поговорим в разделе 11.5.

Три способа ненормального завершения процесса:

1. Вызов функции `abort`. Это особый случай следующего способа, так как данная функция генерирует сигнал `SIGABRT`.
2. При получении процессом некоторых сигналов. (Более подробно о сигналах рассказывается в главе 10.) Сигнал может сгенерировать сам процесс (например, с помощью функции `abort`), другие процессы или ядро. К последним относятся сигналы, передаваемые при попытке обратиться к памяти вне адресного пространства процесса или при попытке деления на ноль.
3. По запросу на завершение последнего потока выполнения. По умолчанию завершение потока происходит с некоторой задержкой: один поток запрашивает завершение другого потока, и через какое-то время указанный поток завершается. Мы обсудим запросы на завершение в разделах 11.5 и 12.7.

Независимо от того, как именно завершается процесс, в конечном итоге ядро выполняет один и тот же код. Этот код закрывает все открытые дескрипторы, освобождает занимаемую процессом память и т. д.

Для любого из перечисленных способов завершающийся процесс должен иметь возможность известить родительский процесс о том, как он завершился. В случае трех функций выхода (`exit`, `_exit` и `_Exit`) родительскому процессу через аргумент функции передается код завершения. А в случае ненормального завершения ядро – не процесс – генерирует код, указывающий причину ненормального завершения процесса. В любом случае родительский процесс может получить код завершения от функции `wait` или `waitpid` (описание этих функций дается в следующем разделе).

Обратите внимание на различие между кодом выхода, который является аргументом одной из трех функций выхода или возвращаемым значением функции `main`, и кодом завершения. Ядро преобразует код выхода в код завершения, когда в заключение вызывается функция `_exit` (рис. 7.1). В табл. 8.1 перечислены способы, с помощью которых родительский процесс может получить код завершения до-

черного процесса. Если дочерний процесс завершился нормально, родительский процесс может получить его код выхода.

Когда мы описывали функцию `fork`, было очевидно, что родительский процесс продолжает существовать после вызова функции `fork`. Сейчас мы говорим о возврате кода завершения родительскому процессу. Но что произойдет, если родительский процесс завершится раньше дочернего? Ответ таков: родителем любого процесса, родительский процесс которого завершился раньше его самого, становится процесс `init`. В таком случае мы говорим, что процесс был унаследован процессом `init`. Обычно при завершении какого-либо процесса ядро проверяет все активные процессы, чтобы узнать, не является ли завершившийся процесс чьим-либо родителем. Если это так, для процесса, оставшегося активным, идентификатором родительского процесса назначается 1 (идентификатор процесса `init`). Благодаря этому удается гарантировать наличие родителя у любого процесса.

Еще один момент, который нужно рассмотреть, — когда дочерний процесс заканчивает работу раньше родительского. Если дочерний процесс полностью исчезнет, родительский процесс не сможет получить его код завершения, когда это потребуется. Ядро сохраняет некоторый объем информации о каждом завершившемся процессе, чтобы она была доступна, когда родительский процесс вызовет функцию `wait` или `waitpid`. В простейшем случае эта информация состоит из идентификатора процесса, кода завершения и количества процессорного времени, затраченного процессом. Ядро может освободить всю память, занимаемую процессом, и закрыть его открытые файлы. В терминологии UNIX процесс, который завершился, но при этом его родительский процесс не уловил этого момента, называют *зомби*. Команда `ps(1)` выводит в поле состояния процесса-зомби символ Z. Если написать долго работающую программу, которая порождает множество дочерних процессов, они будут превращаться в зомби, если программа не станет дожидаться получения от них кодов завершения.

В некоторых системах существует возможность предотвратить появление зомби; в разделе 10.7 будет описано, как именно это сделать.

Наконец, рассмотрим случай, когда заканчивается процесс, унаследованный процессом `init`. Превращается ли он в зомби? Нет, потому что `init` создан так, что всякий раз, когда один из его потомков завершается, `init` вызывает одну из функций `wait`, чтобы забрать код завершения. Таким способом `init` препятствует засорению системы процессами-зомби. Под «потомками процесса `init`» мы подразумеваем как процессы, запущенные непосредственно процессом `init` (например, `getty`, который описывается в разделе 9.2), так и унаследованные, родители которых завершили работу.

8.6. Функции `wait` и `waitpid`

Когда процесс завершается, обычным образом или аварийно, ядро извещает об этом родительский процесс с помощью сигнала `SIGCHLD`. Поскольку завершение дочернего процесса есть событие асинхронное (оно может произойти в любой момент), то и сигнал является асинхронным извещением, посыпаемым ядром роди-

тельскому процессу. Родительский процесс может проигнорировать сигнал или определить функцию, которая будет вызвана по прибытии сигнала, — обработчик сигнала. По умолчанию процессы игнорируют этот сигнал. Мы обсудим возможные варианты поведения в главе 10. А пока достаточно запомнить, что функции `wait` и `waitpid`, вызванные родительским процессом, могут:

- Заблокировать процесс, если все его дочерние процессы продолжают работу.
- Сразу же вернуть управление с кодом завершения дочернего процесса, если он уже закончил работу и ожидает, пока родительский процесс заберет код завершения.
- Сразу же вернуть управление с признаком ошибки, если у вызвавшего процесса нет ни одного дочернего процесса.

Если процесс вызывает `wait` по получении сигнала `SIGCHLD`, функция сразу же вернет управление. Но если `wait` была вызвана в любой произвольный момент времени, она может заблокировать родительский процесс.

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Обе возвращают идентификатор процесса в случае успеха,
—1 — в случае ошибки

Эти функции имеют следующие различия:

- Функция `wait` может заблокировать вызывающий процесс, пока не завершится дочерний процесс, в то время как функция `waitpid` дает возможность предотвратить блокировку.
- Функция `waitpid` не ждет первого завершившегося дочернего процесса — можно указать, завершения какого процесса она должна ожидать.

Если дочерний процесс уже завершился и находится в состоянии зомби, функция `wait` сразу же вернет управление и передаст код его завершения. Иначе она заблокирует вызывающий процесс до момента, пока дочерний процесс не завершит свою работу. Если у вызывающего процесса имеется несколько дочерних процессов, функция `wait` вернет управление, когда завершит работу любой из них. Мы всегда можем узнать, какой из потомков завершился, поскольку функция возвращает идентификатор процесса.

В обеих функциях аргумент `statloc` является указателем на целое число. Если в аргументе передается непустой указатель, по заданному адресу будет записан код завершения дочернего процесса. Если код завершения нас не интересует, можно передать в этом аргументе пустой указатель.

Целочисленный код завершения, возвращаемый этими двумя функциями, традиционно определяется реализацией. В нем несколько битов отводится под код выхода (в случае нормального завершения работы), несколько битов — под номер сигнала (в случае аварийного завершения), один бит указывает, был ли создан

файл дампа памяти (файл `core`), и т. д. Согласно стандарту POSIX.1, в файле `<sys/wait.h>` определяются различные макросы, с помощью которых производится извлечение кодов выхода. Определить, как завершился процесс, можно с помощью четырех взаимоисключающих макросов, имена которых начинаются с префикса `WIF`. В зависимости от того, какой из этих четырех макросов возвращает истину, можно использовать другие макросы, чтобы получить код выхода, номер сигнала и другую информацию. Все четыре макроопределения приводятся в табл. 8.1.

Таблица 8.1. Макроопределения для проверки кода завершения, возвращаемого функциями `wait` и `waitpid`

Макроопределение	Описание
<code>WIFEXITED(status)</code>	Возвращает <code>true</code> , если код <code>status</code> получен в результате нормального завершения дочернего процесса. В этом случае можно извлечь младшие 8 бит из аргумента, который был передан функции <code>exit</code> , <code>_exit</code> или <code>_Exit</code> : <code>WEXITSTATUS(status)</code>
<code>WIFSIGNALED(status)</code>	Возвращает <code>true</code> , если код <code>status</code> получен в результате ненормального (аварийного) завершения дочернего процесса из-за сигнала, который не был перехвачен. В этом случае можно узнать номер сигнала, вызвавшего завершение дочернего процесса: <code>WTERMSIG(status)</code> Кроме того, в некоторых реализациях (но не в Single UNIX Specification) определен макрос <code>WCOREDUMP(status)</code> который возвращает <code>true</code> , если в результате аварийного завершения процесса создан файл с дампом памяти (<code>core</code> -файл)
<code>WIFSTOPPED(status)</code>	Возвращает <code>true</code> , если код <code>status</code> получен в результате остановки дочернего процесса по сигналу. В этом случае можно узнать номер сигнала, который вызвал остановку процесса, с помощью макроса <code>WSTOPSIG(status)</code>
<code>WIFCONTINUED(status)</code>	Возвращает <code>true</code> , если код <code>status</code> получен для дочернего процесса, который продолжил работу после остановки (расширение XSI в стандарте POSIX.1 — только для функции <code>waitpid</code>)

В разделе 9.8, когда речь пойдет об управлении заданиями, мы увидим, как можно остановить процесс.

Пример

Функция `pr_exit` в листинге 8.3 использует макросы из табл. 8.1 для вывода сведений, полученных из кода завершения. В этой книге мы будем использовать ее во многих примерах. Обратите внимание, что эта функция обращается к макросу `WCOREDUMP`, если он определен в системе.

Листинг 8.3. Вывод сведений, полученных из кода завершения

```
#include "apue.h"
#include <sys/wait.h>
```

```
void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("нормальное завершение, код выхода = %d\n",
               WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("аварийное завершение, номер сигнала = %d%s\n",
               WTERMSIG(status),
               #ifdef WCOREDUMP
               WCOREDUMP(status) ? " (создан файл core)" : "");
    #else
               "");
    #endif
    else if (WIFSTOPPED(status))
        printf("дочерний процесс остановлен, номер сигнала = %d\n",
               WSTOPSIG(status));
}
}
```

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 поддерживают макроопределение `WCOREDUMP`. Но некоторые платформы скрывают его определение, если определена константа `_POSIX_C_SOURCE` (раздел 2.7).

Программа в листинге 8.4 демонстрирует интерпретацию различных значений кода завершения с помощью функции `pr_exit`. Запустив ее, мы получим следующие результаты:

```
$ ./a.out
нормальное завершение, код выхода = 7
аварийное завершение, номер сигнала = 6 (создан файл core)
аварийное завершение, номер сигнала = 8 (создан файл core)
```

Сейчас мы получаем номер сигнала от `WTERMSIG`. Чтобы убедиться, что сигнал `SIGABRT` имеет значение 6, а сигнал `SIGFPE` — значение 8, можно заглянуть в файл `<signal.h>`. Переносимый способ отображения номера сигнала в описательное имя будет представлен в разделе 10.22.

Как уже говорилось выше, если родительский процесс имеет несколько процессов-потомков, функция `wait` вернет управление по завершении любого из них. А что делать, если требуется дождаться завершения конкретного дочернего процесса (при условии, что известен его идентификатор)? В ранних версиях UNIX приходилось вызывать функцию `wait` и сравнивать возвращаемый ею идентификатор процесса с интересующим. Если завершившийся процесс оказался не тем, который мы ожидали, приходилось сохранять идентификатор процесса и код его завершения в отдельном списке и снова вызывать функцию `wait`. Этую операцию надо было повторять до тех пор, пока не завершится желаемый процесс. Если после этого нужно было дождаться завершения другого процесса, мы вынуждены были сначала просмотреть список уже завершившихся процессов, и если его в этом списке не было, вызывать функцию `wait`. Таким образом, возникла потребность в функции, которая ожидала бы завершения конкретного процесса. Эта функциональность (и даже больше) заложена в функцию `waitpid`, которая определена стандартом POSIX.1.

Листинг 8.4. Интерпретация различных кодов завершения

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;
    int      status;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)           /* дочерний процесс */
        exit(7);

    if (wait(&status) != pid) /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
    pr_exit(status);          /* и вывести код завершения */

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)           /* дочерний процесс */
        abort();                  /* послать сигнал SIGABRT */

    if (wait(&status) != pid) /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
    pr_exit(status);          /* и вывести код завершения */

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0)           /* дочерний процесс */
        status /= 0;              /* деление на 0 сгенерирует сигнал SIGFPE */

    if (wait(&status) != pid) /* дождаться завершения дочернего процесса */
        err_sys("ошибка вызова функции wait");
    pr_exit(status);          /* и вывести код завершения */

    exit(0);
}

```

Интерпретация аргумента *pid* функцией *waitpid* зависит от его значения:

- pid == -1* Ожидает завершения любого дочернего процесса. В данном случае функция *waitpid* эквивалентна функции *wait*.
- pid > 0* Ожидает завершения процесса с идентификатором, равным *pid*.
- pid == 0* Ожидает завершения любого дочернего процесса с тем же идентификатором группы процессов, что и у вызывающего процесса (группы процессов обсуждаются в разделе 9.4).
- pid < -1* Ожидает завершения любого дочернего процесса с идентификатором группы процессов, совпадающим с *pid*.

Функция *waitpid* возвращает идентификатор завершившегося дочернего процесса и сохраняет его код завершения по адресу в аргументе *statloc*. Функция *wait* может вернуть признак ошибки, только когда процесс не имеет потомков. (Еще одна ошибочная ситуация возможна, если выполнение функции было прервано

сигналом. Мы обсудим этот вариант в главе 10.) Но функция `waitpid` может завершиться ошибкой также, если заданный процесс или группа процессов не существуют или не являются потомками вызывающего процесса.

Аргумент *options* позволяет управлять поведением функции `waitpid`. Он может содержать 0 или значение, полученное в результате поразрядной операции ИЛИ (OR) из констант, перечисленных в табл. 8.2.

FreeBSD 8.0 и Solaris 10 поддерживают одну дополнительную, нестандартную, константу для аргумента options. WNOWAIT вынуждает систему сохранить процесс, код завершения которого возвращается функцией `waitpid`, благодаря чему его можно получить повторно.

Таблица 8.2. Константы для аргумента *options* функции `waitpid`

Константа	Описание
<code>WCONTINUED</code>	Если реализация поддерживает управление заданиями, функция <code>waitpid</code> вернет код состояния потомка, определяемого аргументом <i>pid</i> , который возобновил работу после остановки и чей код состояния еще не был получен (расширение XSI стандарта POSIX.1)
<code>WNOHANG</code>	Функция <code>waitpid</code> не блокирует вызывающий процесс, если потомок, определяемый аргументом <i>pid</i> , еще не изменил свое состояние. В этом случае функция вернет 0
<code>WUNTRACED</code>	Если реализация поддерживает управление заданиями, функция <code>waitpid</code> вернет код состояния дочернего процесса, определяемого аргументом <i>pid</i> , который был остановлен и код состояния которого еще не был получен. Макрокоманда <code>WIFSTOPPED</code> позволяет определить, соответствует ли возвращаемое значение остановленному дочернему процессу

Функция `waitpid` предоставляет три возможности, которых лишена функция `wait`.

1. Функция `waitpid` позволяет указать процесс, завершения которого необходимо дождаться, в то время как `wait` возвращает код состояния первого завершившегося процесса-потомка. Мы вернемся к обсуждению этой возможности, когда будем рассказывать о функции `popen`.
2. Функция `waitpid` дает возможность предотвратить блокировку, когда требуется лишь узнать состояние дочернего процесса и нежелательно, чтобы вызывающий процесс заблокировался.
3. Функция `waitpid` поддерживает управление заданиями с помощью констант `WUNTRACED` и `WCONTINUED`.

Пример

Вернемся к обсуждению процессов-зомби, начатому в разделе 8.5. Если необходимо, чтобы процесс, создавший потомка с помощью функции `fork`, не дождался его завершения и при этом процесс-потомок не превращался в зомби до завершения родительского процесса, функцию `fork` следует вызвать дважды. Этот прием использует программа в листинге 8.5.

Листинг 8.5. Предотвращение появления зомби за счет двойного вызова функции `fork`

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* первый потомок */
        if ((pid = fork()) < 0)
            err_sys("ошибка вызова функции fork");
        else if (pid > 0)
            exit(0);           /* первый потомок, он же */
                               /* родительский процесс для второго потомка */
        /*
         * Здесь продолжает работу второй потомок, для которого родительским
         * стал процесс init, поскольку настоящий родительский процесс
         * вызвал функцию exit() чуть выше.
         * Теперь можно продолжить работу, зная, что когда процесс завершится,
         * его код завершения получит процесс init.
         */
        sleep(2);
        printf("второй потомок, идентификатор родительского процесса = %ld\n",
               (long)getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid) /* ждать завершения первого потомка */
        err_sys("ошибка вызова функции waitpid");

    /*
     * Здесь продолжает работу родительский (первоначальный) процесс,
     * поскольку он не является родительским процессом для второго потомка.
     */
    exit(0);
}
```

Мы приостановили работу второго потомка на две секунды, чтобы гарантировать, что первый потомок завершил свою работу до того, как будет выведен идентификатор родительского процесса. Функция `fork` вернет управление как родительскому, так и дочернему процессу, но мы никогда заранее не знаем, кто из них будет первым. Если бы второй дочерний процесс не был приостановлен и после вызова функции `fork` получил бы управление первым, идентификатор процесса, функция `printf`, вывела бы идентификатор первичного родительского процесса, а не процесса `init`. Запустив программу из листинга 8.5, мы получили

```
$ ./a.out
$ второй потомок, идентификатор родительского процесса = 1
```

Обратите внимание, что командная оболочка вывела приглашение (символ \$), как только первичный процесс завершился, то есть еще до того, как второй потомок вывел идентификатор своего родительского процесса.

8.7. Функция waitid

Расширения XSI стандарта Single UNIX Specification включают дополнительную функцию, которая может получить код выхода процесса. Функция `waitid` очень похожа на функцию `waitpid`, но обладает дополнительными возможностями.

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Подобно `waitpid`, функция `waitid` позволяет процессу указать на потомка, завершения которого необходимо дождаться. Вместо того чтобы передавать эту информацию вместе с идентификатором процесса или группы процессов в закодированном виде через единственный аргумент, функция `waitid` предоставляет два отдельных аргумента. Значение аргумента `id` интерпретируется в зависимости от значения аргумента `idtype`. Возможные значения этого аргумента приводятся в табл. 8.3.

Таблица 8.3. Возможные значения аргумента `idtype` функции `waitid`

Константа	Описание
P_PID	Ждать завершения конкретного процесса. Аргумент <code>id</code> содержит его идентификатор
P_PGID	Ждать завершения дочернего процесса, принадлежащего к указанной группе процессов. Аргумент <code>id</code> содержит идентификатор группы процессов
P_ALL	Ждать завершения любого дочернего процесса. Содержимое аргумента <code>id</code> игнорируется

Аргумент `options` содержит результат поразрядной операции ИЛИ (OR) из констант, перечисленных в табл. 8.4. Эти константы определяют, какие изменения состояния дочернего процесса интересуют вызывающий процесс.

Таблица 8.4. Константы для аргумента `options` функции `waitid`

Константа	Описание
WCONTINUED	Ждать завершения процесса, который возобновил работу после остановки и код состояния которого еще не был получен
WEXITED	Получить информацию о состоянии завершившегося процесса
WNOHANG	Сразу же вернуть управление и не блокировать вызывающий процесс, если код выхода дочернего процесса недоступен
WNOWAIT	Не уничтожать информацию о состоянии дочернего процесса, чтобы затем ее можно было получить с помощью функции <code>wait</code> , <code>waitpid</code> или <code>waitid</code>
WSTOPPED	Ждать завершения процесса, который был остановлен и код состояния которого еще не был получен

В аргументе *options* должна быть указана хотя бы одна из констант: WCONTINUED, WEXITED или WSTOPPED.

Аргумент *infop* – указатель на структуру **siginfo** с подробной информацией о сигнале, вызвавшем изменение состояния дочернего процесса. Структура **siginfo** будет рассмотрена в разделе 10.14.

Из четырех платформ, обсуждаемых в данной книге, только Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 поддерживают функцию waitid. Но имейте в виду, что в Mac OS X 10.6.8 эта функция возвращает в структуре siginfo не всю информацию, которую можно было бы ожидать.

8.8. Функции wait3 и wait4

Большинство реализаций UNIX поддерживают две дополнительные функции: **wait3** и **wait4**. Они впервые появились в ветке BSD. Единственное их преимущество перед **wait**, **waitid** и **waitpid** заключается в дополнительном аргументе, через который ядро может вернуть краткую справку о ресурсах, использованных завершившимся процессом и всеми его дочерними процессами.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Обе возвращают идентификатор процесса в случае успеха,
0 или -1 – в случае ошибки

Информация об использованных ресурсах включает такие сведения, как количество процессорного времени (пользовательского и системного), неудачных попыток обращений к страницам виртуальной памяти, принятых сигналов и т. п. За подробностями обращайтесь к странице справочного руководства **getrusage(2)**. (Эта информация о ресурсах отличается от ограничений на ресурсы, которые обсуждались в разделе 7.11.) В табл. 8.5 приводится информация о различных аргументах, поддерживаемых функциями семейства **wait**.

Таблица 8.5. Аргументы, поддерживаемые функциями семейства **wait** на различных plataформах

Функция	pid	options	rusage	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
wait				✓	✓	✓	✓	✓
waitid	✓	✓		✓		✓	✓	✓
waitpid	✓	✓		✓	✓	✓	✓	✓
wait3		✓	✓		✓	✓	✓	✓
wait4	✓	✓	✓		✓	✓	✓	✓

Функция `wait3` была включена в ранние версии стандарта Single UNIX Specification. Во второй версии стандарта она переведена в разряд устаревших, а в третьей версии вообще исключена из стандарта.

8.9. Гонка за ресурсами

Мы будем называть гонкой за ресурсами состояние, возникающее, когда несколько процессов пытаются одновременно производить некоторые действия с совместно используемыми данными и конечный результат зависит от порядка, в котором эти процессы выполняются. Функция `fork` является собой яркий пример потенциального источника проблем, связанных с гонкой за ресурсами, если логика выполнения программы явно или неявно зависит от того, кто первым получит управление — родительский процесс или дочерний. В общем случае это невозможно предсказать заранее, но даже если бы мы знали наверняка, какой процесс первым получит управление, все равно дальнейшая работа процесса зависит от степени нагрузки на систему и алгоритма планирования, заложенного в ядре.

Мы уже встречались с потенциальной ситуацией гонки за ресурсами в программе из листинга 8.5, когда второй потомок выводил идентификатор родительского процесса. Если второй потомок получит управление раньше первого, его родительским процессом станет первый потомок. Но если сначала получит управление первый потомок и у него будет достаточно времени, чтобы успеть завершиться, родительским процессом для второго потомка станет процесс `init`. Даже вызов функции `sleep`, который использовался в нашем примере, не может гарантировать выполнения процессов в заданном порядке. Если система сильно загружена, даже после двухсекундной задержки второй потомок может получить управление раньше, чем первому потомку удастся завершиться. Проблемы такого рода очень сложны в отладке, потому что в большинстве случаев не проявляются.

Чтобы дождаться завершения потомка, достаточно вызвать одну из функций семейства `wait`. Чтобы дождаться завершения родительского процесса, как в программе из листинга 8.5, можно воспользоваться примерно таким циклом:

```
while (getppid() != 1)
    sleep(1);
```

Однако этот цикл, который называется *опросом* (polling), порождает еще одну проблему. Дело в том, что процесс непроизводительно расходует процессорное время, так как возобновляет работу каждую секунду, чтобы проверить истинность условия.

Чтобы не попасть в состояние гонки за ресурсами и избежать применения опроса, необходимо нечто вроде обмена сигналами между процессами. Для этой цели можно использовать сигналы, и один такой способ будет описан в разделе 10.16. Также могут использоваться различные виды межпроцессных взаимодействий (Interprocess Communication, IPC). Некоторые из них мы рассмотрим в главах 15 и 17.

Для организации взаимоотношений между родительским и дочерним процессами часто используется следующий сценарий. После вызова функции `fork` оба процесса, родительский и дочерний, выполняют некоторые действия. Например, родительский процесс может добавить запись с идентификатором потомка в файл журнала, а потомок может создать файл для родительского процесса. В таком случае требуется, чтобы каждый из процессов имел возможность известить другой процесс о завершении определенных начальных операций и дождался бы завершения этих операций другим процессом, прежде чем продолжить работу. Следующий код иллюстрирует этот сценарий:

```
#include "apue.h"

TELL_WAIT(); /* выполнить подготовительные операции для TELL_xxx и WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) { /* дочерний процесс */

    /* дочерний процесс выполняет необходимые действия ... */

    TELL_PARENT(getppid()); /* сообщить родительскому процессу */
    /* о завершении подготовительных операций */
    WAIT_PARENT();           /* и дождаться ответа родительского процесса */

    /* потомок продолжает работу самостоятельно ... */

    exit(0);
}

/* родительский процесс выполняет необходимые действия ... */

TELL_CHILD(pid); /* сообщить дочернему процессу */
/* о завершении подготовительных операций */
WAIT_CHILD();      /* и дождаться ответа дочернего процесса */

/* родительский процесс продолжает работу самостоятельно ... */

exit(0);
```

Здесь мы исходим из предположения, что все необходимые определения находятся в заголовочном файле `apue.h`. Пять процедур — `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` — должны быть оформлены в виде функций или макроопределений.

Мы покажем различные варианты реализации процедур `TELL` и `WAIT` в последующих главах: в разделе 10.16 будет продемонстрирована реализация на основе сигналов, а в листинге 15.3 — на основе неименованных каналов. А теперь рассмотрим пример, где используются эти пять процедур.

Пример

Программа в листинге 8.6 выводит две строки: одна формируется дочерним процессом, а другая — родительским. Программа подвержена гонке за ресурсами, по-

тому что порядок вывода символов строк зависит от того, какой процесс получает управление и как долго он работает.

Листинг 8.6. Программа, содержащая гонку за ресурсами

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {
        charatatime("от дочернего процесса\n");
    } else {
        charatatime("от родительского процесса\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char     *ptr;
    int      c;

    setbuf(stdout, NULL); /* установить небуферизованный режим */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Мы установили небуферизованный режим для стандартного потока вывода, чтобы запись каждого символа сопровождалась вызовом функции `write`. Это сделано, чтобы ядро могло производить переключение процессов настолько часто, насколько это возможно. Таким способом создается ситуация гонки за ресурсами. (Если этого не сделать, то, быть может, мы никогда и не увидим результатов, показанных ниже. Но если мы их не видим, это не значит, что гонки за ресурсами не существует; это значит лишь, что мы не наблюдаем ее в данной конкретной системе.) Ниже приводится вывод, действительно полученный от программы:

```
$ ./a.out
от дочернего процесса
т родительского процесса
$ ./a.out
от дочернего процесса
т родительского процесса
$ ./a.out
от дочернего процесса
от родительского процесса
```

А теперь изменим программу в листинге 8.6 так, чтобы она использовала функции `TELL` и `WAIT`. Эти изменения представлены в листинге 8.7. Добавленные строки отмечены символом «`++`».

Листинг 8.7. Модификация программы из листинга 8.6, позволяющая избежать гонки за ресурсами

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;
+    TELL_WAIT();
+
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) {
+        WAIT_PARENT(); /* родительский процесс стартует первым */
        charatatime("от дочернего процесса\n");
    } else {
        charatatime("от родительского процесса\n");
+        TELL_CHILD(pid);
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char *ptr;
    int c;

    setbuf(stdout, NULL); /* установить небуферизованный режим */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Запустив эту программу, мы получим то, что и ожидали, — символы, выводимые двумя процессами, более не смешиваются.

В программе из листинга 8.7 родительский процесс стартует первым. Если требуется, чтобы первым стартовал дочерний процесс, нужно изменить строки, следующие за вызовом `fork`:

```
} else if (pid == 0) {
    charatatime("от дочернего процесса\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD(); /* дочерний процесс стартует первым */
    charatatime("от родительского процесса\n");
}
```

Обсуждение этого примера будет продолжено в упражнении 8.4.

8.10. Функции `exec`

Мы уже говорили в разделе 8.3, что функция `fork` часто используется для создания нового процесса, который затем запускает другую программу с помощью одной из функций семейства `exec`. Когда процесс вызывает одну из функций `exec`,

он полностью замещается другой программой, и эта новая программа начинает выполнение собственной функции `main`. Идентификатор процесса при этом не изменяется, поскольку функция `exec` не создает новый процесс, она просто замещает текущий процесс — его сегмент кода, сегмент данных, динамическую область памяти и сегмент стека — другой программой.

Существует семь различных функций `exec`, но мы обычно будем говорить просто о «функции `exec`», подразумевая любую из них. Эти семь функций завершают список примитивов UNIX, предназначенных для управления процессами. С помощью функции `fork` можно создавать новые процессы, с помощью функций `exec` — запускать новые программы. Функция `exit` и функции семейства `wait` обслуживают процедуры выхода и ожидания завершения. Эти примитивы — все, что необходимо для управления процессами. Мы будем использовать их в последующих разделах для создания дополнительных функций, таких как `popen` и `system`.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execvp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Все семь функций возвращают `-1` в случае ошибки,
не возвращают управление в случае успеха

Одно из отличий между этими функциями заключается в том, что первые четыре принимают полный путь к файлу, следующие две — только имя файла и последняя — дескриптор файла. Аргумент `filename` интерпретируется следующим образом:

- если аргумент `filename` содержит символ слеша, он интерпретируется как полный путь к файлу;
- иначе производится поиск выполняемого файла в каталогах, перечисленных в переменной окружения `PATH`.

Переменная окружения `PATH` содержит список каталогов, разделенных двоеточиями; они называются префиксами пути. Например, строка окружения в формате `name=value`

`PATH=/bin:/usr/bin:/usr/local/bin::`

определяет четыре каталога, в которых будет производиться поиск выполняемых файлов. Последним указан текущий каталог. (Пустой префикс также означает те-

кущий каталог. Он может быть определен двоеточием в начале, двумя двоеточиями в середине или двоеточием в конце подстроки *value*.)

По причинам, связанным с безопасностью системы, никогда не включайте текущий каталог в переменную окружения PATH. Подробности в [Garfinkel et al., 2003].

Если функция `exec1p` или `execvp` находит выполняемый файл, используя один из префиксов пути, но этот файл не является двоичным выполняемым файлом, скенированным редактором связей, функция предположит, что найденный файл является сценарием командной оболочки, и попытается вызывать `/bin/sh` с именем файла в качестве аргумента.

Функция `fexecve` вообще перекладывает задачу поиска выполняемого файла на вызывающую программу. Используя файловый дескриптор, вызывающая программа может проверить, действительно ли файл является тем, который требуется выполнить, и запустить его. Иначе злоумышленник с соответствующими привилегиями мог бы подменить выполняемый файл (или часть пути к выполняемому файлу) после того, как он будет найден и проверен, но до того, как программа запустит его (вспомните обсуждение ошибок «проверка перед использованием» (time-of-check-to-timeof-use, TOCTTOU) в разделе 3.3).

Следующее различие касается передачи списка аргументов (символ `l` в имени означает список (*list*), `v` — вектор, или массив (*vector*)). Функции требуют, чтобы каждый из аргументов командной строки новой программы был оформлен в виде отдельного аргумента функции. Конец списка аргументов отмечается пустым указателем. Для других четырех функций (`execv`, `execvp`, `execve` и `fexecve`) необходимо сформировать массив указателей на аргументы командной строки и передать адрес этого массива в качестве аргумента.

До появления прототипов ISO C было принято показывать аргументы командной строки, передаваемые функциям `exec1`, `exec1p` и `execle`, так:

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

В таком прототипе ясно видно, что заключительный аргумент функции является пустым указателем. Если этот пустой указатель задается как `0`, мы должны явно привести его к типу указателя; если этого не сделать, он будет интерпретироваться как целочисленный аргумент. Если при этом размер целочисленного типа не будет совпадать с размером типа `char *`, то фактически функция `exec` получит неверные аргументы.

И последнее различие — передача списка переменных окружения новой программе. Три функции, имена которых оканчиваются на `e` (`execle`, `execve` и `fexecve`), позволяют передать массив указателей на строки окружения. Остальные четыре функции для передачи копии среды окружения новой программе используют переменную `environ` вызывающего процесса. (Вспомните обсуждение строк окружения в разделе 7.9 и загляните в табл. 7.2. Там мы упоминали, что если система поддерживает функции `setenv` и `putenv`, можно изменить текущую среду окружения и среду окружения любых дочерних процессов, но нельзя изменить среду окружения родительского процесса.) Обычно окружение процесса передается дочерним процессам без изменения, но иногда требуется создать особую

среду для дочернего процесса. Пример такого случая — программа `login`, которая инициализирует новую командную оболочку. Обычно программа `login` создает определенное окружение с небольшим количеством переменных и позволяет нам через файл начального запуска командной оболочки добавить свои переменные окружения при входе в систему.

До появления прототипов ISO C аргументы функции `execle` принято было показывать так:

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

В таком прототипе ясно видно, что заключительный аргумент функции является адресом массива указателей на строки окружения. Прототипы стандарта ISO C не показывают этого, в них все аргументы командной строки, пустой указатель и указатель `envp` заменяются многоточием (...).

Аргументы всех семи функций семейства `exec` достаточно сложно запомнить. Но буквы в именах функций немного помогают в этом. Буква `p` означает, что функция принимает аргумент `filename` и использует переменную окружения `PATH`, чтобы найти выполняемый файл. Буква `l` означает, что функция принимает список аргументов, а буква `v` — что она принимает массив (вектор) `argv[]`. Наконец, буква `e` означает, что функция принимает массив `envp[]` вместо текущего окружения. В табл. 8.6 показаны различия между этими семью функциями.

Таблица 8.6. Различия между семью функциями семейства `exec`

Функция	pathname	filename	fd	Список аргументов	argv[]	environ	envp[]
<code>exec1</code>	✓			✓		✓	
<code>execlp</code>		✓		✓		✓	
<code>execle</code>	✓			✓			✓
<code>execv</code>	✓				✓	✓	
<code>execvp</code>		✓			✓	✓	
<code>execve</code>	✓				✓		✓
<code>fexecve</code>			✓		✓		✓
буква в имени		p	f	l	v		e

Каждая система накладывает свои ограничения на размер списков с аргументами командной строки и переменными окружения. Из раздела 2.5.2 и табл. 2.9 следует, что этот предел задается с помощью константы `ARG_MAX`. Для POSIX.1-совместимых систем его значение должно быть не менее 4096 байт. Иногда приходится сталкиваться с этим пределом при использовании масок командного интерпретатора для создания списка файлов. Например, в некоторых системах команда

```
grep getrlimit /usr/share/man/*/*
```

может выдать сообщение об ошибке

`Argument list too long`

то есть «список аргументов слишком велик».

В ранних версиях System V этот предел составлял 5120 байт. Ранние версии BSD имели предел 20 480 байт. В современных системах этот предел намного выше. (См. данные вывода программы из листинга 2.2, приведенные в табл. 2.13.)

Чтобы обойти ограничение на размер списка, можно воспользоваться командой `xargs(1)`, которая способна обрабатывать списки аргументов большого размера. Например, чтобы отыскать все вхождения слова `getrlimit` в страницах справочного руководства вашей системы, можно использовать такую команду:

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

Однако если файлы со страницами справочного руководства сжаты, лучше попробовать так:

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

Мы использовали параметр `-type f` команды `find`, чтобы ограничить список обычными файлами, поскольку команды `grep` не способны производить поиск по шаблону в каталогах и мы хотим избежать ненужных сообщений об ошибках.

Уже отмечалось, что идентификатор процесса не изменяется после вызова функции `exec`. Кроме того, новая программа наследует от вызывающего процесса ряд дополнительных характеристик:

- идентификатор процесса и идентификатор родительского процесса;
- реальный идентификатор пользователя и реальный идентификатор группы;
- идентификаторы дополнительных групп;
- идентификатор группы процессов;
- идентификатор сеанса;
- управляющий терминал;
- время, оставшееся до срабатывания таймера;
- текущий рабочий каталог;
- корневой каталог;
- маску режима создания файлов;
- блокировки файлов;
- маску сигналов процесса;
- сигналы, ожидающие обработки;
- ограничения на ресурсы;
- значение приоритета (в XSI-совместимых системах, как описывается в разделе 8.16);
- значения `tms_utime`, `tms_stime`, `tms_cutime` и `tms_cstime`.

Судьба открытых файлов зависит от значения флага close-on-exes (закрыть-при-вызове-exes) в их дескрипторах. Вспомните рис. 3.1 и упоминание о флаге FD_CLOEXEC в разделе 3.14. Там мы говорили, что каждый дескриптор, открытый процессом, имеет флаг close-on-exes. Если этот флаг установлен, дескриптор закрывается функцией `exec`. Иначе дескриптор остается открытым. По умолчанию после вызова `exec` дескриптор остается открытым, если флаг close-on-exes не был специально установлен с помощью функции `fcntl`.

Стандарт POSIX.1 требует, чтобы открытые каталоги (вспомните функцию `opendir` из раздела 4.22) обязательно закрывались при вызове функции `exec`. Обычно это обеспечивает функция `opendir`, которая вызывает `fcntl`, чтобы установить флаг close-on-exes для дескриптора, соответствующего открытому файлу каталога.

Обратите внимание, что реальные идентификаторы пользователя и группы не изменяются при вызове функции `exec`, но эффективные идентификаторы могут измениться в зависимости от состояния битов set-user-ID и set-group-ID файла запускаемой программы. Если бит set-user-ID установлен, в качестве эффективного идентификатора пользователя процесса принимается идентификатор владельца файла программы. В противном случае эффективный идентификатор пользователя не изменяется (он не устанавливается равным реальному идентификатору пользователя). Эффективный идентификатор группы устанавливается аналогично.

В большинстве реализаций UNIX только одна из этих семи функций, `execve`, является системным вызовом. Остальные шесть — обычные библиотечные функции, которые в конечном счете обращаются к этому системному вызову. На рис. 8.2 изображена схема взаимоотношений между семью функциями `exec`.

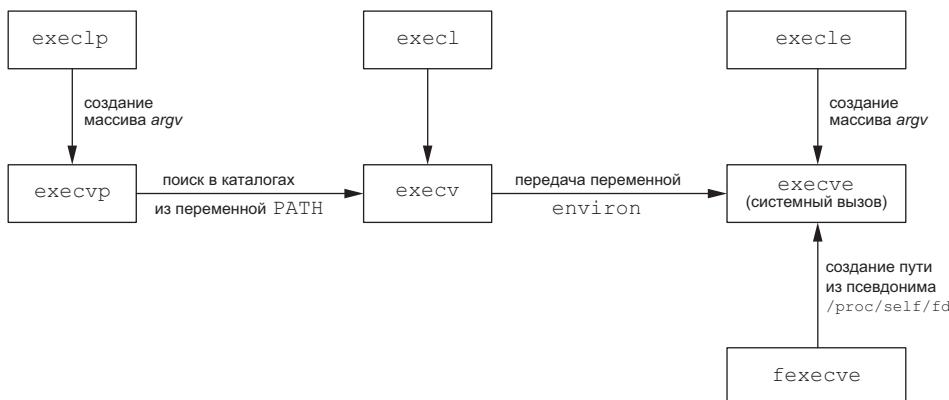


Рис. 8.2. Взаимоотношения между семью функциями `exec`

В соответствии с этой схемой библиотечные функции `execvp` и `execvpe` обрабатывают переменную окружения `PATH` в поисках первого каталога, содержащего выполняемый файл `filename`. Для преобразования дескриптора файла в строку пути к нему, чтобы затем передать ее системному вызову `execve` для запуска программы, библиотечная функция `fexecve` использует `/proc`.

Так реализована функция `fexecve` в OC FreeBSD 8.0 и Linux 3.2.0. В других системах может использоваться иной подход. Например, системы, не имеющие `/proc` или `/dev/fd`, могли бы реализовать `fexecve` как системный вызов, транслирующий дескриптор файла в индексный узел (*i-node*), `execve` — как системный вызов, транслирующий строку пути к файлу в индексный узел, а весь остальной код `exec`, общий для `execve` и `fexecve`, — в отдельную функцию, принимающую индексный узел файла, который требуется выполнить.

Пример

Программа в листинге 8.8 демонстрирует работу с функциями `exec`.

Листинг 8.8. Пример использования функций `exec`

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* задать полный путь к файлу и среду окружения */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                   "MY ARG2", (char *)0, env_init) < 0)
            err_sys("ошибка вызова функции execle");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("ошибка вызова функции wait");

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* задать имя файла, наследовать среду окружения */
        if (execlp("echoall", "echoall", "только 1 аргумент", (char *)0) < 0)
            err_sys("ошибка вызова функции execlp");
    }

    exit(0);
}
```

Сначала программа вызывает функцию `execle`, которая требует указать полный путь к файлу и среду окружения. Далее вызывается функция `execlp`, которой передается имя файла, а среда окружения наследуется новой программой от вызывающего процесса. В данном примере обращение к функции `execlp` не завершается ошибкой только потому, что каталог `/home/sar/bin` входит в переменную `PATH`. Кроме того, обратите внимание, что в качестве первого аргумента (`argv[0]`) командной строки новой программы мы передаем только имя файла. Некоторые командные оболочки передают в этом аргументе полный путь к файлу. Но это просто соглашение, на самом деле в `argv[0]` можно записать любую строку. Команда `login` именно так и поступает, когда запускает командную оболочку. Перед ее запуском `login` добавляет в начало строки `argv[0]` дефис, тем самым сообщая

командной оболочке, что она вызывается как оболочка входа в систему. В этом случае она производит запуск команд начальной настройки, в то время как при обычном вызове командная оболочка этого не делает.

Программа `echoall`, дважды запускаемая программой из листинга 8.8, приведена в листинге 8.9. Эта простенькая программа выводит все аргументы командной строки и список переменных окружения.

Листинг 8.9. Выводит все аргументы командной строки и переменные окружения

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int          i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++) /* вывести все аргументы командной строки */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++) /* и все переменные окружения */
        printf("%s\n", *ptr);

    exit(0);
}
```

После запуска программы из листинга 8.9 мы получили следующие результаты:

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: только 1 аргумент
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                         еще 47 строк здесь не показаны
HOME=/home/sar
```

Обратите внимание, что перед выводом значения `argv[0]` для второго вызова функции `exec` появилось приглашение командной оболочки. Произошло это потому, что родительский процесс не стал ждать, пока этот потомок завершит свою работу.

8.11. Изменение идентификаторов пользователя и группы

В системе UNIX предоставление привилегий (таких, как возможность изменять текущую дату) и управление доступом к файлам (например, право на чтение или запись) основаны на идентификаторах пользователя и группы. Когда программе

необходимы дополнительные привилегии, чтобы получить доступ к ресурсам, недоступным в настоящее время, она должна изменить свой идентификатор пользователя или группы на идентификатор, который имеет соответствующие привилегии. Точно так же, чтобы понизить свои привилегии или предотвратить доступ к некоторым ресурсам, программа должна изменить идентификатор пользователя или группы на идентификатор, не обладающий указанными привилегиями или достаточными правами для обращения к ресурсу.

Вообще при разработке приложений следует использовать принцип *наименьших привилегий*. Следуя этому принципу, приложения должны использовать минимальный набор привилегий, необходимый для решения своих задач. Это уменьшает вероятность, что злоумышленник сможет «обмануть» систему безопасности, используя программы и их привилегии непредусмотренным способом.

Изменить реальный и эффективный идентификаторы пользователя можно с помощью функции `setuid`. Аналогично с помощью функции `setgid` можно изменить реальный и эффективный идентификаторы группы.

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Существуют определенные правила, согласно которым изменяются идентификаторы. Рассмотрим их на примере идентификатора пользователя. (Все перечисленное ниже в равной степени относится к идентификатору группы.)

- Если процесс обладает привилегиями суперпользователя, функция `setuid` устанавливает реальный, эффективный и сохраненный идентификаторы пользователя в соответствии с аргументом `uid`.
- Если процесс не обладает привилегиями суперпользователя, но аргумент `uid` совпадает с реальным или сохраненным идентификатором пользователя, функция `setuid` изменяет только эффективный идентификатор. Реальный и сохраненный идентификаторы не меняются.
- Если ни одно из этих условий не соблюдено, `setuid` возвращает значение `-1` и записывает в переменную `errno` код ошибки `EPERM`.

Здесь предполагается, что конфигурационный параметр `_POSIX_SAVED_IDS` имеет значение `true`. Если эта функциональная возможность не предоставляется вашей системой, исключите из вышеприведенных правил упоминание о сохраненном идентификаторе.

Сохраненные идентификаторы стали обязательными для реализации в версии POSIX.1 от 2001 года. В более ранних версиях POSIX эта функциональная особенность относилась к разряду необязательных. Чтобы узнать, поддерживается ли она системой, приложение может проверить константу `_POSIX_SAVED_IDS` во время компиляции или вызвать функцию `sysconf` с аргументом `_SC_SAVED_IDS` во время выполнения.

Можно сформулировать несколько правил относительно трех идентификаторов пользователя.

- Изменить реальный идентификатор пользователя может только процесс, обладающий привилегиями суперпользователя. Как правило, реальный идентификатор пользователя устанавливается программой `login(1)` при входе в систему и никогда не изменяется. Поскольку `login` является процессом, обладающим привилегиями суперпользователя, с помощью функции `setuid` он устанавливает все три идентификатора пользователя.
- Эффективный идентификатор пользователя устанавливается функцией `exec`, только когда файл программы имеет установленный бит set-user-ID. Если этот бит не установлен, функция `exec` не изменяет эффективный идентификатор пользователя. В любой момент времени можно вызвать функцию `setuid`, чтобы установить эффективный идентификатор равным реальному или сохраненному идентификатору. Но, как правило, нельзя установить эффективный идентификатор пользователя в произвольное значение.
- Функция `exec` копирует эффективный идентификатор пользователя в сохраненный. Если файл программы имеет установленный бит set-user-ID, эта копия сохраняется после того, как функция `exec` установит эффективный идентификатор равным идентификатору владельца файла.

В табл. 8.7 обобщаются возможные варианты изменения этих трех идентификаторов.

Таблица 8.7. Варианты изменения идентификаторов пользователя

Идентификатор	exec		setuid(uid)	
	Бит set-user-ID выключен	Бит set-user-ID включен	Суперпользователь	Непривилегированный пользователь
Реальный	Не изменяется	Не изменяется	Устанавливается в соответствии с uid	Не изменяется
Эффективный	Не изменяется	Устанавливается в соответствии с идентификатором владельца файла программы	Устанавливается в соответствии с uid	
Сохраненный	Копия эффективного идентификатора	Копия эффективного идентификатора	Устанавливается в соответствии с uid	Не изменяется

Обратите внимание, что с помощью функций `getuid` и `geteuid`, описанных в разделе 8.2, можно получить только текущие значения реального и эффективного идентификаторов пользователя. У нас нет возможности получить текущее значение сохраненного идентификатора.

FreeBSD 8.0 и Linux 3.2.0 предоставляют функции `getresuid` и `getresgid` для получения сохраненных идентификаторов пользователя и группы соответственно.

Функции `setreuid` и `setregid`

BSD-системы традиционно поддерживают возможность менять местами реальный и эффективный идентификаторы пользователя с помощью функции `setreuid`.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

В любом из аргументов можно передать значение -1, чтобы указать, что соответствующий идентификатор должен остаться неизменным.

Правило использования этих функций очень простое: непrivилегированный пользователь всегда может поменять местами реальный и эффективный идентификаторы. Это позволяет программам с установленным битом set-user-ID переходить к привилегиям обычного пользователя и затем возвращаться к привилегиям владельца файла программы. Когда в стандарте POSIX.1 появились сохраненные идентификаторы, это правило было расширено, чтобы позволить непrivилегированному процессу также устанавливать эффективный идентификатор пользователя равным сохраненному идентификатору.

Обе функции, `setreuid` и `setregid`, являются расширениями XSI стандарта POSIX.1, поэтому предполагается, что все версии UNIX должны обеспечивать поддержку этих функций.

В версии 4.3BSD отсутствовало понятие сохраненных идентификаторов, описанное выше. Вместо этого использовались функции `setreuid` и `setregid`. Это позволяло непrivилегированному пользователю свободно переключаться между двумя идентификаторами. Однако когда программа, использовавшая эту возможность, запускала командную оболочку, она должна была перед вызовом функции `exec` установить реальный идентификатор пользователя равным идентификатору обычного пользователя. Если этого не сделать, реальный идентификатор может оказаться принадлежащим привилегированному пользователю (как результат вызова функции `setreuid`) и процесс, запущенный из такой командной оболочки, сможет с помощью `setreuid` поменять идентификаторы и получить более высокие привилегии. В качестве меры предосторожности и реальный и эффективный идентификаторы перед вызовом функции `exec` в дочернем процессе устанавливались равными идентификатору обычного пользователя.

Функции `seteuid` и `setegid`

Стандарт POSIX.1 включает еще две функции: `seteuid` и `setegid`. Они очень похожи на функции `setuid` и `setgid`, но изменяют только эффективный идентификатор пользователя и группы.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Непrivилегированный пользователь может установить свой эффективный идентификатор равным реальному или сохраненному идентификатору. Для привилегированного пользователя функция `seteuid` устанавливает только эффективный идентификатор равным значению `uid`. (Этим она отличается от `setuid`, изменяющей все три идентификатора пользователя.)

Рисунок 8.3 наглядно описывает все функции, предназначенные для изменения трех идентификаторов пользователя.

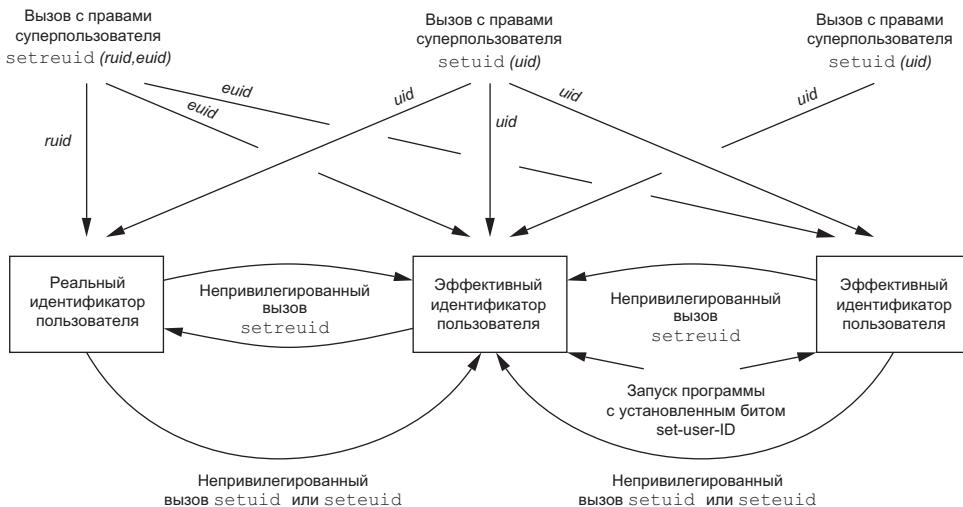


Рис. 8.3. Функции для изменения различных идентификаторов пользователя

Идентификаторы группы

Все, о чем мы говорили в этом разделе, в равной степени относится и к идентификаторам группы. Функции `setgid`, `setregid` и `setegid` не оказывают влияния на идентификаторы дополнительных групп.

Пример

Чтобы увидеть, где может пригодиться сохраненный идентификатор пользователя, рассмотрим программу, которая его использует. В качестве примера возьмем утилиту `at(1)`, которую можно использовать для планирования запуска команд в некоторый момент времени в будущем.

В Linux 3.2.0 программа at имеет установленный бит set-user-ID и принадлежит пользователю daemon. В FreeBSD 8.0, Mac OS X 10.6.8 и Solaris 10 программа at имеет установленный бит set-user-ID и принадлежит пользователю root. Это дает команде at возможность выполнять запись в привилегированные файлы, принадлежащие демону, который будет выполнять требуемые команды от имени пользователя, запустившего команду at. В Linux 3.2.0 программы выполняются демоном atd(8), в FreeBSD 8.0 и Solaris 10 – демоном cron(1M), в Mac OS X 10.6.8 – демоном Launchd(8).

Чтобы предотвратить возможность запуска программ, на выполнение которых пользователь не имеет привилегий, или запись в файлы, недоступные пользователю, команда at и демон, в конечном счете выполняющий команды от имени пользователя, должны переключаться между наборами привилегий: пользовательских и тех, которыми обладает демон. Вот как это происходит:

1. Допустим, что выполняемый файл at имеет установленный бит set-user-ID и его владельцем является пользователь root. Запустив эту программу, мы получили:

реальный идентификатор пользователя	=	идентификатор пользователя, запустившего программу (не изменился)
эффективный идентификатор пользователя	=	root
сохраненный идентификатор пользователя	=	root

2. Первое, что делает команда at, – понижает свои привилегии до уровня привилегий вызвавшего ее пользователя. Для этого она вызывает seteuid, чтобы установить эффективный идентификатор пользователя равным реальному идентификатору пользователя. После чего мы получили:

реальный идентификатор пользователя	=	идентификатор пользователя, запустившего программу (не изменился)
эффективный идентификатор пользователя	=	идентификатор пользователя, запустившего программу (не изменился)
сохраненный идентификатор пользователя	=	root (не изменился)

3. Программа at продолжает выполнение с пользовательскими привилегиями, пока ей не потребуется обратиться к конфигурационным файлам, чтобы сохранить команду, которую требуется выполнить, и время ее запуска. Владельцем этих файлов является демон, выполняющий запуск команд от имени пользователя. Команда at вызывает seteuid, чтобы установить эффективный идентификатор пользователя равным идентификатору пользователя root. Это вполне допустимый вызов, потому что аргумент функции seteuid равен сохраненному идентификатору пользователя. (Именно для этого и нужен сохраненный идентификатор.) После чего получили:

реальный идентификатор пользователя	=	идентификатор пользователя, запустившего программу (не изменился)
эффективный идентификатор пользователя	=	root
сохраненный идентификатор пользователя	=	root (не изменился)

Так как эффективный идентификатор пользователя соответствует идентификатору пользователя `root`, доступ к файлам разрешается.

4. После записи в конфигурационные файлы названия команды, которую требуется запустить, и времени запуска программы `at` вновь понижает свои привилегии вызовом `seteuid`, устанавливая эффективный идентификатор пользователя равным идентификатору пользователя, вызвавшего ее. Это предотвращает возможность неправильного использования повышенных привилегий. В этот момент мы получили:

реальный идентификатор	=	идентификатор пользователя, запустившего пользователя	программу (не изменился)
эффективный идентификатор	=	идентификатор пользователя, запустившего пользователя	программу
сохраненный идентификатор	=	<code>root</code> (не изменился)	пользователя

5. Демон запускается с привилегиями пользователя `root`. Чтобы выполнить запланированную команду от имени пользователя, он вызывает функцию `fork`, и дочерний процесс вызывает `setuid`, чтобы изменить идентификатор пользователя процесса. Поскольку дочерний процесс выполняется с привилегиями `root`, изменяются все три идентификатора. Мы получили:

реальный идентификатор	=	идентификатор пользователя, запустившего пользователя	программу
эффективный идентификатор	=	идентификатор пользователя, запустившего пользователя	программу
сохраненный идентификатор	=	идентификатор пользователя, запустившего пользователя	программу

Теперь демон может безопасно выполнить команду от имени пользователя, потому что она будет иметь доступ только к файлам, и так доступным этому пользователю. Никаких дополнительных привилегий команда не получает.

Используя сохраненный идентификатор пользователя подобным образом, можно пользоваться повышенными привилегиями, которые даются установкой бита set-user-ID для файла программы, только когда они действительно необходимы, а в любой другой момент времени пользоваться обычными привилегиями пользователя, запустившего программу. Если бы у нас отсутствовала возможность переключиться обратно на использование сохраненного идентификатора пользователя, процесс был бы вынужден выполнятьсь с повышенными привилегиями все время (напрашиваясь тем самым на неприятности).

8.12. Интерпретируемые файлы

Все современные версии UNIX поддерживают интерпретируемые файлы. Это обычные текстовые файлы, которые начинаются со строки вида

`#!/ pathname [optional-argument]`

Пробел между восклицательным знаком и параметром *pathname* необязателен. Чаще всего интерпретируемые файлы начинаются со строки

```
#!/bin/sh
```

В *pathname* обычно указывается абсолютный путь к выполняемому файлу интерпретатора, поскольку никаких дополнительных операций над ней не производится (то есть переменная PATH не используется). Распознавание интерпретируемых файлов производится ядром в процессе выполнения системного вызова `exec`. В действительности ядро запускает на выполнение не сам интерпретируемый файл, а программу, указанную в параметре *pathname*, в первой строке. Необходимо понимать разницу между интерпретируемым файлом — обычным текстовым файлом, начинающимся с последовательности `#!` — и интерпретатором, то есть выполняемым файлом, путь к которому указывается в первой строке интерпретируемого файла.

Помните, что существует ограничение на размер первой строки интерпретируемого файла. Это ограничение включает последовательность символов `#!`, параметр *pathname*, необязательные аргументы *optional-argument*, завершающий символ перевода строки и все пробельные символы.

В FreeBSD 8.0 длина первой строки ограничивается 4097 байтами, в Mac OS X 10.6.8 — 513 байтами, в Linux 3.2.0 — 128 байтами, а в Solaris 10 — 1024 байтами.

Пример

Рассмотрим на примере, что делает ядро с параметрами функции `exec`, если запускаемый файл является интерпретируемым файлом и в первой его строке имеется дополнительный аргумент. Программа в листинге 8.10 осуществляет запуск интерпретируемого файла.

Листинг 8.10. Программа, запускающая интерпретируемый файл

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* дочерний процесс */
        if (execel("/home/sar/bin/testinterp",
                   "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("ошибка вызова функции exec1");
    }
    if (waitpid(pid, NULL, 0) < 0) /* родительский процесс */
        err_sys("waitpid error");
    exit(0);
}
```

Ниже приводится содержимое интерпретируемого файла, запускаемого программой из листинга 8.10, и результаты работы программы.

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

Программа `echoarg` (интерпретатор) просто выводит все аргументы, переданные ей в командной строке (это программа из листинга 7.3). Обратите внимание, что когда ядро запускает интерпретатор, в `argv[0]` предается полный путь к выполняемому файлу интерпретатора, `argv[1]` — необязательный аргумент, взятый из интерпретируемого файла, `argv[2]` — полный путь к файлу программы (`/home/sar/bin/testinterp`), а `argv[3]` и `argv[4]` — это второй и третий аргументы функции `exec1` (`myarg1` и `MY ARG2`). Оба аргумента функции `exec1`, `argv[1]` и `argv[2]`, смещаются вправо на две позиции. Обратите внимание: ядро берет аргумент `pathname` из вызова функции `exec1` вместо первого аргумента (`testinterp`), исходя из предположения, что `pathname` содержит больше информации, чем первый аргумент.

Пример

Часто в качестве необязательного аргумента, следующего за именем интерпретатора, передается флаг `-f` для программ, которые его поддерживают. Например, программу `awk(1)` можно запустить как

```
awk -f myfile
```

Таким способом ей сообщается, что текст программы на языке `awk` находится в файле `myfile`.

Системы, производные от System V, часто содержат две версии языка awk. В этих системах awk часто называется «old awk» (старый awk) и соответствует оригинальной версии, распространявшейся в составе Version 7. В противоположность ему nawk (new awk — новый awk) содержит многочисленные расширения и соответствует языку, описанному в [Aho, Kernighan, and Weinberger, 1988]. Эта новая версия предоставляет доступ к аргументам командной строки. ОС Solaris 10 поддерживает обе версии.

Программа awk — это одна из утилит, включенных в POSIX в стандарт 1003.2 (теперь это часть базовых спецификаций POSIX.1 в стандарте Single UNIX Specification). Она также основывается на языке, описанном в книге [Aho, Kernighan, and Weinberger, 1988].

Версия awk в Mac OS X 10.6.8 основана на версии Bell Laboratories, которую компания Lucent сделала свободно распространяемой. В составе FreeBSD 8.0 и в некоторых дистрибутивах Linux распространяется утилита GNU awk, называемая gawk. В этих системах awk является символьической ссылкой на gawk. Утилита gawk соответствует стандарту POSIX, но при этом также включает ряд дополнительных расширений. Поскольку awk от Bell Laboratories и gawk представляют собой более современные версии, им следует отдавать предпочтение перед nawk или «старым awk». (Версия awk от Bell Laboratories доступна по адресу <http://cm.bell-labs.com/cm/cs/awkbook/index.html>.)

Флаг `-f` позволяет оформлять интерпретируемый файл так:

```
#!/bin/awk -f
(далее следует программа на языке awk)
```

Например, в листинге 8.11 приводится содержимое интерпретируемого файла `/usr/local/bin/awkexample`.

Листинг 8.11. Интерпретируемый файл с программой на языке awk

```
#!/bin/awk -f
# Примечание: в ОС Solaris следует использовать nawk
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Если включить каталог `/usr/local/bin` в переменную окружения `PATH`, мы сможем запустить программу из листинга 8.11 (при условии, что у нас есть право на выполнение), как показано ниже.

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

При запуске программа `/bin/awk` получит следующие аргументы командной строки:

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

Интерпретатору передается полный путь к интерпретируемому файлу (`/usr/local/bin/awkexample`). Простого имени файла (которое мы набрали в командной строке) недостаточно, поскольку не предполагается, что интерпретатор (в данном случае `/bin/awk`) сможет определить местонахождение файла, используя переменную окружения `PATH`. Когда `awk` начинает разбирать интерпретируемый файл, он игнорирует первую строку, так как в языке `awk` символ `#` означает начало комментария.

Мы можем проверить аргументы командной строки с помощью следующей последовательности команд:

<pre>\$ /bin/su Password: # mv /bin/awk /bin/awk.save # cp /home/sar/bin/echoarg /bin/awk # suspend</pre>	<i>получаем привилегии суперпользователя вводим пароль суперпользователя сохраним оригинальный файл программы и заменим его на время приостановим работу командной оболочки суперпользователя</i>
<pre>[1] + Stopped /bin/su \$ awkexample file1 FILENAME2 f3 argv[0]: /bin/awk argv[1]: -f argv[2]: /usr/local/bin/awkexample argv[3]: file1 argv[4]: FILENAME2</pre>	

```

argv[5]: f3
$ fg
/bin/su
# mv /bin/awk.save /bin/awk
# exit

```

возобновим работу командной оболочки
 суперпользователя

восстановим оригинальный файл программы
 и покинем командную оболочку

В этом примере флаг `-f` совершенно необходим интерпретатору. Как уже говорилось, он сообщает `awk`, где находится текст программы на языке `awk`. Если убрать этот флаг из интерпретируемого файла, при его запуске мы получим сообщение об ошибке. Точный текст сообщения зависит от местонахождения интерпретируемого файла и от того, представляют ли остальные аргументы существующие файлы. Это происходит потому, что аргументы командной строки приобретают вид

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

и в результате `awk` пытается интерпретировать строку `/usr/local/bin/awkexample` как текст программы на языке `awk`. Если бы отсутствовала возможность передавать хотя бы один необязательный аргумент интерпретатору (в данном случае `-f`), интерпретируемые файлы были бы пригодны к использованию только с командными оболочками.

Действительно ли так необходимы интерпретируемые файлы? На самом деле нет. Но они дают массу удобств для пользователей, хотя и за счет некоторого увеличения нагрузки на ядро (поскольку именно ядро их распознает и запускает указанный интерпретатор). Интерпретируемые файлы удобны по следующим причинам.

1. Они скрывают тот факт, что программа фактически является сценарием на том или ином языке. Так, например, запустить программу из листинга 8.11 можно с помощью такой команды:

```
awkexample необязательные-аргументы
```

Совсем не обязательно помнить, что программа в действительности является сценарием на языке `awk`, который пришлось бы запускать командой

```
awk -f awkexample необязательные-аргументы
```

2. Интерпретируемые сценарии дают выигрыш в эффективности. Вернемся к предыдущему примеру. Мы можем скрыть, что файл является сценарием на языке `awk`, заключив текст программы в сценарий командной оболочки:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

Но такой подход требует от системы дополнительной работы. Прежде всего команная оболочка прочитает команду и попытается выполнить ее с помощью функции `exec1p`. Поскольку сценарий командной оболочки является выполняемым файлом, но не содержит машинных инструкций, возвращается признак ошибки, и `exec1p` предположит, что это файл сценария (как это и есть на самом деле). Тогда запустится программа `/bin/sh`, которой в качестве аргумента пере-

дается имя файла сценария. Командная оболочка запустит сценарий, но чтобы запустить `awk`, она вызовет функции `fork`, `exec` и `wait`. Поэтому «обертывание» сценариев на других языках в сценарии командной оболочки приводит к увеличению нагрузки.

- Интерпретируемые файлы позволяют писать сценарии на языках других командных оболочек, отличных от `/bin/sh`. Когда функция `execlp` обнаруживает, что выполняемый файл не содержит машинных инструкций, она вызывает командную оболочку, и это всегда `/bin/sh`. Однако, используя возможность указания интерпретатора в первой строке интерпретируемого файла, мы можем просто написать

```
#!/bin/csh
(далее следует текст сценария на языке командной оболочки C shell)
```

Опять же, этот код можно завернуть в сценарий командной оболочки `/bin/sh` (которая вызовет C shell), как показано немного выше, но это повлечет за собой дополнительную нагрузку.

Ни один из указанных приемов не работал бы, если бы эти три командные оболочки и `awk` не использовали символ `#` в качестве знака комментария.

8.13. Функция `system`

Функция `system` дает удобный способ выполнения команд внутри программы. Например, допустим, что требуется записать строку с датой и временем в некоторый файл. Для этого можно было бы использовать функции, описанные в разделе 6.10: получить текущее календарное время с помощью функции `time`, затем преобразовать его в структуру `tm` с помощью функции `localtime`, сформировать строку с помощью функции `strftime` и записать результат в файл. Однако гораздо проще сделать так:

```
system("date > file");
```

Функция `system` определяется стандартом ISO C, но порядок взаимодействия с ней зависит от системы. Стандарт POSIX.1 включает интерфейс `system`, расширяя определение ISO C, чтобы уточнить поведение функции в среде POSIX.

```
#include <stdlib.h>
int system(const char *cmdstring);
```

Возвращает: см. ниже

Если в аргументе `cmdstring` передается пустой указатель, функция `system` возвращает ненулевое значение, только если командный процессор доступен. Таким способом можно проверить, поддерживается ли функция `system` в данной системе. В системах UNIX она поддерживается всегда.

Поскольку функция `system` реализована на основе функций `fork`, `exec` и `waitpid`, она может возвращать значения трех типов.

1. Если функция `fork` терпит неудачу или функция `waitpid` возвращает код ошибки, отличный от `EINTR`, функция `system` возвращает значение `-1`.
2. Если функция `exec` терпит неудачу, это означает, что командная оболочка не может быть запущена, и функция `system` возвращает значение, как если бы командная оболочка вызвала функцию `exit(127)`.
3. Когда обращение ко всем трем функциям — `fork`, `exec` и `waitpid` — заканчивается успехом, функция `system` возвращает код завершения командной оболочки в формате, предназначенном для функции `waitpid`.

Некоторые старые реализации функции `system` возвращали код ошибки `EINTR`, если выполнение функции `waitpid` было прервано сигналом. Поскольку нет достаточно ясной стратегии восстановления после такой ошибки (так как идентификатор дочернего процесса скрыт от вызывающей программы), стандарт POSIX позднее добавил требование, чтобы функция `system` не возвращала в этом случае код ошибки. (Прерывание системных вызовов рассматривается в разделе 10.5.)

В листинге 8.12 приводится пример реализации функции `system`. Единственный ее недостаток — отсутствие возможности обработки сигналов. Эту возможность мы добавим в разделе 10.18.

Флаг `-c` сообщает командной оболочке, что следующий за ней аргумент — это команда, которую нужно выполнить. Командная оболочка анализирует эту строку и разбивает ее на отдельные аргументы. Аргумент `cmdstring` может содержать любую допустимую команду оболочки. Например, для перенаправления ввода/вывода можно использовать операторы `<` и `>`.

Чтобы выполнить команду, не прибегая к услугам командной оболочки, потребовалось бы значительные усилия. Прежде всего пришлось бы вызвать функцию `execvp` вместо `exec1`, чтобы использовать переменную окружения `PATH` подобно командной оболочке. Также пришлось бы разбивать командную строку на отдельные аргументы, чтобы передать их функции `execvp`. И наконец, мы не смогли бы воспользоваться метасимволами командной оболочки.

Обратите внимание, что вместо функции `exit` мы вызвали функцию `_exit`. Сделано это для предотвращения сброса буферов ввода/вывода, которые могли быть унаследованы дочерним процессом от родительского при вызове функции `fork`.

Листинг 8.12. Функция `system` без обработки сигналов

```
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int
system(const char *cmdstring) /* версия без обработки сигналов */
{
    pid_t    pid;
    int      status;
```

```

if (cmdstring == NULL)
    return(1); /* UNIX всегда поддерживает командный процессор */

if ((pid = fork()) < 0)
    status = -1; /* превышено максимальное количество процессов */
} else if (pid == 0) { /* дочерний процесс */
    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127); /* ошибка вызова функции execl */
} else { /* родительский процесс */
    while (waitpid(pid, &status, 0) < 0) {
        if (errno != EINTR) {
            status = -1; /* waitpid вернула ошибку, отличную от EINTR */
            break;
        }
    }
}
return(status);
}

```

Мы можем протестировать нашу версию функции `system` с помощью программы в листинге 8.13. (Исходный код функции `pr_exit` вы найдете в листинге 8.3.)

Листинг 8.13. Вызов функции system

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("ошибка вызова функции system()");

    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("ошибка вызова функции system()");

    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("ошибка вызова функции system()");

    pr_exit(status);

    exit(0);
}

```

Запуск программы из листинга 8.13 дал следующие результаты:

```

$ ./a.out
Sat Feb 25 19:36:59 EST 2012
нормальное завершение, код выхода = 0      команда date
sh: nosuchcommand: command not found
нормальное завершение, код выхода = 127      команда nosuchcommand
sar      console Jan  1 14:59
sar      ttys000  Feb  7 19:08

```

```
sar      ttys001 Jan 15 15:28
sar      ttys002 Jan 15 21:50
sar      ttys003 Jan 21 16:02
нормальное завершение, код выхода = 44      команда exit
```

Основное преимущество использования функции `system` вместо прямого обращения к функциям `fork` и `exec` в том, что она производит все необходимые действия по обработке ошибочных ситуаций, а также (в нашей следующей версии этой функции, в разделе 10.18) по обработке сигналов.

Ранние версии UNIX, включая SVR3.2 и 4.3BSD, не имели функции `waitpid`. Вместо этого родительский процесс дожидался завершения работы потомка с помощью примерно такой инструкции:

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
;
```

Однако в этом случае возникает проблема, если у процесса уже имеются дочерние процессы, запущенные до обращения к функции `system`. Поскольку показанный выше цикл `while` продолжает работу, пока не завершится дочерний процесс, созданный функцией `system`, если какой-либо из дочерних процессов, запущенных ранее, завершится до процесса, указанного в переменной `pid`, его идентификатор и код завершения будут потеряны в цикле `while`. Эта неспособность функции `wait` ждать завершения определенного дочернего процесса — одна из причин добавления функции `waitpid`, приводимых в обосновании «POSIX.1 Rationale». В разделе 15.3 мы увидим, что та же проблема возникает при работе с функциями `popen` и `pclose`, если система не поддерживает функцию `waitpid`.

Программы с установленным битом set-user-ID

Что произойдет, если вызвать функцию `system` из программы с установленным битом set-user-ID? Такой вызов создает брешь в системе безопасности, и этого нельзя допускать никогда. В листинге 8.14 приводится программа, которая просто вызывает функцию `system` для обработки своих аргументов командной строки.

Листинг 8.14. Обработка аргументов командной строки с помощью функции `system`

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int      status;

    if (argc < 2)
        err_quit("требуется хотя бы один аргумент командной строки");

    if ((status = system(argv[1])) < 0)
        err_sys("ошибка вызова функции system()");

    pr_exit(status);

    exit(0);
}
```

Скомпилируем эту программу в выполняемый файл `tsys`.

В листинге 8.15 приводится другая простая программа, которая выводит значения реального и эффективного идентификаторов пользователя.

Листинг 8.15. Вывод реального и эффективного идентификаторов пользователя

```
#include "apue.h"
```

```
int
main(void)
{
    printf("реальный uid = %d, эффективный uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Скомпилируем эту программу в выполняемый файл `printuids`. Запуск обеих программ дал следующие результаты:

```
$ tsys printuids          обычный запуск без дополнительных привилегий
реальный uid = 205, эффективный uid = 205
нормальное завершение, код выхода = 0
$ su                      получаем права суперпользователя
Password:                 вводим пароль суперпользователя
# chown root tsys         меняем владельца файла
# chmod u+s tsys          устанавливаем бит set-user-ID
# ls -l tsys               проверяем владельца и права доступа
-rwsrwxr-x 1 root 7888 Feb 25 22:13 tsys
# exit                     покидаем командную оболочку суперпользователя
$ tsys printuids          реальный uid = 205, эффективный uid = 0     вот она, брешь в системе безопасности
нормальное завершение, код выхода = 0
```

Привилегии суперпользователя, которые мы дали программе `tsys`, сохранились после вызовов функций `fork` и `exec`, которые производит функция `system`.

Некоторые реализации закрывают эту брешь в системе безопасности, изменяя командную оболочку `/bin/sh` так, чтобы она устанавливалась эффективный идентификатор пользователя равным реальному, если они не совпадают. В таких системах данный пример будет работать не так, как показано выше. Вместо этого будет выводиться одно и то же значение эффективного идентификатора пользователя, независимо от значения бита set-user-ID у файла программы, вызываемой функцией system.

Если предполагается, что программа будет работать с повышенными привилегиями — с установленными битами `set-user-ID` или `set-group-ID` — и порождать другие процессы, она должна делать это непосредственно с помощью функций `fork` и `exec`, выполняя переход к привилегиям обычного пользователя после вызова `fork` и перед вызовом `exec`. Функция `system` никогда не должна использоваться в программах с установленными битами `set-user-ID` или `set-group-ID`.

Еще одна из причин заключается в том, что функция system вызывает командную оболочку для разбора аргументов командной строки, а сама оболочка использует значение переменной окружения IFS в качестве разделителя полей во входной строке. Ранние версии

командной оболочки при запуске не сбрасывали эту переменную к стандартному набору символов. Это позволяло злоумышленнику изменить значение переменной окружения *IFS* до вызова функции *system* и заставить ее выполнить совсем другую команду.

8.14. Учет использования ресурсов процессами

Большинство версий UNIX дают возможность вести учет использования ресурсов процессами. Когда режим учета включен, ядро создает запись всякий раз, когда процесс завершает работу. Обычно это небольшой блок двоичных данных, в котором хранится имя команды, количество использованного процессорного времени, идентификаторы пользователя и группы, время запуска и т. п. В этом разделе мы поближе рассмотрим записи учета, поскольку это дает возможность взглянуть на процессы с другой стороны, используя для этого функцию *freadd* из раздела 5.9.

Возможность учета использования ресурсов процессами не определяется ни одним из стандартов. Поэтому все реализации имеют существенные различия. Например, учет ввода/вывода в OC Solaris 10 производится в байтах, тогда как в FreeBSD 8.0 и Mac OSX 10.6.8 – в блоках, хотя при этом не делается никаких различий между размерами блоков, что делает такой учет достаточно бесполезным. С другой стороны, Linux 3.2.0 вообще не поддерживает учет операций ввода/вывода.

*Кроме того, каждая реализация имеет собственный набор административных команд для работы с учетной информацией. Например, Solaris предоставляет для сбора, обработки и вывода учетных сведений команды *runacct(1m)* и *acctcom(1)*, а FreeBSD – команду *sa(8)*.*

Функция, о которой пока ничего не рассказывалось (*acct*), включает и выключает режим сбора статистической информации о процессах. Эта функция используется в единственной программе — *accton(8)* (она, к счастью, на всех платформах называется одинаково). Чтобы включить режим учета, суперпользователь должен запустить команду *accton* с аргументом командной строки, в котором передается полный путь к файлу для записи учетной информации, обычно */var/account/acct* в FreeBSD и Mac OS X, */var/log/account/pacct* — в Linux и */var/adm/pacct* — в Solaris. Учет выключается, когда команда *accton* запускается без параметров.

Структура записи с учетной информацией определена в заголовочном файле *<sys/acct.h>* и выглядит, как показано ниже, хотя в разных системах могут быть некоторые отличия:

```
typedef u_short comp_t; /* 3-битная, по основанию 8, экспонента;
                         /* 13 бит – мантисса */
struct acct
{
    char ac_flag;      /* флаг (табл. 8.8) */
    char ac_stat;      /* код завершения (только номер сигнала)
                         /* и признак создания файла core) (только в Solaris) */
    uid_t ac_uid;      /* реальный идентификатор пользователя */
    gid_t ac_gid;      /* реальный идентификатор группы */
    dev_t ac_tty;      /* управляющий терминал */
```

```

time_t ac_bttime; /* календарное время запуска */
comp_t ac_utime; /* пользовательское время */
comp_t ac_stime; /* системное время */
comp_t ac_etime; /* общее время работы */
comp_t ac_mem; /* средний расход памяти */
comp_t ac_io; /* количество переданных байтов (функции read и write) */
/* в "блоках" для BSD-систем */
comp_t ac_rw; /* количество прочитанных и записанных блоков */
/* (отсутствует в BSD-системах) */
char ac_comm[8]; /* имя команды: [8] в Solaris, */
/* [10] в Mac OS X, [16] в FreeBSD и [17] в Linux */
};


```

Значения времени в большинстве платформ хранятся в тактах, кроме FreeBSD, где время хранится в микросекундах. В поле `ac_flag` заносится информация о некоторых событиях, зафиксированных во время работы процесса. Эти события перечислены в табл. 8.8.

Все необходимые данные, такие как количество использованного процессорного времени или объем операций ввода/вывода, хранятся в таблице процессов и инициализируются при создании нового процесса после вызова функции `fork`. Каждая запись формируется и записывается в файл в момент завершения процесса. Эта особенность имеет два следствия.

Во-первых, мы не сможем получить учетную информацию для процессов, которые никогда не завершаются. Процессы, такие как `init`, выполняющиеся от запуска до завершения системы, не генерируют записи с учетной информацией. То же относится и к демонам ядра, которые обычно не завершают работу.

Во-вторых, записи в файле следуют в порядке завершения процессов, а не в порядке запуска. Чтобы определить порядок запуска, придется просмотреть файл с учетной информацией и отсортировать его по календарному времени запуска процессов. Но это даст не совсем точный порядок запуска, так как календарное время измеряется в секундах (раздел 1.10), а на протяжении одной секунды может быть запущено несколько процессов. С другой стороны, общее время работы дается в тактах системных часов (обычно от 60 до 128 тактов в секунду). Но мы не знаем точное время окончания работы процесса; все, что у нас есть, — это время запуска и порядок завершения процессов. Это означает, что даже при том, что общее время работы процесса измеряется более точно, чем время запуска, мы все еще не в состоянии определить точный порядок запуска процессов по тем данным, которые имеются в файле учета.

Каждая запись с учетной информацией соответствует процессу, а не программе. Новая запись создается ядром только при создании нового дочернего процесса вызовом функции `fork`, а не в момент, когда запускается новая программа. Хотя вызов функции `exec` и не приводит к созданию новой записи, тем не менее имя команды изменяется и поэтому сбрасывается флаг `AFORK`. Это означает, что если программа А запускает В, В запускает С и после этого С завершает работу, то такой последовательности запущенных программ будет соответствовать всего одна запись с учетной информацией. Имя команды в этой записи будет соответствовать программе С, но процессорное время будет представлять сумму времени, потраченного всеми тремя программами.

Таблица 8.8. Значения флага ac_flag структуры acct

ac_flag	Описание	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
AFORK	Процесс порожден функцией <code>fork</code> , но без вызова функции <code>exec</code>	✓	✓	✓	✓
ASU	Процесс использовал привилегии суперпользователя		✓	✓	✓
ACORE	Был создан файл с дампом памяти процесса (<code>core</code>)	✓	✓	✓	
AXSIG	Процесс завершился по сигналу	✓	✓	✓	
AEXPND	Расширенная запись с учетными данными				✓
ANVER	Новый формат записи	✓			

Пример

Чтобы получить некоторый объем данных для исследования, создадим тестовую программу, которая реализует следующую схему действий (рис. 8.4).

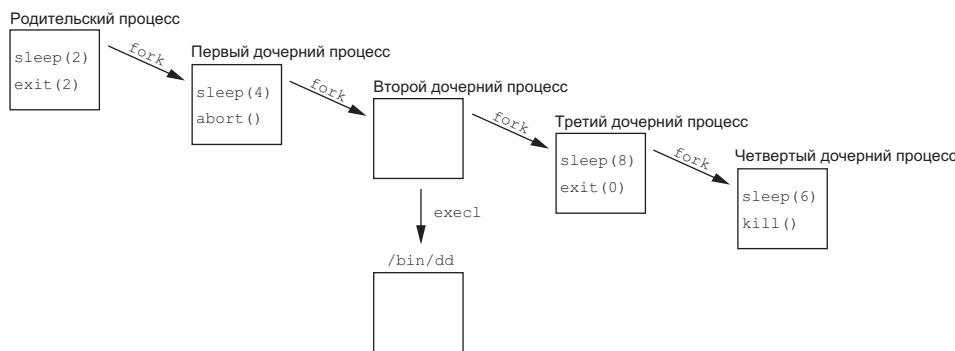


Рис. 8.4. Структура процесса, на примере которого будет рассматриваться учетная информация

Исходный текст программы приводится в листинге 8.16. Эта программа вызывает функцию `fork` четыре раза. Каждый из дочерних процессов выполняет некоторые действия и завершает работу.

Листинг 8.16. Программа генерации учетной информации

```
#include "apue.h"

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0)
```

```

        err_sys("ошибка вызова функции fork");
else if (pid != 0) { /* родительский процесс */
    sleep(2);
    exit(2);           /* завершение с кодом 2 */
}

if ((pid = fork()) < 0)
    err_sys("ошибка вызова функции fork");
else if (pid != 0) { /* первый дочерний процесс */
    sleep(4);
    abort();           /* завершение с созданием файла core */
}

if ((pid = fork()) < 0)
    err_sys("ошибка вызова функции fork");
else if (pid != 0) { /* второй дочерний процесс */
    execl("/bin/dd", "dd", "if=/etc/passwd", "of=/dev/null", NULL);
    exit(7);           /* программа не должна доходить до этой точки */
}

if ((pid = fork()) < 0)
    err_sys("ошибка вызова функции fork");
else if (pid != 0) { /* третий дочерний процесс */
    sleep(8);
    exit(0);           /* нормальный выход */
}

sleep(6);             /* четвертый дочерний процесс */
kill(getpid(),SIGKILL); /* завершить по сигналу без создания файла core */
exit(6);              /* программа не должна доходить до этой точки */
}

```

Запустим эту программу в Solaris и затем выведем учетную информацию с помощью программы из листинга 8.17.

Листинг 8.17. Вывод учетной информации из системного файла учетных данных

```

#include "apue.h"
#include <sys/acct.h>

#if defined(BSD) /* В FreeBSD структура определена иначе */
#define acct acctv2
#define ac_flag ac_trailer.ac_flag
#define FMT "%-*.*s e = %.0f, chars = %.0f, %c %c %c %c\n"
#elif defined(HAS_SA_STAT)
#define FMT "%-*.*s e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#if defined(LINUX)
#define acct acct_v3 /* В OC Linux структура определена иначе */
#endif

#if !defined(HAS_ACORE)
#define ACORE 0
#endif
#if !defined(HAS_AXSIG)
#define AXSIG 0
#endif

#if !defined(BSD)

```

```

static unsigned long
compt2ulong(comp_t comptime) /* преобразовать comp_t в unsigned long */
{
    unsigned long val;
    int exp;

    val = comptime & 0xffff; /* 13 бит - мантисса */
    exp = (comptime >> 13) & 7; /* 3 бита - экспонента (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
#endif

int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE           *fp;

    if (argc != 2)
        err_quit("Использование: pracct имя_файла");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
               (int)sizeof(acdata.ac_comm), acdata.ac_comm,
#ifdef BSD
               acdata.ac_etime, acdata.ac_io,
#else
               compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#endif
#ifdef HAS_AC_STAT
               (unsigned char) acdata.ac_stat,
#endif
               acdata.ac_flag & ACORE ? 'D' : ' ',
               acdata.ac_flag & AXSIG ? 'X' : ' ',
               acdata.ac_flag & AFORK ? 'F' : ' ',
               acdata.ac_flag & ASU ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("ошибка вызова функции read");
    exit(0);
}

```

На платформах, производных от BSD, поле `ac_stat` в структуре `acct` не поддерживается, поэтому мы определили константу `HAS_SA_STAT` для платформ, которые поддерживают это поле. Использование константы с именем, соответствующим функциональной особенности вместо имени платформы, дает более удобочитаемый исходный текст программы и позволяет легко модифицировать ее простым добавлением дополнительных определений в строку команды компиляции. В качестве альтернативы можно было бы использовать в тексте программы

```
#if !defined(BSD) || !defined(MACOS)
```

что по мере переноса программы на другие платформы делает ее все более громоздкой.

Мы определили аналогичные константы, чтобы установить факт поддержки платформой флагов `ACORE` и `AXSIG`. Мы не можем использовать просто имена флагов, потому что в Linux они определены в виде перечисления `enum` и их нельзя использовать в выражении `#ifdef`.

Для тестирования нам необходимо:

1. Обладая привилегиями суперпользователя, включить сбор информации командой `accton`. Обратите внимание, что к моменту, когда команда `accton` завершится, сбор информации уже должен быть включен; поэтому первая запись в учетном файле должна относиться к этой команде.
2. Выйти из командной оболочки суперпользователя и запустить программу из листинга 8.16. В результате в файле учета должно появиться шесть дополнительных записей: одна соответствует завершившейся командной оболочке, в которой мы работали с привилегиями суперпользователя, одна — родительскому процессу тестовой программы и четыре — дочерним процессам, порожденным тестовой программой.

Второй дочерний процесс не создает новый процесс с помощью функции `exec1`. Поэтому ему будет соответствовать одна запись в файле учета.

3. Получить привилегии суперпользователя и отключить сбор информации. Поскольку к моменту завершения команды `accton` сбор информации уже должен быть выключен, в файле учета не должно появиться новой записи, соответствующей этой команде.
4. Запустить программу из листинга 8.17, которая выведет информацию, собранную в учетном файле.

Ниже приводится вывод программы, полученный на шаге 4. Для последующего обсуждения в конце некоторых строк добавлено описание процесса (курсивом).

```
accton  e =      1, chars =   336, stat =  0:  S
sh      e =  1550, chars = 20168, stat =  0:  S
dd      e =      2, chars =  1585, stat =  0:      второй потомок
a.out   e =     202, chars =    0, stat =  0:      родительский процесс
a.out   e =     420, chars =    0, stat = 134: F  первый потомок
a.out   e =     600, chars =    0, stat =   9: F  четвертый потомок
a.out   e =     801, chars =    0, stat =   0: F  третий потомок
```

Для данной системы общее затраченное время измеряется в тактах системных часов. В табл. 2.12 показано, что частота хода системных часов составляет 100 тактов в секунду. Например, вызов `sleep(2)` в родительском процессе соответствует 202 тактам системных часов. Первый потомок на выполнение функции `sleep(4)` затратил 420 тактов системных часов. Обратите внимание, что время, на которое процесс был приостановлен функцией `sleep`, измеряется не совсем точно. (Мы вернемся к этой функции в главе 10.) Вызовы функций `fork` и `exit` также занимают некоторое количество времени.

Обратите также внимание, что поле `ac_stat` соответствует не действительному коду завершения процесса, а той его части, которая обсуждалась в разделе 8.6. Единственная информация, которая хранится в этом поле, — номер сигнала (обычно младшие семь разрядов) и признак создания файла `core` (обычно старший бит), если процесс завершился аварийно. Если процесс завершился нормаль-

но, мы не сможем получить код выхода из файла учета. Для первого потомка код завершения имеет значение 128+6, где 128 – это признак создания файла `core`, а 6 – номер сигнала `SIGABRT` для данной системы, который генерируется функцией `abort`. Значение кода завершения 9 четвертого потомка соответствует номеру сигнала `SIGKILL`. Данный набор учетной информации ничего не сообщает о том, что код выхода (аргумент функции `exit`) родительского процесса равен числу 2, а аргумент функции `exit` в третьем потомке равен 0.

Размер файла `/etc/passwd`, который был скопирован процессом `dd` во втором потомке, составляет 777 байт. Объем операций ввода/вывода в два раза превышает это значение, поскольку 777 байт сначала читаются и затем те же 777 байт записываются. Даже несмотря на то, что вывод производится в пустое устройство, эти байты все равно учитываются. Дополнительные 31 байт – это длина сообщения, выводимого командой `dd` в `stdout`.

Значения поля `ac_flag` в точности соответствуют нашим ожиданиям. Флаг `F` установлен у всех дочерних процессов, за исключением второго, который вызвал функцию `exec1`. Флаг `F` отсутствует у родительского процесса, поскольку команда оболочка, запустившая его, вызвала функцию `fork`, а затем `exec` для файла `a.out`. Первый дочерний процесс вызвал функцию `abort`, которая сгенерировала сигнал `SIGABRT`, что вызвало создание файла `core`. Обратите внимание, что в полученных результатах отсутствуют флаги `D` и `X`, поскольку они не поддерживаются в Solaris; информацию, которую они представляют, можно извлечь из поля `ac_stat`. Четвертый дочерний процесс также завершился по сигналу, но сигнал `SIGKILL` не вызывает создание файла `core`, он просто завершает процесс.

И последнее замечание: первый дочерний процесс имеет нулевой объем операций ввода/вывода, хотя завершился созданием файла `core`. Это говорит о том, что объем операций ввода/вывода, который требуется для создания файла `core`, не учитывается ядром.

8.15. Идентификация пользователя

Любой процесс может узнать свои реальные и эффективные идентификаторы пользователя и группы. Но иногда требуется узнать имя пользователя, запустившего программу. Для этой цели можно было бы вызвать `getpwuid(getuid())`, но что делать, если один и тот же пользователь имеет несколько учетных записей с разными именами, но с одним и тем же числовым идентификатором? (В файле паролей может быть несколько записей с одинаковым числовым идентификатором, чтобы пользователь мог запускать различные командные оболочки при входе в систему.) Как правило, система отслеживает имена, под которыми осуществлялся вход (раздел 6.8), и позволяет получить имя пользователя с помощью функции `getlogin`.

```
#include <unistd.h>
char *getlogin(void);
```

Возвращает указатель на строку с именем пользователя в случае успеха, `NULL` – в случае ошибки

Эта функция может завершиться неудачей, если процесс не присоединен к терминалу, с которого произведен вход пользователя в систему. Такие процессы обычно называются демонами. Они будут обсуждаться в главе 13.

Получив имя пользователя, можно с помощью функции `getpwname` найти соответствующую запись в файле паролей, чтобы, например, определить тип командной оболочки.

Чтобы найти имя пользователя, операционные системы UNIX традиционно вызывали функцию `ttypname` (раздел 18.9) и затем пытались отыскать соответствующую запись в файле `utmp` (раздел 6.8). FreeBSD и Mac OS X сохраняют имя пользователя в структуре сеанса, связанного с записью в таблице процессов, и предоставляют системные вызовы для получения и сохранения этого имени.

ОС System V поддерживала функцию `cuserid`, с помощью которой можно получить имя пользователя. Эта функция вызывала `getLogin` или, в случае ее неудачи, `getpwuid(getuid())`. Стандарт IEEE 1003.11988 определял функцию `cuserid`, но для получения имени она использовала эффективный, а не реальный идентификатор пользователя. Функция `cuserid` была исключена в версии стандарта POSIX.1 1990 года.

Программа `Login(1)` обычно записывает имя пользователя в переменную окружения `LOGNAME`, которая наследуется командной оболочкой после входа в систему. Однако следует помнить, что пользователь может изменить значение этой переменной окружения, поэтому на нее нельзя полагаться при проверке имени пользователя. Вместо этого должна использоваться функция `getLogin`.

8.16. Планирование процессов

Традиционно системами UNIX поддерживается лишь очень грубая настройка приоритетов процессов для планирования. Политика планирования и поддержки приоритетов определяется ядром. Процесс может понизить свой приоритет, изменив так называемый коэффициент уступчивости (то есть изменив значение «коэффициента уступчивости» («nice»), процесс может стать более «уступчивым» и уменьшить выделяемую ему долю процессорного времени). Только привилегированные процессы могут увеличивать свой приоритет.

Расширения реального времени в стандарте POSIX добавляют интерфейсы, позволяющие выбирать из множества классов планирования и производить подстройку их поведения. Здесь мы рассмотрим только интерфейсы, используемые для изменения приоритета процесса; они являются частью расширений XSI в POSIX.1. Дополнительную информацию по расширениям реального времени можно найти в [Gallmeister, 1995].

Согласно стандарту Single UNIX Specification, значения коэффициента уступчивости изменяются в диапазоне от 0 до $(2 * \text{NZERO}) - 1$, хотя некоторые реализации поддерживают диапазон от 0 до $2 * \text{NZERO}$. Более низкие значения коэффициента уступчивости соответствуют более высокому приоритету. Кому-то такое соответствие может показаться странным, но в нем есть определенный смысл: чем более уступчив процесс, тем ниже его приоритет. Константа `NZERO` определяет значение по умолчанию для коэффициента уступчивости в данной системе.

Помните, что в разных системах константа NZERO определяется в разных заголовочных файлах. Кроме того, в Linux 3.2.0 значение NZERO можно получить не только из заголовочного файла, но и вызвав функцию `sysconf` с нестандартным аргументом `_SC_NZERO`.

Процесс может получать и изменять свой коэффициент уступчивости с помощью функции `nice`. Эта функция позволяет процессу воздействовать только на собственный коэффициент уступчивости — ее нельзя использовать для изменения коэффициента уступчивости другого процесса.

```
#include <unistd.h>
int nice(int incr);
```

Возвращает новое значение коэффициента уступчивости —NZERO
в случае успеха, —1 — в случае ошибки

Аргумент `incr` определяет значение, которое будет добавлено к коэффициенту уступчивости вызывающего процесса. Если значение `incr` окажется слишком большим, система просто уменьшит его до максимально допустимого. Аналогично, если значение `incr` окажется слишком маленьким, система увеличит его до минимально допустимого. Так как —1 является допустимым возвращаемым значением, перед вызовом функции `nice` следует очистить переменную `errno` и проверить ее после вызова, если `nice` вернет —1. Если `nice` выполнилась успешно и вернула —1, в переменной `errno` сохранится нулевое значение. Если `errno` будет содержать ненулевое значение, следовательно, вызов `nice` потерпел неудачу.

Функция `getpriority`, как и `nice`, позволяет получить значение коэффициента уступчивости процесса. Но помимо этого `getpriority` может также вернуть значение коэффициента уступчивости для группы родственных процессов.

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

Возвращает значение коэффициента уступчивости между —NZERO
и NZERO—1 в случае успеха, —1 — в случае ошибки

Аргумент `which` может принимать одно из трех значений: `PRIOR_PROCESS`, если операция выполняется для процесса; `PRIOR_PGRP`, если операция выполняется для группы процессов; и `PRIOR_USER`, если операция выполняется для пользователя. Аргумент `which` управляет интерпретацией аргумента `who`, который, в свою очередь, определяет порядок выбора процесса или процессов. Если в аргументе `who` передать 0, операция выполнится для текущего процесса, группы процессов или пользователя (в зависимости от значения аргумента `which`). Когда в аргументе `which` передается `PRIOR_USER`, а в аргументе `who` — 0, выбор процессов производится по реальному идентификатору пользователя текущего процесса. Если аргумент `which` соответствует более чем одному процессу, возвращается наибольший приоритет (наименьшее значение) из всех выбранных процессов.

Для изменения приоритета процесса, группы процессов или всех процессов, принадлежащих определенному пользователю, можно использовать функцию `setpriority`.

```
#include <sys/resource.h>
int setpriority(int which, id_t who, int value);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Назначение аргументов `which` и `who` в точности соответствует назначению одноименных аргументов функции `getpriority`. К значению `value` добавляется константа `NZERO`, и результат становится новым значением коэффициента уступчивости.

Системный вызов nice появился в ранних версиях Research UNIX System для PDP11. Функции getpriority и setpriority появились в 4.2BSD.

Стандарт Single UNIX Specification оставляет за реализациями определение порядка наследования коэффициента уступчивости дочерними приложениями после вызова `fork`. Однако XSI-совместимые системы должны сохранять коэффициент уступчивости после вызова `exec`.

Bo всех четырех обсуждаемых plataформах — FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 — дочерний процесс наследует значение коэффициента уступчивости от своего родителя.

Пример

Программа в листинге 8.18 измеряет эффект изменения приоритета процесса. Два процесса, выполняющиеся одновременно, наращивают свои счетчики. Родительский процесс выполняется с коэффициентом уступчивости по умолчанию, а дочерний изменяет свой коэффициент в соответствии с параметром командной строки. По истечении 10 секунд оба процесса выводят значения своих счетчиков и завершаются. Сравнивая значения счетчиков, полученные процессами с разными приоритетами, можно примерно судить, как значение коэффициента уступчивости влияет на планирование процесса системой.

Листинг 8.18. Определение эффекта влияния изменения коэффициента уступчивости

```
#include "apue.h"
#include <errno.h>
#include <sys/time.h>

#if defined(MACOS)
#include <sys/syslimits.h>
#elif defined(SOLARIS)
#include <limits.h>
#elif defined(BSD)
#include <sys/param.h>
#endif

unsigned long long count;
```

```

struct timeval end;

void
checktime(char *str)
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    if (tv.tv_sec >= end.tv_sec && tv.tv_usec >= end.tv_usec) {
        printf("%s счетчик = %lld\n", str, count);
        exit(0);
    }
}

int
main(int argc, char *argv[])
{
    pid_t pid;
    char *s;
    int nzero, ret;
    int adj = 0;

    setbuf(stdout, NULL);
#ifndef NZERO
    nzero = NZERO;
#else
    #if defined(_SC_NZERO)
        nzero = sysconf(_SC_NZERO);
    #else
        #error NZERO undefined
    #endif
#endif
    printf("NZERO = %d\n", nzero);
    if (argc == 2)
        adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10; /* выполняться в течение 10 секунд */

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid == 0) { /* дочерний процесс */
        s = "потомок";
        printf("текущий коэффициент уступчивости в потомке %d, изменяется на %d\n",
               nice(0)+nzero, adj);
        errno = 0;
        if ((ret = nice(adj)) == -1 && errno != 0)
            err_sys("ошибка изменения приоритета в потомке");
        printf("сейчас коэффициент уступчивости в потомке %d\n", ret+nzero);
    } else { /* родительский процесс */
        s = "родитель";
        printf("текущий коэффициент уступчивости в родителе %d\n", nice(0)+nzero);
    }
    for(;;) {
        if (++count == 0)
            err_quit("%s переполнение счетчика", s);
        checktime(s);
    }
}

```

Мы запустили программу дважды: один раз — с приоритетом по умолчанию и второй раз — с максимально допустимым значением коэффициента уступчивости

(низший приоритет). Тестирование производилось в однопроцессорной системе Linux, чтобы показать, как планировщик делит процессорное время между процессами с разными приоритетами. В многопроцессорной и ненагруженной системе (или в системе с многоядерным процессором) оба процесса могли бы выполняться на разных процессорах, не мешая друг другу, и разница между процессами с разными приоритетами была бы не так очевидна.

```
$ ./a.out
NZERO = 20
текущий коэффициент уступчивости в родителе 20
текущий коэффициент уступчивости в потомке 20, изменяется на 0
сейчас коэффициент уступчивости в потомке 20
потомок счетчик = 1859362
родитель счетчик = 1845338
$ ./a.out 20
NZERO = 20
текущий коэффициент уступчивости в родителе 20
текущий коэффициент уступчивости в потомке 20, изменяется на 20
сейчас коэффициент уступчивости в потомке 39
родитель счетчик = 3595709
потомок счетчик = 52111
```

Когда оба процесса имели одинаковый приоритет, родительский процесс получил 50,2% процессорного времени, а потомок — 49,8%. То есть оба получили практически одинаковое количество процессорного времени. Небольшая разница объясняется некоторой неточностью в процедуре планирования, а также тем, что родительский и дочерний процессы выполняют разное количество операций между моментом, когда вычисляется время завершения, и моментом, когда начинается цикл увеличения счетчика.

Напротив, когда коэффициент уступчивости дочернего процесса был увеличен до максимального значения (получил минимальный приоритет), родительскому процессу было отдано 98,5% процессорного времени, а дочернему — всего 1,5%. Эти результаты могут отличаться в разных системах UNIX, в зависимости от особенностей использования коэффициента планировщиком процессов.

8.17. Временные характеристики процесса

В разделе 1.10 описывались три временные характеристики, которые можно измерить: общее время выполнения, пользовательское время и системное время. Любой процесс может вызвать функцию `times`, чтобы получить эти три значения для себя самого и для любого из завершивших работу потомков.

```
#include <sys/types.h>
clock_t times(struct tms *buf);
```

Возвращает количество тактов общего времени выполнения процесса
в случае успеха, -1 — в случае ошибки

Эта функция заполняет структуру `tms`, адрес которой передается в аргументе `buf`:

```
struct tms {
    clock_t tms_utime; /* пользовательское время */
    clock_t tms_stime; /* системное время */
    clock_t tms_cutime; /* пользовательское время для завершившегося потомка */
    clock_t tms_cstime; /* системное время для завершившегося потомка */
};
```

Обратите внимание, что структура не содержит общего времени выполнения. Мы получаем его в виде возвращаемого значения. Это время отмеряется от произвольного момента в прошлом, поэтому вместо абсолютных следует использовать относительные значения. Например, вызвать функцию `times` и сохранить возвращаемое значение. Через какое-то время еще раз вызвать функцию `times` и вычесть сохраненное ранее значение из нового значения. Разница будет определять время, прошедшее между двумя вызовами функции `times`. (Вполне возможно, хотя и маловероятно, что у долгоживущего процесса произойдет переполнение счетчика общего времени, см. упражнение 1.5.)

Два поля структуры отводятся для хранения временных характеристик дочернего процесса, но только того, завершение которого ожидалось с помощью функции `wait`, `waitid` или `waitpid`.

Все значения типа `clock_t`, возвращаемые функцией, можно преобразовать в секунды путем деления на количество тактов системных часов в секунду — значение параметра `_SC_CLK_TCK`, возвращаемое функцией `sysconf` (раздел 2.5.4).

Большинство реализаций предоставляют функцию `getrusage(2)`. Она возвращает затраченное процессорное время и еще 14 значений, характеризующих использование различных ресурсов. Исторически эта функция происходит из ОС BSD, поэтому все системы, производные от BSD, как правило, поддерживают большее количество полей, чем другие реализации.

Пример

Программа в листинге 8.19 запускает команды оболочки, переданные ей в аргументах, засекает время их выполнения и выводит значения полей структуры `tms`.

Листинг 8.19. Запуск команд и определение времени их работы

```
#include "apue.h"
#include <sys/types.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int      i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* один раз для каждого аргумента командной строки */
    exit(0);
}
```

```

    }

static void
do_cmd(char *cmd) /* запускает и измеряет время работы "cmd" */
{
    struct tms tmsstart, tmssend;
    clock_t start, end;
    int status;

    printf("\nкоманда: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1) /* начальные значения */
        err_sys("ошибка вызова функции times");

    if ((status = system(cmd)) < 0)          /* запустить команду */
        err_sys("ошибка вызова функции system()");

    if ((end = times(&tmssend)) == -1) /* конечные значения */
        err_sys("ошибка вызова функции times");

    pr_times(end - start, &tmsstart, &tmssend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmssend)
{
    static long clktck = 0;

    if (clktck == 0) /* получить количество тактов в сек. в первый раз*/
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("ошибка вызова функции sysconf");

    printf(" real: %7.2f\n", real / (double) clktck);
    printf(" user: %7.2f\n",
           (tmssend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf(" sys: %7.2f\n",
           (tmssend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf(" child user: %7.2f\n",
           (tmssend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf(" child sys: %7.2f\n",
           (tmssend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

```

Запустив эту программу, мы получили

```
$ ./a.out "sleep 5" "date" "man bash >/dev/null"
```

```
команда: sleep 5
real: 5.01
user: 0.00
sys: 0.00
child user: 0.00
child sys: 0.00
нормальное завершение, код выхода = 0
```

```
команда: date
Sun Feb 26 18:39:23 EST 2012
real: 0.00
```

```
user: 0.00
sys: 0.00
child user: 0.00
child sys: 0.00
нормальное завершение, код выхода = 0

команда: man bash >/dev/null
real: 1.46
user: 0.00
sys: 0.00
child user: 1.32
child sys: 0.07
нормальное завершение, код выхода = 0
```

Первые две команды выполняются слишком быстро, чтобы с имеющейся точностью можно было определить затраченное процессорное время. Но третья действует достаточно долго, чтобы заметить, что все затраченное процессорное время относится к дочернему процессу, которым является командная оболочка, запускающая команду.

8.18. Подведение итогов

Глубокое понимание механизма управления процессами в UNIX совершенно необходимо для профессионального программирования в этой операционной системе. Для этого нужно освоить лишь несколько функций: `fork`, семейство функций `exec`, `_exit`, `wait` и `waitpid`. Эти примитивы широко используются во многих приложениях. Кроме всего прочего, функция `fork` дала нам возможность увидеть ситуацию гонки за ресурсами.

Изучение функции `system` и возможностей учета расходования ресурсов процессами помогло нам посмотреть на функции управления процессами под другим углом. Мы также рассмотрели еще одну возможность функции `exec` — интерпретацию файлов и то, как эта интерпретация выполняется. Понимание различий между различными идентификаторами пользователя и группы — реальным, эффективным и сохраненным идентификаторами — особенно важно для обеспечения безопасности программ с установленным битом set-user-ID.

Обладая новыми знаниями о функционировании отдельного процесса и его потомков, мы рассмотрим в следующей главе взаимоотношения между разными процессами — сеансы и управление заданиями. Затем в главе 10 завершим тему процессов описанием сигналов.

Упражнения

- 8.1** При обсуждении программы из листинга 8.2 мы говорили, что если вызов функции `_exit` заменить на `exit`, стандартный поток вывода может оказаться закрытым и функция `printf` вернет признак ошибки — число `-1`. Измените программу, чтобы проверить, присущее ли такое поведение вашей реализации. Если нет, как эту ситуацию можно смоделировать?

- 8.2 Вспомните типичную организацию памяти процесса, изображенную на рис. 7.3. Каждому вызову функции обычно соответствует свой кадр стека, но поскольку после вызова функции `vfork` дочерний процесс продолжает работать в адресном пространстве родительского процесса, что может произойти, если обращение к `vfork` производится не из функции `main`, а из другой функции и при этом дочерний процесс выполняет возврат из этой функции после вызова `vfork`? Напишите тестовую программу для проверки этой ситуации и нарисуйте схему происходящего.
- 8.3 Перепишите программу в листинге 8.4, чтобы вместо `wait` она использовала `waitpid`. Не прибегая к функции `pr_exit`, определите эквивалентную информацию из структуры `siginfo`.
- 8.4 Запустив программу из листинга 8.7 один раз:

```
$ ./a.out
```

результаты не выглядят необычными. Но если запустить ее несколько раз подряд:

```
$ ./a.out ; ./a.out ; ./a.out
от родительского процесса
от родительского процесса
ото дроочерного процесса
дительского процесса
от дочернего процесса
черного процесса
```

вывод не будет соответствовать нашим ожиданиям. Почему? Как это можно исправить? Останется ли эта проблема, если дочерний процесс будет производить вывод первым?

- 8.5 В программе из листинга 8.10 мы вызывали функцию `exec1`, которой передавали *полный путь* к интерпретируемому файлу. Если вместо этого использовать функцию `exec1p`, передав ей только *имя файла* `testinterp`, и если каталог `/home/sar/bin` указан в переменной окружения `PATH`, что выведет программа в качестве `argv[2]`?
- 8.6 Напишите программу, которая создает процесс-зомби и затем с помощью функции `system` запускает команду `ps(1)`, чтобы проверить, действительно ли процесс превратился в зомби.
- 8.7 Как было отмечено в разделе 8.10, стандарт POSIX.1 требует, чтобы все открытые каталоги закрывались при вызове функции `exec`. Проверить это можно так: откройте корневой каталог с помощью функции `opendir`, уточните содержимое структуры `DIR` в своей системе и выведите состояние флага `close-on-exes`. Затем откройте тот же каталог для чтения с помощью функции `open` и выведите состояние флага `close-on-exes`.

9

Взаимоотношения между процессами

9.1. Введение

В предыдущей главе мы узнали, что процессы связаны определенными взаимоотношениями. Прежде всего, каждый процесс имеет «родителя» (начальный процесс уровня ядра обычно сам является собственным родителем). Когда дочерний процесс завершает работу, родительский процесс извещается об этом и может получить код выхода своего потомка. Мы также упоминали группы процессов, когда описывали функцию `waitpid` (раздел 8.6), которая может ожидать завершения любого процесса из указанной группы.

В этой главе мы более подробно рассмотрим группы процессов, а также коснемся понятия сеансов, введенного стандартом POSIX.1. Также мы рассмотрим отношения между командной оболочкой входа, которая запускается во время входа в систему, и всеми процессами, запускаемыми из этой оболочки.

Невозможно говорить о взаимоотношениях процессов без упоминания сигналов, а чтобы обсуждать сигналы, необходимо иметь представление о понятиях, рассматриваемых в этой главе. Если вы совершенно не знакомы с механизмом сигналов UNIX, вероятно, стоит сначала просмотреть главу 10.

9.2. Вход с терминала

Для начала рассмотрим программы, которые запускаются при входе пользователя в систему UNIX. Во времена ранних версий UNIX, таких как Version 7, пользователи входили в систему через терминалы ввода/вывода, соединенные кабелем с главным компьютером. Терминалы могли быть локальными (связанными непосредственно с компьютером) и удаленными (связанными с компьютером через модем). И в том и в другом случае вход в систему осуществлялся через драйвер устройства терминала в ядре. Например, наиболее типичными терминальными устройствами на PDP-11 были DH-11 и DZ-11. Машина имела фиксированное количество таких устройств, поэтому заранее было известно максимальное количество пользователей, которые могли одновременно войти в систему.

После появления графических терминалов были разработаны системы с оконным графическим интерфейсом, чтобы дать пользователю новый, более удобный способ взаимодействия с компьютером. Для эмуляции алфавитно-цифровых терми-

налов стали разрабатываться приложения, которые создавали «окно терминала», что позволяло пользователю взаимодействовать с главной машиной привычным способом (то есть через командную строку).

В настоящее время некоторые системы дают возможность запустить оконный интерфейс после входа, а другие запускают его автоматически. В последнем случае вам, возможно, все равно придется вводить имя и пароль — в зависимости от конфигурации оконной системы (в некоторых системах может быть настроен автоматический вход).

Процедура, которую мы сейчас описываем, используется для входа в систему UNIX посредством терминала. Она не зависит от типа терминала — это может быть алфавитно-цифровой терминал, графический терминал, эмулирующий простой алфавитно-цифровой терминал или графический терминал с оконной системой.

Вход в систему с терминала в BSD-системах

Эта процедура практически не изменилась за последние 40 лет. Системный администратор создает файл, обычно `/etc/ttys`, в котором каждая строка соответствует одному терминальному устройству. В каждой строке определяется имя устройства и другие параметры для программы `getty` — например, скорость передачи данных. Во время загрузки системы ядро создает процесс с идентификатором 1, то есть процесс `init`, который переводит систему в многопользовательский режим. Процесс `init` читает файл `/etc/ttys` и для каждого терминала запускает программу `getty` с помощью функций `fork` и `exec`. Это дает нам схему процессов, изображенную на рис. 9.1.

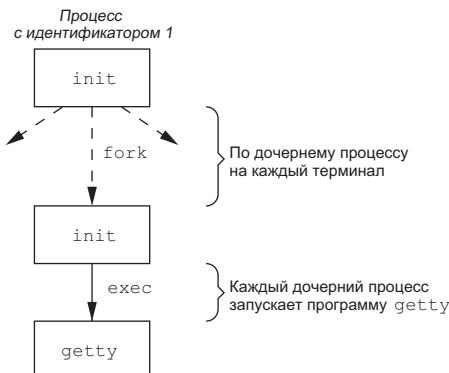


Рис. 9.1. Процессы, порождаемые `init`, чтобы разрешить вход в систему с терминалов

Все процессы, изображенные на рис. 9.1, имеют реальный и эффективный идентификатор пользователя 0 (то есть обладают привилегиями суперпользователя). Кроме того, процесс `init` запускает программу `getty` с пустым окружением.

Программа `getty` с помощью функции `open` открывает устройство терминала на чтение и на запись. Если устройство является модемом, функция `open` может отра-

ботать с некоторой задержкой внутри драйвера устройства, пока модем набирает номер и устанавливает соединение. Когда устройство открыто, ему назначаются файловые дескрипторы с номерами 0, 1 и 2. Далее `getty` выводит некоторую строку, например `login:`, и ожидает ввода имени пользователя. Если терминал поддерживает обмен данными на разных скоростях, программа `getty` в состоянии распознать специальные управляющие последовательности, которые указывают ей изменить скорость передачи. За дополнительными сведениями о программе `getty` и файлах данных (`gettytab`), управляющих ее действиями, обращайтесь к справочному руководству по вашей операционной системе.

После ввода имени пользователя программа `getty` завершает работу и передает управление программе `login` примерно так:

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

(Файл `gettytab` может содержать ссылки на другие программы, но по умолчанию вызывается программа `login`.) Процесс `init` запускает программу `getty` с пустым окружением, а `getty` создает для программы `login` (аргумент `envp`) окружение с именем терминала (что-нибудь вроде `TERM=foo`, где `foo` — тип терминала, который берется из файла `gettytab`) и другими переменными окружения, определенными в файле `gettytab`. Флаг `-p` сообщает программе `login`, что она должна сохранить предыдущую среду окружения и добавить к ней новую среду, не заменяя существующую. На рис. 9.2 показано состояние процессов сразу после запуска программы `login`.

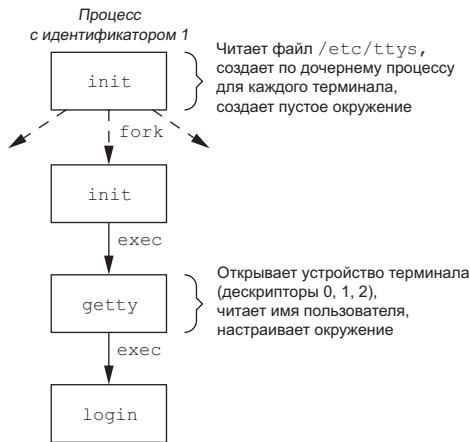


Рис. 9.2. Состояние процессов после запуска программы `login`

Все процессы на рис. 9.2 наследуют привилегии суперпользователя от первоначального процесса `init`. Три последних процесса получают одинаковые идентификаторы, потому что `exec` не изменяет идентификатор процесса. Кроме того, все процессы, кроме первоначального `init`, имеют идентификатор родительского процесса 1. Программа `login` выполняет множество различных действий. Поскольку у нее уже есть имя пользователя, она может вызвать функцию `getpwname`, чтобы полу-

чить строку учетной записи из файла паролей. Затем программа `login` вызывает функцию `getpass(3)`, чтобы вывести приглашение к вводу `Password:` и прочитать пароль (при этом, разумеется, отображение вводимых символов отключено). Далее вызывается функция `crypt(3)`, которая шифрует введенный пароль, и полученный результат сравнивается с полем `pw_passwd` из записи в теневом файле паролей. Если попытка входа в систему не удалась из-за неверно введенного пароля (после нескольких попыток), `login` вызывает функцию `exit` с аргументом 1. Такое завершение программы `login` будет замечено родительским процессом (`init`), и он с помощью функций `fork` и `exec` снова запустит программу `getty` для возобновления процедуры входа на данном терминале.

Это традиционная процедура аутентификации, используемая в UNIX. Современные системы UNIX поддерживают большое количество других процедур. Например, FreeBSD, Linux, Mac OS X и Solaris поддерживают более гибкую схему, известную как PAM (Pluggable Authentication Modules — сменные модули аутентификации). Эта схема позволяет системным администраторам настраивать методы аутентификации для обращения к службам, которые разработаны для использования с библиотекой PAM.

Если приложение требует проверки прав пользователя на выполнение определенных задач, можно либо жестко защитить механизм аутентификации в код приложения, либо создать аналогичную функциональность, используя библиотеку PAM. Преимущество PAM в том, что администратор может настроить разные способы аутентификации пользователей для выполнения различных задач, основываясь на локальной политике безопасности.

Если вход в систему выполнен корректно, программа `login`:

- изменит домашний каталог (`chdir`);
- изменит владельца терминала (`chown`);
- изменит права доступа к устройству терминала, чтобы дать возможность производить операции чтения и записи;
- установит идентификатор группы вызовом функций `setgid` и `initgroups`;
- инициализирует окружение той информацией, которой располагает программа `login`: это домашний каталог пользователя (`HOME`), командная оболочка (`SHELL`), имя пользователя (`USER` или `LOGNAME`) и список каталогов для поиска выполняемых файлов (`PATH`);
- изменит идентификатор пользователя (`setuid`) и запустит командную оболочку входа в систему
`execl("/bin/sh", "-sh", (char *)0);`

Символ «-» в качестве первого символа `argv[0]` сообщает командной оболочке, что она запущена как оболочка входа в систему. Командная оболочка, обнаружив этот признак, может соответственно изменить перечень действий, выполняемых при запуске.

На самом деле программа `login` выполняет гораздо больше действий. Она, например, может выводить так называемое сообщение дня, проверять поступление

новой почты и выполнять ряд других задач. Но сейчас нас интересует только та функциональность, которую мы описали.

В разделе 8.11, при обсуждении функции `setuid`, мы говорили, что при вызове с привилегиями суперпользователя она изменяет все три идентификатора пользователя – реальный, эффективный и сохраненный. Вызов функции `setgid`, который производится программой `login` чуть раньше, так же воздействует на все три идентификатора группы.

Итак, оболочка входа запущена. Ее родителем является процесс `init` (с идентификатором 1), то есть когда командная оболочка завершит работу, процесс `init` получит уведомление (сигнал `SIGCHLD`) и сможет снова запустить процедуру входа на данном терминале. Файловые дескрипторы 0, 1 и 2 в командной оболочке входа связаны с терминальным устройством. Это состояние изображено на рис. 9.3.

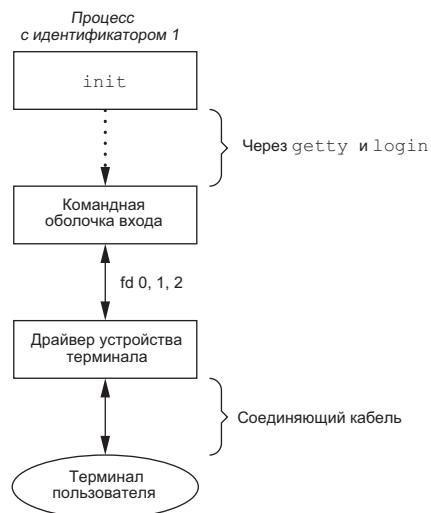


Рис. 9.3. Состояние процессов после входа пользователя в систему с терминала

Далее оболочка входа читает файлы начальной загрузки (`.profile` для Bourne shell и Korn shell; `.bash_profile`, `.bash_login` или `.profile` для GNU Bourne-again shell; `.cshrc` и `.login` для C shell). В них обычно изменяются значения некоторых переменных окружения и добавляется множество новых. Например, большинство пользователей создают свой список каталогов поиска (`PATH`) и устанавливают правильный тип терминала (`TERM`). Когда файлы начального запуска обработаны, мы наконец видим приглашение командной оболочки и можем вводить команды.

Вход в систему с терминала в Mac OS X

В Mac OS X вход в систему осуществляется так же, как в BSD, поскольку Mac OS X частично основана на FreeBSD, но имеет некоторые отличия:

- работу процесса `init` выполняет процесс `launchd`;
- вся процедура входа выполняется через графический интерфейс.

Вход в систему с терминала в Linux

Процедура входа в Linux очень напоминает процедуру входа в BSD-системах. И действительно, команда `login` в Linux является производной от команды `login` в 4.3BSD. Главное отличие между процедурами входа в Linux и BSD заключается в способе настройки терминала.

Некоторые дистрибутивы Linux распространяются с версией программы `init`, использующей конфигурационные файлы в формате, подобном формату файлов для программы `init` в System V. В этих системах конфигурационная информация, определяющая терминальные устройства, для которых должна вызываться программа `getty`, хранится в файле `/etc/inittab`.

Другие дистрибутивы Linux, такие как Ubuntu, распространяются с версией программы `init`, известной под названием «Upstart». Она использует конфигурационные файлы с именами вида `*.conf`, которые хранятся в каталоге `/etc/init`. Например, настройки для запуска `getty` на устройстве `/dev/tty1` находятся в файле `/etc/init/ttymode.conf`.

В зависимости от используемой версии `getty` характеристики терминалов указываются либо в командной строке (команда `getty`), либо в файле `/etc/gettydefs` (команда `mgetty`).

Вход в систему с терминала в Solaris

Solaris поддерживает два вида входа в систему с терминала: (а) с помощью `getty`, как описано выше для BSD-систем, и (б) с помощью `ttymon` — возможности, появившейся в SVR4. Как правило, для входа с консоли используется `getty`, а для входа с других терминальных устройств — `ttymon`.

Команда `ttymon` является частью большого программного механизма, называемого SAF — Service Access Facility (механизм доступа к службам). Основная цель SAF — обеспечить единый механизм управления службами, предоставляемыми доступ к системе. (За дополнительной информацией обращайтесь к главе 6 [Rago, 1993].) В нашем случае конечный результат действия этого механизма соответствует тому, что изображено на рис. 9.3, однако между процессом `init` и запуском оболочки входа выполняются несколько иные действия. Процесс `init` является родительским для процесса `sac` (service access controller — контроллер доступа к службам), который с помощью `fork` и `exec` запускает программу `ttymon`, когда система переходит в многопользовательский режим. Программа `ttymon` контролирует все терминальные порты, перечисленные в конфигурационном файле, и запускает дочерний процесс после ввода имени пользователя. Этот дочерний процесс с помощью функции `exec` запускает программу `login`, а уже она запрашивает пароль. После ввода пароля `login` запускает командную оболочку входа, и система приходит в состояние, изображенное на рис. 9.3. Единственное отличие — родителем для командной оболочки становится процесс `ttymon`, а в схеме с использованием программы `getty` — процесс `init`.

9.3. Вход в систему через сетевое соединение

Главное физическое отличие между входом в систему с терминала, соединенного с главной машиной последовательным кабелем, и через сетевое соединение состоит в том, что сетевое соединение не построено по принципу «точка-точка». В данном случае `login` — это просто служба, подобная другим сетевым службам, таким как FTP или SMTP.

В ситуациях, описанных в предыдущем разделе, процесс `init` знает, с каких устройств разрешен вход, и порождает процесс `getty` для каждого из них. Но в случае входа в систему через сетевое соединение все запросы поступают через драйвер сетевого интерфейса (например, драйвер Ethernet) и мы заранее не знаем, сколько таких запросов поступит. Вместо запуска отдельного процесса для каждого возможного запроса на вход в систему мы теперь ожидаем прибытия запросов на соединение.

Чтобы одно и то же программное обеспечение могло обрабатывать вход в систему с терминала и через сетевые соединения, используется программный драйвер, который называется *псевдотерминалом*. Этот драйвер эмулирует поведение обычного терминала, отображая операции с терминалом в сетевые операции и наоборот. (Подробнее о псевдотерминалах мы поговорим в главе 19.)

Вход в систему через сетевое соединение в BSD

В системах BSD большинство сетевых соединений устанавливается с помощью единственного процесса — `inetd`, который иногда называют *Internet superserver*. В этом разделе мы рассмотрим последовательность действий, которая выполняется при входе в BSD-систему через сетевое соединение. Нас не интересуют все подробности программной реализации этих процессов — их вы найдете в [Stevens, Fenner, and Rudoff, 2004].

Во время запуска системы процесс `init` вызывает командный интерпретатор, который выполняет сценарий командной оболочки `/etc/rc`. Одним из демонов, запускаемых этим сценарием, является `inetd`. По окончании работы сценария родительским процессом для `inetd` становится процесс `init`. Процесс `inetd` ожидает запросов на соединение по протоколу TCP/IP. Когда поступает очередной запрос, демон `inetd` с помощью функций `fork` и `exec` запускает соответствующую программу для его обработки.

Предположим, что по адресу сервера TELNET пришел запрос на TCP-соединение. TELNET — это служба удаленного входа в систему, которая использует протокол TCP. Пользователь, находящийся за другим компьютером (соединенным с сервером сетью) или за тем же самым компьютером, инициирует вход в систему, запустив клиент TELNET:

```
telnet hostname
```

Клиент открывает TCP-соединение с узлом сети `hostname`, и на стороне сервера запускается программа, которая называется сервером TELNET. После этого клиент и сервер начинают обмен данными по прикладному протоколу TELNET. Таким способом пользователь, запустивший клиентскую программу, выполняет

вход в систему на сервере. (Разумеется, лишь в том случае, если у пользователя имеется учетная запись на сервере.) На рис. 9.4 показана последовательность процессов, сопутствующих запуску сервера TELNET с именем `telnetd`.

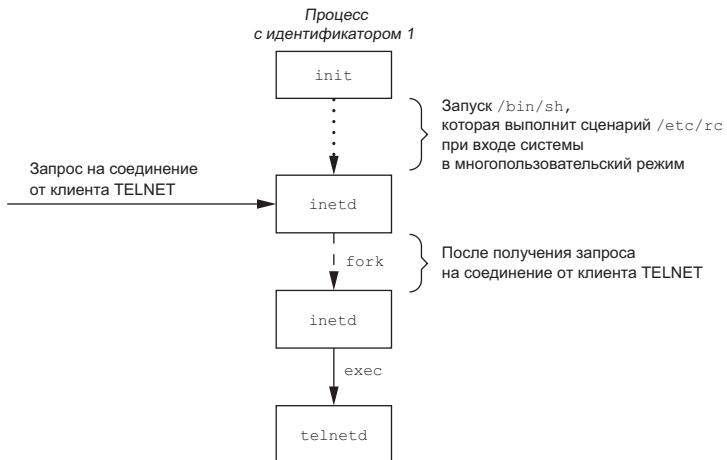


Рис. 9.4. Последовательность действий, приводящая к запуску сервера TELNET

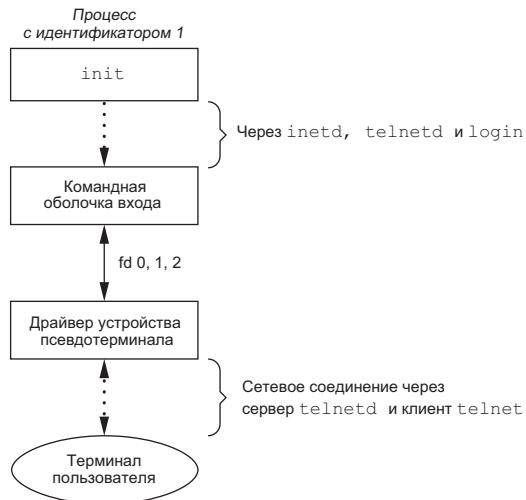


Рис. 9.5. Состояние процессов после входа пользователя через сетевое соединение

Затем процесс `telnetd` открывает устройство псевдотерминала и с помощью функции `fork` разделяется на два процесса. Родительский процесс продолжает обслуживать сетевое соединение, а дочерний запускает программу `login`. Родительский процесс связан с дочерним через псевдотерминал. Перед вызовом функции `exec` дочерний процесс присоединяет файловые дескрипторы 0, 1 и 2 к псевдотерминалу. В случае удачного входа в систему программа `login` выполняет действия,

описанные в разделе 9.2: она делает текущим домашний каталог пользователя, устанавливает идентификаторы пользователя и группы и инициализирует окружение. Затем программа `login` замещает себя командной оболочкой входа посредством функции `exec`. На рис. 9.5 показано состояние процессов в этот момент.

Очевидно, что между драйвером псевдотерминала и пользовательским терминалом действует еще множество процессов. Мы рассмотрим их в главе 19, когда будем говорить о псевдотерминалах более подробно.

Важно понимать, что независимо от того, входим мы в систему через терминал (см. рис. 9.3) или через сетевое соединение (см. рис. 9.5), мы получаем командную оболочку входа со стандартным вводом, стандартным выводом и стандартным выводом сообщений об ошибках, которые связаны либо с устройством терминала, либо с устройством псевдотерминала. В последующих разделах мы увидим, что оболочка входа открывает сеанс POSIX.1, а терминал или псевдотерминал становится управляющим терминалом сеанса.

Вход в систему через сетевое соединение в Mac OS X

В Mac OS X процедура входа в систему через сетевое соединение совпадает с процедурой в BSD, поскольку Mac OS X частично основана на FreeBSD. Только в Mac OS X демон `telnet` запускается из `launchd`.

По умолчанию в Mac OS X демон telnet отключен (но его можно включить командой launchctl(1)). Для входа в систему через сетевое соединение в Mac OS X предпочтительнее использовать ssh, защищенную командную оболочку.

Вход в систему через сетевое соединение в Linux

Процедура входа через сетевое соединение в Linux практически такая же, как в BSD, только вместо процесса `inetd` используется его альтернатива — `xinetd` (extended Internet services daemon — расширенный демон сетевых служб). Демон `xinetd` дает возможность более точного управления запуском сетевых служб по сравнению с `inetd`.

Вход в систему через сетевое соединение в Solaris

Сценарий входа через сетевое соединение в Solaris по большей части идентичен соответствующим сценариям в BSD и Linux. В Solaris, как и в BSD, используется сервер `inetd`, но версия в Solaris выполняется механизмом управления службами (Service Management Facility, SMF) как *рестартер* (restarter). Рестартер — это демон, ответственный за запуск других демонов и повторный их запуск, когда они завершаются по ошибке. Хотя сервер `inetd` запускается главным рестартером (master restarter) в SMF, сам главный рестартер запускается процессом `init`, и в результате мы приходим к состоянию, изображенному на рис. 9.5.

Механизм управления службами (Service Management Facility) в Solaris — это целая инфраструктура управления системными службами, поддерживающая возможность их возобновления в случае отказов. За дополнительной информацией о механизме управления службами обращайтесь к [Adams, 2005] и страницам smf(5) и inetd(1M) справочного руководства в Solaris.

9.4. Группы процессов

Каждый процесс не только имеет идентификатор процесса, но и принадлежит к определенной группе процессов. Мы еще будем встречаться с группами процессов при обсуждении сигналов в главе 10.

Группа процессов — это коллекция из одного или более процессов, обычно связанных с выполнением одного и того же задания (управление заданиями рассматривается в разделе 9.8), которые могут принимать сигналы от одного и того же терминала. Каждая группа процессов имеет уникальный идентификатор. Идентификатор группы процессов напоминает идентификатор процесса: это целое положительное число, которое может храниться в переменной типа `pid_t`. Функция `getpgrp` возвращает идентификатор группы процессов вызывающего процесса.

```
#include <unistd.h>
pid_t getpgrp(void);
```

Возвращает идентификатор группы процессов вызывающего процесса

В ранних версиях BSD-систем функция `getpgrp` принимала аргумент `pid` и возвращала группу процессов для заданного процесса. Стандарт Single UNIX Specification определил в качестве расширения XSI функцию `getpgid`, которая имитирует это поведение.

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Возвращает идентификатор группы процессов в случае успеха,
-1 — в случае ошибки

Если в аргументе `pid` передать 0, функция вернет групповой идентификатор вызывающего процесса. То есть вызов

`getpgid(0);`

эквивалентен вызову

`getpgrp();`

Каждая группа процессов может иметь лидера. Идентификатор группы процессов лидера группы совпадает с его идентификатором процесса.

Вполне допустима ситуация, когда лидер группы создает группу процессов, затем запускает процессы в этой группе и завершается. Группа процессов продолжит существовать, пока в ней остается хотя бы один процесс, вне зависимости от того, завершил работу лидер группы или нет. Период от момента создания группы и до момента, когда последний процесс в группе покинет ее, называется временем жизни группы процессов. Последний оставшийся в группе процесс может либо завершиться, либо войти в состав другой группы процессов.

Процесс может присоединиться к группе или создать новую группу процессов с помощью функции `setpgid`. (В следующем разделе мы увидим, что функция `setsid` также создает новую группу процессов.)

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Эта функция устанавливает для процесса с идентификатором *pid* идентификатор группы процессов *pgid*. Если аргументы имеют одинаковые значения, процесс, заданный идентификатором *pid*, становится лидером группы процессов. Если в аргументе *pid* передается 0, в качестве идентификатора процесса используется идентификатор вызывающего процесса. Если в аргументе *pgid* передается 0, в качестве идентификатора группы процессов используется значение аргумента *pid*.

Процесс может установить идентификатор группы только для себя самого и для своих дочерних процессов. Но процесс не может изменить идентификатор группы процессов дочернего процесса, который вызвал одну из функций семейства `exec`.

В большинстве командных оболочек, поддерживающих управление заданиями, функция `setpgid` вызывается после `fork`, чтобы родительский процесс мог назначить идентификатор группы процессов дочернему процессу, а дочерний процесс — установить свой собственный идентификатор группы процессов.

Один из этих вызовов является излишним, но, выполняя оба, мы гарантируем, что дочерний процесс будет помещен в его собственную группу процессов в любом случае. Иначе мы столкнулись бы с ситуацией гонки за ресурсами, когда членство дочернего процесса зависело бы от того, какой из процессов первым получит управление.

При обсуждении сигналов мы увидим, как послать сигналциальному процессу (по идентификатору процесса) или группе процессов (по идентификатору группы процессов). Аналогично функция `waitpid` позволяет дождаться завершения конкретного процесса или одного из процессов в заданной группе.

9.5. Сеансы

Сеанс — это коллекция из одной или более групп процессов. Рассмотрим в качестве примера ситуацию, изображенную на рис. 9.6. Здесь мы имеем три группы процессов в одном сеансе.

Процессы обычно помещаются в группу командной оболочки при конвейерной обработке данных. Например, состояния процессов, изображенного на рис. 9.6, можно достичь последовательностью команд:

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

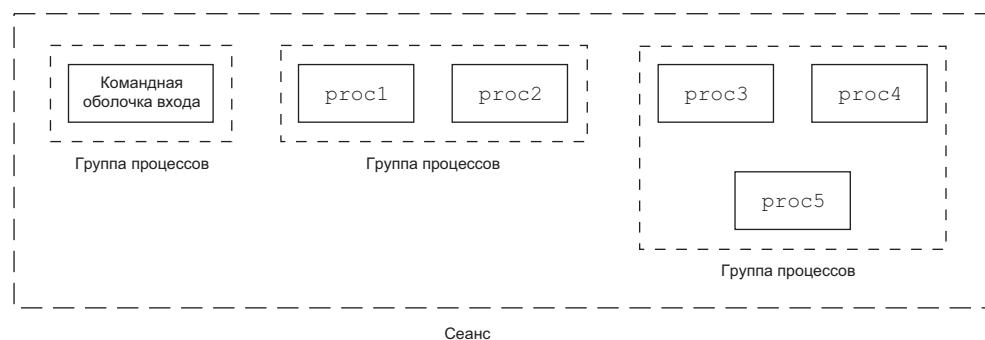


Рис. 9.6. Распределение процессов по группам процессов и сеансам

Создание нового сеанса производится с помощью вызова функции `setsid`.

```
#include <unistd.h>
pid_t setsid(void);
```

Возвращает идентификатор группы процессов в случае успеха,
–1 – в случае ошибки

Если вызывающий процесс не является лидером группы, функция создает новый сеанс. При этом происходит следующее.

1. Процесс становится *лидером нового сеанса*. (Лидер сеанса – это процесс, который создает сеанс.) Этот процесс – единственный в новом сеансе.
2. Процесс становится лидером новой группы процессов. Идентификатором новой группы процессов становится идентификатор вызывающего процесса.
3. Процесс теряет управляющий терминал. (Управляющие терминалы обсуждаются в следующем разделе.) Если у процесса был управляющий терминал перед вызовом функции `setsid`, связь с ним разрывается.

Эта функция возвращает признак ошибки, если вызывающий процесс уже является лидером группы. Чтобы избежать этой ошибки, обычно вызывают функцию `fork`, затем родительский процесс завершается, а дочерний процесс продолжает работу. В этом случае можно гарантировать, что дочерний процесс не будет лидером группы, поскольку его идентификатор группы процессов наследуется от родительского процесса, но сам он получит новый идентификатор процесса. Соответственно идентификатор дочернего процесса никогда не будет совпадать с унаследованным идентификатором группы процессов.

Стандарт Single UNIX Specification оговаривает только определение лидера сеанса, но в нем нет определения идентификатора сеанса. Очевидно, что лидер сеанса – это отдельный процесс, имеющий уникальный идентификатор процесса, поэтому можно утверждать, что идентификатор сеанса – это идентификатор процесса, являющегося лидером сеанса. Такое понимание идентификатора сеанса

было введено в SVR4. Традиционно BSD-системы не поддерживали это понятие, но впоследствии положение изменилось.

Некоторые реализации, такие как Solaris, следуя Single UNIX Specification, избегают понятия «идентификатор сеанса»; вместо этого они используют термин «идентификатор группы процессов лидера сеанса». Эти два понятия эквивалентны, так как лидер сеанса всегда является лидером группы процессов.

Функция `getsid` возвращает идентификатор группы процессов лидера сеанса.

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Возвращает идентификатор группы процессов лидера сессии
в случае успеха, -1 — в случае ошибки

Если в аргументе `pid` передать 0, `getsid` вернет идентификатор группы процессов лидера сеанса, которой принадлежит вызывающий процесс. Из соображений безопасности некоторые реализации могут ограничивать возможность получения идентификатора группы процессов лидера сеанса, если в аргументе `pid` передается идентификатор процесса, не принадлежащего тому же сеансу, что и вызывающий процесс.

9.6. Управляющий терминал

Сеансы и группы процессов обладают еще некоторыми характеристиками.

- Сеанс может иметь только один *управляющий терминал*. Обычно это устройство терминала (в случае входа в систему с терминала) или устройство псевдо-терминала (в случае входа в систему через сетевое соединение), с которого был произведен вход в систему.
- Лидер сеанса, который устанавливает соединение с управляющим терминалом, называется *управляющим процессом*.
- Группы процессов в пределах одного сеанса могут подразделяться на единственную *группу процессов переднего плана* и одну или более *групп фоновых процессов*.
- Если сеанс имеет управляющий терминал, в нем будет одна группа процессов переднего плана, а все остальные группы процессов в сеансе будут группами фоновых процессов.
- Когда мы вводим с клавиатуры терминала символ прерывания (обычно `DELETE` или `Control-C`), всем процессам в группе процессов переднего плана посыпается сигнал прерывания.
- Когда мы вводим с клавиатуры терминала символ завершения (обычно `Control-\`), всем процессам в группе процессов переднего плана посыпается сигнал завершения.

- О Если интерфейс терминала обнаруживает разрыв связи с модемом или сетью, управляющему процессу (лидеру сеанса) посыпается сигнал, оповещающий о разрыве связи.

Эти характеристики показаны на рис. 9.7.

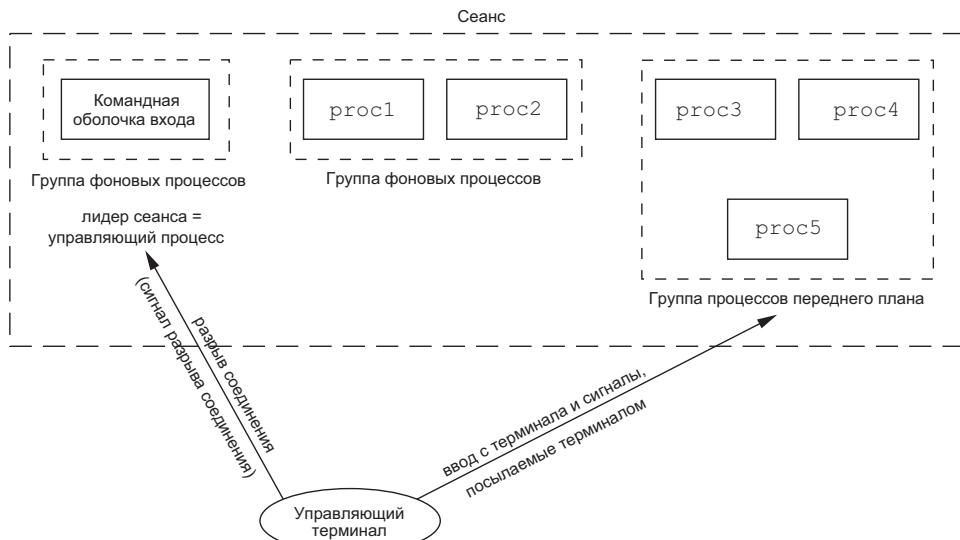


Рис. 9.7. Группы процессов, сеансы и управляющий терминал

Обычно не приходится беспокоиться об управляющем терминале — он устанавливается автоматически после входа в систему.

Стандарт POSIX.1 оставляет выбор механизма размещения управляющего терминала за конкретной реализацией. Фактические действия мы будем рассматривать в разделе 19.4.

Системы, производные от System V, размещают управляющий терминал сеанса в тот момент, когда лидер сеанса открывает первое устройство терминала, еще не связанное с сеансом. Это предполагает, что лидер сеанса, вызывая функцию open, не указывает флаг O_NOSTTY (раздел 3.3).

Системы, основанные на BSD, размещают управляющий терминал сеанса, когда лидер сеанса вызывает функцию ioctl, передавая ей в аргументе request значение TIOCSETTY (третий аргумент — пустой указатель). Чтобы вызов завершился успехом, сеанс не должен иметь управляющего терминала. (Обычно вызов функции ioctl следует за вызовом функции setsid — это гарантирует, что процесс является лидером сеанса без управляющего терминала.) Флаг O_NOSTTY функции open не используется в BSD-системах, за исключением случаев, когда необходима поддержка совместимости с другими системами.

В табл. 9.1 перечислены способы размещения управляющих терминалов в каждой из систем, обсуждаемых в этой книге. Обратите внимание: хотя Mac OS X 10.6.8 ведет свою родословную от BSD, она действует как System V, когда размещает управляющий терминал.

Таблица 9.1. Размещение управляющих терминалов в различных реализациях

Метод	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
open без флага O_NOCTTY		✓	✓	✓
передача команды TIOCSCTTY системному вызову ioctl	✓	✓	✓	✓

Иногда возникают ситуации, когда программа должна произвести обмен данными с управляющим терминалом, даже когда стандартные потоки ввода/вывода перенаправлены. Чтобы обеспечить возможность такого обмена, необходимо вызовом функции `open` открыть файл `/dev/tty`. Этот специальный файл является синонимом управляющего терминала в ядре. Разумеется, если программа не имеет управляющего терминала, попытка открыть его окончится неудачей.

Классический пример — функция `getpass(3)`, которая читает пароль при вводе с клавиатуры (естественно, при отключенном отображении вводимых символов). Эта функция вызывается программой `crypt(1)` и может быть использована в конвойере с другими командами. Например, команда

```
crypt < salaries | lpr
```

расшифрует содержимое файла `salaries` и отправит результат на принтер. Поскольку программа `crypt` читает входной файл со стандартного ввода, он не может использоваться для ввода пароля. Кроме того, программа `crypt` спроектирована так, что при каждом вызове она заставляет нас снова вводить пароль и не дает сохранить его в файле (иначе это стало бы лазейкой в системе безопасности).

Существуют способы, позволяющие взломать шифр, используемый программой `crypt`. За дополнительной информацией о шифровании файлов обращайтесь к [Garfinkel et al., 2003].

9.7. Функции tcgetpgrp, tcsetpgrp и tcgetsid

Теперь нам нужен способ сообщить ядру, какая группа процессов является группой переднего плана, чтобы драйвер терминала знал, какому процессу передавать ввод с терминала и кому отправлять сигналы (рис. 9.7).

```
#include <unistd.h>
pid_t tcgetpgrp(int fd);
```

Возвращает идентификатор группы процессов переднего плана
в случае успеха, -1 — в случае ошибки

```
int tcsetpgrp(int fd, pid_t pgrp_id);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Функция `tcgetpgrp` возвращает идентификатор группы процессов переднего плана, связанной с открытым файловым дескриптором терминала `fd`.

Если процесс обладает управляющим терминалом, он может вызывать функцию `tcsetpgrp`, чтобы назначить группу процессов с идентификатором `pgid` группой процессов переднего плана. Значение аргумента `pgid` должно быть идентификатором группы процессов в том же сеансе, а аргумент `fd` должен быть дескриптором управляющего терминала сеанса.

Большинство приложений не вызывают эти две функции напрямую. Обычно они вызываются командными оболочками, которые поддерживают управление заданиями.

Функция `tcgetsid` позволяет приложению получить идентификатор группы процессов лидера сеанса по заданному файловому дескриптору управляющего терминала.

```
#include <termios.h>
pid_t tcgetsid(int fd);
```

Возвращает идентификатор группы процессов лидера сеанса в случае успеха, `-1` — в случае ошибки

Приложения, которым необходимо взаимодействовать с управляющим терминалом, могут использовать функцию `tcgetsid`, чтобы получить идентификатор сеанса для лидера сеанса, владеющего управляющим терминалом (что эквивалентно идентификатору группы процессов лидера сеанса).

9.8. Управление заданиями

Возможность управления заданиями была добавлена в BSD около 1980 года.

Она позволяет запустить несколько заданий (групп процессов) с одного терминала и определить, какие из них получат доступ к терминалу, а какие будут выполняться в фоновом режиме. Управление заданиями поддерживается, если соблюдаются следующие условия.

1. Командная оболочка должна поддерживать управление заданиями.
2. Драйвер терминала в ядре должен поддерживать управление заданиями.
3. Ядро должно поддерживать ряд сигналов, с помощью которых осуществляется управление заданиями.

В SVR3 имелась возможность управления заданиями в иной форме, которая называлась уровнями командной оболочки (shell layers). Однако стандарт POSIX.1 выбрал форму управления заданиями, реализованную в BSD; именно она здесь и описывается. В ранних версиях стандарта поддержка управления заданиями была необязательной, однако теперь это обязательное требование.

первый POSIX.1 требует, чтобы все POSIX-совместимые платформы поддерживали эту возможность.

Для нас сейчас важно, что возможность управления заданиями позволяет запустить задание на переднем плане или в фоновом режиме. Задание — это просто набор процессов, часто объединенных в конвейер. Например, команда

```
vi main.c
```

запустит задание, содержащее один процесс переднего плана. Команды

```
pr *.c | lpr &
make all &
```

запустят два фоновых задания. Все процессы, запускаемые в рамках этих заданий, являются фоновыми.

Как уже говорилось, чтобы пользоваться управлением заданиями, необходима командная оболочка, поддерживающая эту возможность. Довольно просто перечислить командные оболочки, которые поддерживали управление заданиями в старых системах. Так, C shell имела поддержку управления заданиями, Bourne shell — нет, а Korn shell — в зависимости от того, поддерживала ли управление заданиями сама платформа. Но позднее командная оболочка C shell была перенесена на системы, которые не поддерживали управление заданиями (например, ранние версии System V), а в SVR4 можно было включить поддержку управления заданиями в командной оболочке Bourne shell, запустив ее командой `jsh` вместо `sh`. В настоящее время возможность управления заданиями в Korn shell зависит от того, поддерживает ли эту возможность сама система. Командная оболочка Bourne-again shell поддерживает управление заданиями. Далее мы будем просто упоминать, что командная оболочка поддерживает управление заданиями в противоположность оболочке, которая не имеет такой поддержки, если различия между конкретными оболочками для нас несущественны.

При запуске задания в фоновом режиме командная оболочка присваивает ему идентификатор задания и выводит один или более идентификаторов процессов. Ниже показано, как это делает командная оболочка Korn shell:

```
$ make all > Make.out &
[1] 1475
$ pr *.c | lpr &
[2] 1490
$ просто нажмите клавишу Enter
[2] + Done pr *.c | lpr &
[1] + Done make all > Make.out &
```

Задание с номером 1 представлено программой `make`, а соответствующий ей процесс имеет идентификатор 1475. Задание с номером 2 представлено конвейером, в котором первый процесс имеет идентификатор 1490. По завершении заданий, после нажатия клавиши `Enter`, командная оболочка сообщает, какие задания завершились. Она не выводит сообщений об изменении состояния фоновых заданий по своей инициативе — только перед тем, как выведет приглашение, которое позволяет вводить новые команды. Иначе сообщения оболочки

могли бы смешиваться с вводимыми символами. Поэтому чтобы вызвать сообщение о состоянии фоновых заданий, после появления приглашения нужно нажать клавишу **Enter**.

Управление заданием переднего плана через драйвер терминала осуществляется с помощью ввода специальных символов, например символа приостановки (обычно **Control-Z**). Ввод этого символа заставляет драйвер послать сигнал **SIGTSTP** всем процессам группы переднего плана. Задания, выполняемые в фоновом режиме, при этом не затрагиваются. Драйвер терминала посыпает сигналы процессам переднего плана при вводе трех специальных символов.

- ввод символа прерывания (обычно **DELETE** или **Control-C**) порождает сигнал **SIGINT**;
- ввод символа завершения (обычно **Control-**) порождает сигнал **SIGQUIT**;
- ввод символа приостановки (обычно **Control-Z**) порождает сигнал **SIGTSTP**.

В главе 18 мы увидим, как привязать эту функциональность к любым другим символам и как запретить обработку этих специальных символов драйвером терминала.

Драйверу терминала приходится обрабатывать и другие ситуации, связанные с управлением заданиями. Так как у нас может быть одно задание переднего плана и одно или более фоновых заданий, необходимо разобраться, какие из них будут получать символы, вводимые с терминала. Ввод с терминала получает только задание переднего плана. Попытка прочитать ввод с терминала в фоновом задании не считается ошибкой, но будет обнаружена драйвером, который пошлет специальный сигнал **SIGTTIN** фоновому заданию. Этот сигнал обычно приводит к остановке фонового задания, командная оболочка выводит сообщение об этом, и мы можем перевести задание на передний план, чтобы оно получило возможность прочитать ввод с терминала, например:

```
$ cat > temp.foo & программа запущена в фоновом режиме, но пытается
читать со стандартного ввода
[1] 1681
$ нажимаем клавишу Enter
[1] + Stopped (SIGTTIN) cat > temp.foo &
$ fg %1 переводим задание с номером 1 на передний план
cat > temp.foo оболочка сообщает, какое задание находится
на переднем плане
hello, world вводим одну строку
^D вводим символ EOF (конец файла)
$ cat temp.foo проверяем, попала ли введенная строка в файл
hello, world
```

Обратите внимание, что данный пример не работает в Mac OS X 10.6.8. При попытке перевести команду **cat** на передний план вызов функции **read** в ней завершается с ошибкой **EINTR** в **errno**. Так как Mac OS X основана на FreeBSD, а FreeBSD работает в соответствии с ожиданиями, такое поведение, скорее всего, обусловлено ошибкой в Mac OS X.

Командная оболочка запускает в фоновом режиме процесс **cat**, который пытается прочитать символы со стандартного ввода (управляющий терминал). Драйвер терминала знает, что это фоновое задание, и посыпает ему сигнал

SIGTTIN. Командная оболочка определяет изменение состояния своего дочернего процесса (вспомните обсуждение функций `wait` и `waitpid` в разделе 8.6) и сообщает, что задание приостановлено. После этого мы с помощью команды `fg` перемещаем приостановленное задание на передний план. (За дополнительной информацией о командах управления заданиями, таких как `fg` или `bg`, и способах идентификации заданий обращайтесь к страницам справочного руководства по командной оболочке.) В результате оболочка переместила задание в группу процессов переднего плана (`tcsetpgrp`) и послала группе сигнал продолжения работы (`SIGCONT`). Поскольку теперь задание принадлежит к группе процессов переднего плана, оно получает возможность читать данные с управляющего терминала.

А что произойдет, если фоновое задание попытается вывести что-нибудь на терминал? Мы можем разрешить или запретить эту возможность, обычно для этого используется команда `stty(1)`. (В главе 18 мы покажем, как управлять этой возможностью из программы.) Например:

```
$ cat temp.foo & запустить в фоновом режиме
[1] 1719
$ hello, world вывод фонового задания появляется после приглашения
командной оболочки; нажимаем клавишу Enter
[1] + Done cat temp.foo &
$ stty tostop запретить фоновым заданиям вывод в терминал
$ cat temp.foo & попробуем еще раз запустить команду в фоновом режиме
[1] 1721
$ нажимаем Enter и обнаруживаем, что задание приостановлено
[1] + Stopped(SIGTTOU) cat temp.foo &
$ fg %1 возобновим работу задания на переднем плане
cat temp.foo оболочка сообщила, какое из заданий выполняется
на переднем плане
hello, world а это вывод задания
```

Когда мы запретили возможность вывода в терминал для фоновых заданий, утилита `cat` была заблокирована при попытке записи в стандартный вывод, так как драйвер терминала определил, что запись производится из фонового процесса, и передал ему сигнал `SIGTTOU`. Как и в предыдущем примере, мы с помощью команды `fg` перевели задание на передний план, благодаря чему оно получило возможность успешно отработать.

На рис. 9.8 изображена схема управления заданиями, описанная выше. Сплошные линии внутри драйвера терминала означают, что ввод/вывод на терминал и сигналы, посыпаемые терминалом, всегда связаны с группой процессов переднего плана. Пунктирная линия, соответствующая сигналу `SIGTTOU`, означает, что возможность вывода на терминал для фоновых процессов может отсутствовать.

Является ли управление заданиями необходимым или только желательным? Изначально управление заданиями было спроектировано и реализовано еще до появления и широкого распространения терминалов, предоставляющих многооконный интерфейс. Одни берут на себя смелость утверждать, что хорошо продуманная многооконная система ликвидирует потребность в управлении заданиями. Другие выражают недовольство чрезмерно сложной реализацией

управления заданиями, которое должно поддерживаться ядром, драйвером терминала, командной оболочкой и отдельными приложениями. Третий используют и управление заданиями, и многооконную систему, утверждая, что обе возможности одинаково необходимы. Однако, независимо от вашего мнения, эта функциональность является обязательной для реализации согласно стандарту POSIX.1.

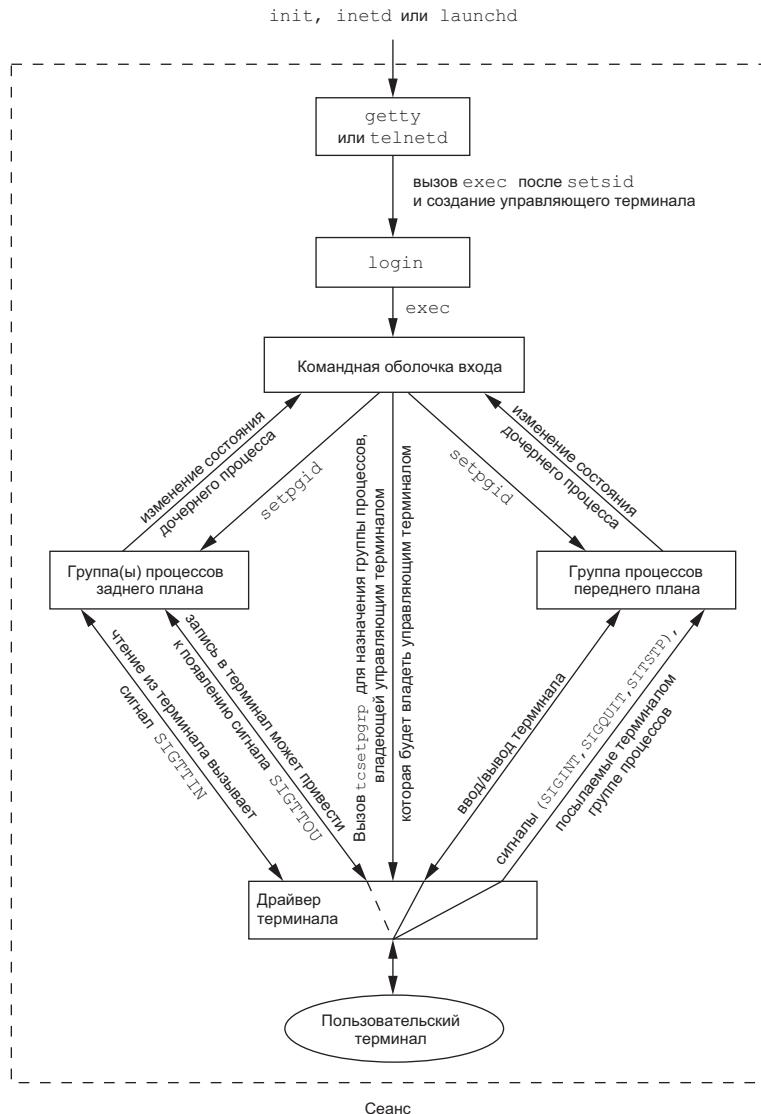


Рис. 9.8. Схема взаимодействия заданий переднего плана и фонового режима с драйвером терминала

9.9. Выполнение программ командной оболочкой

Рассмотрим, как командная оболочка запускает программы и как это связано с понятиями группы процессов, управляющего терминала и сеанса. Для этого воспользуемся командой `ps`.

Для начала возьмем командную оболочку, которая не поддерживает управление заданиями, — это классическая Bourne shell под управлением Solaris. Запустив команду

```
ps -o pid,ppid,pgid,sid,comm
```

мы получим

```
PID PPID PGID SID COMMAND
949 947 949 949 sh
1774 949 949 949 ps
```

Как мы и ожидали, родительским процессом для `ps` является командная оболочка. И командная оболочка, и команда `ps` находятся в одном сеансе и принадлежат одной группе процессов переднего плана (949). Мы говорим, что число 949 представляет группу процессов переднего плана, потому что в командных оболочках без поддержки управления заданиями мы получаем именно группу процессов.

На некоторых платформах команда ps может выводить идентификатор группы процессов, связанный с управляющим терминалом сеанса. Это значение отображается в столбце TPGID. К сожалению, вывод команды ps часто различается в разных версиях UNIX. Например, Solaris 10 не поддерживает такую возможность. В FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 команда

```
ps -o pid,ppid,pgid,sid,tpgid,comm
```

выведет то, что нам необходимо.

Обратите внимание: было бы неправильно ассоциировать процесс с идентификатором группы процессов терминала (столбец TPGID — terminal process group ID). У процесса нет такого признака, как группа процессов терминала. Процесс принадлежит группе процессов, а группа процессов принадлежит сеансу. Сеанс может иметь управляющий терминал, а может не иметь. Если сеанс имеет управляющий терминал, то терминальное устройство знает идентификатор группы процессов переднего плана. Это значение можно установить в драйвере терминала с помощью функции tcsetpgrp, как это видно на рис. 9.8. Идентификатор группы процессов переднего плана — это атрибут терминала, а не процесса. Значение, выводимое командой ps в колонке TPGID, берется из драйвера терминала. Если окажется, что сеанс не имеет управляющего терминала, команда ps выведет в этой колонке значение –1.

Если запустить команду в фоновом режиме:

```
ps -o pid,ppid,pgid,sid,comm &
```

единственное, что изменится, — это идентификатор процесса команды:

```
PID PPID PGID SID COMMAND
949 947 949 949 sh
1812 949 949 949 ps
```

Эта командная оболочка не поддерживает управление заданиями, поэтому фоновое задание не помещается в собственную группу процессов и не теряет связь с управляющим терминалом.

А теперь посмотрим, как Bourne shell обслуживает конвейеры. После запуска команды

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

мы получаем

```
PID PPID PGID SID COMMAND
949 947 949 949 sh
1823 949 949 949 cat1
1824 1823 949 949 ps
```

(Программа `cat1` — это просто копия программы `cat`, сохраненная под другим именем. У нас есть еще одна копия программы `cat` под именем `cat2`, которую мы используем чуть позже в этом же разделе. Запуск двух копий программы `cat` в одном конвейере дает нам возможность различать их.) Обратите внимание, что последний процесс в конвейере является дочерним процессом командной оболочки, а первый — дочерним по отношению к последнему. Похоже, что командная оболочка создала собственную копию, которая затем в обратном порядке породила каждый из процессов в конвейере.

Если запустить ту же команду в фоновом режиме:

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

изменятся только идентификаторы процессов. Поскольку командная оболочка поддерживает управление заданиями, идентификатор группы процессов фонового режима сохраняет значение 949, равно как и идентификатор сеанса.

Что произойдет, если в этой оболочке фоновый процесс попытается прочитать ввод из управляющего терминала? Например, предположим, что мы запустили такую команду:

```
cat > temp.foo &
```

При наличии поддержки управления заданиями, если фоновое задание, находящееся в группе процессов фонового режима, попытается произвести чтение из управляющего терминала, ему будет послан сигнал `SIGTTIN`. При отсутствии поддержки управления заданиями командная оболочка автоматически перенаправляет стандартный ввод фонового процесса в устройство `/dev/null`, если процесс не перенаправит его самостоятельно. При попытке чтения из устройства `/dev/null` приложение получает признак конца файла. Это означает, что фоновый процесс `cat` сразу же прочитает признак конца файла и завершит работу нормальным образом.

Предыдущий абзац описывает случай, когда фоновый процесс обращается к управляющему терминалу через стандартный ввод, но что произойдет, если фоновый процесс попытается открыть устройство `/dev/tty` и прочитать входные данные из него? Ответ: «Это зависит от реализации», но, наверное, это не то, что нам нужно. Например, команда

```
crypt < salaries | lpr &
```

является таким конвейером. Мы запускаем эту команду в фоновом режиме, но программа `crypt` открывает `/dev/tty`, изменяет характеристики терминала (запрещает отображение вводимых символов), читает из устройства и восстанавливает характеристики терминала. Если запустить такой конвейер в фоновом режиме, на экране появится приглашение `Password:`, но введенный нами пароль для шифрования будет прочитан командной оболочкой, которая воспримет введенную строку как команду и попытается ее запустить. Следующая строка, введенная в командной оболочке, будет воспринята как пароль, в результате файл будет расшифрован неправильно и на принтер будет отправлен бессмысленный набор символов. Здесь присутствуют два процесса, которые одновременно пытаются читать из одного и того же устройства, и конечный результат таких попыток зависит от системы. Управление заданиями, как было описано ранее, позволяет лучше организовать совместное использование одного терминала несколькими процессами. Вернемся к примеру с Bourne shell, запустим три процесса в конвейере и посмотрим, как эта оболочка осуществляет управление процессами:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

Эта команда выведет следующее:

```
PID PPID PGID SID COMMAND
949 947 949 949 sh
1988 949 949 949 cat2
1989 1988 949 949 ps
1990 1988 949 949 cat1
```

Пусть вас не тревожит, если в вашей системе вы получите неверные имена команд. Иногда можно получить примерно такой результат:

```
PID PPID PGID SID COMMAND
949 947 949 949 sh
1831 949 949 949 sh
1832 1831 949 949 ps
1833 1831 949 949 sh
```

Дело в том, что процесс `ps` конкурирует с командной оболочкой за обладание процессором, когда та запускает команды `cat` с помощью функций `fork` и `exec`. В ситуации, показанной выше, командная оболочка еще не успела завершить вызовы функции `exec`, а команда `ps` уже вывела список процессов.

И опять последний процесс является дочерним процессом командной оболочки, а все предыдущие процессы — дочерними процессами последнего процесса. Рисунок 9.9 показывает смысл происходящего.

Так как последний процесс в конвейере (`cat2`) является дочерним по отношению к командной оболочке, она получит извещение о его завершении.

Теперь исследуем те же самые примеры, но уже в Linux с командной оболочкой, поддерживающей управление заданиями, и посмотрим, как эти командные оболочки обслуживают фоновые задания. В этой серии экспериментов мы использовали оболочку Bourne-again shell; результаты в других оболочках практически идентичны. Команда

```
ps -o pid,ppid,pgid,sid,tgid,comm
```

выводит следующее:

```
PID PPID PGRP SID TPGID COMMAND
2837 2818 2837 2837 5796 bash
5796 2837 5796 2837 5796 ps
```

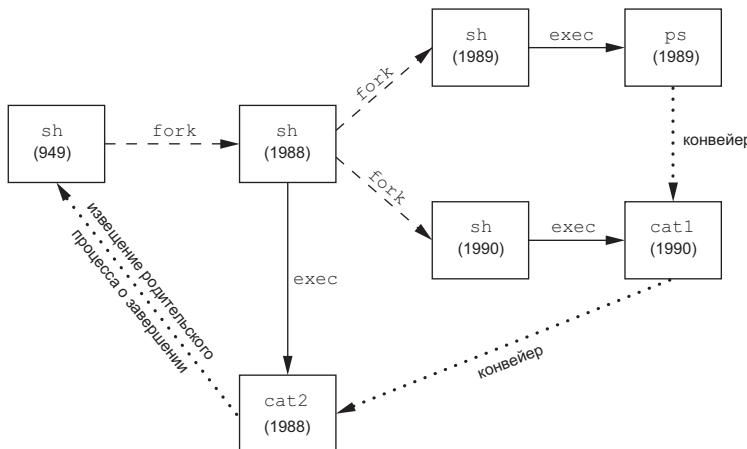


Рис. 9.9. Процессы в конвейере ps | cat1 | cat2, запущенном в оболочке Bourne shell

(Начиная с этого примера мы будем отмечать группу процессов переднего плана жирным шрифтом.) Здесь сразу же видны отличия от Bourne shell. Оболочка Bourne-again shell помещает задание переднего плана (`ps`) в собственную группу (5796). Команда `ps` — лидер группы процессов и единственный процесс в этой группе. Кроме того, эта группа является группой процессов переднего плана, так как имеет управляющий терминал. На время выполнения команды `ps` командная оболочка становится группой фоновых процессов. Но обратите внимание, что обе группы процессов, 2837 и 5796, принадлежат одному сеансу. В примерах в этом разделе мы увидим, что сеанс никогда не изменяется.

Запуск этой же команды в фоновом режиме:

```
ps -o pid,ppid,pgrp,sid,tpgid,comm &
```

дает

```
PID PPID PGRP SID TPGID COMMAND
2837 2818 2837 2837 2837 bash
5797 2837 5797 2837 2837 ps
```

И опять команда `ps` была помещена в собственную группу процессов (5797), но на этот раз она уже не является группой процессов переднего плана. Теперь это группа фоновых процессов. Значение `TPGID` (2837) указывает, что группа процессов переднего плана соответствует командной оболочке.

Запуск двух команд в конвейере:

```
ps -o pid,ppid,pgrp,sid,tpgid,comm | cat1
```

дает

```
PID PPID PGRP SID TPGID COMMAND
2837 2818 2837 2837 5799 bash
5799 2837 5799 2837 5799 ps
5800 2837 5799 2837 5799 cat1
```

Оба процесса, `ps` и `cat1`, теперь помещены в отдельную группу процессов (5799) – группу процессов переднего плана. Здесь также имеются отличия от аналогичного примера для Bourne shell. Командная оболочка Bourne shell первым запускала последний процесс в конвейере, и этот процесс становился родительским по отношению к первому процессу в конвейере. Здесь же родительским процессом для обеих команд становится Bourne-again shell. Если этот же конвейер запустить в фоновом режиме:

```
ps -o pid,ppid,pgrp,sid,tpgid,comm | cat1 &
```

результаты будут похожими, но на этот раз `ps` и `cat1` окажутся в одной группе фоновых процессов:

```
PID PPID PGRP SID TPGID COMMAND
2837 2818 2837 2837 2837 bash
5801 2837 5801 2837 2837 ps
5802 2837 5801 2837 2837 cat1
```

Обратите внимание, что порядок, в котором создаются процессы, может зависеть от выбранной командной оболочки.

9.10. Осиrotевшие группы процессов

Мы уже говорили, что процесс, родитель которого завершился, называется осиротевшим и наследуется процессом `init`. Теперь посмотрим, что произойдет, если осиротеет вся группа процессов, и как стандарт POSIX.1 регламентирует эту ситуацию.

Пример

Рассмотрим процесс, который порождает дочерний процесс и завершает работу. В этом нет ничего необычного (такое случается постоянно), тем не менее что произойдет, если дочерний процесс будет приостановлен (с помощью управления заданиями), а родительский в это время завершится? Как возобновить работу дочернего процесса и узнает ли он, что осиротел? Эта ситуация показана на рис. 9.10: родительский процесс порождает дочерний, затем потомок приостанавливается, а родительский процесс завершается.

Программа, создающая эту ситуацию, приводится в листинге 9.1. В ней имеется ряд новых для нас особенностей. Мы подразумеваем, что она будет выполняться под управлением командной оболочки, поддерживающей управление заданиями. В предыдущем разделе уже говорилось, что командная оболочка помещает процесс переднего плана в его собственную группу (в данном примере – 6099), а сама

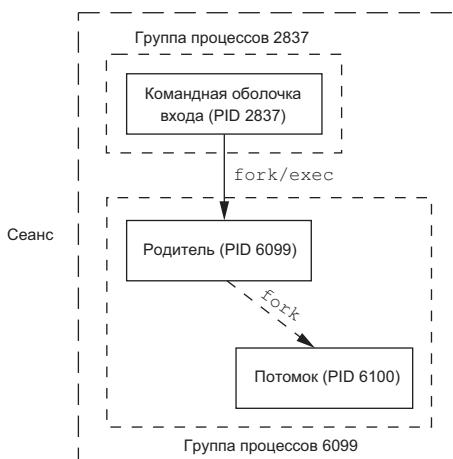


Рис. 9.10. Пример осиротевшей группы процессов

остается в своей группе (2837). Дочерний процесс наследует группу процессов от родителя (6099). Итак, после вызова функции `fork`:

- Родительский процесс приостанавливается на 5 секунд. Это наш (пусть и несовершенный) способ позволить дочернему процессу первым продолжить работу, прежде чем родительский завершится.
- Дочерний процесс устанавливает обработчик сигнала `SIGHUP`. Так мы сможем узнать, посыпался ли сигнал `SIGHUP` дочернему процессу. (Обработчики сигналов будут обсуждаться в главе 10.)
- Дочерний процесс с помощью функции `kill` посыпает самому себе сигнал `SIGTSTP`. Благодаря этому дочерний процесс приостанавливается точно так же, как приостанавливается задание переднего плана при вводе символа остановки (Control-Z).
- Когда родительский процесс завершается, дочерний становится «сиротой» и обретает себе родителя в лице процесса `init` с идентификатором 1.
- С этого момента дочерний процесс становится членом *осиротевшей группы процессов*. Стандарт POSIX.1 определяет осиротевшую группу процессов как группу, в которой родительский процесс любого члена группы либо сам является членом группы, либо не является членом сеанса, которому принадлежит группа. Другими словами, группа процессов не считается осиротевшей, пока в группе есть процесс, который имеет родителя в другой группе процессов, но в том же сеансе. Если группа процессов не является осиротевшей, есть шанс, что один из родительских процессов, расположенных в других группах процессов, но в том же самом сеансе, перезапустит приостановленный дочерний процесс. В нашем же случае родитель каждого процесса в группе принадлежит другому сеансу (так, для процесса 6100 родительским является процесс с идентификатором 1).
- Поскольку группа процессов оказывается осиротевшей, когда завершается родительский процесс, каждому приостановленному процессу в этой группе (как наш дочерний процесс) посыпается сигнал `SIGHUP` и вслед за ним сигнал `SIGCONT`, как того требует стандарт POSIX.1.

- Это приводит к тому, что дочерний процесс возобновляет работу после обработки сигнала `SIGHUP`. Реакция по умолчанию на этот сигнал — завершение процесса, поэтому мы должны предусмотреть функцию-обработчик, чтобы перехватить его. Соответственно мы предполагаем, что вызов функции `printf` сначала будет произведен в функции `sig_hup`, а затем в функции `pr_ids`.

Листинг 9.1. Создание осиротевшей группы процессов

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
printf("принят сигнал SIGHUP, pid = %ld\n", (long)getpid());
}

static void
pr_ids(char *name)
{
printf("%s: pid = %ld, ppid = %ld, pgid = %ld, tpgid = %ld\n",
name, (long)getpid(), (long)getppid(), (long)getpgrp(),
(long)tcgetpgrp(STDIN_FILENO));
fflush(stdout);
}

int
main(void)
{
char c;
pid_t pid;

pr_ids("родитель");
if ((pid = fork()) < 0) {
err_sys("ошибка вызова функции fork");
} else if (pid > 0) { /* родительский процесс */
sleep(5); /* приостановиться, чтобы дать потомку отработать первым */
} else { /* дочерний процесс */
pr_ids("потомок");
signal(SIGHUP, sig_hup); /* установить обработчик сигнала */
kill(getpid(), SIGSTP); /* остановить самого себя */
pr_ids("потомок"); /* вывести данные, */
/* когда процесс будет возобновлен */
if (read(STDIN_FILENO, &c, 1) != 1)
printf("ошибка чтения из управляемого TTY, errno = %d\n",
errno);
}
exit(0);
}
```

Вот результаты работы программы из листинга 9.1:

```
$ ./a.out
родитель: pid = 6099, ppid = 2837, pgid = 6099, tpgid = 6099
потомок: pid = 6100, ppid = 6099, pgid = 6099, tpgid = 6099
$ принят сигнал SIGHUP, pid = 6100
потомок: pid = 6100, ppid = 1, pgid = 6099, tpgid = 2837
ошибка чтения из управляемого TTY, errno = 5
```

Обратите внимание, что среди строк, выводимых дочерним процессом, появилось приглашение командной оболочки. Произошло это потому, что вывод на терми-

нал осуществляют два процесса — командная оболочка и дочерний процесс. Как мы и ожидали, родителем стал процесс с идентификатором 1.

После вызова функции `pr_ids` в дочернем процессе производится попытка чтения со стандартного ввода. Как уже говорилось выше в этой главе, когда процесс из группы фоновых процессов пытается читать из управляющего терминала, группе передается сигнал `SIGTTIN`. Но здесь мы имеем дело с осиротевшей группой процессов; если ядро остановит этим сигналом процесс из такой группы, он, скорее всего, никогда не будет возобновлен. Стандарт POSIX.1 требует, чтобы в такой ситуации функция `read` возвращала признак ошибки с кодом `EIO` в переменной `errno` (в данной системе этот код имеет значение 5).

Наконец, обратите внимание, что дочерний процесс был помещен в группу фоновых процессов, когда его родительский процесс завершился, так как родитель выполнялся командной оболочкой как задание переднего плана.

В разделе 19.5 мы увидим другой пример осиротевших групп процессов, когда будем рассматривать программу `pty`.

9.11. Реализация в FreeBSD

Теперь, когда мы поговорили о различных атрибутах процесса, группах процессов, сессиях и управляющих терминалах, настало время посмотреть, как все это может быть реализовано. Мы коротко рассмотрим реализацию в FreeBSD. Некоторые подробности реализации в SVR4 вы сможете найти в [Williams, 1989]. На рис. 9.11 показаны различные структуры данных, используемые в FreeBSD.

Рассмотрим все показанные поля структур, начиная со структуры `session`. Для каждого сеанса в памяти размещается отдельная структура `session` (это происходит, например, при обращении к функции `setsid`).

- `s_count` — количество групп процессов в сеансе. Когда этот счетчик обнуляется, память, занимаемая структурой, освобождается.
- `s_leader` — указатель на структуру `proc` лидера сеанса.
- `s_ttyvp` — указатель на структуру `vnode` управляющего терминала.
- `s_ttyp` — указатель на структуру `tty` управляющего терминала.
- `s_sid` — идентификатор сеанса. Помните, что понятие идентификатора сеанса не определяется стандартом Single UNIX Specification.

Во время вызова функции `setsid` в памяти ядра размещается новая структура `session`. Значение поля `s_count` устанавливается равным 1, в поле `s_leader` заносится указатель на структуру `proc` вызывающего процесса, в поле `s_sid` заносится идентификатор процесса и в поля `s_ttyvp` и `s_ttyp` — пустые указатели, поскольку новый сеанс не имеет управляющего терминала.

Теперь перейдем к структуре `tty`. Для каждого устройства терминала или псевдотерминала в памяти ядра размещается одна такая структура. (Более подробно о псевдотерминалах мы поговорим в главе 19.)

- `t_session` — указатель на структуру `session`, для которой этот терминал является управляющим. (Обратите внимание, что структура `tty` содержит указа-

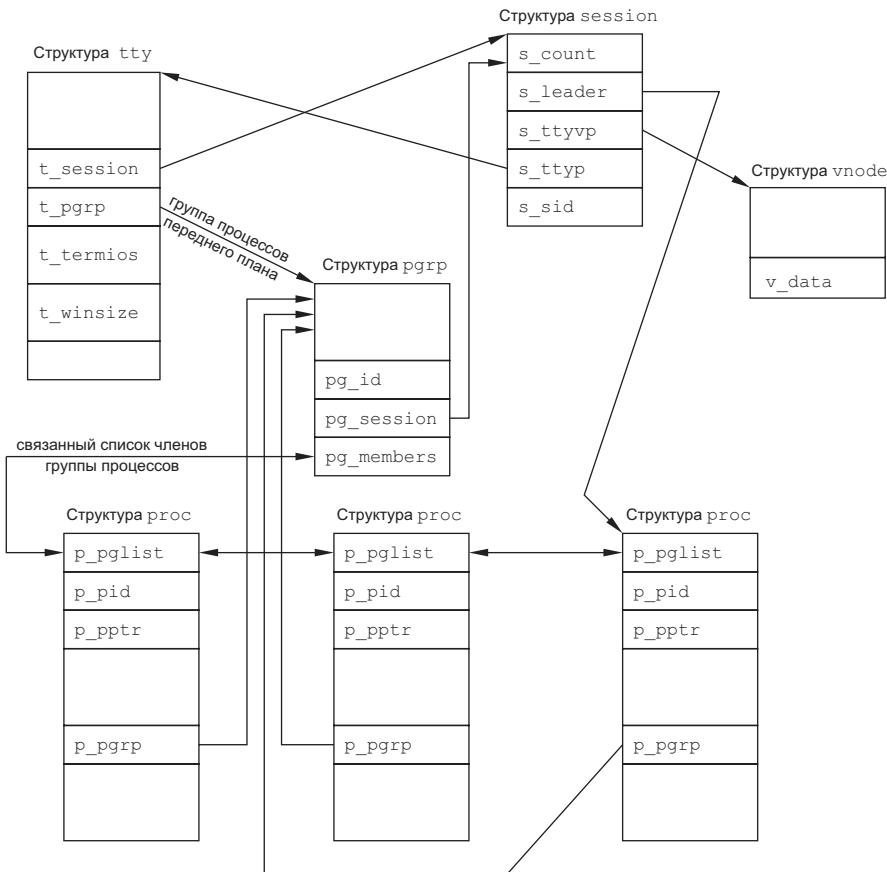


Рис. 9.11. Реализация сеансов и групп процессов в FreeBSD

тель на структуру **session**, а не наоборот.) Этот указатель используется терминалом для передачи сигнала **SIGHUP** лидеру сеанса, когда система теряет связь с терминалом (см. рис. 9.7).

- **t_pgrp** — указатель на структуру **pggrp** группы процессов переднего плана. Он используется терминалом для передачи сигналов группе процессов переднего плана. Это те самые три сигнала, которые генерируются в результате ввода специальных символов (сигнал прерывания, завершения и остановки).
- **t_termios** — структура **winsize**, которая содержит все специальные символы и дополнительную информацию о данном терминале, такую как скорость передачи данных, отображение вводимых символов (включено или выключено) и т. п. Мы еще вернемся к этой структуре в главе 18.
- **t_winsize** — структура **winsize**, которая содержит текущие размеры окна терминала. При изменении размеров окна терминала группе процессов переднего плана передается сигнал **SIGWINCH**. В разделе 18.12 мы покажем, как узнать и изменить размеры окна терминала.

Поиск группы процессов переднего плана для заданного сеанса ядро начинает со структуры `session`. Следуя по указателю `s_ttyp`, ядро находит структуру `tty` управляющего терминала, а затем по указателю `t_pgrp` отыскивает структуру `pgrp` группы процессов переднего плана. Структура `pgrp` содержит все необходимые сведения о данной группе процессов переднего плана.

- `pg_id` — идентификатор группы процессов.
- `pg_session` — указатель на структуру `session` для сеанса, которому принадлежит данная группа процессов.
- `pg_members` — указатель на список структур `proc`, соответствующих процессам, входящим в состав данной группы. Структура `p_pglist`, входящая в структуру `proc`, содержит два поля — указатели на предыдущую и следующую структуры `proc` — для организации двусвязного списка процессов в группе. Структура `proc` содержит всю информацию о процессе.
- `p_pid` — идентификатор процесса.
- `p_pptr` — указатель на структуру `proc` родительского процесса.
- `p_pgrp` — указатель на структуру `pgrp` группы, которой принадлежит процесс.
- `p_pglist` — структура, которая содержит указатели на предыдущий и следующий процессы в группе.

И наконец, структура `vnode`. Эта структура размещается в памяти в момент открытия устройства управляющего терминала. Все обращения к устройству `/dev/tty` из процесса проходят через структуру `vnode`.

9.12. Подведение итогов

В этой главе были описаны взаимоотношения между группами процессов — сеансы, которые состоят из групп процессов. Управление заданиями в настоящее время поддерживается большинством версий UNIX, и мы показали, как оно осуществляется в командной оболочке, которая поддерживает эту функциональную возможность. Понятие управляющего терминала также связано со взаимоотношениями между процессами.

Мы много раз упоминали сигналы, которые широко используются для организации взаимодействий между процессами. В следующей главе обсуждение сигналов будет продолжено и мы подробно рассмотрим все сигналы системы UNIX.

Упражнения

- 9.1 Вспомните обсуждение файлов `utmp` и `wtmp` в разделе 6.8 и ответьте на вопрос: почему запись о выходе из системы производится процессом `init`? Происходит ли то же самое в случае завершения сеанса, открытого через сетевое соединение?
- 9.2 Напишите программу, которая с помощью функции `fork` порождает дочерний процесс, создающий, в свою очередь, новый сеанс. Проверьте, становится ли дочерний процесс лидером группы и теряет ли он управляющий терминал.

10 Сигналы

10.1. Введение

Сигналы — это программные прерывания. Большинство серьезных приложений опираются на работу с сигналами. Сигналы дают возможность обработки асинхронных событий — например, когда пользователь вводит символ прерывания, чтобы остановить программу, или когда одна из программ в конвейере завершается аварийно.

Сигналы появились в самых ранних версиях UNIX, но модель сигналов, которую предоставляли такие системы, как Version 7, была недостаточно надежна. Сигналы могли теряться, и процессу было довольно сложно отключить отдельные сигналы на время выполнения критических фрагментов кода. Существенные изменения в модель сигналов внесли 4.3BSD и SVR3, в этих версиях были добавлены так называемые *надежные сигналы*. Но изменения, сделанные в Беркли и в AT&T, оказались несовместимы между собой. К счастью, POSIX.1 стандартизировал функции обслуживания надежных сигналов, и именно их мы обсудим.

Эта глава начинается с краткого обзора, где описывается назначение каждого сигнала. Затем мы рассмотрим проблемы, имевшие место в ранних реализациях. Чтобы разобраться во всех тонкостях, иногда очень важно понять, какие проблемы были связаны с реализацией. В этой главе приводится большое количество примеров, которые не совсем корректны, и обсуждаются имеющиеся в них недочеты.

10.2. Концепция сигналов

Прежде всего, каждый сигнал имеет собственное имя. Имена всех сигналов начинаются с последовательности `SIG`. Например, `SIGABRT` — это сигнал прерывания, который генерируется, когда процесс вызывает функцию `abort`. Сигнал `SIGALRM` генерируется, когда таймер, установленный функцией `alarm`, отмерит указанный промежуток времени. В Version 7 было 15 различных сигналов, в SVR4 и 4.4BSD — уже 31 сигнал. FreeBSD 8.0 поддерживает 32 разных сигнала, Mac OS X 10.6.8 и Linux 3.2.0 — 31 сигнал, а Solaris 10 — 40 различных сигналов. Кроме того, FreeBSD, Linux и Solaris поддерживают дополнительные сигналы, определяемые приложениями в виде расширений реального времени (расширения ре-

ального времени POSIX не рассматриваются в данной книге, поэтому за дополнительной информацией обращайтесь к [Gallmeister, 1995]).

Все имена сигналов определены как константы с положительными числовыми значениями (номерами сигналов) в заголовочном файле `<signal.h>`.

Фактически реализации определяют сигналы в отдельных заголовочных файлах, которые подключаются файлом `<signal.h>`. Вообще, считается дурным тоном в исходных текстах ядра подключать заголовочные файлы, предназначенные для приложений пользовательского уровня. Поэтому если и приложение и ядро нуждаются в одних и тех же определениях, информация размещается в заголовочном файле ядра, который затем подключается в заголовочном файле пользовательского уровня. Так, FreeBSD 8.0 и Mac OS X 10.6.8 определяют сигналы в файле `<sys/signalf.h>`. Linux 3.2.0 определяет сигналы в файле `<bits/signum.h>`, а Solaris 10 – в файле `<sys/iso/signal_iso.h>`.

Сигнала с номером 0 не существует. В разделе 10.9 мы увидим, что функция `kill` использует номер сигнала 0 в особых случаях. Стандарт POSIX.1 называет такой сигнал *null signal* (пустой сигнал).

Сигналы могут порождаться различными условиями.

- Сигналы, генерируемые терминалом, возникают, когда пользователь вводит определенные символы. Нажатие клавиши **DELETE** (или **Control-C** – в большинстве систем) порождает сигнал прерывания (**SIGINT**). Таким способом можно прервать выполнение программы, вышедшей из-под контроля. (В главе 18 мы увидим, что этот сигнал можно привязать к любой клавише на клавиатуре.)
- Аппаратные ошибки – деление на 0, ошибка доступа к памяти и прочее – также приводят к генерации сигналов. Эти ошибки обычно обнаруживаются аппаратным обеспечением, которое извещает ядро об их появлении. После этого ядро генерирует соответствующий сигнал и передает его процессу, который выполнялся в момент появления ошибки. Например, сигнал **SIGSEGV** посыпается процессу при попытке обратиться к неверному адресу в памяти.
- Функция `kill(2)` позволяет процессу передать любой сигнал другому процессу или группе процессов. Естественно, здесь существуют свои ограничения: необходимо быть владельцем процесса, которому посыпается сигнал, или обладать привилегиями суперпользователя.
- Команда `kill(1)` позволяет передавать сигналы другим процессам. Эта программа является простым интерфейсом к функции `kill`. Зачастую эта команда используется для принудительного завершения вышедших из-под контроля фоновых процессов.
- Сигналы могут порождаться при условиях, определяемых программно, например, когда нужно известить приложение о наступлении некоторого события. Эти условия определяются не аппаратурой (как, например, деление на 0), а программным обеспечением. Примерами таких сигналов могут служить **SIGURG** (посыпается, когда через сетевое соединение приходят экстренные (out-of-band) данные), **SIGPIPE** (посыпается пишущему процессу, когда он пытается записать данные в канал после завершения процесса, читающего из канала) и **SIGALRM** (посыпается процессу по истечении установленного им таймера).

Сигналы являются собой классический пример асинхронных событий. Сигнал может быть передан процессу в любой момент. Чтобы выяснить причину, породившую сигнал, процесс не может просто проверить некоторую переменную (как, например, `errno`), вместо этого он должен обратиться к ядру с предложением: «если появится этот сигнал — сделай то-то и то-то».

В случае появления сигнала можно запросить ядро произвести одно из трех действий. Они называются *диспозициями* сигнала, или *действиями*, связанными с сигналом.

1. Игнорировать сигнал. Это действие возможно для большинства сигналов, но два сигнала, `SIGKILL` и `SIGSTOP`, нельзя игнорировать. Причина, почему эти два сигнала не могут быть проигнорированы, заключается в том, что ядру и суперпользователю необходима возможность завершить или остановить любой процесс. Кроме того, если проигнорировать некоторые из сигналов, возникающих в результате аппаратных ошибок (таких, как деление на 0 или попытка обращения к несуществующей памяти), поведение процесса может стать непредсказуемым.
2. Перехватить сигнал. Для этого нужно сообщить ядру адрес функции, которая будет обрабатывать сигнал. В этой функции можно предусмотреть действия по обработке условия, породившего сигнал. Например, создавая командный интерпретатор, можно предусмотреть в нем прерывание выполняемой команды, запущенной пользователем, и возврат к главному циклу, когда пользователь пошлет сигнал прерывания. Если пойман сигнал `SIGCHLD`, который означает завершение дочернего процесса, функция, перехватившая сигнал, может вызвать функцию `waitpid`, чтобы получить идентификатор дочернего процесса и код его завершения. Еще один пример: если процесс создает временные файлы, имеет смысл написать функцию обработки сигнала `SIGTERM` (сигнал завершения, посыпаемый командой `kill` по умолчанию), которая будет удалять временные файлы. Имейте в виду, что сигналы `SIGKILL` и `SIGSTOP` нельзя перехватить.
3. Применить действие по умолчанию. Каждому сигналу поставлено в соответствие некоторое действие по умолчанию (перечислены в табл. 10.1). Заметьте, что для большинства сигналов действие по умолчанию заключается в завершении процесса.

В табл. 10.1 перечислены имена всех сигналов и указано, какими системами они поддерживаются и каково действие по умолчанию для каждого сигнала. Если в колонке SUS (Single UNIX Specification) стоит галочка, значит, сигнал определен как часть базовой спецификации POSIX.1, а если указана аббревиатура XSI — сигнал определен как расширение XSI.

Если в колонке «Действие по умолчанию» указано «завершить + core», это означает, что образ памяти процесса сохраняется в файле `core` в текущем рабочем каталоге процесса. (Имя файла `core` наглядно демонстрирует, как давно эта функциональная особенность появилась в UNIX.) Большинство отладчиков могут использовать этот файл для выяснения причин, породивших преждевременное завершение процесса.

Таблица 10.1. Сигналы UNIX

Имя	Описание	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Действие по умолчанию
SIGABRT	Аварийное завершение (<code>abort</code>)	✓	✓	✓	✓	✓	✓	Завершить + core
SIGALRM	Истекло время таймера (<code>alarm</code>)		✓	✓	✓	✓	✓	Завершить
SIGBUS	Аппаратная ошибка		✓	✓	✓	✓	✓	Завершить + core
SIGCANCEL	Для внутреннего использования библиотекой <code>threads</code>						✓	Игнорировать
SIGCHLD	Изменение состояния дочернего процесса		✓	✓	✓	✓	✓	Игнорировать
SIGCONT	Возобновить работу приостановленного процесса		✓	✓	✓	✓	✓	Продолжить/игнорировать
SIGEMT	Аппаратная ошибка			✓	✓	✓	✓	Завершить + core
SIGFPE	Арифметическая ошибка	✓	✓	✓	✓	✓	✓	Завершить + core
SIGFREEZE	Закрепление контрольной точки						✓	Игнорировать
SIGHUP	Обрыв связи		✓	✓	✓	✓	✓	Завершить
SIGILL	Недопустимая инструкция	✓	✓	✓	✓	✓	✓	Завершить + core
SIGINFO	Запрос состояния с клавиатуры			✓		✓		Игнорировать
SIGINT	С терминала введен символ прерывания	✓	✓	✓	✓	✓	✓	Завершить
SIGIO	Асинхронный ввод/вывод			✓	✓	✓	✓	Завершить/игнорировать

Имя	Описание	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Действие по умолчанию
SIGIOT	Аппаратная ошибка			✓	✓	✓	✓	Завершить + core
SIGJVM1	Для внутреннего использования виртуальной машиной Java						✓	Игнорировать
SIGJVM2	Для внутреннего использования виртуальной машиной Java						✓	Игнорировать
SIGKILL	Завершение		✓	✓	✓	✓	✓	Завершить
SIGLOST	Ресурс потерян						✓	Завершить
SIGLWP	Для внутреннего использования библиотекой <code>threads</code>			✓			✓	Завершить/игнорировать
SIGPIPE	Запись в канал, из которого никто не читает		✓	✓	✓	✓	✓	Завершить
SIGPOLL	Событие опроса (<code>poll</code>)				✓		✓	Завершить
SIGPROF	Истекло время профилирующего таймера (<code>setitimer</code>)			✓	✓	✓	✓	Завершить
SIGPWR	Падение напряжения питания/перезапуск				✓		✓	Завершить/игнорировать
SIGQUIT	С терминала введен символ завершения		✓	✓	✓	✓	✓	Завершить + core
SIGSEGV	Ошибка доступа к памяти	✓	✓	✓	✓	✓	✓	Завершить + core
SIGSTKFLT	Ошибка, связанная со стеком сопроцессора				✓			Завершить

Таблица 10.1 (продолжение)

Имя	Описание	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Действие по умолчанию
SIGSTOP	Приостановить процесс		✓	✓	✓	✓	✓	Остановить процесс
SIGSYS	Неверный системный вызов		XSI	✓	✓	✓	✓	Завершить + core
SIGTERM	Завершение	✓	✓	✓	✓	✓	✓	Завершить
SIGTHAW	Освобождение контрольной точки						✓	Игнорировать
SIGTHR	Для внутреннего использования библиотекой <i>threads</i>			✓				Завершить
SIGTRAP	Аппаратная ошибка		XSI	✓	✓	✓	✓	Завершить + core
SIGTSTP	С терминала введен символ приостановки		✓	✓	✓	✓	✓	Остановить процесс
SIGTTIN	Чтение из управляющего терминала фоновым процессом		✓	✓	✓	✓	✓	Остановить процесс
SIGTTOU	Запись в управляющий терминал фоновым процессом		✓	✓	✓	✓	✓	Остановить процесс
SIGURG	Экстренное событие (сокеты)		✓	✓	✓	✓	✓	Игнорировать
SIGUSR1	Определяемый пользователем сигнал		✓	✓	✓	✓	✓	Завершить
SIGUSR2	Определяемый пользователем сигнал		✓	✓	✓	✓	✓	Завершить
SIGVTALRM	Истекло время виртуального таймера (<i>setitimer</i>)		XSI	✓	✓	✓	✓	Завершить

Имя	Описание	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Действие по умолчанию
SIGWAITING	Для внутреннего использования библиотекой <code>threads</code>						✓	Игнорировать
SIGWINCH	Изменение размеров окна терминала			✓	✓	✓	✓	Игнорировать
SIGXCPU	Исчерпан лимит процессорного времени (<code>setrlimit</code>)		XSI	✓	✓	✓	✓	Завершить/закончить + core
SIGXFSZ	Превышено ограничение на размер файла (<code>setrlimit</code>)		XSI	✓	✓	✓	✓	Завершить/закончить + core
SIGXRES	Превышено ограничение на использование ресурса						✓	Завершить + core/игнорировать

Возможность создания файла `core` – функциональная особенность, присущая большинству версий UNIX. Хотя она и не является частью POSIX.1, тем не менее она упоминается в расширении XSI стандарта Single UNIX Specification как возможное, зависящее от реализации действие.

Имя `core`-файла варьируется в разных реализациях. В FreeBSD 8.0, например, `core`-файл получает имя `cmdname.core`, где `cmdname` – имя команды, соответствующей процессу, получившему сигнал. В Mac OS X 10.6.8 файл `core` получает имя `core.pid`, где `pid` – идентификатор процесса, получившего сигнал. (Эти системы позволяют настроить правила именования файлов `core` через параметр `sysctl`. В Linux 3.2.0 имя настраивается через файл `/proc/sys/kernel/core_pattern`.)

Большинство реализаций сохраняют файл `core` в текущем рабочем каталоге соответствующего процесса, но Mac OS X помещает все файлы `core` в каталог `/cores`.

Файл `core` не создается, если (а) файл программы имеет установленный бит set-user-ID, а текущий пользователь не является его владельцем; (б) файл программы имеет установленный бит set-group-ID, а текущий пользователь не принадлежит к группе владельца файла; (в) пользователь не имеет права на запись в текущий каталог; (г) файл уже существует и пользователь не имеет права на запись в него; (д) файл слишком велик (вспомните предел `RLIMIT_CORE` из раздела 7.11). Файл `core` (если он еще не существует) обычно создается с правами на запись и на чтение для владельца, хотя в Mac OS X выдается только право на чтение для владельца.

В табл. 10.1 сигналы с описанием «аппаратная ошибка» соответствуют зависящим от реализации аппаратным ошибкам. Многие из них взяты из оригинальной реализации UNIX для PDP-11. Проверьте справочное руководство по вашей операционной системе и уточните, каким именно ошибкам соответствуют эти сигналы.

А теперь опишем каждый сигнал подробнее.

SIGABRT Генерируется вызовом функции `abort` (раздел 10.17). Процесс завершается аварийно.

SIGALRM Генерируется по истечении таймера, установленного функцией `alarm` (раздел 10.10). Также генерируется по истечении таймера, установленного функцией `setitimer(2)`.

SIGBUS Соответствует аппаратной ошибке, определяемой реализацией. Обычно этот сигнал генерируется в случае некоторых ошибок, связанных с памятью, которые мы рассмотрим в разделе 14.8.

SIGCANCEL Используется библиотекой `threads` в Solaris. Не предназначен для общего использования.

SIGCHLD Когда процесс завершается или останавливается, родительскому процессу посыпается сигнал `SIGCHLD`. По умолчанию этот сигнал игнорируется, но родительский процесс может перехватить его, если желает получать извещения об изменении состояния дочерних процессов. Функция, перехватывающая этот сигнал, обычно вызывает одну из функций семейства `wait`, чтобы получить идентификатор дочернего процесса и код завершения.

В ранних версиях System V имелся похожий сигнал с именем `SIGCLD` (без `H`). Семантика этого сигнала отличалась от семантики других сигналов, и еще справочное руководство SVR2 рекомендовало не использовать его в новых программах. (Как ни странно, в SVR3 и SVR4 из справочного руководства это предупреждение исчезло.) Приложения должны использовать сигнал `SIGCHLD`, но нужно знать, что многие версии UNIX определяют сигнал `SIGCLD`, идентичный сигналу `SIGCHLD`, для сохранения обратной совместимости. Если вам понадобится определить семантику сигнала `SIGCLD` в своей системе, обратитесь к страницам справочного руководства. Эти два сигнала мы обсудим в разделе 10.7.

SIGCONT Посыпается остановленным процессам, чтобы возобновить их работу. Действие по умолчанию заключается в продолжении работы процесса, если он был остановлен, — для работающего процесса сигнал игнорируется. Полноэкранный редактор, например, может перехватывать этот сигнал, чтобы использовать функцию-обработчик для перерисовки экрана. Дополнительная информация об этом сигнале приводится в разделе 10.21.

SIGEMT Соответствует аппаратной ошибке, определяемой реализацией. Имя `EMT` происходит от инструкции PDP11 — «emulator trap» (ловушка эмулятора). Этот сигнал поддерживается не всеми платформами. В Linux, например, этот сигнал поддерживается только для некоторых аппаратных архитектур, таких как SPARC, MIPS и PARISC.

SIGFPE Свидетельствует об арифметической ошибке, такой как деление на 0 или переполнение числа с плавающей точкой.

SIGFREEZE Определен только в Solaris. Используется для извещения процессов, которые должны предпринять дополнительные действия перед фиксацией состояния системы, что обычно происходит, когда система уходит в спящий или в ждущий режим.

SIGHUP Посыпается управляющему процессу (лидеру сеанса), связанному с управляющим терминалом, если обнаружен обрыв связи с терминалом. На рис. 9.11 видно, что сигнал посыпается процессу, на который указывает поле `s_leader` в структуре `session`. Этот сигнал генерируется, только если сброшен флаг терминала `CLOCAL`. (Флаг `CLOCAL` устанавливается для локального терминала. Этот флаг сообщает драйверу, что он должен игнорировать все управляющие сигналы модема. В главе 18 мы расскажем, как устанавливается этот флаг.)

Обратите внимание, что лидер сеанса, которому передается сигнал, может быть фоновым процессом (см. рис. 9.7). Это отличает данный сигнал от других сигналов, генерируемых терминалом (прерывание, завершение и останов), которые всегда посыпаются группе процессов переднего плана.

Этот сигнал также генерируется в случае завершения лидера сеанса. В такой ситуации сигнал посыпается всем процессам из группы процессов переднего плана. Нередко этот сигнал используется для извещения процессов-демонов (глава 13) о необходимости перечитать конфигурационные файлы. Причина, по которой для этой цели выбирается именно сигнал `SIGHUP`, заключается в том, что если не послать этот сигнал явно, демоны никогда не примут его, поскольку у них нет управляющего терминала.

SIGILL Указывает, что процесс выполнил недопустимую машинную инструкцию. В 4.3BSD этот сигнал генерировался функцией `abort`. Теперь она генерирует сигнал `SIGABRT`.

SIGINFO Этот BSD-сигнал генерируется драйвером терминала при нажатии клавиши запроса состояния (часто — `Control-T`). Посыпается всем процессам из группы процессов переднего плана (см. рис. 9.8). Обычно этот сигнал используется для вывода информации о состоянии процессов в группе процессов переднего плана.

Linux не поддерживает сигнал SIGINFO, за исключением платформы Alpha, где он определен с тем же номером, что и сигнал SIGPWR. Эта поддержка была добавлена для совместимости с программным обеспечением, разработанным для OSF/1.

SIGINT Генерируется драйвером терминала при нажатии клавиши прерывания (часто — `DELETE` или `Control-C`). Посыпается всем процессам из группы процессов переднего плана (см. рис. 9.8). Этот сигнал часто используется для прерывания выполнения приложений, вышедших из-под контроля, особенно когда они начинают выводить ненужную информацию на экран.

SIGIO Указывает на событие асинхронной операции ввода/вывода. Мы обсудим его в разделе 14.6.2.

В табл. 10.1 указано, что действие по умолчанию для сигнала SIGIO — завершить процесс либо игнорировать. К сожалению, выбор действия по умолчанию зависит от реализации. В System V сигнал SIGIO идентичен сигналу SIGPOLL, поэтому по умолчанию он завершает процесс. В BSD этот сигнал по умолчанию игнорируется.

Linux 3.2.0 и Solaris 10 определяют сигнал SIGIO с тем же номером, что и SIGPOLL, поэтому по умолчанию он завершает процесс. FreeBSD 8.0 и Mac OS X 10.6.8 по умолчанию игнорируют этот сигнал.

SIGIOT Соответствует аппаратной ошибке, определяемой реализацией.

Имя IOT происходит от мнемоники инструкции PDP11 – «input/output TRAP» (ловушка ввода/вывода). В ранних версиях System V этот сигнал генерировался функцией `abort`. Теперь она генерирует сигнал SIGABRT.

В FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 сигнал SIGIOT определен с тем же номером, что и сигнал SIGABRT.

SIGJVM1 Сигнал, зарезервированный в Solaris для использования виртуальной машиной Java.

SIGJVM2 Еще один сигнал, зарезервированный в Solaris для использования виртуальной машиной Java.

SIGKILL Один из двух сигналов, которые нельзя перехватить или игнорировать в приложении. Дает возможность системному администратору уничтожить любой процесс.

SIGLOST Применяется в Solaris для извещения клиентов NFSv4 (сетевой файловой системы) об ошибке повторного приобретения блокировки при попытке восстановления.

SIGLWP Используется библиотекой `threads` в Solaris и недоступен для общего использования. В FreeBSD сигнал SIGLWP определен как псевдоним сигнала SIGTHR.

SIGPIPE Посыпается процессу, который предпринял попытку записи в канал, когда процесс, производивший чтение из канала, уже завершился. Мы обсудим каналы в разделе 15.2. Этот сигнал также генерируется при попытке выполнить запись в сокет типа SOCK_STREAM, когда соединение уже разорвано. Сокеты обсуждаются в главе 16.

SIGPOLL Отмечен как устаревший в SUSv4 и может быть удален в следующих версиях стандарта. Может генерироваться при наступлении определенного события в опрашиваемом устройстве. Мы рассмотрим этот сигнал при обсуждении функции `poll` в разделе 14.4.2. Сигнал SIGPOLL появился в SVR3 и в некоторой степени соответствует сигналам SIGIO и SIGURG в BSD.

В Linux и Solaris сигнал SIGPOLL определен с тем же номером, что и сигнал SIGIO.

SIGPROF Отмечен как устаревший в SUSv4 и может быть удален в следующих версиях стандарта. Генерируется по истечении интервала времени профилирующего таймера, установленного функцией `setitimer(2)`.

SIGPWR Реализация этого сигнала зависит от системы. В основном он используется в системах, снабженных источником бесперебойного питания (UPS). Обнаружив сбой в электросети, источник бесперебойного питания извещает об этом систему и принимает на себя обеспечение питания системы. Пока ничего более не предпринимается, так как система продолжает пытаться от аккумуляторных батарей. Но когда напряжение в сети отсутствует продолжительное время и напряжение аккумуляторов падает ниже определенного уровня, программное обеспечение

обычно извещается об этом повторно, и с этого момента у системы остается примерно 15–30 секунд, чтобы корректно завершить работу. В этот момент посыпается сигнал **SIGPWR**. В большинстве систем имеется процесс, который получает извещение о падении напряжения аккумуляторов и посыпает сигнал **SIGPWR** процессу **init**, а **init** берет на себя заботу об остановке системы.

Для этих целей в Solaris 10 и в некоторых дистрибутивах Linux предусматриваются специальные записи в **inittab**: **powerfail** и **powerwait** (или **powerokwait**).

В табл. 10.1 указано, что по умолчанию сигнал **SIGPWR** завершает процесс или игнорируется. К сожалению, действие по умолчанию зависит от реализации. В Linux по умолчанию этот сигнал завершает процесс, в Solaris — игнорируется.

SIGQUIT Генерируется драйвером терминала при вводе символа завершения (частью **Control-**). Посыпается всем процессам из группы процессов переднего плана (см. рис. 9.8). При этом происходит не только завершение группы процессов переднего плана (как в случае сигнала **SIGINT**), но и создание файла **core**.

SIGSEGV Показывает, что процесс обратился к недопустимому адресу в памяти (что обычно служит признаком ошибки в программе, такой как разыменование пустого указателя).

Имя **SEGV** происходит от фразы «*segmentation violation*» (нарушение правил сегментации).

SIGSTKFLT Определен только в Linux. Появился в самых ранних версиях Linux и предназначался для обнаружения ошибок, связанных со стеком арифметического сопроцессора. Этот сигнал не генерируется ядром, но остается для сохранения обратной совместимости.

SIGSTOP Останавливает процесс. Похож на сигнал **SIGTSTP**, порождаемый драйвером терминала, но в отличие от него **SIGSTOP** нельзя перехватить или игнорировать.

SIGSYS Свидетельствует о неверном системном вызове. Каким-то образом процесс выполнил машинную инструкцию, которая была воспринята ядром как системный вызов, но параметр инструкции указывал на неверный тип системного вызова. Это может произойти, если скомпилировать программу, использующую недавно появившийся системный вызов, и попытаться запустить двоичный выполняемый файл в более старой версии системы, которая этот системный вызов не поддерживает.

SIGTERM Сигнал завершения процесса, который посыпается командой **kill(1)** по умолчанию. Так как его можно перехватить, обработка сигнала **SIGTERM** дает программам возможность корректно завершить работу, освободив занятые ресурсы (в противоположность сигналу **SIGKILL**, который нельзя перехватить или игнорировать).

SIGTHAW Определен только в Solaris и используется для извещения процессов о том, что они должны предпринять определенные действия после выхода системы из ждущего или спящего режима.

SIGTHR Используется библиотекой **threads** в FreeBSD. Определен с тем же номенклатурой, что и сигнал **SIGLWP**.

SIGTRAP Соответствует аппаратной ошибке, определяемой реализацией.

Имя сигнала произошло от инструкции PDP11 – TRAP (ловушка). Реализации часто используют его для передачи управления отладчикам по достижении точки останова.

SIGTSTP Этот сигнал приостановки генерируется драйвером терминала при вводе символа приостановки (часто Control-Z). Посыпается всем процессам из группы процессов переднего плана (рис. 9.8).

К сожалению, термин «приостановка» может иметь несколько смыслов. Когда обсуждалась возможность управления заданиями, мы говорили о приостановке и возобновлении работы. Однако если речь идет о драйвере терминала, для обозначения остановки и возобновления вывода на терминал с помощью клавиш Control-S и Control-Q традиционно используется термин «останов». Таким образом, в случае драйвера терминала символ, который приводит к генерации сигнала **SIGTSTP**, называется символом приостановки, а не останова.

SIGTTIN Генерируется драйвером терминала, когда фоновый процесс пытается выполнить операцию чтения из управляющего терминала (раздел 9.8). В особых случаях – если (а) процесс, выполняющий чтение, игнорирует или блокирует этот сигнал или (б) группа процессов, в которой находится читающий процесс, является осиротевшей, – сигнал не генерируется; вместо этого операция чтения завершается с признаком ошибки и в переменную `errno` записывается код **EIO**.

SIGTTOU Генерируется драйвером терминала, когда фоновый процесс пытается выполнить запись в управляющий терминал (раздел 9.8). В отличие от только что описанного сигнала **SIGTTIN**, в данном случае процесс может разрешить фоновым процессам запись в управляющий терминал. Эту возможность мы рассмотрим в главе 18.

Если запись в терминал для фоновых процессов запрещена, то, как и в случае сигнала **SIGTTIN**, возможны два особых случая: когда (а) пишущий процесс игнорирует или блокирует этот сигнал или (б) группа пишущего процесса является осиротевшей. В этих случаях сигнал не генерируется, а в случае (б) операция записи завершается с признаком ошибки и в переменную `errno` записывается код **EIO**.

Независимо от того, разрешено ли фоновому процессу выполнять запись в терминал, некоторые другие операции с терминалом также могут генерировать сигнал **SIGTTOU**: `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow` и `tcsetpgrp`. Эти операции рассматриваются в главе 18.

SIGURG Сообщает процессу, что произошло экстренное событие. Может генерироваться при поступлении экстренных (out-of-band) данных через сетевое соединение.

SIGUSR1 Определяется пользователем и предназначен для внутреннего использования в приложениях.

SIGUSR2 Еще один сигнал, определяемый пользователем. Подобно сигналу **SIGUSR1**, также предназначен для внутреннего применения в приложениях.

SIGVTALRM Генерируется по истечении периода времени, назначенного виртуальному таймеру функцией `setitimer(2)`.

SIGWAITING Предназначен для внутреннего использования в библиотеке `threads` в Solaris.

SIGWINCH Ядро управляет изменением размера окна, связанного с каждым терминалом или псевдотерминалом. Процесс может получить и изменить размер окна с помощью функции `ioctl`, которую мы рассмотрим в разделе 18.12. Если процесс изменяет размер окна с помощью команды `set-window-size` функции `ioctl`, ядро посыпает сигнал `SIGWINCH` группе процессов переднего плана.

SIGXCPU Стандарт Single UNIX Specification поддерживает как расширение XSI идею ограничений ресурсов (раздел 7.11). Если процесс достигает мягкого предела на использование центрального процессора, ему посыпается сигнал `SIGXCPU`.

В табл. 10.1 указано, что по умолчанию сигнал `SIGXCPU` завершает процесс или завершает с созданием файла `core`. Действие по умолчанию зависит от реализации. В Linux 3.2.0 и Solaris 10 по умолчанию этот сигнал завершает процесс и создает файл `core`, тогда как в FreeBSD 8.0 и Mac OS X 10.6.8 он завершает процесс без создания файла `core`. Стандарт Single UNIX Specification требует, чтобы по умолчанию происходило аварийное завершение процесса. Создавать ли при этом файл `core`, каждая из реализаций может определять самостоятельно.

SIGXFSZ Посыпается процессу, который превысил мягкий предел на размер файла (раздел 7.11).

Так же как в случае с сигналом `SIGXCPU`, действие по умолчанию зависит от операционной системы. В Linux 3.2.0 и Solaris 10 по умолчанию процесс завершается с созданием файла `core`, тогда как в FreeBSD 8.0 и Mac OS X 10.6.8 процесс завершается без создания файла `core`. Стандарт Single UNIX Specification требует, чтобы по умолчанию происходило аварийное завершение процесса. Создавать ли при этом файл `core`, каждая из реализаций может определять самостоятельно.

SIGXRES Определен только в Solaris. Может применяться для извещения процессов о превышении установленного ограничения на использование ресурса. Механизм управления ресурсами в Solaris дает возможность управлять ресурсами, разделяемыми между несколькими независимыми приложениями.

10.3. Функция `signal`

Функция `signal` — это простейший интерфейс к сигналам UNIX.

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Возвращает предыдущую диспозицию сигнала (см. далее) в случае успеха, `SIG_ERR` — в случае ошибки

Функция `signal` определена стандартом ISO C, который ничего не говорит о многозадачности, группах процессов, терминальном вводе/выводе и т. п. Поэтому определение сигналов в этом стандарте практически бесполезно для систем UNIX.

Реализации, происходящие от System V, поддерживают функцию `signal`, но она представляет устаревшую семантику механизма ненадежных сигналов (которая описана

в разделе 10.4). Эта функция обеспечивает обратную совместимость с приложениями, требующими устаревшей семантики. Новые приложения не должны использовать ненадежные сигналы.

4.4BSD также поддерживает функцию `signal`, но она реализована в терминах функции `sigaction` (которая описана в разделе 10.14), то есть функция `signal` в 4.4BSD предоставляет новую семантику надежных сигналов. Большинство современных систем следуют этой стратегии, но Solaris 10 следует семантике, принятой в System V.

Поскольку семантика функции `signal` различается в разных реализациях, вместо нее следует использовать функцию `sigaction`. В разделе 10.14 мы представим реализацию функции `signal` на основе `sigaction`. Все примеры в этой книге используют функцию `signal` из листинга 10.12, обеспечивающую единство семантики работы с сигналами независимо от конкретной платформы.

Аргумент `signo` — это имя сигнала из табл. 10.1. В качестве аргумента `func` можно передать (а) константу `SIG_IGN`, или (б) константу `SIG_DFL`, или (в) адрес функции, которая будет вызвана при получении сигнала. Константа `SIG_IGN` сообщает системе, что сигнал должен игнорироваться. (Не забывайте, что два сигнала, `SIGKILL` и `SIGSTOP`, не могут игнорироваться.) Если указана константа `SIG_DFL`, с сигналом связывается действие по умолчанию (последняя колонка в табл. 10.1). Если указан адрес функции, она будет вызываться при получении сигнала, то есть будет «перехватывать» сигнал. Такие функции называются *обработчиками* или *перехватчиками* сигналов.

Прототип функции `signal` показывает, что она принимает два аргумента и возвращает указатель на функцию, которая не имеет возвращаемого значения (`void`). Первый аргумент функции `signal`, `signo`, представляет собой целое число. Второй аргумент — указатель на функцию, которая не возвращает значение и принимает единственный целочисленный аргумент. Функция, адрес которой возвращает функция `signal`, также принимает один целочисленный аргумент (последний (`int`)). Проще говоря, функции — обработчику сигнала передается единственный аргумент (целое число — номер сигнала), и она ничего не возвращает. Когда функция `signal` вызывается, чтобы установить обработчик сигнала, второй аргумент должен быть указателем на функцию. Возвращаемое значение функции `signal` — указатель на предыдущий обработчик сигнала.

Большинство систем вызывают обработчик сигнала с дополнительными, зависящими от реализации, аргументами. Этот вопрос мы рассмотрим в разделе 10.14.

Довольно сложный для восприятия прототип функции `signal`, приведенный в начале раздела, можно определить проще, через использование следующей инструкции `typedef` [Plauger, 1992]:

```
typedef void Sigfunc(int);
```

Тогда прототип самой функции `signal` будет выглядеть так:

```
Sigfunc *signal(int, Sigfunc *);
```

Мы включили это определение в заголовочный файл `apue.h` (приложение В) и будем использовать его в наших примерах.

Заглянув в файл `<signal.h>`, можно обнаружить следующие объявления:

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

Эти константы можно использовать вместо «указателя на функцию, которая принимает один целочисленный аргумент и ничего не возвращает» — это второй аргумент функции `signal` и одновременно ее возвращаемое значение. Значения констант не обязательно должны быть -1 , 0 и 1 . Но они должны быть такими, чтобы их нельзя было принять за адреса функций. В большинстве реализаций используются значения, приведенные выше.

Пример

В листинге 10.1 показан простейший обработчик сигнала, который перехватывает два сигнала, определяемые пользователем, и выводит их номера. Функцию `pause` мы рассмотрим в разделе 10.10, она просто приостанавливает процесс до получения сигнала.

Листинг 10.1. Простейшая программа, которая перехватывает сигналы SIGUSR1 и SIGUSR2

```
#include "apue.h"

static void sig_usr(int); /* один обработчик для двух сигналов */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo) /* аргумент — номер сигнала */
{
    if (signo == SIGUSR1)
        printf("принят сигнал SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("принят сигнал SIGUSR2\n");
    else
        err_dump("принят сигнал %d\n", signo);
}
```

Мы запускали эту программу как фоновый процесс и с помощью команды `kill(1)` посыпали ей сигналы. Обратите внимание, что термин `kill` (уничтожить) в UNIX представляет пример не вполне корректного именования. Команда `kill(1)` и функция `kill(2)` просто посыпают сигнал процессу или группе процессов. Завершится процесс при получении сигнала или нет, зависит от того, какой сигнал был послан и перехватывается ли этот сигнал процессом.

```
$ ./a.out &           запустить процесс в фоновом режиме
[1]    7216             командная оболочка вывела номер задания
$ kill -USR1 7216      и идентификатор процесса
принят сигнал SIGUSR1
$ kill -USR2 7216      послать сигнал SIGUSR1
принят сигнал SIGUSR2
$ kill 7216            послать сигнал SIGUSR2
теперь послать сигнал SIGTERM
[1]+ Terminated ./a.out
```

Когда был послан сигнал SIGTERM, процесс завершился, поскольку он не перехватывает этот сигнал, а по умолчанию он завершает процесс.

Запуск программы

В момент запуска программы всем сигналам назначаются действия по умолчанию или сигналы игнорируются. Обычно для всех сигналов назначаются действия по умолчанию, если только процесс, вызвавший функцию `exec`, не игнорирует какие-либо сигналы. Фактически функции семейства `exec` изменяют диспозицию тех сигналов, которые перехватываются, на действия по умолчанию и оставляют без изменения все остальные. (Это вполне естественно, поскольку сигнал, который перехватывается процессом, вызвавшим функцию `exec`, не может быть перехвачен той же функцией в новой программе, так как адрес функции-перехватчика в вызывающей программе наверняка потеряет смысл в новой программе.)

Вот один характерный пример, как интерактивная командная оболочка обращается с сигналами SIGINT и SIGQUIT фонового процесса. Если командная оболочка не поддерживает управление заданиями, то при запуске фонового процесса, например

```
cc main.c &
```

командная оболочка автоматически установит диспозицию этих сигналов для фонового процесса в значение SIG_IGN, поэтому ввод символа прерывания не оказывает никакого влияния на фоновый процесс. Если бы этого не было сделано, при вводе символа прерывания завершился бы не только процесс переднего плана, но и все фоновые процессы.

Многие интерактивные программы, перехватывающие эти два сигнала, содержат примерно такой код:

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

Этот код устанавливает перехватчик сигнала, только если сигнал не игнорируется.

Эти два вызова наглядно демонстрируют недостаток функции `signal` — отсутствие возможности определить текущую диспозицию сигнала без ее изменения. Далее в этой главе мы увидим, что функция `sigaction` предоставляет такую возможность.

Создание процесса

Когда процесс вызывает функцию `fork`, дочерний процесс наследует диспозиции сигналов от родительского процесса. В данном случае, поскольку дочерний процесс представляет собой полную копию родительского процесса, адреса функций-обработчиков не теряют своего смысла в дочернем процессе.

10.4. Ненадежные сигналы

В ранних версиях UNIX (таких, как Version 7) сигналы были ненадежными. То есть сигналы могли теряться: иными словами, процесс мог не получить посланный ему сигнал. Кроме того, процесс имел весьма ограниченные возможности управления сигналами: он мог либо перехватить сигнал, либо игнорировать его. Иногда может возникнуть потребность заблокировать сигнал, то есть не игнорировать его, а просто отложить посылку сигнала до момента, когда приложение будет готово принять его.

Такое положение дел было исправлено в 4.2BSD, когда появились так называемые надежные сигналы. Затем ряд других изменений, также обеспечивающих поддержку надежных сигналов, был внесен в SVR3. Стандарт POSIX.1 в качестве образца выбрал модель BSD.

Одна из проблем, связанных с ранними версиями, заключалась в том, что действие сигнала сбрасывалось в значение по умолчанию после передачи сигнала. (В предыдущем примере, когда мы запускали программу из листинга 10.1, эта проблема не возникала, поскольку сигнал перехватывался всего один раз.) Классический пример из книг по программированию, описывающий обработку сигнала прерывания в ранних версиях UNIX, обычно выглядит примерно так:

```
int sig_int();                  /* функция-обработчик */
...
signal(SIGINT, sig_int);        /* установить функцию-обработчик */
...
sig_int()
{
    signal(SIGINT, sig_int); /* переустановить функцию-обработчик */
    ...                      /* обработка сигнала ... */
}
```

(Функция-обработчик объявлена как возвращающая целое число потому, что в ранних версиях UNIX отсутствовала поддержка типа `void` стандарта ISO C.)

Этот пример также не лишен недостатков. Проблема здесь в том, что существует некоторый промежуток времени между моментом посыпки сигнала и моментом вызова функции `signal` в функции-обработчике, когда сигнал `SIGINT` может быть послан повторно. Этот повторный сигнал может вызвать выполнение действия по умолчанию — завершение процесса. Это один из примеров, когда все работает правильно большую часть времени, заставляя нас думать, что все в порядке, хотя на самом деле это не так.

Еще одна проблема в ранних версиях UNIX состояла в том, что процесс был не в состоянии отключить сигнал на время, когда его появление нежелательно. Про-

цесс мог полностью игнорировать сигнал, но он не мог сообщить системе: «Следующие сигналы не должны поступать ко мне, но система должна запомнить, что они были посланы». Классический пример, демонстрирующий этот недостаток, представлен фрагментом кода, который перехватывает сигнал и устанавливает флаг, отмечающий появление сигнала:

```
int sig_int();           /* функция-обработчик сигнала */
int sig_int_flag;        /* ненулевое значение, если был получен сигнал */

main()
{
    signal(SIGINT, sig_int); /* установить функцию-обработчик */
    ...
    while (sig_int_flag == 0)
        pause();            /* приостановить работу в ожидании сигнала */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* переустановить функцию-обработчик */
    sig_int_flag = 1;        /* установить флаг для проверки в основной программе */
}
```

Здесь процесс вызывает функцию `pause`, ожидая, пока сигнал не будет перехвачен. При получении сигнала функция-обработчик просто устанавливает флаг `sig_int_flag` в ненулевое значение. Ядро автоматически возобновляет работу процесса после выхода из функции-обработчика, после чего процесс обнаруживает введенный флаг и выполняет все необходимые действия. Но здесь опять же существует промежуток времени, когда все может пойти не так, как мы ожидали. Если сигнал будет послан в промежутке времени между проверкой флага `sig_int_flag` и вызовом функции `pause`, процесс рискует приостановиться навсегда (при условии, что сигнал больше никогда не будет послан). В данной ситуации сигнал будет потерян. Это еще один пример, когда ошибочный код корректно работает большую часть времени. Обнаружение и отладка подобных ошибок — чрезвычайно сложная задача.

10.5. Прерванные системные вызовы

Ранние версии UNIX обладали одним свойством: если процесс, заблокированный в «медленном» системном вызове, перехватывал сигнал, выполнение системного вызова прерывалось. В этом случае системный вызов возвращал признак ошибки с кодом `EINTR` в переменной `errno`. Так было сделано в предположении, что раз сигнал был послан и процесс перехватил его, возможно, произошло что-то, из-за чего системный вызов должен прервать ожидание и вернуть управление процессу.

Здесь следует понимать разницу между системными вызовами и обычными функциями. Имеется в виду, что при перехвате сигнала прерывается выполнение именно системного вызова в ядре.

Для поддержки такого поведения системные вызовы были разделены на «медленные» и все остальные. Медленные системные вызовы — это такие вызовы, которые могут заблокировать процесс навсегда. В эту категорию попали:

- Операция чтения, которая может навсегда заблокировать вызывающий процесс при отсутствии данных в файлах некоторых типов (каналы, устройства терминалов и сетевые устройства).
- Операция записи, которая может навсегда заблокировать вызывающий процесс, если записываемые данные не могут быть немедленно отправлены в файлы перечисленных выше типов.
- Операция открытия, которая может заблокировать вызывающий процесс, пока не будут выполнены некоторые условия при открытии файлов определенных типов (таких, как устройства терминалов, которые ожидают установления модемного соединения).
- Функция `pause` (которая по определению приостанавливает выполнение процесса до получения сигнала) и функция `wait`.
- Некоторые операции функции `ioctl`.
- Некоторые функции межпроцессного взаимодействия (глава 15).

Известное исключение составляют операции ввода/вывода с дисковыми устройствами. Несмотря на то что операции чтения и записи на дисковое устройство могут временно заблокировать выполнение процесса (пока драйвер дискового устройства ставит запрос в очередь, чтобы затем выполнить его), тем не менее при отсутствии аппаратных ошибок операции ввода/вывода всегда завершаются достаточно быстро и разблокируют вызывающий процесс.

Один из случаев, которые могут привести к прерыванию системного вызова, — когда процесс инициирует операцию чтения из терминала, а пользователь в это время уходит на неопределенно долгое время. В этом случае процесс может быть заблокирован на многие часы или даже дни и оставаться в таком состоянии, если система не будет остановлена.

Семантика прерванных системных вызовов `read` и `write` была изменена в версии стандарта POSIX.1 2001 года. Предшествующие версии стандарта оставляли за реализацией выбор, как обрабатывать операции чтения и записи, которые уже частично передали некоторый объем данных. Если вызов `read` прочитал данные и поместил их в буфер приложения, но к моменту прерывания был получен не весь объем запрошенных данных, тогда решение вопроса, завершить ли системный вызов с кодом ошибки `EINTR` или позволить ему завершиться без признака ошибки и вернуть частично полученные данные, оставлялось на усмотрение операционной системы. Точно так же, если операция записи была прервана после передачи некоторого объема данных из буфера приложения, система могла завершить системный вызов с кодом ошибки `EINTR` или позволить ему завершиться без признака ошибки и вернуть информацию о количестве записанных данных. Исторически сложилось так, что реализации, происходящие от System V, завершают системный вызов с признаком ошибки, тогда как реализации, производные от BSD, возвращают управление без признака ошибки с информацией о фактически выполненной работе. Начиная с версии 2001 года стандарт POSIX.1 требует соблюдения BSD-подобной семантики.

Проблема с прерванными системными вызовами заключается в том, что приходится явно обрабатывать возможные ошибочные ситуации. Типичная последовательность инструкций (в случае операции чтения, когда необходимо прочитать полный объем данных, даже если операция чтения была прервана) может быть следующей:

```
again:
    if ((n = read(fd, buf, BUFSIZE)) < 0) {
        if (errno == EINTR)
            goto again; /* просто прерванный системный вызов */
        /* обработать другие возможные ошибки */
    }
```

Чтобы избавить приложения от необходимости обрабатывать ситуации прерванных системных вызовов, в 4.2BSD было введено понятие автоматического перезапуска прерванных системных вызовов. К системным вызовам, которые перезапускаются автоматически, были отнесены `ioctl`, `read`, `readv`, `write`, `writev`, `wait` и `waitpid`. Как мы уже упоминали, первые пять вызовов прерываются сигналами, только если они взаимодействуют с медленными устройствами. Системные вызовы `wait` и `waitpid` всегда прерываются перехваченными сигналами. Так как это породило другую проблему для приложений, которые не желали, чтобы системный вызов автоматически перезапускался в случае его прерывания, в 4.3BSD у процессов появилась возможность изменить такое поведение для отдельных сигналов.

Стандарт POSIX.1 требует от реализаций перезапускать системные вызовы, только когда для прерывающего сигнала действует флаг `SA_RESTART`. Как будет показано в разделе 10.14, этот флаг используется с функцией `sigaction`, чтобы позволить приложениям запрашивать перезапуск прерванных системных вызовов.

Исторически, когда для установки обработчиков сигналов использовалась функция `signal`, разные реализации по-разному обрабатывали прерванные системные вызовы. По умолчанию System V никогда не перезапускала их. С другой стороны, BSD перезапускает их, если они были прерваны сигналами. По умолчанию FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 перезапускают системные вызовы, прерванные сигналами. Однако в Solaris 10 по умолчанию должна возвращаться ошибка (`EINTR`). Используя собственную версию функции `signal` (из листинга 10.12), мы можем избавиться от этих различий.

Одна из причин, почему в 4.2BSD был введен автоматический перезапуск, заключается в том, что иногда мы просто не знаем, является ли устройство ввода/вывода медленным устройством. Если мы пишем программу, которая может работать в интерактивном режиме, ей, вероятно, придется работать с медленным устройством, так как терминалы относятся к этой категории. Если эта программа перехватывает сигналы, то в случае, когда система не поддерживает возможность перезапуска системных вызовов, пришлось бы выполнять проверку каждой операции чтения и записи на предмет появления ошибки `EINTR` и возобновлять прерванную операцию.

В табл. 10.2 перечислены функции, предназначенные для работы с сигналами, и их семантика для некоторых реализаций.

Таблица 10.2. Функции, предоставляемые различными реализациями

Функция	Система	Обработчик сигнала остается установленным	Возможность блокировать сигналы	Автоматический перезапуск прерванных системных вызовов
signal	ISO C, POSIX.1	Не определено	Не определено	Не определено
	V7, SVR2, SVR3, SVR4, Solaris			Никогда
	4.2BSD	✓	✓	Всегда
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	✓	✓	По умолчанию
sigaction	POSIX.1, 4.4BSD, SVR4, FreeBSD, Mac OS X, Linux, Solaris	✓	✓	По выбору

Имейте в виду, что версии UNIX от других производителей могут иметь значения, отличные от перечисленных в таблице. Например, функция `sigaction` в SunOS 4.1.2 по умолчанию перезапускает прерванные системные вызовы, что отличает ее от платформ, представленных в табл. 10.2.

В листинге 10.12 приводится наша версия функции `signal`, которая автоматически пытается перезапустить прерванные системные вызовы (за исключением сигнала `SIGALRM`). Другая функция, `signal_intr`, которая приводится в листинге 10.13, никогда не пытается выполнить перезапуск.

Мы еще будем говорить о прерванных системных вызовах в разделе 14.4, при обсуждении функций `select` и `poll`.

10.6. Реентерабельные функции

Когда процесс обрабатывает перехваченный сигнал, нормальная последовательность выполнения инструкций временно нарушается обработчиком сигнала. После этого процесс продолжает работу, но выполняет инструкции уже в функции-обработчике. Если обработчик сигнала возвращает управление (а не вызывает, например, функцию `exit` или `longjmp`), процесс продолжает выполнение нормальной последовательности инструкций, прерванной перехваченным сигналом. (Это напоминает ситуацию, когда работа приложения прерывается аппаратным прерыванием.) Но, находясь внутри обработчика сигнала, мы не можем определить, в каком месте процесса произошло прерывание. А что, если процесс вызвал функцию `malloc`, чтобы распределить дополнительную память, и обработчик сигнала также вызвал функцию `malloc?` Или если процесс произвел вызов функции, такой как `getpwnam` (раздел 6.2), сохраняющей результат статически, и обработчик сигнала также вызвал ту же самую функцию? В случае функции `malloc` результаты такого вызова могут оказаться разрушительными для приложения, поскольку обычно функция `malloc` поддерживает связанный список всех выделенных ею

областей памяти и вызов из обработчика сигнала может произойти как раз в тот момент, когда она вносит изменения в этот список. В случае с функцией `getpwname` информация, записанная по запросу процесса, может оказаться затертый информацией, запрошенной из обработчика сигнала.

Стандарт Single UNIX Specification определяет перечень функций, которые должны обеспечивать безопасность вызова из обработчиков сигналов. Эти функции являются реентерабельными и в стандарте Single UNIX Specification называются *безопасными для использования в обработчиках асинхронных сигналов*. Помимо обеспечения реентерабельности, они блокируют доставку любых сигналов до своего завершения, если это может вызвать нарушения в работе приложения. Эти функции перечислены в табл. 10.3. Большинство функций, отсутствующих в табл. 10.3, не были внесены в список либо потому, что (а) известно, что они

Таблица 10.3. Реентерабельные функции, которые можно вызывать из обработчиков сигналов

abort	faccessat	linkat	select	socketpair
accept	fchmod	listen	sem_post	stat
access	fchmodat	lseek	send	symlink
aio_error	fchown	lstat	sendmsg	symlinkat
aio_return	fchownat	mkdir	sendto	tcdrain
aio_suspend	fcntl	mkdirat	setgid	tcflow
alarm	fdatsasync	mkfifo	setpgid	tcflush
bind	fexecv	mkfifoat	setsid	tcgetattr
cfgetispeed	fork	mknode	setsockopt	tcgetpgrp
cfgetospeed	fstat	mknodeat	setuid	tcsendbreak
cfsetispeed	fstatat	open	shutdown	tcsetattr
cfsetospeed	fsync	openat	sigaction	tcsetpgrp
chdir	ftruncate	pause	sigaddset	time
chmod	futimens	pipe	sigdelset	timer_getoverrun
chown	getegid	poll	sigemptyset	timer_gettime
clock_gettime	geteuid	posix_trace_event	sigfillset	timer_settime
close	getgid	pselect	sigismember	times
connect	getgroups	raise	signal	umask
creat	getpeername	read	sigpause	uname
dup	getpgrp	readlink	sigpending	unlink
dup2	getpid	readlinkat	sigprocmask	unlinkat
execl	getppid	recv	sigqueue	utime
execle	getsockname	recvfrom	sigset	utimensat
execv	getsockopt	recvmsg	sigsuspend	utimes
execve	getuid	rename	sleep	wait
_Exit	kill	renameat	socketmark	waitpid
_exit	link	rmdir	socket	write

используют структуры данных, размещаемые статически, либо (б) они вызывают функцию `malloc` или `free`, либо (в) входят в стандартную библиотеку ввода/вывода. Большинство реализаций стандартной библиотеки ввода/вывода используют глобальные структуры данных способом, исключающим реентерабельность. Обратите внимание, что хотя мы вызываем функцию `printf` из обработчиков сигналов в некоторых наших примерах, нельзя гарантировать, что этот вызов даст предсказуемый результат, так как обработчик сигнала может быть вызван в процессе работы функции `printf`, вызванной в другом месте программы.

Имейте также в виду, что даже если вызов одной из функций, перечисленных в табл. 10.3, производится из обработчика сигнала, для каждого из потоков управления существует единственная переменная `errno` (вспомните раздел 1.7) и мы можем изменить ее значение. Например, обработчик сигнала может быть вызван сразу же после того, как код ошибки был записан в переменную `errno` в главной программе. Если обработчик сигнала вызывает, например, функцию `read`, она может изменить значение этой переменной, затерев значение, только что записанное в главной программе. Поэтому при вызове функций из табл. 10.3 всегда сохраняйте значение переменной `errno` в начале обработчика и восстанавливайте перед возвратом в главную программу. (Чаще всего при перехвате сигнала `SIGCHLD` функция-обработчик обращается к одной из функций `wait`, которая может изменить значение переменной `errno`.)

Обратите внимание на отсутствие функций `longjmp` (раздел 7.10) и `siglongjmp` (раздел 10.15) в табл. 10.3, поскольку обработчик сигнала может быть вызван как раз в тот момент, когда эти функции выполняют обновление структур данных нереентерабельным способом. Эти структуры данных могут оказаться обновленными частично, если вместо обычного возврата из функции-обработчика вызвать функцию `siglongjmp`. Если необходимо сделать что-то с глобальными структурами данных в то время, когда может быть вызван обработчик сигнала, вызывающий функцию `siglongjmp`, приложение должно блокировать сигналы на время обновления этих данных.

Пример

В листинге 10.2 представлена программа, которая обращается к нереентерабельной функции `getpwnam` из обработчика сигнала, вызываемого один раз в секунду. Функцию `alarm` мы рассмотрим в разделе 10.10. Мы использовали ее для генерации сигнала `SIGALRM` каждую секунду.

Листинг 10.2. Вызов нереентерабельной функции из обработчика сигнала

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd *rootptr;

    printf("внутри обработчика сигнала\n");
    if ((rootptr = getpwnam("root")) == NULL)
        err_sys("ошибка вызова функции getpwnam(root)");
```

```

alarm(1);
}

int
main(void)
{
    struct passwd *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("ошибка вызова функции getpwnam");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("возвращаемое значение повреждено!, pw_name = %s\n",
                   ptr->pw_name);
    }
}

```

Когда мы запустили эту программу, она начала выдавать совершенно непредсказуемые результаты. Обычно программа завершалась по сигналу SIGSEGV, когда обработчик сигнала возвращал управление после нескольких итераций. Исследование файла `core` показало, что проблема возникала, когда `main` вызывала функцию `getpwnam`, которая, в свою очередь, вызывает функцию `free`, и в этот момент происходило прерывание — обработчик сигнала также вызывал функцию `getpwnam`, которая обращается к функции `free`. В результате структуры данных, которыми управляют функции `malloc` и `free`, оказывались поврежденными. Иногда программе удавалось проработать несколько секунд, прежде чем она получала сигнал SIGSEGV. Когда функция `main` работала вполне корректно после вызова обработчика сигнала, возвращаемое значение `getpwnam` иногда оказывалось поврежденным, а иногда — нет.

Как показывает этот пример, вызов нереентерабельных функций из обработчиков сигналов может дать самые непредсказуемые результаты.

10.7. Семантика сигнала SIGCLD

Два сигнала, которые продолжают вносить сумятицу в умы программистов, — SIGCLD и SIGCHLD. Прежде всего, SIGCLD (без H) — это имя сигнала, пришедшее из System V. Семантика этого сигнала отличается от семантики BSD-сигнала с именем SIGCHLD. Соответствующий сигнал из стандарта POSIX.1 также получил имя SIGCHLD.

Семантика сигнала SIGCHLD подобна семантике всех остальных сигналов. Когда изменяется состояние дочернего процесса, генерируется сигнал SIGCHLD, и мы должны вызвать одну из функций семейства `wait`, чтобы узнать, что произошло. Однако System V традиционно обслуживает сигнал SIGCLD иначе, чем остальные сигналы. Системы, основанные на SVR4, продолжают эту сомнительную (в смысле совместимости) традицию, если диспозиция этого сигнала устанавливается функциями `signal` или `sigset` (устаревшие SVR3-совместимые функции, предназначенные для изменения диспозиции сигнала). Этот способ обслуживания сигнала SIGCLD заключается в следующем:

- Если процесс установит его диспозицию в значение `SIG_IGN`, дочерние процессы не будут порождать процессы-зомби. Обратите внимание: это действие отличается от действия по умолчанию `SIG_DFL`, которое, в соответствии с табл. 10.1, просто игнорирует сигнал. Вместо этого по завершении дочернего процесса его код завершения просто теряется. Если затем вызвать одну из функций `wait`, вызывающий процесс окажется заблокированным, пока все его дочерние процессы не завершат работу, после чего `wait` вернет значение `-1` с кодом `ECHILD` в переменной `errno`. (Диспозиция сигнала по умолчанию — игнорировать сигнал, но это не означает, что его семантика будет следовать семантике `SIG_IGN`. Поэтому мы должны явно установить диспозицию сигнала в значение `SIG_IGN`.)

Стандарт POSIX.1 не определяет поведение системы в случае, когда сигнал `SIGCHLD` игнорируется, поэтому подобное поведение вполне допустимо. Расширение XSI требует, чтобы такое поведение поддерживалось для сигнала `SIGCHLD`.

В 4.4BSD, если сигнал `SIGCHLD` игнорируется, это всегда приводит к созданию зомби. Чтобы избежать появления зомби, мы должны вызывать функцию `wait` для дочерних процессов. Если в SVR4 вызывается функция `signal` или `sigset`, чтобы установить диспозицию сигнала `SIGCHLD` в значение `SIG_IGN`, зомби никогда не появляются. Все четыре платформы, описываемые в этой книге, в своем поведении следуют за SVR4.

При использовании функции `sigaction` можно установить флаг `SA_NOCLDWAIT` (табл. 10.5), чтобы избежать появления зомби. Это поведение также поддерживается всеми четырьмя платформами.

- Если для сигнала `SIGCLD` назначена функция-обработчик, ядро сразу же проверяет наличие дочерних процессов и вызывает обработчик сигнала `SIGCLD`, если такие процессы имеются. Пункт 2 меняет алгоритм обработчика сигнала, как показано в следующем примере.

Пример

Как уже говорилось в разделе 10.4, первое, что нужно сделать на входе в обработчик сигнала, — переустановить его с помощью функции `signal`. (Тем самым минимизировав интервал времени, когда сигнал может быть потерян в результате временного сброса диспозиции в значение по умолчанию.) Этот прием демонстрируется в листинге 10.3. Данная программа не работает на традиционных платформах System V. Она будет выводить непрерывный поток сообщений `принят сигнал SIGCLD`, пока, наконец, процесс не завершится аварийно в результате исчерпания пространства, отведенного под стек.

В FreeBSD 8.0 и Mac OS X 10.6.8 эта проблема не проявляется, потому что системы, основанные на BSD, вообще не поддерживают семантику System V для сигнала `SIGCLD`. Linux 3.2.0 также лишена этого недостатка, потому что не вызывает обработчик сигнала `SIGCHLD` сразу после его установки — даже при том, что сигналы `SIGCLD` и `SIGCHLD` определены с одним и тем же номером. С другой стороны, в Solaris 10 обработчик сигнала в такой ситуации действительно вызывается, но в ядро включен дополнительный код, который помогает избежать описанной проблемы.

Хотя все четыре рассматриваемые в этой книге платформы разрешили данную проблему, тем не менее есть такие системы (например, AIX), в которых она все еще существует.

Листинг 10.3. Обработчик сигнала SIGCLD из System V, который не работает

```
#include "apue.h"
#include <sys/wait.h>

static void sig_cld(int);

int
main()
{
    pid_t pid;

    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("ошибка вызова функции signal");
    if ((pid = fork()) < 0) {
        perror("ошибка вызова функции fork");
    } else if (pid == 0) /* дочерний процесс */
        sleep(2);
        _exit(0);
    }

    pause(); /* родительский процесс */
    exit(0);
}

static void
sig_cld(int signo) /* прерывает функцию pause() */
{
    pid_t pid;
    int status;

    printf("принят сигнал SIGCLD\n");

    if (signal(SIGCLD, sig_cld) == SIG_ERR) /* переустановить обработчик */
        perror("ошибка вызова функции signal");

    if ((pid = wait(&status)) < 0) /* получить состояние дочернего процесса */
        perror("ошибка вызова функции wait");

    printf("pid = %d\n", pid);
}
```

Проблема этой программы в том, что она вызывает функцию `signal` в самом начале функции-обработчика, и это приводит к тому, что ядро проверяет наличие дочерних процессов (а дочерний процесс имеется, поскольку мы уже находимся в обработчике сигнала `SIGCLD`), что, в свою очередь, приводит к очередному вызову обработчика сигнала. Функция-обработчик опять вызывает функцию `signal`, и цикл повторяется.

Чтобы исправить ошибку, нужно поместить вызов `signal` после вызова функции `wait`. Благодаря этому вызов функции `signal` будет производиться после получения кода завершения дочернего процесса, и в следующий раз сигнал будет сгенерирован ядром, только если завершится один из дочерних процессов.

Формулировка стандарта POSIX.1 не определяет, должен ли генерироваться сигнал, когда к моменту установки обработчика сигнала SIGCHLD существует завершившийся дочерний процесс, код завершения которого еще не был получен. Это допускает реализа-

цию поведения, описанного выше. Но поскольку стандарт POSIX.1 не предполагаетброс диспозиции сигнала в значение по умолчанию после его появления (здесь мы предполагаем, что для изменения диспозиции сигнала используется функция POSIX.1 — `sigaction`), то и нет никакой потребности в том, чтобы переустанавливать обработчик сигнала `SIGCHLD` в теле функции-обработчика.

Поинтересуйтесь семантикой сигнала `SIGCHLD` в вашей системе. В особенности обратите внимание на определение, которое в одних системах выглядит как `#define SIGCHLD SIGCLD`, а в других — наоборот. Изменение имени сигнала может устраниь проблемы при сборке программы, которая была написана для другой системы, но если эта программа зависит от конкретной семантики, возможно, что она не будет работать.

На четырех платформах, обсуждаемых в данной книге, сигнал `SIGCLD` эквивалентен сигналу `SIGCHLD`.

10.8. Надежные сигналы. Терминология и семантика

Мы должны определить некоторые термины, используемые при обсуждении сигналов. Прежде всего для процесса *генерируется* (или процессу посыпается) сигнал, когда происходит некоторое событие. Таким событием может быть аппаратная ошибка (например, деление на 0), программное событие (например, истечение интервала времени, отмеряемого таймером), сигнал, сгенерированный терминатором, или вызов функции `k11`. Когда генерируется сигнал, ядро, как правило, устанавливает некий флаг в таблице процессов.

Когда выполняется действие, предусмотренное для сигнала, мы говорим, что сигнал *доставлен* процессу. Интервал времени между генерацией сигнала и его доставкой называется *периодом ожидания обработки*.

Процесс может заблокировать доставку сигнала. Если процессу посыпается сигнал, который был заблокирован, и при этом для сигнала установлено либо действие по умолчанию, либо перехват, сигнал остается в состоянии ожидания обработки, пока процесс (а) не разблокирует сигнал или (б) не установит диспозицию сигнала в значение `SIG_IGN`. Что делать с сигналом, определяется системой в момент доставки, но не в момент генерации. Это позволяет процессам изменить диспозицию сигнала до того, как он будет доставлен. Процесс может получить перечень ожидающих или заблокированных сигналов с помощью функции `sigpending` (раздел 10.13).

Что произойдет, если заблокированный сигнал будет сгенерирован несколько раз, прежде чем процесс разблокирует его? Стандарт POSIX.1 допускает доставку как единственного сигнала, так и всех сгенерированных сигналов. Если система доставляет процессу все сгенерированные сигналы, мы говорим, что сигналы ставятся в очередь. Однако большинство версий UNIX при отсутствии расширений реального времени POSIX.1 не ставят сигналы в очередь, то есть ядро доставляет единственный сигнал.

В SUSv4 определение поддержки сигналов реального времени было перемещено из раздела расширений в раздел базовых спецификаций. С течением времени все больше систем будут поддерживать очереди сигналов, даже если они не поддерживают расширения реального времени. Очереди сигналов мы обсудим ниже, в разделе 10.20.

Справочное руководство SVR2 утверждало, что сигнал SIGCLD ставится в очередь, если процесс в данный момент выполняет функцию обработки сигнала SIGCLD. Возможно, это было истинно только на концептуальном уровне, поскольку фактическая реализация была иной. На самом деле ядро повторно генерировало сигнал, как было описано в разделе 10.7. В SVR3 текст справочного руководства претерпел некоторые изменения: сообщается, что в момент обработки сигнала SIGCLD последующие его доставки игнорируются. В справочном руководстве SVR4 вообще исчезло любое упоминание о том, что происходит с сигналом SIGCLD, полученным в то время, когда процесс выполняет код функции — обработчика этого сигнала.

Страница справочного руководства SVR4 к функции `sigaction(2)` в [AT&T, 1990e] утверждает, что существует надежный способ поставить сигнал в очередь с помощью флага SA_SIGINFO (табл. 10.5). Это не соответствует истине. Очевидно, такая возможность была частично реализована в ядре, но она не используется в SVR4. Любопытно, что руководство SVID не делает подобных заявлений.

Что произойдет, если сразу несколько сигналов одновременно будут готовы к доставке? Стандарт POSIX.1 не определяет порядок доставки сигналов. Однако POSIX.1 Rationale предлагает в первую очередь доставлять сигналы, которые имеют отношение к текущему состоянию процесса (один из таких сигналов — SIGSEGV).

Каждый процесс имеет *маску сигналов*, с помощью которой определяется множество блокируемых сигналов. Ее можно представлять как битовую маску, в которой каждый бит соответствуетциальному сигналу. Если некоторый бит включен, доставка соответствующего ему сигнала блокируется. Процесс может проверить и изменить маску с помощью функции `sigprocmask`, которая будет описана в разделе 10.12.

Поскольку существует вероятность, что количество сигналов превысит количество битов в целочисленном типе, стандарт POSIX.1 предусматривает специальный тип данных `sigset_t` для хранения набора сигналов. В разделе 10.11 мы рассмотрим пять функций, предназначенных для работы с наборами сигналов.

10.9. Функции `kill` и `raise`

Функция `kill` посылает сигнал процессу или группе процессов. Функция `raise` позволяет процессу послать сигнал себе самому.

Изначально функция `raise` была определена стандартом ISO C. Стандарт POSIX.1 включает эту функцию, чтобы соблюсти соответствие со стандартом ISO C, однако он расширяет ее спецификацию для обеспечения работы с потоками выполнения (взаимодействие потоков с сигналами мы обсудим в разделе 12.8). Поскольку стандарт ISO C ничего не говорит о многозадачности, в нем отсутствует определение функции `kill`, которая требует передачи идентификатора процесса в качестве одного из аргументов.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Вызов

```
raise(signo);
```

эквивалентен вызову

```
kill(getpid(), signo);
```

Интерпретация аргумента *pid* функции *kill* производится в соответствии со следующими правилами.

pid > 0 Сигнал посыпается процессу с идентификатором *pid*.

pid == 0 Сигнал посыпается всем процессам с идентификатором группы процессов, равным идентификатору группы процессов посыпающего процесса, которым данный процесс имеет право посыпать сигналы. Обратите внимание, что в понятие *всем процессам* не входят системные процессы, определяемые реализацией. В большинстве версий UNIX под системными понимаются процессы ядра и процесс *init* (идентификатор процесса 1).

pid < 0 Сигнал посыпается всем процессам с идентификатором группы процессов, равным абсолютному значению *pid*, которым данный процесс имеет право посыпать сигналы. Опять же в понятие *всем процессам* не входят системные процессы, определяемые реализацией.

pid == -1 Сигнал посыпается всем процессам в системе, которым посыпающий процесс имеет право посыпать сигналы. Здесь точно так же из понятия *всем процессам* исключаются некоторые системные процессы.

Как уже упоминалось, процесс должен обладать определенными правами, чтобы посыпать сигналы другим процессам. Так, суперпользователь может послать сигнал любому процессу. В остальных случаях должно соблюдаться основное правило — реальный или эффективный идентификатор пользователя процесса, посыпающего сигнал, должен совпадать с реальным или эффективным идентификатором пользователя процесса, принимающего сигнал. Если реализация поддерживает возможность *_POSIX_SAVED_IDS* (которая ныне считается обязательной), вместо эффективного идентификатора пользователя проверяется сохраненный идентификатор пользователя. Из этого правила существует одно исключение: сигнал *SIGCONT* можно послать любому другому процессу, принадлежащему тому же сеансу.

Стандарт POSIX.1 определяет сигнал с номером 0 как пустой сигнал. Если аргумент *signo* имеет значение 0, функция *kill* выполнит обычную проверку на наличие ошибок, но сам сигнал не пошлет. Это часто используется, чтобы определить, существует ли еще некоторый процесс. Если несуществующему процессу послать

пустой сигнал, функция `kill` вернет `-1` и код ошибки `ESRCH` в переменной `errno`. Однако следует иметь в виду, что через некоторый промежуток времени идентификаторы процессов могут использоваться повторно, поэтому наличие процесса с заданным идентификатором вовсе не означает, что это тот самый процесс, который вам нужен.

Кроме того, проверка существования процесса не является атомарной операцией. К моменту, когда функция `kill` вернет управление в вызывающую программу, проверяемый процесс уже может завершиться, что сильно ограничивает область применения такого приема.

Если в результате вызова функции `kill` генерируется сигнал для вызывающего процесса и при этом сигнал не заблокирован, тогда либо сигнал с номером `signo`, либо другой ожидающий обработки сигнал будет доставлен процессу еще до того, как функция `kill` вернет управление. (В случае использования нескольких потоков выполнения возникает ряд дополнительных вариантов; за информацией обращайтесь к разделу 12.8.)

10.10. Функции `alarm` и `pause`

Функция `alarm` позволяет установить таймер, который по истечении заданного времени сгенерирует сигнал `SIGALRM`. Если этот сигнал не игнорируется и не перехватывается приложением, он вызывает завершение процесса.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Возвращает 0 или количество секунд до истечения периода времени, установленного ранее

Аргумент `seconds` определяет количество секунд, через которое должен быть сгенерирован сигнал. Следует помнить, что между моментом генерации сигнала и моментом его доставки приложению пройдет некоторое время.

В ранних версиях UNIX оговаривалось, что сигнал может быть сгенерирован чуть раньше (на секунду или менее). Стандарт POSIX.1 не допускает этого.

Каждый процесс может обладать только одним таким таймером. Если функция `alarm` вызывается до истечения таймера, установленного ранее, она возвращает количество оставшихся секунд, а ранее установленный интервал времени заменяется новым.

Если ранее установленный интервал времени еще не истек и в аргументе `seconds` передается значение 0, введенный таймер останавливается, а функция возвращает количество секунд, оставшихся до истечения таймера.

По умолчанию сигнал `SIGALRM` завершает процесс, но большинство приложений перехватывают его. Если в результате получения этого сигнала приложение должно завершить работу, оно может выполнить все необходимые заключительные

операции перед выходом. Если предполагается перехват сигнала `SIGALRM`, необходимо установить обработчик сигнала до вызова функции `alarm`. Если функцию `alarm` вызвать первой и она успеет генерировать сигнал до установки обработчика сигнала, процесс завершится.

Функция `pause` приостанавливает вызывающий процесс, пока не будет перехвачен сигнал.

```
#include <unistd.h>
int pause(void);
```

Возвращает значение `-1` с кодом ошибки `EINTR` в переменной `errno`

Функция `pause` возвращает управление, только когда отработает функция — обработчик сигнала. В этом случае она возвращает значение `-1` с кодом ошибки `EINTR` в переменной `errno`.

Пример

С помощью функций `alarm` и `pause` можно приостановить процесс на определенный промежуток времени. На первый взгляд функция `sleep1` из листинга 10.4 выполняет эту задачу, однако в ней кроется ряд ошибок, о которых мы вскоре поговорим.

Листинг 10.4. Простейшая, но не полная реализация функции `sleep`

```
#include <signal.h>
#include <unistd.h>

static void
sig_alrm(int signo)
{
    /* ничего не делаем, просто возвращаем управление */
}

unsigned int
sleep1(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(seconds);
    alarm(seconds); /* запустить таймер */
    pause();          /* следующий перехваченный сигнал возобновит работу процесса */
/*
    return(alarm(0)); /* выключить таймер и вернуть время,
                       /* оставшееся до его истечения */
}
```

Эта функция напоминает функцию `sleep`, которая будет описана в разделе 10.19, но в данной реализации кроются три проблемы.

- Если вызывающий процесс уже установил таймер, его значение будет затерто первым вызовом функции `alarm`. Мы можем исправить эту ошибку, проанализировав возвращаемое функцией значение. Если оставшееся количество секунд меньше значения аргумента `seconds`, мы должны оставить прежнее значение таймера. Если значение таймера больше значения аргумента `seconds`, после

того, как таймер сработает через заданное количество секунд, мы должны переставить его, чтобы он повторно сработал в указанное ранее время.

2. Наша функция изменяет диспозицию сигнала `SIGALRM`. Если мы предполагаем использовать функцию в других приложениях, то должны сохранять диспозицию сигнала при вызове функции и восстанавливать ее по завершении. Этую ошибку можно исправить, сохраняя возвращаемое значение функции `signal` и восстанавливая прежнюю диспозицию перед выходом.
 3. Между первым вызовом функции `alarm` и вызовом функции `pause` возникает состояние гонки за ресурсами. При значительной нагрузке на систему есть вероятность, что таймер сработает и функция-обработчик будет вызвана еще до вызова функции `pause`. В этом случаезывающий процесс навсегда приостановится в функции `pause` (если, конечно, он не перехватит какой-нибудь другой сигнал).

Ранние реализации функции `sleep` выглядели подобно нашей программе, но ошибки 1 и 2 были исправлены, как описано выше. Для исправления третьей ошибки существует два пути. Первый – использовать функцию `setjmp`, этот подход мы продемонстрируем в следующем примере. Второй – использовать функции `sigprocmask` и `sigsuspend`, которые мы рассмотрим в разделе 10.19.

Пример

В SVR2 реализация функции `sleep`, во избежание гонки за ресурсами, использовала функции `setjmp` и `longjmp` (раздел 7.10). Простейшая версия этой функции, которую мы назвали `sleep2`, приводится в листинге 10.5. (Чтобы сократить размер листинга, мы не включили устранение ошибок 1 и 2.)

Листинг 10.5. Другая (неполная) реализация функции sleep

В функции `sleep2` исключается возможность попасть в состояние гонки за ресурсами. Даже если функция `pause` никогда не будет вызвана, `sleep2` все равно вернет управление после доставки сигнала `SIGALRM`.

В данной версии существует еще одна малозаметная ошибка, которая связана с взаимодействием с другими сигналами. Если сигнал `SIGALRM` будет получен при выполнении функции — обработчика другого сигнала, вызов функции `longjmp` оборвет обработку этого сигнала. Программа из листинга 10.6 демонстрирует такое развитие событий. Цикл в обработчике сигнала `SIGINT` построен так, что в операционной системе, которую использовал автор книги, он выполняется дольше 5 секунд. Это нужно, чтобы время работы обработчика было больше, чем значение аргумента функции `sleep2`. Переменная `k` объявлена со спецификатором `volatile`, чтобы предотвратить нарушение цикла в результате оптимизации, которую выполняет компилятор.

Листинг 10.6. Вызов функции `sleep2` из программы, которая перехватывает другие сигналы

```
#include "apue.h"

unsigned int      sleep2(unsigned int);
static void      sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    unslept = sleep2(5);
    printf("функция sleep2 вернула значение: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int            i, j;
    volatile int   k;

    /*
     * Настройте параметры циклов так, чтобы они выполнялись
     * дольше 5 секунд в системе, где запускается программа.
     */
    printf("\nфункция sig_int начала обработку\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
            k += i * j;
    printf("функция sig_int закончила обработку\n");
}
```

Запустив программу из листинга 10.6 и прервав ее вводом символа прерывания, мы получили результаты:

```
$ ./a.out
^C                                введен символ прерывания
функция sig_int начала обработку
функция sleep2 вернула значение: 0
```

Как видите, вызов `longjmp` из `sleep2` оборвал работу другого обработчика сигнала (`sig_int`), не дав ему завершиться. С этим вы столкнетесь, если будете смешивать использование функции `sleep` с обработкой других сигналов (упражнение 10.3).

Цель этих двух примеров функций `sleep1` и `sleep2` в том, чтобы продемонстрировать возможные проблемы при работе с сигналами. В следующем разделе мы покажем приемы, помогающие избежать этих проблем и надежно обрабатывать сигналы, не вступая в конфликт с другими участками кода.

Пример

Часто функция `alarm` используется в паре с функцией `pause`, чтобы установить предельное время выполнения некоторых операций, которые могут блокировать процесс. Например, если выполняется операция чтения из «медленного» устройства (раздел 10.5), которая может заблокировать процесс, у нас может появиться желание ограничить время работы функции `read` некоторым промежутком времени. Программа из листинга 10.7 читает данные со стандартного ввода и выводит их на стандартный вывод, ограничивая при этом время операции чтения.

Листинг 10.7. Вызов функции `read` с тайм-аутом

```
#include "apue.h"

static void sig_alrm(int);

int
main(void)
{
    int      n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("ошибка вызова функции read");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* ничего не делать, просто прервать работу функции read */
}
```

Такая последовательность инструкций — обычное дело для приложений в UNIX, но в этой программе кроются две проблемы.

1. Программа из листинга 10.7 имеет один из недостатков, присущих программе из листинга 10.4: вероятность попасть в состояние гонки за ресурсами между первым вызовом функции `alarm` и вызовом функции `read`. Если ядро успеет

заблокировать процесс между этими двумя вызовами на больший период, чем период срабатывания таймера, процесс рискует оказаться навсегда заблокированным в функции `read`. В большинстве подобных случаев используются длительные тайм-ауты, порядка минуты или больше, но тем не менее вероятность попасть в состояние гонки за ресурсами все равно сохраняется.

2. Если системные вызовы перезапускаются автоматически, выполнение функции `read` не будет прервано после выхода из обработчика сигнала `SIGALRM`. В этом случае установка тайм-аута ничего не даст.

Здесь нам требуется прервать медленный системный вызов. Более переносимый способ будет представлен в разделе 10.14.

Пример

Давайте перепишем предыдущий пример так, чтобы он использовал функцию `longjmp`. Это позволит прервать работу медленного системного вызова в любом случае.

Листинг 10.8. Вызов функции read с тайм-аутом с использованием функции longjmp

```
#include "apue.h"
#include <setjmp.h>

static void sig_alrm(int);
static jmp_buf env_alrm;

int
main(void)
{
    int      n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");
    if (setjmp(env_alrm) != 0)
        err_quit("работа функции read прервана по тайм-ауту");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("ошибка вызова функции read");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Эта версия работает так, как мы и ожидали, независимо от того, перезапускает система прерванные системные вызовы или нет. Но не забывайте, что эта версия все еще подвержена проблеме, связанной с обработкой других сигналов.

Если требуется ограничить время выполнения операций ввода/вывода, следует использовать функцию `longjmp`, как показано выше, но при этом не забывать о возможных конфликтах с другими обработчиками сигналов. Другой способ ограничения выполнения операций по времени предоставляют функции `select` и `poll`, которые будут рассматриваться в разделах 14.4.1 и 14.4.2.

10.11. Наборы сигналов

Для представления множества сигналов нам необходим специальный тип данных — набор сигналов. Он используется такими функциями, как `sigprocmask` (описывается в следующем разделе), чтобы передать ядру набор сигналов, которые должны быть заблокированы. Как уже говорилось выше, количество различных сигналов может превышать количество битов в целочисленном типе, поэтому в большинстве случаев нельзя использовать тип `int` для представления набора сигналов, в котором каждому сигналу отводится отдельный бит. Стандарт POSIX.1 определяет для этих целей специальный тип `sigset_t`, который может хранить набор сигналов и с которым работают следующие пять функций.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

Все четыре возвращают 0 в случае успеха, -1 — в случае ошибки

```
int sigismember(const sigset_t *set, int signo);
```

Возвращает 1 (истина), 0 (ложь), -1 — в случае ошибки

Функция `sigemptyset` инициализирует пустой набор сигналов, на который указывает аргумент `set`. Функция `sigfillset` инициализирует набор сигналов, в который включены все сигналы. Все приложения должны вызывать функцию `sigemptyset` или `sigfillset` для каждого набора сигналов перед его использованием, потому что нельзя предполагать, что инициализация глобальных или статических переменных, выполняемая языком C, соответствует реализации сигналов в заданной системе.

После инициализации в набор сигналов можно добавлять или удалять из него сигналы. Добавление одного сигнала в существующий набор производится функцией `sigaddset`, а удаление сигнала из набора — функцией `sigdelset`. Все функции, которым передается набор сигналов, в виде аргумента всегда получают указатель на набор сигналов.

Реализация

Если количество сигналов в реализации меньше количества разрядов в целочисленном типе, набор сигналов может быть реализован на основе этого типа и пред-

ставлять каждый сигнал отдельным битом. Далее в этом разделе мы будем исходить из предположения, что реализация насчитывает 31 сигнал, а для представления целых чисел используется 32 разряда. Таким образом, функция `sigemptyset` обнуляет целое число, а функция `sigfillset` — взводит все биты в целом числе. Эти две функции могут быть реализованы в виде макроопределений в заголовочном файле `<signal.h>`:

```
#define sigemptyset(ptr) (*(ptr) = 0)
#define sigfillset(ptr) (*(ptr) = ~(sigset_t)0, 0)
```

Обратите внимание: поскольку функция `sigfillset` должна устанавливать все биты в наборе сигналов и возвращать значение 0, в данном определении использован оператор языка С «запятая», возвращающий в качестве общего результата значение, стоящее после запятой.

В такой реализации функция `sigaddset` включает, а функция `sigdelset` выключает один разряд в наборе. Функция `sigismember` проверяет состояние указанного разряда. Поскольку сигнал с номером 0 отсутствует, при определении номера разряда из номера сигнала вычитается 1. В листинге 10.9 показана реализация этих функций.

Листинг 10.9. Реализация функций `sigaddset`, `sigdelset` и `sigismember`

```
#include <signal.h>
#include <errno.h>

/*
 * Обычно в файле <signal.h> имеется определение константы NSIG,
 * которая учитывает сигнал с номером 0.
 */
#define SIGBAD(signo) ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    *set |= 1 << (signo - 1); /* включить бит */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    *set &= ~(1 << (signo - 1)); /* выключить бит */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
```

```

if (SIGBAD(signo)) {
    errno = EINVAL;
    return(-1);
}
return((*set & (1 << (signo - 1))) != 0);
}

```

У нас может возникнуть желание реализовать эти функции в виде коротких макроопределений в заголовочном файле `<signal.h>`, но стандарт POSIX.1 требует проверки аргумента с номером сигнала, чтобы в случае недопустимого номера устанавливалась переменная `errno`. В функции сделать это гораздо проще, чем в макроопределении.

10.12. Функция `sigprocmask`

В разделе 10.8 мы говорили, что маска сигналов процесса — это набор сигналов, доставка которых процессу в текущий момент заблокирована. Процесс может получить текущее значение маски, изменить маску или выполнить сразу обе операции с помощью следующей функции.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Прежде всего, если в аргументе `oset` передается непустой указатель, через негоозвращается текущая маска сигналов процесса.

Далее, если в аргументе `set` передается непустой указатель, то значение аргумента `how` определяет, как должна измениться маска сигналов. В табл. 10.4 приводятся возможные значения аргумента `how`. Операция `SIG_BLOCK` представляет логическую операцию включающего ИЛИ, тогда как `SIG_SETMASK` — обычное присваивание. Обратите внимание, что сигналы `SIGKILL` и `SIGSTOP` нельзя заблокировать.

Таблица 10.4. Способы изменения текущего значения маски сигналов с помощью функции `sigprocmask`

how	Описание
<code>SIG_BLOCK</code>	Новая маска сигналов является объединением текущей маски сигналов с набором, на который указывает аргумент <code>set</code> . Это означает, что аргумент <code>set</code> содержит дополнительные сигналы, которые требуется заблокировать.
<code>SIG_UNBLOCK</code>	Новая маска сигналов является пересечением текущей маски сигналов с набором, на который указывает аргумент <code>set</code> . Это означает, что аргумент <code>set</code> содержит сигналы, которые требуется разблокировать.
<code>SIG_SETMASK</code>	Новая маска сигналов в аргументе <code>set</code> замещает текущую маску сигналов.

Если в аргументе `set` передается пустой указатель, маска сигналов процесса не изменяется и значение аргумента `how` игнорируется.

После вызова функции `sigprocmask`, если имеются какие-либо разблокированные сигналы, ожидающие обработки, по меньшей мере один из них будет доставлен приложению перед возвратом из функции.

Функция `sigprocmask` определена только для однопоточных процессов. Для работы с масками сигналов в многопоточных приложениях предоставляются отдельные функции. Мы обсудим их в разделе 12.8.

Пример

В листинге 10.10 приводится функция, которая выводит имена сигналов, составляющих маску сигналов вызывающего процесса. Мы будем использовать эту функцию в листингах 10.14 и 10.15.

Листинг 10.10. Вывод маски сигналов процесса

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t      sigset;
    int          errno_save;

    errno_save = errno; /* функция может вызываться из обработчиков сигналов */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_ret("ошибка вызова функции sigprocmask");
    } else {
        printf("%s", str);
        if (sigismember(&sigset, SIGINT))
            printf(" SIGINT ");
        if (sigismember(&sigset, SIGQUIT))
            printf(" SIGQUIT ");
        if (sigismember(&sigset, SIGUSR1))
            printf(" SIGUSR1 ");
        if (sigismember(&sigset, SIGALRM))
            printf(" SIGALRM ");

        /* здесь можно продолжить список сигналов */
        printf("\n");
    }
    errno = errno_save; /* восстановить errno */
}
```

Чтобы не «раздувать» листинг, мы не выполняем проверку наличия в маске сигналов каждого сигнала из табл. 10.1 (см. упражнение 10.9).

10.13. Функция `sigpending`

Функция `sigpending` возвращает набор сигналов, доставка которых заблокирована и которые в данный момент ожидают обработки. Набор сигналов возвращается через аргумент `set`.

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Пример

Программа в листинге 10.11 демонстрирует многие из описанных выше возможностей сигналов.

Листинг 10.11. Пример работы с наборами сигналов и с функцией sigprocmask

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t      newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("невозможно перехватить сигнал SIGQUIT");

    /*
     * Заблокировать SIGQUIT и сохранить маску сигналов.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка вызова sigprocmask с аргументом SIG_BLOCK");

    sleep(5); /* здесь SIGQUIT останется в ожидании обработки */

    if (sigpending(&pendmask) < 0)
        err_sys("ошибка вызова функции sigpending");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nsignal SIGQUIT ожидает обработки\n");

    /*
     * Восстановить маску сигналов, которая разблокирует SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка вызова sigprocmask с аргументом SIG_SETMASK");
    printf("сигнал SIGQUIT разблокирован\n");

    sleep(5); /* здесь SIGQUIT завершит приложение с созданием файла core */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("перехвачен сигнал SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("невозможно переустановить диспозицию сигнала SIGQUIT");
}
```

Процесс блокирует сигнал **SIGQUIT**, предварительно сохранив текущую маску сигналов для последующего восстановления, и затем приостанавливается на 5 секунд. Любой сигнал **SIGQUIT**, сгенерированный в этот промежуток времени, заблокируется и не будет доставлен процессу, пока не окажется разблокированным. Перед последней 5-секундной приостановкой проверяется наличие ожидающего обработки сигнала **SIGQUIT**, после чего блокировка сигнала снимается.

Обратите внимание, что сначала мы сохранили маску сигналов, а затем заблокировали сигнал. Чтобы разблокировать его, мы воспользовались операцией `SIG_SETMASK`, с помощью которой восстановили прежнее значение маски сигналов. Как вариант, можно было бы использовать для разблокирования сигнала операцию `SIG_UNBLOCK`. Однако необходимо понимать, что если мы пишем функцию, которая может быть использована в других приложениях, и в этой функции мы должны заблокировать некоторый сигнал, мы не можем использовать операцию `SIG_UNBLOCK` для разблокирования сигнала. В таких случаях для восстановления первоначального значения маски следует использовать операцию `SET_SIGMASK`, потому что возможно, что перед обращением к функции вызывающая программа уже заблокировала этот сигнал. Мы увидим это на примере функции `system` в разделе 10.18.

Если сигнал `SIGQUIT` будет сгенерирован во время этой приостановки, ожидающий обработки сигнал окажется разблокированным и будет доставлен процессу перед возвратом из функции `sigprocmask`. Мы обнаружим это, так как вызов функции `printf` в обработчике сигнала произойдет раньше, чем вызов функции `printf`, следующий за вызовом `sigprocmask`.

После этого процесс приостановится еще на 5 секунд. Если в течение этого периода будет сгенерирован сигнал `SIGQUIT`, он завершит процесс, поскольку в момент перехвата сигнала мы переустановили его диспозицию в значение по умолчанию. В следующем ниже выводе символы `\^` показаны там, где мы нажимали комбинацию клавиш `Control-\` — терминальный символ завершения процесса.

```
$ ./a.out
^\
сигнал SIGQUIT ожидает обработки
перехвачен сигнал SIGQUIT
сигнал SIGQUIT разблокирован
^Quit(coredump)
$ ./a.out
^\\^\\^\\^\\^\\^\\^\\^\\^\\^\\

сигнал SIGQUIT ожидает обработки
перехвачен сигнал SIGQUIT
сигнал SIGQUIT разблокирован
^Quit(coredump)
```

сгенерировать сигнал *SIGQUIT*
(до завершения 5-секундной задержки)
по окончании задержки
в обработчике сигнала
после выхода из *sigprocmask*
повторная генерация сигнала *SIGQUIT*

сгенерировать сигнал *SIGQUIT* 10 раз
(до завершения 5-секундной задержки)

доставлен только один сигнал

повторная генерация сигнала *SIGQUIT*

Сообщение **Quit(coredump)** выводится командной оболочкой, когда она обнаруживает аварийное завершение дочернего процесса. Обратите внимание, что, запустив программу повторно, мы десять раз генерировали сигнал **SIGQUIT**, а когда разблокировали его, процессу был доставлен только один сигнал. Это говорит о том, что в данной системе сигналы не помещаются в очередь.

10.14. Функция `sigaction`

Функция `sigaction` позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Эта функция служит заменой функции `signal` из ранних версий UNIX. В конце этого раздела мы продемонстрируем реализацию функции `signal` на основе `sigaction`.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Через аргумент `signo` передается номер сигнала, диспозицию которого требуется получить или изменить. Если в аргументе `act` передается непустой указатель, диспозиция сигнала изменяется. Если в аргументе `oact` передается непустой указатель, функция возвращает предыдущее значение диспозиции сигнала, помещая его по указанному в `oact` адресу. Эта функция использует следующую структуру:

```
struct sigaction {
    void      (*sa_handler)(int); /* адрес функции — обработчика сигнала, */
                                   /* или SIG_IGN, или SIG_DFL */
    sigset_t  sa_mask;          /* дополнительные блокируемые сигналы */
    int       sa_flags;         /* флаги, табл. 10.5 */

    /* альтернативный обработчик сигнала */
    void      (*sa_sigaction)(int, siginfo_t *, void *);
};
```

Если при изменении диспозиции сигнала поле `sa_handler` содержит адрес функции-обработчика (а не константы `SIG_IGN` или `SIG_DFL`), поле `sa_mask` определяет набор сигналов, которые будут добавлены к маске сигналов процесса перед вызовом функции-обработчика. Перед возвратом из обработчика сигнала маска сигналов будет автоматически восстановлена в прежнее состояние. Таким способом можно блокировать определенные сигналы на время работы функции-обработчика. Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал также включается в маску сигналов; тем самым блокируется доставка того же самого сигнала на время выполнения обработчика. В разделе 10.8 уже говорилось, что заблокированные сигналы обычно не помещаются в очередь. Если сигнал был сгенерирован пять раз за период времени, когда он был заблокирован, функция-обработчик, как правило, вызывается только один раз.

После установки диспозиции сигнала она остается неизменной, пока явно не изменится вызовом функции `sigaction`. В отличие от ранних версий UNIX с их недостаточно надежными сигналами, стандарт POSIX.1 требует, чтобы действие сигнала оставалось неизменным, пока явно не будет изменено программой.

Поле `sa_flags` структуры `act` определяет различные параметры обработки этого сигнала. В табл. 10.5 приводится подробное описание всех возможных флагов. Если в колонке SUS стоит галочка, соответствующий флаг определен как часть

базовых спецификаций стандарта POSIX.1. Если в этой колонке стоит аббревиатура **XSI**, флаг определен как расширение XSI.

Поле `sa_sigaction` представляет альтернативную функцию обработки сигнала при использовании флага `SA_SIGINFO`. Реализации могут использовать для хранения указателей из полей `sa_handler` и `sa_sigaction` одну и ту же область памяти, поэтому приложения должны заполнять только одно из них.

Таблица 10.5. Флаги (`sa_flags`) обработки каждого сигнала

Флаг	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
<code>SA_INTERRUPT</code>			✓			Системный вызов, прерываемый сигналом, не должен перезапускаться автоматически (в XSI — по умолчанию для <code>sigaction</code>). Дополнительная информация в разделе 10.5
<code>SA_NOCLDSTOP</code>	✓	✓	✓	✓	✓	Для сигнала <code>SIGCHLD</code> — не генерировать этот сигнал при приостановке дочернего процесса. Разумеется, при завершении дочернего процесса этот сигнал все равно будет сгенерирован (но обратите внимание на флаг <code>SA_NOCLDWAIT</code> ниже). Если установлен этот флаг, сигнал <code>SIGCHLD</code> также не будет генерироваться и при возобновлении работы дочернего процесса после приостановки
<code>SA_NOCLDWAIT</code>	✓	✓	✓	✓	✓	Для сигнала <code>SIGCHLD</code> — предотвращает создание процессов-зомби по завершении дочерних процессов. Если родительский процесс впоследствии вызовет функцию <code>wait</code> , он окажется заблокированным, пока последний дочерний процесс не завершится, после чего <code>wait</code> вернет значение <code>-1</code> и код ошибки <code>ECHILD</code> в переменной <code>errno</code> (раздел 10.7)
<code>SA_NODEFER</code>	✓	✓	✓	✓	✓	Не блокировать сигнал автоматически при вызове функции-обработчика (если, конечно, сигнал не включен в маску <code>sa_mask</code>). Заметьте, что такое поведение соответствует поведению ненадежных сигналов в ранних версиях UNIX

Таблица 10.5 (окончание)

Флаг	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Описание
SA_ONSTACK	XSI	✓	✓	✓	✓	Доставлять сигнал на альтернативном стеке, если таковой был объявлен обращением к функции <code>sigaltstack(2)</code>
SA_RESETHAND	✓	✓	✓	✓	✓	На входе в функцию-обработчик установить диспозицию сигнала в значение <code>SIG_DFL</code> и сбросить флаг <code>SA_SIGINFO</code> . Обратите внимание, что такое поведение соответствует поведению не-надежных сигналов в ранних версиях UNIX. Однако диспозицию сигналов <code>SIGILL</code> и <code>SIGTRAP</code> нельзя переустановить автоматически. При наличии этого флага функция <code>sigaction</code> ведет себя, как если бы был установлен флаг <code>SA_NODEFER</code>
SA_RESTART	✓	✓	✓	✓	✓	Производить автоматический перезапуск системных вызовов, прерванных данным сигналом (раздел 10.5)
SA_SIGINFO	✓	✓	✓	✓	✓	По этому флагу в обработчик сигнала передается дополнительная информация: указатели на структуру <code>siginfo</code> и контекст процесса

Обычно прототип функции – обработчика сигнала выглядит так:

```
void handler(int signo);
```

но если установлен флаг `SA_SIGINFO`, прототип функции-обработчика выглядит иначе:

```
void handler(int signo, siginfo_t *info, void *context);
```

Структура `siginfo_t` хранит информацию о причинах появления сигнала. Пример определения структуры приводится ниже. Все POSIX-совместимые реализации должны включать в эту структуру как минимум поля `si_signo` и `si_code`. Дополнительно XSI-совместимые реализации должны включать в состав структуры следующие поля:

```
struct siginfo {
    int           si_signo; /* номер сигнала */
```

```

int      si_errno; /* если не 0, то значение errno из <errno.h> */
int      si_code;  /* дополнительная информация (зависит от сигнала) */
pid_t    si_pid;   /* идентификатор процесса-отправителя */
uid_t    si_uid;   /* реальный идентификатор пользователя */
                  /* процесса-отправителя */
void     *si_addr; /* адрес, где возникла ошибка */
int      si_status; /* код завершения или номер сигнала */
union sigval si_value; /* значение, зависящее от приложения */
/* далее могут быть определены дополнительные поля */
};


```

Объединение `sigval` содержит следующие поля:

```

int sival_int;
void *sival_ptr;

```

При доставке сигнала приложение передает целое число в поле `si_value.sival_int` или указатель в поле `si_value.sival_ptr`.

В табл. 10.6 перечислены значения поля `si_code` для различных сигналов, определяемых стандартом Single UNIX Specification. Обратите внимание, что реализации могут определять дополнительные значения.

Для сигнала `SIGCHLD` устанавливаются значения полей `si_pid`, `si_status` и `si_uid`. Для сигналов `SIGBUS`, `SIGILL`, `SIGFPE` и `SIGSEGV` в поле `si_addr` заносится адрес обнаруженной ошибки, хотя адрес может быть неточным. Поле `si_errno` содержит код ошибки, который соответствует ситуации, вызвавшей появление сигнала, хотя это во многом зависит от реализации.

В аргументе `context` обработчику сигнала передается нетипизированный указатель, который можно привести к типу `struct ucontext_t`, идентифицирующему контекст процесса в момент доставки сигнала. Эта структура содержит следующие поля:

```

ucontext_t *uc_link; /* указатель на контекст, который будет */
                     /* восстановлен при выходе из текущего контекста */
sigset_t   uc_sigmask; /* хранит сигналы, блокируемые в данном контексте */
stack_t    uc_stack;  /* стек, используемый данным контекстом */
mcontext_t uc_mcontext; /* аппаратное представление сохраненного контекста */

```

Поле `uc_stack` описывает стек, используемый текущим контекстом. Оно содержит по меньшей мере следующие поля:

```

void *ss_sp; /* указатель на дно стека */
size_t ss_size; /* размер стека */
int ss_flags; /* флаги */

```

Если реализация поддерживает расширения сигналов реального времени, установка обработчика сигнала с флагом `SA_SIGINFO` гарантирует, что сигналы будут ставиться в очередь. Для приложений реального времени зарезервирован отдельный диапазон сигналов. Через структуру `siginfo` можно передавать данные приложения при условии, что сигнал будет генерироваться функцией `sigqueue` (раздел 10.20).

Таблица 10.6. Значения кодов в структуре siginfo_t

Сигнал	Код	Значение
SIGILL	ILL_ILLOPC	Недопустимая инструкция
	ILL_ILLOPN	Недопустимый операнд
	ILL_ILLADR	Недопустимый режим адресации
	ILL_ILLTRP	Некорректная ловушка
	ILL_PRVOPC	Привилегированная операция
	ILL_PTVREG	Привилегированный регистр
	ILL_COPROC	Ошибка сопроцессора
	ILL_BADSTK	Внутренняя ошибка стека
SIGFPE	FPE_INTDIV	Деление на ноль при работе с целыми числами
	FPE_INTOVF	Переполнение при работе с целыми числами
	FPE_FLTDIV	Деление на ноль при работе с вещественными числами
	FPE_FLTOVF	Переполнение при работе с вещественными числами
	FPE_FLTUND	Нехватка значащих разрядов при работе с вещественными числами
	FPE_FLTRES	Потеря точности при работе с вещественными числами
	FPE_FLTINV	Неверная операция при работе с вещественными числами
	FPE_FLTSUB	Выход индекса за границы диапазона
SIGSEGV	SEGV_MAPPER	Адрес не соответствует объекту
	SEGV_ACCERR	Недостаточно прав для доступа к объекту
SIGBUS	BUS_ADRALN	Неправильное выравнивание адреса
	BUS_ADRERR	Несуществующий физический адрес
	BUS_OBJERR	Аппаратная ошибка, специфичная для объекта
SIGTRAP	TRAP_BRKPT	Точка останова процесса
	TRAP_TRACE	Ловушка трассировки процесса
SIGCHLD	CLD_EXITED	Дочерний процесс завершился
	CLD_KILLED	Дочерний процесс завершился аварийно (без файла core)
	CLD_DUMPED	Дочерний процесс завершился с созданием файла core
	CLD_TRAPPED	Сработала ловушка в отлаживаемом дочернем процессе
	CLD_STOPPED	Дочерний процесс приостановлен
	CLD_CONTINUED	Приостановленный дочерний процесс продолжил выполнение
Любой сигнал	SI_USER	Сигнал послан функцией <code>kill</code>
	SI_QUEUE	Сигнал послан функцией <code>sigqueue</code>
	SI_TIMER	Истекло время таймера, установленного функцией <code>timer_gettime</code>
	SI_ASYNCIO	Завершено выполнение запрошеннной асинхронной операции ввода/вывода
	SI_MESGQ	В очередь сообщений поступило новое сообщение (расширение реального времени)

Пример — функция signal

Теперь перейдем к реализации функции `signal` на основе функции `sigaction`. Такую реализацию предусматривают многие платформы (POSIX.1 Rationale утверждает, что это и было замыслом стандарта POSIX). С другой стороны, операционные системы с ограниченной совместимостью на уровне двоичных кодов могут предоставлять функцию `signal`, поддерживающую устаревшую семантику ненадежных сигналов. Если вам не требуется поддержка устаревшей семантики (например, для сохранения обратной совместимости), используйте приводимую ниже реализацию функции `signal` или непосредственно функцию `sigaction`. (Как вы уже наверняка догадались, чтобы вернуться к устаревшей семантике, нужно вызывать функцию `sigaction` с флагами `SA_RESETHAND` и `SA_NODEFER`.) Все примеры в этой книге, которые обращаются к функции `signal`, используют функцию, представленную в листинге 10.12.

Листинг 10.12. Реализация функции signal на основе функции sigaction

```
#include "apue.h"

/* Надежная версия функции signal() с использованием функции sigaction() */
/* стандарта POSIX. */
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction      act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
        act.sa_flags |= SA_RESTART;
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Обратите внимание, что для инициализации поля `sa_mask` следует использовать функцию `sigemptyset`. Нельзя гарантировать, что `act.sa_mask = 0`; сделает то же самое.

Мы преднамеренно пробуем установить флаг `SA_RESTART` для всех сигналов, кроме `SIGALRM`, что дает возможность автоматически перезапускать системные вызовы, прерванные другими сигналами. Сигнал `SIGALRM` исключен из этого списка по той причине, что нам понадобится задавать тайм-ауты для операций ввода/вывода (листинг 10.7).

В некоторых старых системах, таких как SunOS, определен флаг `SA_INTERRUPT`. В этих системах перезапуск прерванных системных вызовов производится по умолчанию, поэтому установка этого флага предотвратит автоматический перезапуск прерванных системных вызовов. Linux определяет флаг `SA_INTERRUPT`

для совместимости с приложениями, использующими его, но в ней системные вызовы не перезапускаются по умолчанию, если обработчик установлен функцией `sigaction`. Стандарт Single UNIX Specification оговаривает, что функция `sigaction` не должна перезапускать прерванные системные вызовы, если явно не указан флаг `SA_RESTART`.

Пример — функция `signal_intr`

В листинге 10.13 приводится версия функции `signal`, которая пытается предотвратить перезапуск любого прерванного системного вызова.

Листинг 10.13. Функция `signal_intr`

```
#include "apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction      act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifndef SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Чтобы повысить переносимость функции, мы используем флаг `SA_INTERRUPT` для предотвращения перезапуска прерванных системных вызовов, если он определен в системе.

10.15. Функции `sigsetjmp` и `siglongjmp`

В разделе 7.10 мы говорили о функциях `setjmp` и `longjmp`, которые используются для выполнения дальних, или нелокальных, переходов. Функция `longjmp` достаточно часто используется в обработчиках сигналов, когда нужно вернуться в главный цикл программы, не выполняя возврат из обработчика. Мы продемонстрировали это в листингах 10.5 и 10.8.

Однако использование функции `longjmp` сопряжено с одной проблемой. Когда сигнал перехвачен, перед входом в функцию обработки производится автоматическое добавление текущего сигнала к маске сигналов процесса. Это препятствует прерыванию обработчика этим же сигналом. Как вы думаете, что произойдет с маской сигналов, если выполнить дальний переход (`longjmp`) из функции-обработчика?

B FreeBSD 8.0 и Mac OS X 10.6.8 функции `setjmp` и `Longjmp` сохраняют и восстанавливают маску сигналов. Однако Linux 3.2.0 и Solaris 10 этого не делают, хотя Linux предоставляет возможность поддержки поведения BSD-систем. FreeBSD и Mac OS X предоставляют функции `_setjmp` и `_Longjmp`, которые не сохраняют и не восстанавливают маску сигналов.

Стандарт POSIX.1 не оговаривает воздействие функций `setjmp` и `longjmp` на маску сигналов — вместо этого он определяет две новые функции, `sigsetjmp` и `siglongjmp`. Для выхода из обработчика сигнала всегда должны использоваться эти две функции.

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
```

Возвращает 0, если вызвана непосредственно, и ненулевое значение, если возврат произошел в результате обращения к функции `siglongjmp`

```
void siglongjmp(sigjmp_buf env, int val);
```

Единственное их отличие от функций `setjmp` и `longjmp` заключается в том, что функция `sigsetjmp` принимает один дополнительный аргумент. Если аргумент `savemask` содержит ненулевое значение, `sigsetjmp` сохраняет текущую маску сигналов процесса в буфере `env`. При вызове `siglongjmp`, если аргумент `env` был сохранен в результате вызова `sigsetjmp` с ненулевым значением `savemask`, маска сигналов восстанавливается из сохраненного значения.

Пример

Программа в листинге 10.14 демонстрирует, как производится автоматическое включение сигнала в маску сигналов при вызове функции-обработчика. Здесь также показано использование функций `sigsetjmp` и `siglongjmp`.

Листинг 10.14. Пример работы с маской сигналов и функций `sigsetjmp` и `siglongjmp`

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>

static void                sig_usr1(int);
static void                sig_almr(int);
static sigjmp_buf          jmpbuf;
static volatile sig_atomic_t canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR1)");
    if (signal(SIGALRM, sig_almr) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGALRM)");

    pr_mask("в начале функции main: "); /* листинг 10.10 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("в конце функции main: ");
        exit(0);
    }
    canjump = 1; /* теперь можно выполнить переход */

    for ( ; ; )
```

```

        pause();
}

static void
sig_usr1(int signo)
{
    time_t starttime;
    if (canjump == 0)
        return; /* получен неожиданный сигнал, игнорировать */
    pr_mask("в начале функции sig_usr1: ");
    alarm(3); /* запланировать SIGALRM через 3 секунды */
    starttime = time(NULL);
    for ( ; ; ) /* ждать 5 секунд */
        if (time(NULL) > starttime + 5)
            break;
    pr_mask("в конце функции sig_usr1: ");
    canjump = 0;
    siglongjmp(jmpbuf, 1); /* переход в функцию main - не возврат */
}

static void
sig_alrm(int signo)
{
    pr_mask("в функции sig_alrm: ");
}

```

Эта программа демонстрирует методику, которая должна применяться всякий раз, когда планируется использовать функцию `siglongjmp` в обработчике сигнала. Здесь переменная `canjump` устанавливается в значение, отличное от нуля, только после вызова функции `sigsetjmp`. Значение этой переменной проверяется в обработчике сигнала, и функция `siglongjmp` вызывается, только когда значение переменной `canjump` отлично от нуля. Это предотвращает несвоевременный вызов обработчика сигнала, когда буфер перехода еще не подготовлен функцией `sigsetjmp`. (В этой достаточно простой программе все заканчивается практически сразу же после вызова `siglongjmp`, но в больших программах обработчик сигнала может оставаться установленным и после вызова `siglongjmp`.) Подобного рода защита обычно не требуется при использовании функции `longjmp` в обычных функциях языка C (в противоположность обработчикам сигналов). Однако учитывая, что сигнал может быть сгенерирован в любой момент, мы вынуждены предусматривать меры предосторожности в обработчике сигналов.

Здесь мы использовали тип данных `sig_atomic_t`, который определен стандартом ISO C для переменных, запись в которые не может быть прервана. Это означает, например, что переменные этого типа не должны пересекать границы страниц виртуальной памяти и доступ к ним должен осуществляться единственной машинной инструкцией. Кроме того, мы всегда используем спецификатор `volatile` с этим типом данных, поскольку доступ к переменной возможен из двух различных потоков управления — из функции `main` и из обработчика асинхронного сигнала. На рис. 10.1 приводится временная диаграмма для этой программы. Этот рисунок можно разделить на три части: левая часть соответствует функции `main`,

центральная часть – функции `sig_usr1` и правая часть – функции `sig_alm`. Пока выполнение процесса происходит в левой части, маска сигналов пуста (нет блокируемых сигналов). В центральной части в маске сигналов находится сигнал `SIGUSR1`. В правой части в маске сигналов находятся сигналы `SIGUSR1` и `SIGALRM`.

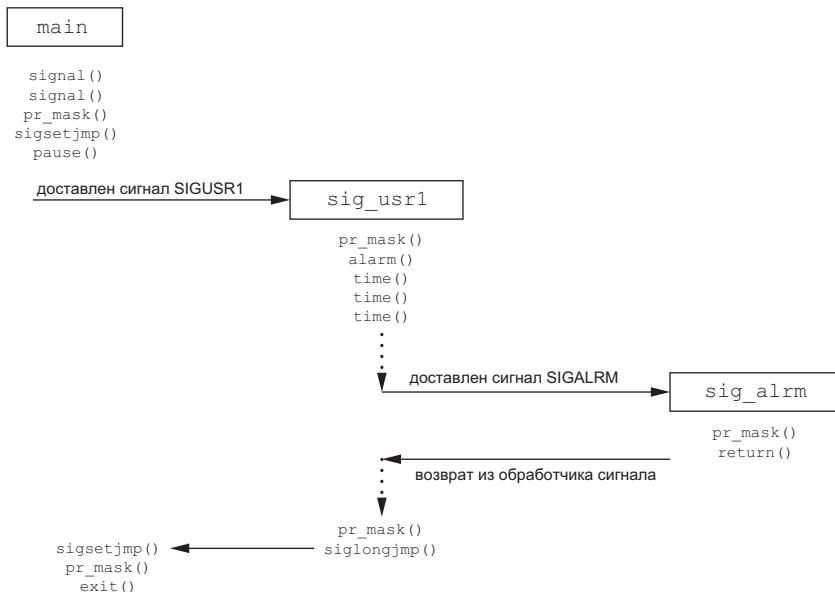


Рис. 10.1. Временная диаграмма программы, обрабатывающей два сигнала

А теперь посмотрим, что выведет программа из листинга 10.14 после запуска:

```

$ ./a.out &           запуск процесса в фоновом режиме
в начале функции main:
[1] 531                 командная оболочка вывела идентификатор процесса
$ kill -USR1 531        посыпаем процессу сигнал SIGUSR1
в начале функции sig_usr1: SIGUSR1
$ в функции sig_alm: SIGUSR1 SIGALRM
в конце функции sig_usr1: SIGUSR1
в конце функции main:   нажимаем ввод
[1] + Done      ./a.out &

```

Как мы и ожидали, на входе в обработчик сигнала перехваченный сигнал добавляется в маску сигналов процесса. После выхода из обработчика маска сигналов восстанавливается. Кроме того, функция `siglongjmp` восстанавливает маску сигналов, сохраненную вызовом функции `sigsetjmp`.

Если в программе из листинга 10.14 заменить функции `sigsetjmp` и `siglongjmp` на `setjmp` и `longjmp` в Linux (или `_setjmp` и `_longjmp` в FreeBSD), последняя строка, выведенная программой, будет такой

в конце функции main: SIGUSR1

Это означает, что после вызова `longjmp` функция `main` продолжит работу с заблокированным сигналом `SIGUSR1`, а это, скорее всего, не то, что нам нужно.

10.16. Функция `sigsuspend`

Мы рассмотрели порядок изменения маски сигналов процесса, с помощью которой можно заблокировать или разблокировать отдельные сигналы. Эту методику можно также использовать для защиты критических участков программы, выполнение которых не должно прерываться сигналами. А если понадобится разблокировать сигнал и затем с помощью функции `pause` дождаться его доставки? Если предположить, что ожидаемый сигнал — `SIGINT`, неправильная реализация могла бы выглядеть так:

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* заблокировать SIGINT и сохранить текущую маску сигналов */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("ошибка выполнения операции SIG_BLOCK");

/* критический участок программы */

/* восстановить прежнюю маску сигналов, в которой SIGINT не заблокирован */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("ошибка выполнения операции SIG_SETMASK");

/* интервал времени, когда доставка сигнала станет проблемой */
pause(); /* дождаться доставки сигнала */

/* продолжить работу */
```

Если послать сигнал процессу, когда он еще заблокирован, доставка сигнала будет отложена до тех пор, пока процесс не разблокирует его. С точки зрения приложения это выглядит так, как если бы сигнал был сгенерирован между операцией разблокирования и вызовом функции `pause` (в зависимости от реализации механизма сигналов в ядре). Если все происходит именно так или если сигнал действительно будет доставлен в промежутке между снятием блокировки и вызовом функции `pause`, возникнут определенные сложности. Сигнал, доставленный в этот промежуток, приведет к тому, что функция `pause` может заблокировать процесс «навечно», если сигнал не будет сгенерирован еще хотя бы раз. Это еще одна проблема, связанная с ранними ненадежными сигналами.

Чтобы преодолеть ее, необходим способ, с помощью которого можно было бы производить восстановление маски сигналов и приостанавливать процесс атомарно. Такую возможность дает функция `sigsuspend`.

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

Возвращает `-1` и записывает в переменную `errno` код ошибки `EINTR`

Маска сигналов, передаваемая в аргументе `sigmask`, переносится в маску сигналов процесса. Затем процесс приостанавливается, пока не будет перехвачен ожидаемый сигнал или какой-то другой сигнал не завершит процесс. Если сигнал был перехвачен и функция-обработчик вернула управление, функция `sigsuspend` также вернет управление вызывающему процессу и при этом восстановит маску сигналов процесса в состояние, предшествовавшее ее вызову.

Обратите внимание, что эта функция всегда возвращает признак ошибки с кодом `EINTR` в переменной `errno` (который говорит о том, что выполнение системного вызова было прервано сигналом).

Пример

В листинге 10.15 демонстрируется корректный способ защиты критического участка программы от конкретного сигнала.

Листинг 10.15. Защита критического участка программы от сигнала

```
#include "apue.h"

static void sig_int(int);

int
main(void)
{
    sigset_t      newmask, oldmask, waitmask;

    pr_mask("в начале программы: ");

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Заблокировать SIGINT и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");

    /*
     * Критический участок программы.
     */
    pr_mask("внутри критического участка: ");

    /*
     * Промежуток времени, когда может быть доставлен любой сигнал,
     * кроме SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("ошибка вызова функции sigsuspend");

    pr_mask("после выхода из sigsuspend: ");

    /*

```

```

 * Восстановить прежнюю маску сигналов, которая разблокирует SIGINT.
 */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("ошибка выполнения операции SIG_SETMASK");

/*
 * И продолжить работу ...
 */
pr_mask("в конце программы: ");

exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nb функции sig_int: ");
}

```

Возвращая управление, функция `sigsuspend` восстанавливает маску сигналов в состояние, предшествовавшее ее вызову. В данном примере к моменту вызова этой функции сигнал `SIGINT` был заблокирован. Поэтому мы восстанавливаем маску сигналов, записывая туда значение, сохраненное ранее (`oldmask`).

Запустив программу из листинга 10.15, мы получили

```

$ ./a.out
в начале программы:
внутри критического участка: SIGINT
^C               ввод символа прерывания
в функции sig_int: SIGUSR1
после выхода из sigsuspend: SIGINT
в конце программы:

```

Перед обращением к функции `sigsuspend` в существующую маску сигналов мы добавили сигнал `SIGUSR1`. Затем внутри обработчика маска изменилась. Далее видно, что когда `sigsuspend` возвращала управление, она восстановила маску сигналов.

Пример

Функция `sigsuspend` также используется для приостановки процесса, пока обработчик сигнала не установит глобальную переменную. Программа в листинге 10.16 перехватывает два сигнала, `SIGINT` и `SIGQUIT`, но при этом продолжение работы возможно только после перехвата сигнала `SIGQUIT`.

Листинг 10.16. Функция `sigsuspend` приостанавливает процесс, пока не будет установлена глобальная переменная

```

#include "apue.h"

volatile sig_atomic_t quitflag; /* обработчик сигнала записывает сюда */
                                /* ненулевое значение */

static void
sig_int(int signo) /* единый обработчик для SIGINT и SIGQUIT */
{
    if (signo == SIGINT)
        printf("\nпрерывание\n");

```

```

    else if (signo == SIGQUIT)
        quitflag = 1; /* установить флаг для главного цикла */
}

int
main(void)
{
    sigset_t      newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGQUIT)");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Заблокировать SIGQUIT и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /*
     * Сигнал SIGQUIT был перехвачен и к настоящему моменту
     * опять заблокирован.
     */
    quitflag = 0;

    /*
     * Восстановить маску сигналов, в которой SIGQUIT разблокирован.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");

    exit(0);
}

```

Примерный результат работы программы:

```

$ ./a.out
^C                  ввод символа прерывания
прерывание
^C                  ввод символа прерывания еще раз
прерывание
^C                  и еще раз
прерывание
^\$                 а теперь ввод символа завершения

```

Для сохранения переносимости между POSIX-совместимыми системами и системами, не-совместимыми со стандартом POSIX, но поддерживающими стандарт ISO C, необходимо только одно: внутри обработчика сигнала присвоить некоторое значение переменной типа `sig_atomic_t`. Стандарт POSIX.1 пошел дальше и определил список функций, которые можно безопасно вызывать из обработчика сигнала (см. табл. 10.3), но в этом случае программа, вероятно, не будет правильно работать в системах, не поддерживающих стандарт POSIX.

Пример

Следующий пример показывает, как с помощью сигналов синхронизировать работу родительского и дочернего процессов. В листинге 10.17 представлена реализация пяти процедур — TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT и WAIT_CHILD — из раздела 8.9.

Листинг 10.17. Процедуры для синхронизации родительского и дочернего процессов

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* устанавливается обработчиком */
                                   /* в ненулевое значение */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo) /* единый обработчик для сигналов SIGUSR1 и SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR1)");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGUSR2)");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /*
     * Заблокировать сигналы SIGUSR1 и SIGUSR2,
     * и сохранить текущую маску сигналов.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("ошибка выполнения операции SIG_BLOCK");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2); /* сообщить родительскому процессу, что мы готовы */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* ждать ответа от родительского процесса */
    sigflag = 0;

    /*
     * Восстановить маску сигналов в начальное состояние.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
```

```
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1); /* сообщить дочернему процессу, что мы готовы */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* дождаться ответа от дочернего процесса */
    sigflag = 0;

    /*
     * Восстановить маску сигналов в начальное состояние.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
}
```

В этом примере мы использовали сигналы, определяемые пользователем: сигнал `SIGUSR1` передается от родительского процесса дочернему, а `SIGUSR2` — от дочернего родительскому. В листинге 15.3 мы покажем другую реализацию этих пяти функций с использованием неименованных каналов.

Функция `sigsuspend` прекрасно подходит для случая, когда процесс должен пристановить работу, пока ему не будет доставлен сигнал (как это было в двух предыдущих примерах), но что, если нам необходимо во время ожидания сигнала обращаться к другим системным функциям? К сожалению, эта проблема не имеет достаточно надежного решения, за исключением выполнения приложения в нескольких потоках, из которых один выделяется специально для обработки сигналов, о чем мы будем говорить в разделе 12.8.

Если отказаться от многопоточной модели, лучшее, что можно предложить, — это записывать определенные значения в глобальные переменные во время обработки сигналов. Например, если мы выполняем перехват сигналов `SIGINT` и `SIGALRM` и устанавливаем обработчики сигналов с помощью функции `signal_intr`, доставка сигналов будет прерывать любые медленные системные вызовы. Чаще всего эти сигналы будут доставляться во время работы функции `read`, ожидающей окончания операции ввода с медленного устройства. (Особенно это относится к сигналу `SIGALRM`, который используется для прерывания затянувшихся операций ввода/вывода.) Вот как мог бы выглядеть код, обслуживающий такую ситуацию:

```
if (intr_flag) /* флаг устанавливается обработчиком сигнала SIGINT */
    handle_intr();
if (alrm_flag) /* флаг устанавливается обработчиком сигнала SIGALRM */
    handle_alrm();

/* сигналы, появившиеся в этот момент времени, будут утеряны */

int n;
while ((n = read(...)) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
```

```

        handle_alarm();
    else if (intr_flag)
        handle_intr();
} else {
    /* обработка других ошибок */
}
}

if (n == 0) {
    /* конец файла */
} else {
    /* обработать прочитанные данные */
}

```

Мы проверяем значения каждой глобальной переменной перед вызовом `read` и всякий раз после того, как она возвращает ошибку прерывания системного вызова. Проблема возникает, когда происходит перехват сигнала между первыми двумя условными операторами и последующим вызовом функции `read`. Сигналы, доставленные в этом промежутке времени, будут утеряны, что отмечено в комментарии. Обработчики сигналов, разумеется, будут вызваны, и они установят соответствующие глобальные переменные, но `read` никогда не вернет управление (если, конечно, какие-либо данные не будут готовы для чтения).

Нам требуется выполнить следующую последовательность действий.

1. Заблокировать сигналы `SIGINT` и `SIGALRM`.
2. Проверить значения двух глобальных переменных, чтобы проверить, был ли доставлен какой-либо сигнал, и при необходимости выполнить соответствующие действия.
3. Вызвать `read` (или любой другой системный вызов) и разблокировать оба сигнала атомарно.

Функция `sigsuspend` может помочь, только если на шаге 3 используется операция `pause`.

10.17. Функция `abort`

Ранее упоминалось, что вызов функции `abort` приводит к аварийному завершению процесса.

```
#include <stdlib.h>

void abort(void);
```

Эта функция никогда не возвращает управление

Эта функция посыпает сигнал `SIGABRT` вызывающему процессу. (Процессы не должны игнорировать этот сигнал.) Стандарт ISO C определяет, что функция `abort` должна извещать операционную систему об аварийном завершении с помощью вызова `raise(SIGABRT)`.

Стандарт ISO C требует, чтобы функция `abort` никогда не возвращала управление, даже когда приложение перехватывает сигнал `SIGABRT`. Если сигнал перехватывается, единственный способ для обработчика не вернуть управление в вызывающий процесс — вызвать одну из функций: `exit`, `_exit`, `_Exit`, `longjmp` или `siglongjmp`. (Различия между `longjmp` и `siglongjmp` обсуждались в разделе 10.15.) Помимо этого, стандарт POSIX.1 указывает, что функция `abort` должна выполняться даже в том случае, если процесс заблокировал или игнорирует сигнал `SIGABRT`.

Процессу позволено перехватывать сигнал `SIGABRT`, чтобы он мог выполнить необходимые действия перед завершением. Если процесс не завершается из обработчика сигнала, стандарт POSIX.1 указывает, что функция `abort` должна завершить процесс, когда обработчик сигнала вернет управление.

Стандарт ISO C оставляет на усмотрение реализации решение вопроса о сбросе буферов ввода/вывода и удалении временных файлов (раздел 5.13). Стандарт POSIX.1 пошел гораздо дальше и требует, чтобы функция `abort`, если она завершает процесс, воздействовала на открытые потоки ввода/вывода так же, как функция `fclose`.

В ранних версиях System V функция `abort` генерировала сигнал `SIGIOT`. Кроме того, допускалась возможность игнорировать или перехватывать сигнал. В последнем случае, если обработчик возвращал управление как обычно, то и функция `abort` возвращала управление вызывающему процессу.

В 4.3BSD генерировался сигнал `SIGILL`, но перед этим функция `abort` снимала блокировку с сигнала и сбрасывала его диспозицию в значение `SIG_DFL` (завершение с созданием файла `core`). Это не позволяло процессам игнорировать сигнал или перехватывать его.

Традиционно различные реализации функции `abort` по-разному обслуживали потоки ввода/вывода. Если необходимо, чтобы перед аварийным завершением все потоки ввода/вывода сбрасывались должным образом, это нужно сделать перед вызовом функции `abort`. Именно так делает наша функция `err_dump` (приложение B).

Поскольку в большинстве реализаций UNIX функция `tmpfile` сразу же вызывает `unlink`, проблема удаления временных файлов, о которой предупреждает стандарт ISO C, отпадает сама собой.

Пример

В листинге 10.18 приводится реализация функции `abort`, соответствующая требованиям стандарта POSIX.1.

Листинг 10.18. Реализация функции `abort`, соответствующая требованиям стандарта POSIX.1

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void) /* функция abort() в стиле POSIX */
{
```

```

sigset_t          mask;
struct sigaction  action;

/*
 * Вызывающий процесс не может игнорировать SIGABRT,
 * иначе – сбросить диспозицию сигнала в значение по умолчанию.
 */
sigaction(SIGABRT, NULL, &action);
if (action.sa_handler == SIG_IGN) {
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);
}
if (action.sa_handler == SIG_DFL)
    fflush(NULL); /* сбросить все буферы потоков ввода-вывода */

/*
 * Вызывающий процесс не может заблокировать SIGABRT;
 * убедитесь, что он не заблокирован.
 */
sigfillset(&mask);
sigdelset(&mask, SIGABRT);      /* в маске разблокирован только SIGABRT */
sigprocmask(SIG_SETMASK, &mask, NULL);
kill(getpid(), SIGABRT);        /* послать сигнал */

/*
 * Если мы вернулись сюда, значит, процесс обработал SIGABRT
 * и вернул управление.
 */
fflush(NULL);                  /* сбросить все буферы */
action.sa_handler = SIG_DFL;
sigaction(SIGABRT, &action, NULL); /* установить диспозицию сигнала */
                                    /* в значение по умолчанию*/
sigprocmask(SIG_SETMASK, &mask, NULL); /* на всякий случай ... */
kill(getpid(), SIGABRT);           /* и еще раз */
exit(1);                         /* этот вызов никогда не выполнится... */
}

```

Прежде всего мы проверяем, будет ли выполнено действие по умолчанию для сигнала, — если это так, сбрасываем все буферы стандартных потоков ввода/вывода. Это не равносильно вызову функции `fclose` (поскольку мы лишь сбрасываем буфера, а не закрываем потоки), но система сама закроет все открытые файлы, когда процесс завершится. Если процесс перехватил сигнал и вернул управление, мы опять сбрасываем все буферы ввода/вывода, поскольку процесс мог выводить некоторые данные в обработчике сигнала. Единственное, чего мы не сможем, — вызвать функцию `_exit` или `_Exit` из обработчика. В этом случае все данные, оставшиеся в буферах ввода/вывода, будут потеряны. Но мы будем исходить из предположения, что вызывающий процесс просто не пожелал сбрасывать содержимое буферов.

В разделе 10.9 мы говорили, что если вызов функции `kill` генерирует сигнал для вызывающего процесса и этот сигнал не заблокирован (что гарантирует функция из листинга 10.18), этот сигнал (или любой другой незаблокированный сигнал, ожидающий обработки) будет доставлен процессу еще до выхода из функции `kill`. В данном случае мы блокируем доставку всех сигналов, за исключением `SIGABRT`, поэтому мы наверняка знаем, что если вызов `kill` вернул управление, значит, сигнал был перехвачен и обработан процессом.

10.18. Функция system

В разделе 8.13 мы приводили пример реализации функции `system`. Однако эта версия не обрабатывала сигналы. Стандарт POSIX.1 требует, чтобы функция `system` игнорировала сигналы `SIGINT` и `SIGQUIT` и блокировала сигнал `SIGCHLD`. Прежде чем продемонстрировать версию, которая обрабатывает сигналы, мы расскажем, почему это необходимо.

Пример

Программа в листинге 10.19 использует версию функции `system` из раздела 8.13 для вызова редактора `ed(1)`. (Этот редактор уже давно входит в состав UNIX. Мы использовали его потому, что он перехватывает и обрабатывает сигналы `SIGINT` и `SIGQUIT`. Если запустить редактор `ed` из командной оболочки и ввести символ прерывания, он перехватит его и выведет символ «?». Кроме того, программа `ed` устанавливает диспозицию сигнала `SIGQUIT` в значение `SIG_IGN`.) Программа из листинга 10.19 перехватывает сигналы `SIGINT` и `SIGCHLD`. Запустив ее, мы получим следующее:

```
$ ./a.out
a                                включить режим добавления текста в буфер редактора
Это одна строка текста
.
1,$p                            точка на отдельной строке выключает режим добавления
                                 вывести строки из буфера с первой по последнюю,
                                 чтобы увидеть его содержимое
Это одна строка текста
w temp.foo                         записать буфер в файл
23                               редактор сообщает, что записано 23 байта
q                                выйти из редактора
перехвачен сигнал SIGCHLD
```

Когда редактор завершает работу, система посыпает родительскому процессу (`a.out`) сигнал `SIGCHLD`. Мы перехватываем его и возвращаем управление из обработчика сигнала. Родительский процесс должен это делать, если желает знать, когда завершился дочерний процесс. Доставка этого сигнала родительскому процессу должна быть заблокирована на время работы функции `system`, как того требует стандарт POSIX.1. Иначе процесс, запустивший функцию `system`, будет думать, что завершился один из его собственных дочерних процессов. После доставки сигнала вызывающий процесс должен обратиться к одной из функций семейства `wait`, чтобы получить код завершения дочернего процесса.

Листинг 10.19. Вызов редактора ed с помощью функции system

```
#include "apue.h"

static void
sig_int(int signo)
{
    printf("перехвачен сигнал SIGINT\n");
}

static void
sig_chld(int signo)
```

```
{
    printf("перехвачен сигнал SIGCHLD\n");
}

int
main(void)
{
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGCHLD)");
    if (system("/bin/ed") < 0)
        err_sys("ошибка вызова функции system()");
    exit(0);
}
```

Если запустить программу еще раз и отправить ей сигнал SIGINT, мы получим следующий результат:

```
$ ./a.out
a
привет, мир
.
1,$p
привет, мир
w temp.foo
12
^C
?
перехвачен сигнал SIGINT
q
перехвачен сигнал SIGCHLD
```

включить режим добавления текста в буфер редактора

точка на отдельной строке выключает режим добавления

вывести строки из буфера с первой по последнюю,

чтобы увидеть его содержимое

записать содержимое буфера в файл

редактор сообщает, что записано 12 байт

ввод символа прерывания

редактор перехватил сигнал и вывел знак вопроса

и то же самое сделал родительский процесс

выход из редактора

В разделе 9.6 уже говорилось, что ввод символа прерывания приводит к посыпке сигнала SIGINT всем процессам из группы процессов переднего плана. На рис. 10.2 показана схема процессов после запуска редактора.

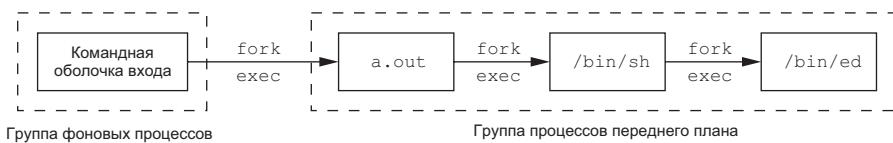


Рис. 10.2. Группы процессов переднего плана и фонового режима

В данном примере сигнал SIGINT посыпается всем трем процессам переднего плана. (Командная оболочка игнорирует его.) Как видно из вывода программы, оба процесса, *a.out* и редактор, перехватывают сигнал. Но когда мы запускаем программу с помощью функции *system*, у нас не должно получаться так, что и родительский и дочерний процессы перехватывают сигналы SIGINT и SIGQUIT, сгенерированные терминалом. В действительности эти сигналы должны посыпаться только запущенной программе — дочернему процессу. Программа, запускаемая функцией *system*, может быть интерактивной (как программа *ed* в этом примере),

а процесс, вызывающий функцию `system`, отдает управление другой программе, ожидая ее завершения, поэтому он не должен принимать эти два сигнала, генерируемые терминалом. По этой причине стандарт POSIX.1 требует, чтобы функция `system` игнорировала эти два сигнала, пока она ожидает завершения команды.

Пример

В листинге 10.20 приводится реализация функции `system`, которая предусматривает необходимую обработку сигналов.

Листинг 10.20. Корректная реализация функции `system`, соответствующая

стандарту POSIX.1

```
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

int
system(const char *cmdstring) /* предусматривает обработку сигналов */
{
    pid_t          pid;
    int            status;
    struct sigaction  ignore, saveintr, savequit;
    sigset(SIGCHLD, chldmask, savemask);

    if (cmdstring == NULL)
        return(1); /* UNIX всегда поддерживает командный процессор */

    ignore.sa_handler = SIG_IGN; /* игнорировать SIGINT и SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;

    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);      /* заблокировать SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1; /* вероятно, превышено максимальное */
                      /* количество процессов */
    } else if (pid == 0) { /* дочерний процесс */
        /* восстановить предыдущие действия сигналов и сбросить маску */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);           /* ошибка вызова функции exec */
    } else {               /* родительский процесс */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* получен код ошибки, отличный от EINTR */
                break;
            }
    }
}
```

```

        }

/* восстановить предыдущие действия сигналов и сбросить маску */
if (sigaction(SIGINT, &saveintr, NULL) < 0)
    return(-1);
if (sigaction(SIGQUIT, &savequit, NULL) < 0)
    return(-1);
if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
    return(-1);

return(status);
}

```

Если собрать программу из листинга 10.19 с этой версией функции `system`, работа программы претерпит следующие изменения.

1. Вызывающий процесс не будет получать сигналы `SIGINT` и `SIGQUIT`, сгенерированные терминалом.
2. По завершении редактора сигнал `SIGCHLD` не будет послан вызывающему процессу. Вместо этого он заблокируется, пока не будет разблокирован последним вызовом функции `sigprocmask` уже после того, как функция `waitpid` получит код завершения дочернего процесса.

Стандарт POSIX.1 указывает, что если функция `wait` или `waitpid` получает код завершения дочернего процесса, когда сигнал `SIGCHLD` находится в ожидании обработки, то сигнал `SIGCHLD` не должен доставляться процессу, если не существует неполученного кода завершения другого дочернего процесса. FreeBSD 8.0, Mac OS X 10.6.8 и Solaris 10 реализуют эту семантику, а Linux 3.2.0 — нет. Вместо этого сигнал `SIGCHLD` остается ждать обработки после того, как функция `system` вызовет `waitpid`. Когда блокировка сигнала снимается, он доставляется вызывающему процессу. Вызвав функцию `wait` в функции `sig_chld` из листинга 10.19, мы получили бы признак ошибки с кодом `ECHILD`, поскольку код завершения дочернего процесса уже был получен функцией `system`.

Во многих устаревших руководствах в качестве примера, как игнорировать сигналы `SIGINT` и `SIGQUIT`, приводится следующий код:

```

if ((pid = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) {
    /* дочерний процесс */
    execl(...);
    _exit(127);
}

/* родительский процесс */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);

```

Проблема этого кода заключается в том, что нельзя заранее точно сказать, какой из процессов первым получит управление после вызова функции `fork` — родительский или дочерний. Если первым начнет работу дочерний процесс, может

получиться так, что сигнал будет сгенерирован еще до того, как родительский процесс получит возможность изменить его диспозицию. По этой причине в листинге 10.20 мы изменяем диспозиции сигналов еще до вызова `fork`.

Обратите внимание, что диспозиции этих сигналов в дочернем процессе необходимо переустановить до вызова `exec1`. Это позволит функции `exec1` изменить их диспозиции на значения по умолчанию на основе диспозиций сигналов вызывающего процесса, как это было описано в разделе 8.10.

Возвращаемое значение функции system

Будьте осторожны с возвращаемым значением функции `system`. Это код завершения командной оболочки, который не всегда совпадает с кодом завершения самой команды. В листинге 8.13 мы видели ряд примеров, когда результаты оказывались вполне ожидаемыми: если выполнялась простая команда, такая как `date`, код завершения был равен 0. Команда `exit 44` дала код завершения — 44. А что случится, если команда во время выполнения получит сигнал?

Запустим программу из листинга 8.14 и попробуем посыпать сигналы выполняемым командам:

```
$ tsys "sleep 30"
^Снормальное завершение, код выхода = 130      мы нажали клавишу
                                                       прерывания (Control-C)
$ tsys "sleep 30"
^sh: 946 Quit                                мы нажали клавишу завершения
нормальное завершение, код выхода = 131
```

Когда мы прервали команду `sleep` сигналом `SIGINT`, функция `pr_exit` (листинг 8.3) восприняла это как нормальное завершение. То же произошло, когда мы прервали команду `sleep` сигналом `SIGQUIT`. Дело в том, что командная оболочка Bourne shell имеет плохо документированную особенность — она возвращает 128 плюс номер сигнала, если работа команды была прервана сигналом. Это можно наблюдать и в интерактивном сеансе работы с командной оболочкой.

```
$ sh                                     убедимся, что запущена Bourne shell
$ sh -c "sleep 30"                         нажали клавишу прерывания
^C                                         вывести код завершения последней команды
$ echo $?                                 нажали клавишу завершения
130                                         вывести код завершения последней команды
$ sh -c "sleep 30"                         нажали клавишу завершения
^sh: 962 Quit - core dumped               вывести код завершения последней команды
$ echo $?                                 выйти из Bourne shell
131
```

В системе, где сигнал `SIGINT` имеет номер 2, а сигнал `SIGQUIT` — номер 3, мы получили коды завершения 130 и 131 соответственно.

Теперь сделаем то же самое, но на этот раз пошлем сигналы самой командной оболочки и посмотрим, что возвращает функция `system`:

```
$ tsys "sleep 30" &                     на этот раз запустим в фоновом режиме
9257                                         посмотрим идентификаторы процессов
$ ps -f
```

```

UID  PID  PPID  TTY      TIME CMD
sar  9260  949   pts/5    0:00 ps -f
sar  9258  9257  pts/5    0:00 sh -c sleep 30
sar  949   947   pts/5    0:01 /bin/sh
sar  9257  949   pts/5    0:00 tsys sleep 30
sar  9259  9258  pts/5    0:00 sleep 30
$ kill -KILL 9258          завершим саму командную оболочку
аварийное завершение, номер сигнала = 9

```

Значение, возвращаемое функцией `system`, свидетельствует об аварийном завершении, только когда сама командная оболочка завершается аварийно.

Другие оболочки ведут себя иначе при обработке сигналов, вызывающих завершение, таких как `SIGINT` и `SIGQUIT`. В оболочках `bash` и `dash`, например, нажатие клавиши прерывания или завершения приведет к возврату кода завершения, свидетельствующего об аварийном завершении с соответствующим номером сигнала. Однако если послать сигнал процессу непосредственно, когда он ожидает в вызове функции `sleep`, чтобы сигнал был доставлен отдельному процессу, а не всей группе процессов переднего плана, обнаружится, что эти оболочки действуют подобно `Bourne` и завершаются с нормальным кодом завершения – 128 плюс номер сигнала.

При использовании функции `system` в своих программах уделите особое внимание правильной интерпретации возвращаемого значения. Если вызывать непосредственно `fork`, `exec` и `wait`, код завершения дочернего процесса не будет соответствовать возвращаемому значению функции `system`.

10.19. Функции `sleep`, `nanosleep` и `clock_nanosleep`

Мы уже пользовались функцией `sleep` во многих примерах и даже продемонстрировали две ее реализации в листингах 10.4 и 10.5; впрочем, у них есть определенные недостатки.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Возвращает 0 или количество секунд, оставшихся до окончания приостановки

Эта функция приостанавливает выполнение вызывающего процесса, пока:

1. Не истечет установленный интервал времени.
2. Не будет получен сигнал и обработчик сигнала не вернет управление.

Как и в случае с функцией `alarm`, функция `sleep` может вернуть управление чуть позже, чем было запрошено, в зависимости от загруженности системы.

В первом случае функция возвращает 0. Если выход из функции происходит раньше из-за того, что процессу был доставлен сигнал (второй случай), возвращаемое

значение содержит количество секунд, оставшихся до истечения запрошенного интервала времени (заданное количество секунд минус количество секунд, прошедших с момента вызова функции).

Функция `sleep` может быть реализована на базе функции `alarm` (раздел 10.10), но это совсем не обязательно. Однако если в основе реализации функции `sleep` лежит функция `alarm`, могут возникнуть взаимовлияния этих двух функций. Стандарт POSIX.1 никак не оговаривает возможность взаимного влияния. Например, что произойдет, если сначала вызвать `alarm(10)`, а затем, спустя 3 секунды, вызвать `sleep(5)`? Функция `sleep` вернет управление через 5 секунд (разумеется, если процессом не был перехвачен какой-либо сигнал), но будет ли генерирован сигнал `SIGALRM` через 2 секунды после этого? Решение этих вопросов остается за реализацией.

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 реализуют функцию sleep на основе функции nanosleep, устраняющей зависимость от сигналов и таймера, вводимого функцией alarm. Для сохранения переносимости приложений не следует делать какие-либо предположения о реализации функции sleep, но если понадобится смешивать вызовы функции sleep с любыми другими функциями, отмеряющими интервалы времени, вам придется побеспокоиться о возможных взаимовлияниях этих функций.

Пример

В листинге 10.21 показана реализация POSIX.1-совместимой функции `sleep`. Эта функция является модификацией функции из листинга 10.4, она надежно обслуживает сигналы и избегает состояния гонки за ресурсами, которое наблюдалось в предыдущей реализации. Однако она по-прежнему не учитывает, что функция `alarm` могла предварительно установить таймер. (Как уже упоминалось, стандарт POSIX.1 не оговаривает явно возможность взаимного влияния этих двух функций.)

Листинг 10.21. Надежная реализация функции sleep

```
#include "apue.h"

static void
sig_alarm(int signo)
{
    /*
     * ничего не делаем, просто возвращаем управление,
     * чтобы выйти из функции sigsuspend()
     */
}

unsigned int
sleep(unsigned int seconds)
{
    struct sigaction      newact, oldact;
    sigset_t              newmask, oldmask, suspmask;
    unsigned int           unslept;

    /* установить свой обработчик, сохранив предыдущую информацию */
    newact.sa_handler = sig_alarm;
    sigemptyset(&newact.sa_mask);
```

```

newact.sa_flags = 0;
sigaction(SIGALRM, &newact, &oldact);

/* заблокировать сигнал SIGALRM и сохранить текущую маску сигналов */
sigemptyset(&newmask);
sigaddset(&newmask, SIGALRM);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

alarm(seconds);
suspmask = oldmask;

/* убедиться, что SIGALRM не заблокирован */
sigdelset(&suspmask, SIGALRM);

/* дождаться, пока не будет перехвачен какой-либо сигнал */
sigsuspend(&suspmask);

/* был перехвачен некоторый сигнал, сейчас SIGALRM заблокирован */
unslept = alarm(0);

/* восстановить предыдущее действие */
sigaction(SIGALRM, &oldact, NULL); /* восстановить предыдущее действие */

/* восстановить маску сигналов, в которой сигнал SIGALRM разблокирован */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
return(unslept);
}

```

Чтобы создать более надежную реализацию, потребовался больший объем кода, чем в листинге 10.4. Мы не используем функции дальних переходов (как это делалось в листинге 10.5, чтобы избежать гонки за ресурсами) и поэтому не оказываем влияния на другие обработчики сигналов, которые могли выполняться в момент доставки сигнала SIGALRM.

Функция `nanosleep` похожа на функцию `sleep`, но обеспечивает точность измерения времени до наносекунд.

```
#include <time.h>

int nanosleep(const struct timespec *reqtp, struct timespec *remtp);
```

Возвращает 0, если установленное время истекло,
или -1 — в случае ошибки

Эта функция приостанавливает вызывающий процесс, пока не истечет установленный интервал времени или выполнение функции не будет прервано сигналом. Параметр `reqtp` определяет интервал времени в секундах и наносекундах. Если выполнение функции будет прервано сигналом и процесс при этом не завершится, в структуре `timespec`, на которую указывает параметр `remtp`, будет возвращено время, оставшееся до окончания приостановки. В этом параметре можно передать `NULL`, если оставшееся время не представляет интереса.

Если система не поддерживает точность измерения времени до наносекунд, запрошенный интервал будет округлен. Поскольку функция `nanosleep` не опирается

ся на механизм сигналов, ее можно смело использовать, не беспокоясь о возможном влиянии других функций.

Функция nanosleep прежде определялась расширением Timers стандарта Single UNIX Specification, но в версии SUSv4 была перемещена в раздел базовых спецификаций.

С введением поддержки множества системных часов (раздел 6.10) возникла необходимость иметь возможность приостанавливать вызывающий поток выполнения, определяя интервал времени относительно определенных часов. Такую возможность дает функция `clock_nanosleep`.

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                     const struct timespec *reqtp, struct timespec *remtp);
```

Возвращает 0, если установленное время истекло, или код ошибки — в случае неудачи

Аргумент `clock_id` определяет часы, которые будут отмерять интервал приостановки. Идентификаторы часов перечислены в табл. 6.7. Аргумент `flags` определяет тип задержки — относительная или абсолютная. Если в аргументе `flags` передать 0, время приостановки будет интерпретироваться как относительное (то есть продолжительность приостановки). Если передать значение `TIMER_ABSTIME`, время приостановки будет интерпретироваться как абсолютное (то есть поток выполнения будет приостановлен до достижения указанного момента времени).

Другие аргументы, `reqtp` и `remtp`, имеют то же назначение, что и в функции `nanosleep`. Однако когда определяется абсолютное время, аргумент `remtp` не используется, так как в этом нет необходимости; мы можем вызывать `clock_nanosleep` с одним и тем же значением `reqtp`, пока указанные часы не достигнут установленного абсолютного значения времени.

Обратите внимание, что, кроме возвращаемого кода ошибки, вызов

```
clock_nanosleep(CLOCK_REALTIME, 0, reqtp, remtp);
```

эквивалентен вызову

```
nanosleep(reqtp, remtp);
```

Проблема с относительными интервалами времени состоит в том, что некоторым приложениям требуется очень высокая точность измерения интервала приостановки, а при использовании относительного интервала процесс может быть приостановлен на более длительное время. Например, если приложению требуется выполнять некоторую операцию через регулярные интервалы времени, в нем следует получить текущее время, прибавить продолжительность интервала до следующего запуска операции и затем вызвать `nanosleep`. Между моментом получения текущего времени и вызовом `nanosleep` процесс может быть приостановлен планировщиком, чтобы выделить квант времени другому процессу, что, в свою очередь, увеличит продолжительность приостановки при использовании относи-

тельного интервала. Использование абсолютного времени увеличивает точность, даже при том, что планировщик процессов не дает гарантии, что процесс возобновит работу сразу после окончания установленного интервала.

В прежних версиях стандарта Single UNIX Specification функция `clock_nanosleep` определялась расширением `Clock`, но в версии SUSv4 была перемещена в раздел базовых спецификаций.

10.20. Функция `sigqueue`

В разделе 10.8 говорилось, что большинство реализаций UNIX не ставят сигналы в очередь. Однако, следуя требованиям расширений реального времени стандарта POSIX.1, некоторые системы стали добавлять поддержку очередей сигналов. В версии SUSv4 поддержка очередей сигналов была перемещена из расширений реального времени в раздел базовых спецификаций.

Вообще, сигнал несет единственный бит информации: сам факт сигнала. Помимо поддержки очередей сигналов, эти расширения позволяют приложениям передавать вместе с сигналами дополнительную информацию (вспомните раздел 10.14). Эта информация встраивается в структуру `siginfo`. Наряду с системной информацией приложения могут передать целое число или указатель на буфер с дополнительной информацией для обработчика сигнала.

Чтобы задействовать очереди сигналов, необходимо выполнить следующее.

1. Указать флаг `SA_SIGINFO` в ходе установки обработчика сигналов вызовом `sigaction`. Если этого не сделать, сигнал будет послан, но при этом реализация сама решит, следует ли поместить его в очередь.
2. Передать обработчик сигнала в поле `sa_sigaction` структуры `sigaction` вместо поля `sa_handler`. Реализации могут позволять использовать поле `sa_handler`, но в этом случае обработчик не сможет получать дополнительную информацию, отправляемую с помощью функции `sigqueue`.
3. Посыпать сигналы вызовом функции `sigqueue`.

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value)
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Функция `sigqueue` напоминает функцию `kill`, кроме того, что позволяет посыпать сигналы только одному процессу и использовать аргумент `value` для передачи обработчику сигнала целого числа или указателя на буфер.

Сигналы не могут добавляться в очередь до бесконечности. Вспомните предел `SIGQUEUE_MAX` из табл. 2.9 и 2.11. По достижении этого предела `sigqueue` может терпеть неудачу, возвращая признак ошибки с кодом `EAGAIN`.

В расширениях реального времени вводится отдельный диапазон сигналов для использования в приложениях: от **SIGRTMIN** по **SIGRTMAX** включительно. Не забывайте, что по умолчанию эти сигналы приводят к завершению процесса.

В табл. 10.7 перечислены отличия поддержки очередей сигналов между реализациями, описываемыми в этой книге.

Таблица 10.7. Поведение поддержки очередей сигналов в различных платформах

Поведение	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Поддержка <code>sigqueue</code>	✓	✓	✓		✓
Возможность помещения в очередь других сигналов, не из диапазона <code>SIGRTMIN .. SIGRTMAX</code>	Не обязательно	✓			✓
Сигналы помещаются в очередь, даже без использования флага <code>SA_SIGINFO</code>	Не обязательно	✓	✓		

ОС Mac OS X 10.6.8 не поддерживает ни функцию `sigqueue`, ни сигналы реального времени. В Solaris 10 функция `sigqueue` находится в библиотеке реального времени `librt`.

10.21. Сигналы управления заданиями

Шесть сигналов из перечисленных в табл. 10.1 стандарт POSIX.1 рассматривает как сигналы управления заданиями.

SIGCHLD Дочерний процесс приостановлен или завершен.

SIGCONT Возобновление работы приостановленного процесса.

SIGSTOP Сигнал останова (не может быть проигнорирован).

SIGTSTP Интерактивный сигнал приостановки.

SIGTTIN Чтение из управляющего терминала процессом из группы процессов фонового режима.

SIGTTOU Запись в управляющий терминал процессом из группы процессов фонового режима.

Большинство программ не обрабатывают эти сигналы, за исключением сигнала **SIGCHLD**. Обычно все необходимые действия по их обработке принимают на себя интерактивные командные оболочки. При вводе символа приостановки (обычно **Control-Z**) всем процессам переднего плана передается сигнал **SIGTSTP**. Когда мы даем команду возобновить работу фонового задания или задания переднего плана, командная оболочка посыпает всем процессам в задании сигнал **SIGCONT**. Аналогично, когда процесс получает сигнал **SIGTTIN** или **SIGTTOU**, по умолчанию он приостанавливается, а командная оболочка, распознав эту ситуацию, уведомляет нас о ней.

Исключение составляют процессы, которые управляют терминалом, например редактор **vi(1)**. Такие программы должны знать, когда пользователь хочет при-

остановить их работу, чтобы восстановить состояние терминала, предшествовавшее запуску программы. Кроме того, программы, подобные редактору `vi`, при возобновлении работы должны надлежащим образом перенастроить терминал и перерисовать экран. Позднее мы увидим на примере, как программа, подобная `vi`, выполняет все необходимые действия.

Сигналы управления заданиями оказывают определенное влияние друг на друга. Когда генерируется любой из четырех сигналов, вызывающих приостановку процесса (`SIGSTOP`, `SIGTSTP`, `SIGTTIN` или `SIGTTOU`), система отменяет ожидающий обработки сигнал `SIGCONT` для этого же процесса. Аналогично, когда генерируется сигнал `SIGCONT`, система отменяет все ожидающие обработки сигналы приостановки.

Обратите внимание, что по умолчанию сигнал `SIGCONT` возобновляет процесс, если он был приостановлен, иначе сигнал игнорируется. Обычно при получении этого сигнала ничего делать не нужно. Когда генерируется сигнал `SIGCONT`, приостановленный процесс возобновляет работу, даже если этот сигнал заблокирован или игнорируется.

Пример

Программа в листинге 10.22 демонстрирует обычную последовательность действий, выполняемую при обработке сигналов управления заданиями. Данная программа просто копирует данные со стандартного ввода в стандартный вывод; в тех местах, где обычно осуществляется управление терминалом, даны соответствующие комментарии.

Листинг 10.22. Обработка сигнала SIGTSTP

```
#include "apue.h"

#define BUFFSIZE 1024

static void
sig_tstp(int signo) /* Обработчик сигнала SIGTSTP */
{
    sigset_t      mask;

    /* ...переместить курсор в левый нижний угол, установить режим терминала...
     */

    /*
     * Разблокировать SIGTSTP, так как он был заблокирован системой.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL); /* установить диспозицию в значение SIG_DFL */

    kill(getpid(), SIGTSTP); /* и послать сигнал самому себе */

    /*
     * Функция kill не вернет управление,
     * пока работа процесса не будет возобновлена.
     */
}
```

```
signal(SIGTSTP, sig_tstp); /* переустановить обработчик сигнала */

/* ... переустановить режим терминала, перерисовать экран ... */
}

int
main(void)
{
    int      n;
    char    buf[BUFFSIZE];

    /*
     * Сигнал SIGTSTP следует перехватывать только в том случае,
     * если командная оболочка поддерживает управление заданиями.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("ошибка вызова функции write");

    if (n < 0)
        err_sys("ошибка вызова функции read");

    exit(0);
}
```

При запуске программа устанавливает обработчик сигнала `SIGTSTP`, только если его диспозиция имеет значение `SIG_DFL`. Причина в том, что когда программа запускается из командной оболочки, не поддерживающей управление заданиями (например, `/bin/sh`), диспозиция сигнала должна быть установлена в значение `SIG_IGN`. На самом деле командная оболочка явно не устанавливает диспозиции трех сигналов (`SIGTSTP`, `SIGTTIN` и `SIGTTOU`) в значение `SIG_IGN`, изначально это делает процесс `init`, после чего эти диспозиции наследуются всеми оболочками входа. И только те оболочки, которые обладают возможностью управления заданиями, переустанавливают диспозиции этих трех сигналов в значение `SIG_DFL`.

Когда мы вводим символ приостановки, процесс получает сигнал `SIGTSTP` и вызывает обработчик сигнала. На этом этапе нужно выполнить все необходимые действия, связанные с терминалом: переместить курсор в нижний левый угол, восстановить режим работы терминала и т. п. После этого процесс отправляет самому себе этот же сигнал, предварительно разблокировав его и установив его диспозицию в значение `SIG_DFL`. Разблокирование сигнала должно производиться обязательно, так как в это самое время ведется обработка этого же сигнала и система автоматически заблокировала его в момент вызова обработчика. Здесь процесс приостанавливается системой. Он возобновит работу только при получении сигнала `SIGCONT` (который обычно посыпается в ответ на команду `fg`). Мы не перехватываем сигнал `SIGCONT`. По умолчанию он должен возобновить работу приостановленного процесса — когда это произойдет, программа продолжит выполнение, как если бы функция `kill` вернула управление. В этот момент восстанавливается диспозиция сигнала `SIGTSTP` и выполняются необходимые действия с терминалом (например, перерисовка экрана).

10.22. Имена и номера сигналов

В этом разделе мы рассмотрим взаимосвязь между номерами и именами сигналов. В некоторых системах имеется массив

```
extern char *sys_siglist[];
```

Этот массив индексируется номерами сигналов и содержит указатели на строки с именами сигналов.

Этот массив существует в OC FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8. В Solaris 10 также имеется этот массив, но он носит имя _sys_siglist.

Для вывода переносимым способом строк символов, соответствующих номерам сигналов, можно использовать функцию `psignal`.

```
#include <signal.h>
void psignal(int signo, const char *msg);
```

Она выводит в стандартный поток вывода сообщений об ошибках строку `msg` (обычно включающую имя программы), за которой следуют двоеточие, пробел, описание сигнала и символ перевода строки. Если в аргументе `msg` передать пустой указатель `NULL`, будет выведено только описание сигнала. Эта функция очень похожа на `perror` (раздел 1.7).

Если обработчику сигнала, установленному альтернативной функцией `sigaction`, передается структура `siginfo`, информацию о сигнале можно вывести вызовом функции `psiginfo`.

```
#include <signal.h>
void psiginfo(const siginfo_t *info, const char *msg);
```

Она действует подобно функции `psignal`, но имеет доступ к дополнительной информации, а не только к номеру сигнала. Что именно выводится в виде дополнительной информации, зависит от платформы.

Если необходимо вывести лишь описание сигнала и необязательно в стандартный поток вывода сообщений об ошибках (например, в файл журнала), можно воспользоваться функцией `strsignal`. Она напоминает функцию `strerror` (раздел 1.7).

```
#include <string.h>
char *strsignal(int signo);
```

Возвращает указатель на строку с описанием сигнала

По заданному номеру сигнала возвращается строка с описанием. Эта строка может использоваться приложениями для формирования сообщений об ошибках при получении сигналов.

Все обсуждаемые в этой книге платформы поддерживают функции `psignal` и `strsignal`, но в их реализациях имеются различия. В Solaris 10 функция `strsignal` возвращает пустой указатель, если задан некорректный номер сигнала, тогда как в FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 возвращается указатель на строку с сообщением о том, что сигнал не распознан.

Функция `psiginfo` поддерживается только в Linux 3.2.0 и Solaris 10.

Solaris предоставляет пару функций — для отображения номеров сигналов в их имена и обратно.

```
#include <signal.h>

int sig2str(int signo, char *str);

int str2sig(const char *str, int *signop);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Эти функции удобны при разработке интерактивных программ, которые должны принимать и выводить номера сигналов и их имена.

Функция `sig2str` преобразует номер сигнала в строку и сохраняет результат в памяти по адресу, переданному в аргументе `str`. Вызывающий процесс должен предоставить буфер достаточного размера для хранения строки максимально возможной длины, с учетом завершающего нулевого символа. Для этих целей Solaris предусматривает в заголовочном файле `<signal.h>` константу `SIG2STR_MAX`, которая представляет максимальный размер строки, возвращаемой функцией `sig2str`. Возвращаемая строка содержит имя сигнала без префикса `SIG`. Например, если функции передать номер сигнала `SIGKILL`, она вернет строку «`KILL`» в буфере, на который указывает аргумент `str`.

Функция `str2sig` преобразует заданное имя сигнала в его номер. Номер сигнала сохраняется в целочисленной переменной, на которую указывает аргумент `signop`. В качестве имени сигнала можно передать имя без префикса `SIG` или строку с десятичным номером сигнала (например, «9»).

Обратите внимание, что функции `sig2str` и `str2sig` отступают от общепринятой практики и в случае неудачи не сохраняют код ошибки в переменной `errno`.

10.23. Подведение итогов

Сигналы широко используются в большинстве серьезных приложений. Понимание, как и зачем обрабатываются сигналы, является основой профессионального подхода к программированию для системы UNIX. В этой главе представлен достаточно объемный и полный обзор сигналов UNIX. Мы начали с рассмотрения недостатков, присущих ранним реализациям сигналов, и того, как они проявляются. Затем мы перешли к обсуждению надежных сигналов POSIX.1 и связанных с ними функций. Разобравшись со всеми тонкостями, мы смогли реализовать

свои версии функций `abort`, `system` и `sleep`. И наконец, мы закончили главу рассмотрением сигналов управления заданиями и способов преобразования между именами сигналов и их номерами.

Упражнения

- 10.1 В листинге 10.1 удалите оператор `for(;;)`. Что произойдет и почему?
- 10.2 Реализуйте функцию `sig2str`, описанную в разделе 10.22.
- 10.3 Нарисуйте схему, которая показывает кадры стека программы из листинга 10.6.
- 10.4 В листинге 10.8 мы продемонстрировали методику использования функций `setjmp` и `longjmp`, которая достаточно часто применяется для ограничения времени выполнения продолжительных операций ввода/вывода. Можно также представить следующий код:

```
signal(SIGALRM, sig_alarm);
alarm(60);
if (setjmp(env_alarm) != 0) {
    /* обработать ситуацию выхода по тайм-ауту */
    ...
}
...
```

Скажите, что в нем неправильно.

- 10.5 Используя единственный системный таймер (`alarm` или `setitimer` — таймер с высоким разрешением), реализуйте набор функций, которые представляли бы в распоряжение процесса произвольное количество таймеров.
- 10.6 Напишите программу, с помощью которой можно было бы проверить функции синхронизации родительского и дочернего процессов из листинга 10.17. Процесс должен создавать файл и записывать в него число 0. Затем вызывать функцию `fork`, после чего родительский и дочерний процессы должны по очереди увеличивать число, прочитанное из файла. При каждом увеличении счетчика процесс должен выводить информацию о том, кто произвел увеличение — родитель или потомок.
- 10.7 В функции из листинга 10.18 предусмотренброс диспозиции сигнала `SIGABRT` в значение по умолчанию и повторный вызов функции `kill` на случай, если обработчик сигнала вернет управление. Почему в этой ситуации нельзя просто вызвать функцию `_exit`?
- 10.8 Как вы думаете, почему структура `siginfo` (раздел 10.14) помещает в поле `si_uid` реальный, а не эффективный идентификатор пользователя?
- 10.9 Перепишите функцию из листинга 10.10 так, чтобы она могла обслуживать все сигналы из табл. 10.1. Функция должна выполнять одну итерацию цикла для каждого включенного в маску, а не для каждого возможного сигнала.

- 10.10** Напишите программу, которая вызывала бы `sleep(60)` в бесконечном цикле. Каждые пять проходов цикла (то есть каждые 5 минут) программа должна получать текущее время суток и выводить содержимое поля `tm_sec`. Запустите программу на ночь и объясните полученные результаты. Помогите, как может быть реализована программа, которая «просыпается» каждую минуту, как демон `cron`?
- 10.11** Измените программу из листинга 3.3 следующим образом: (а) константу `BUFSIZE` установите в значение 100, (б) перехватите сигнал `SIGXFSZ` с помощью функции `signal_intr`, выведите сообщение при выходе из обработчика сигнала и (в) выведите значение, полученное от функции `write`, если она не смогла записать заданное количество байтов. Измените «мягкий» предел `RLIMIT_FSIZE` (раздел 7.11), установив его в значение 1024, и с помощью измененной программы попробуйте скопировать файл, размер которого превышает 1024 байта. (Попробуйте установить новое значение предела из командной оболочки. Если это не удастся, вызовите функцию `setrlimit` прямо из программы.) Запустите эту программу в другой системе, которая вам доступна. Что произошло и почему?
- 10.12** Напишите программу, которая передает функции `fwrite` буфер гигантского размера (порядка нескольких сотен мегабайт). Перед обращением к `fwrite` вызовите `alarm`, чтобы запланировать генерацию сигнала через одну секунду. Ваш обработчик сигнала должен просто выводить сообщение, что сигнал перехвачен, и возвращать управление. Успеет ли функция `fwrite` завершить работу? Объясните, что произойдет?

11

Потоки

11.1. Введение

В предыдущих главах мы обсуждали процессы. Мы рассмотрели окружение процессов в UNIX, взаимоотношения между процессами и способы управления ими.

В этой главе мы продолжим изучение внутреннего устройства процессов и узнаем, как можно использовать несколько *потоков выполнения* (или просто *потоков* (threads)) для решения нескольких задач в рамках единственного процесса. Все потоки внутри процесса имеют доступ к одним и тем же компонентам процесса, таким как файловые дескрипторы или переменные.

Всякий раз при попытке организовать одновременный доступ нескольких пользователей к одному и тому же ресурсу приходится сталкиваться с проблемой согласования доступа. В конце этой главы мы рассмотрим механизмы синхронизации потоков, которые позволяют предотвратить доступ разных потоков к разделяемым ресурсам, находящимся в несогласованном состоянии.

11.2. Понятие потоков

Типичный процесс в UNIX можно представить как имеющий единственный поток управления — каждый процесс в один момент времени решает только одну задачу. При использовании нескольких потоков управления можно спроектировать приложение, которое будет решать одновременно несколько задач в рамках единственного процесса, где каждый поток решает отдельную задачу. Такой подход имеет следующие преимущества.

- Можно значительно упростить код, обрабатывающий асинхронные события, привязав каждый тип события кциальному потоку. В результате каждый поток сможет обслуживать свое событие, используя для этого синхронную модель программирования, которая намного проще асинхронной.
- Чтобы организовать совместный доступ нескольких процессов к одним и тем же ресурсам, таким как общая память или файловые дескрипторы, необходимо использовать достаточно сложные механизмы синхронизации, предоставляемые операционной системой (об этом — в главах 15 и 17). Потоки же, в отличие от процессов, автоматически получают доступ к одному и тому же адресному пространству и файловым дескрипторам.

- Решение некоторых задач можно разбить на более мелкие подзадачи, что может дать прирост производительности программы. Однопоточный процесс, выполняющий решение нескольких задач, неявно вынужден решать их последовательно, поскольку имеет только один поток управления. При наличии нескольких потоков управления независимые друг от друга задачи могут решаться одновременно отдельными потоками. Две задачи могут решаться одновременно только при условии, что они не зависят друг от друга.
- Аналогично, интерактивные программы могут сократить время отклика на действия пользователя, используя многопоточную модель, чтобы отделить обработку ввода/вывода пользователя от других частей программы.

У многих многопоточное программирование ассоциируется с многопроцессорными системами. Однако преимущества многопоточной модели проявляют себя, даже если программа работает в однопроцессорной системе. Независимо от количества процессоров, программу можно упростить благодаря многопоточной модели, поскольку количество процессоров не влияет на структуру программы. Кроме того, в то время как однопоточный процесс вынужден периодически простояивать при последовательном решении нескольких задач, многопоточный может повысить производительность и в однопроцессорной системе, так как часть потоков могут продолжать работу, когда другие простоявают, ожидая наступления некоторых событий.

Поток содержит набор информации, необходимой для представления контекста выполнения внутри процесса. Сюда включаются идентификатор потока, отличающий поток внутри процесса, набор значений в регистрах процессора, стек, приоритет, маска сигналов, переменная `errno` (раздел 1.7) и дополнительные данные, специфичные для потока (раздел 12.6). Все компоненты процесса, включая выполняемый код программы, глобальные переменные и динамическую память, стеки и файловые дескрипторы, могут совместно использоваться различными потоками этого процесса.

Интерфейс потоков, о котором мы будем говорить, определяется стандартом POSIX.1-2001. Этот интерфейс, известный также как «*pthreads*» (от «*POSIX threads*»), первоначально представлял собой дополнительную функциональную возможность, включенную в стандарт POSIX.1-2001, но в версии SUSv4 был перемещен в раздел базовых спецификаций. Поддержку потоков POSIX можно проверить с помощью макропределения `_POSIX_THREADS`. Приложения могут выполнять проверку поддержки потоков во время компиляции, используя команду условной компиляции `#ifdef`, или во время выполнения, вызывая функцию `sysconf` с аргументом `_SC_THREADS`. Системы, соответствующие стандарту SUSv4, определяют символ `_POSIX_THREADS` со значением 200809L.

11.3. Идентификация потоков

Как любой процесс обладает идентификатором процесса, так и каждый поток имеет свой идентификатор потока. В отличие от процессов, идентификаторы которых являются уникальными в пределах системы, идентификатор потока имеет смысл только в контексте процесса, которому он принадлежит.

Мы уже говорили, что идентификатор процесса представлен типом `pid_t` и является целым неотрицательным числом. Идентификатор потока представлен типом `pthread_t`. Реализациям разрешается использовать структуру для представления типа `pthread_t`, поэтому, чтобы сохранить переносимость приложений, мы не должны рассматривать этот тип как целое число. Следовательно, сравнение двух идентификаторов потоков должно выполняться с помощью функции.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Возвращает ненулевое значение, если идентификаторы равны,
0 – в противном случае

Для представления типа `pthread_t` Linux 3.2.0 использует тип `Long int`, Solaris 10 – `unsigned int`, а FreeBSD 8.0 и Mac OS X 10.6.8 в качестве типа `pthread_t` используют указатель на структуру `pthread`.

Поскольку тип `pthread_t` может быть структурой, не существует простого переносимого способа вывести его значение. Иногда в процессе отладки программы бывает удобно выводить идентификаторы потоков, но, как правило, в других случаях в этом нет необходимости. В самом худшем случае это приводит к написанию непереносимого отладочного кода, поэтому данное ограничение можно считать несущественным.

Поток может получить собственный идентификатор, обратившись к функции `pthread_self`.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Возвращает идентификатор вызывающего потока

Эта функция может использоваться совместно с `pthread_equal`, если внутри потока возникнет необходимость самоидентификации. Например, главный поток может размещать задания в некоторой очереди и сопровождать их идентификаторами потоков, чтобы каждый поток мог выполнять задания, предназначенные конкретно для него. Эта методика показана на рис. 11.1. Главный поток помещает новые задания в очередь, а три рабочих потока извлекают их из очереди. Вместо того чтобы позволить произвольному потоку извлекать очередное задание из начала очереди, главный поток, используя идентификаторы потоков, назначает задания конкретным потокам. В этом случае рабочий поток извлекает из очереди только те задания, которые отмечены его идентификатором.

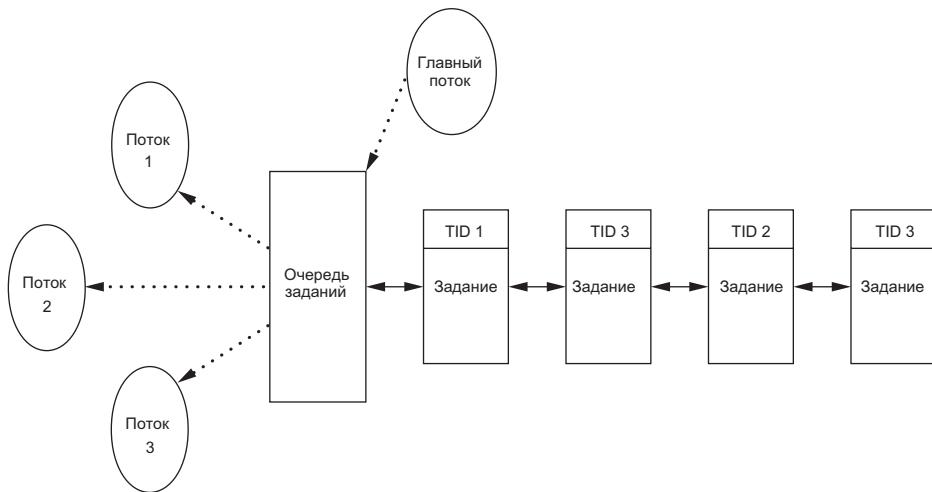


Рис. 11.1. Пример очереди заданий

11.4. Создание потока

Традиционная модель процессов в UNIX поддерживает только один поток управления на процесс. Концептуально это то же, что и модель, основанная на потоках, в случае, когда каждый процесс состоит из одного потока. При наличии поддержки `pthread` программа также запускается как процесс, состоящий из одного потока управления. Поведение такой программы ничем не отличается от поведения традиционного процесса, пока она не создаст дополнительные потоки управления. Создание дополнительных потоков производится с помощью функции `pthread_create`.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void *), void *restrict arg);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Аргумент *tidp* — это указатель на область памяти, где будет размещен идентификатор созданного потока, если вызов функции `pthread_create` завершится успешно. Аргумент *attr* используется для настройки различных атрибутов потока. Об атрибутах потока мы поговорим в разделе 12.3, а пока будем передавать в этом аргументе пустой указатель (`NULL`), что соответствует созданию потока со значениями атрибутов по умолчанию.

Вновь созданный поток начинает выполнение с функции `start_rtn`. Эта функция принимает единственный аргумент `arg` — нетипизированный указатель. Если функции `start_rtn` потребуется передать значительный объем информации, ее следует сохранить в виде структуры и передать в `arg` указатель на структуру.

При создании нового потока нельзя заранее предположить, кто первым получит управление — вновь созданный поток или поток, вызвавший функцию `pthread_create`. Новый поток имеет доступ к адресному пространству процесса и наследует от вызывающего потока среду окружения арифметического сопроцессора и маску сигналов, однако набор сигналов, ожидающих обработки, для нового потока очищается.

Обратите внимание, что функции семейства `pthread`, как правило, возвращают код ошибки в случае неудачи. Они не изменяют значение переменной `errno` подобно другим функциям POSIX. Экземпляр переменной `errno` для каждого потока предоставляется только для сохранения совместимости с существующими функциями, которые используют эту переменную. Вообще, при работе с потоками принято возвращать код ошибки из функций, что дает возможность локализовать ошибку, а не полагаться на некоторую глобальную переменную, которая могла быть изменена в результате побочного эффекта.

Пример

Несмотря на отсутствие переносимого способа вывода значений идентификаторов потоков, все же можно написать небольшую программу, которая делает это, и тем самым получить представление о некоторых особенностях потоков. Программа в листинге 11.1 выводит идентификатор процесса и идентификаторы начального и вновь созданного потоков.

Листинг 11.1. Вывод идентификаторов потоков

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t      pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("новый поток: ");
    return((void *)0);
}
```

```
int
main(void)
{
    int      err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток");
    printids("главный поток:");
    sleep(1);
    exit(0);
}
```

В этом примере есть два интересных момента, связанных с возможностью гонки за ресурсами между основным и вновь созданным потоками. (Далее в этой же главе мы рассмотрим более правильные способы синхронизации потоков.) В первую очередь необходимо приостановить основной поток. Если этого не сделать, основной поток может завершиться и тем самым завершить весь процесс еще до того, как новый поток получит возможность начать работу. Такое поведение потоков во многом зависит от реализации потоков в операционной системе и алгоритма планирования.

Второй интересный момент заключается в том, что новый поток получает свой идентификатор с помощью функции `pthread_self`, а не берет его из глобальной переменной или из аргумента запускающей функции. При описании функции `pthread_create` мы уже говорили, что она возвращает идентификатор созданного потока через аргумент `tidp`. В нашем примере основной поток сохраняет идентификатор в переменной `ntid`, но новый поток не может его использовать. Если новый поток получит управление первым, еще до того, как функция `pthread_create` вернет управление в основной поток, вместо идентификатора новый поток обнаружит неинициализированное значение переменной `ntid`.

Запустив программу из листинга 11.1 в Solaris, мы получили следующие результаты:

```
$ ./a.out
главный поток: pid 20075 tid 1 (0x1)
новый поток:   pid 20075 tid 2 (0x2)
```

Как мы и ожидали, оба потока обладают одним и тем же идентификатором процесса, но разными идентификаторами потоков. Запуск программы из листинга 11.1 в FreeBSD дал следующие результаты:

```
$ ./a.out
главный поток: pid 37396 tid 673190208 (0x28201140)
новый поток:   pid 37396 tid 673280320 (0x28217140)
```

В этом случае потоки также имеют один и тот же идентификатор процесса. Если рассматривать идентификаторы потоков как целые десятичные числа, они могут показаться достаточно странными, но если их рассматривать в шестнадцатеричном представлении, они приобретают некоторый смысл. Как уже отмечалось выше, в качестве идентификатора потока FreeBSD использует указатель на структуру с данными потока.

В Mac OS X можно было бы ожидать похожих результатов, однако идентификаторы главного потока и потока, созданного с помощью функции `pthread_create`, принадлежат к разным диапазонам адресов.

```
$ ./a.out
главный поток: pid 31807 tid 140735073889440 (0x7fff70162ca0)
новый поток:   pid 31807 tid 4295716864 (0x1000b7000)
```

Запуск программы в Linux дал несколько иные результаты:

```
$ ./a.out
главный поток: pid 17874 tid 140693894424320 (0x7ff5d9996700)
новый поток:   pid 17874 tid 140693886129920 (0x7ff5d91ad700)
```

Идентификаторы потоков в Linux больше напоминают указатели, хотя в действительности являются целыми числами без знака.

Реализация потоков изменилась при переходе от версии Linux 2.4 к Linux 2.6. В Linux 2.4 подсистема LinuxThreads реализовала потоки как отдельные процессы. Это мешало реализации поведения потоков так, чтобы оно соответствовало требованиям стандарта POSIX. В Linux 2.6 ядро и библиотека поддержки потоков были полностью переделаны под новую реализацию потоков под названием Native POSIX Thread Library (NPTL). Она поддерживает модель выполнения множества потоков в рамках единственного процесса и упрощает поддержку семантики потоков в соответствии со стандартом POSIX.

11.5. Завершение потока

Если один из потоков вызовет функцию `exit`, `_exit` или `_Exit`, будет завершен весь процесс. Аналогично, если потоку будет послан сигнал, действие которого заключается в завершении процесса, этот сигнал завершит весь процесс (более подробно о взаимодействиях между сигналами и потоками мы поговорим в разделе 12.8).

Завершить работу единственного потока, то есть без завершения всего процесса, можно тремя способами.

- Поток может просто вернуть управление из запускающей процедуры. Возвращаемое значение этой процедуры — код завершения потока.
- Поток можно принудительно завершить из другого потока в том же процессе.
- Поток может вызвать функцию `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

Аргумент `rval_ptr` — это нетипизированный указатель, аналогичный аргументу, передаваемому запускающей процедуре. Этот указатель смогут получить другие потоки процесса, вызвавшие функцию `pthread_join`.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Вызывающий поток будет заблокирован, пока указанный поток не вызовет функцию `pthread_exit`, не вернет управление из запускающей процедуры или не будет принудительно завершен другим потоком. Если поток просто выйдет из запускающей процедуры, `rval_ptr` будет содержать возвращаемое значение. Если поток был принудительно завершен, по адресу `rval_ptr` будет записано значение `PTHREAD_CANCELED`.

Вызов функции `pthread_join` автоматически переводит поток в обособленное состояние (вскоре мы обсудим это), которое позволяет вернуть ресурсы потока обратно. Если он уже находится в обособленном состоянии, поток, вызвавший `pthread_join`, получит код ошибки `EINVAL`.

Если нас не интересует возвращаемое значение потока, мы можем передать пустой указатель в аргументе `rval_ptr`. В этом случае обращение к функции `pthread_join` позволит дождаться завершения указанного потока, но не вернет код его завершения.

Пример

В листинге 11.2 показано, как получить код выхода завершившегося потока.

Листинг 11.2. Получение кода выхода потока

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("поток 1: выход\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: выход\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
```

```

if (err != 0)
    err_exit(err, "невозможно создать поток 1");
err = pthread_create(&tid2, NULL, thr_fn2, NULL);
if (err != 0)
    err_exit(err, "невозможно создать поток 2");
err = pthread_join(tid1, &tret);
if (err != 0)
    err_exit(err, "невозможно присоединить поток 1");
printf("код выхода потока 1: %ld\n", (long)tret);
err = pthread_join(tid2, &tret);
if (err != 0)
    err_exit(err, "невозможно присоединить поток 2");
printf("код выхода потока 2: %ld\n", (long)tret);
exit(0);
}

```

Запустив программу из листинга 11.2, мы получили:

```

$ ./a.out
поток 1: выход
поток 2: выход
код выхода потока 1: 1
код выхода потока 2: 2

```

Как видите, когда поток завершается вызовом функции `pthread_exit` или просто возвращая управление из запускающей процедуры, другой поток может получить код выхода вызовом `pthread_join`.

Нетипизированный указатель, передаваемый функциям `pthread_create` и `pthread_exit`, может использоваться для передачи более одного значения. В этом указателе можно передать адрес структуры, содержащей большой объем информации. Помните, что этот адрес должен оставаться действительным после выхода из вызывающей функции. Если, к примеру, структура размещается на стеке вызывающей функции, ее содержимое может оказаться измененным к моменту, когда она будет использована. Если поток размещает структуру на стеке и передает указатель на нее функции `pthread_exit`, стек этого потока может оказаться разрушенным, а память, занимаемая им, может быть использована повторно для других целей к моменту, когда поток, вызвавший `pthread_join`, попытается обратиться к ней.

Пример

Программа в листинге 11.3 демонстрирует проблему, связанную с использованием переменной с автоматическим классом размещения (на стеке) в качестве аргумента функции `pthread_exit`.

Листинг 11.3. Некорректное использование аргумента функции `pthread_exit`

```

#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void

```

```
printfoo(const char *s, const struct foo *fp)
{
    printf("%s", s);
    printf("  структура по адресу 0x%lx\n", (unsigned long)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo foo = {1, 2, 3, 4};

    printfoo("поток 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: идентификатор - %lu\n", (unsigned long)pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    struct foo   *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    sleep(1);
    printf("родительский процесс создает второй поток\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 2");
    sleep(1);
    printfoo("родительский процесс:\n", fp);
    exit(0);
}
```

Запустив эту программу в Linux, мы получили:

```
$ ./a.out
поток 1:
структура по адресу 0x7f2c83682ed0
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 139829159933696
```

родительский процесс:

```
структура по адресу 0x7f2c83682ed0
foo.a = -2090321472
foo.b = 32556
foo.c = 1
foo.d = 0
```

Разумеется, результаты зависят от архитектуры памяти, компилятора и реализации библиотеки функций для работы с потоками. В Solaris были получены похожие результаты:

```
$ ./a.out
поток 1:
структура по адресу 0xffffffff7f0fbf30
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 3
родительский процесс:
структура по адресу 0xffffffff7f0fbf30
foo.a = -1
foo.b = 2136969048
foo.c = -1
foo.d = 2138049024
```

Как видите, содержимое структуры (размещенной на стеке потоком *tid1*) изменилось к тому моменту, когда главный поток получил к ней доступ. Обратите внимание, как стек второго потока (*tid2*) наложился на стек первого потока. Чтобы решить эту проблему, можно либо использовать глобальную память, либо размещать структуру с помощью функции `malloc`.

В Mac OS X мы получили иные результаты:

```
$ ./a.out
поток 1:
структура по адресу 0x1000b6f00
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 4295716864
родительский процесс:
структура по адресу 0x1000b6f00
Segmentation fault (core dumped)
```

В данном случае, когда родитель попытался получить доступ к структуре, переданной ему при выходе из первого потока, ее адрес оказался недействительным и родителю был послан сигнал `SIGSEGV`.

В FreeBSD память не была затерта к моменту, когда родитель обратился к ней, и мы получили:

```
поток 1:
структура по адресу 0xbff9fef88
foo.a = 1
```

```
foo.b = 2
foo.c = 3
foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 673279680
родительский процесс:
структура по адресу 0xbff9fef88
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
```

Но даже при том что после выхода из потока память оказалась не повреждена, мы не должны полагать, что так будет всегда. Кроме того, на других платформах мы наблюдаем как раз противоположное поведение.

Один поток может передать запрос на принудительное завершение другого потока в том же процессе, обратившись к функции `pthread_cancel`.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

По умолчанию вызов `pthread_cancel` заставляет указанный поток вести себя так, словно он вызвал функцию `pthread_exit` с аргументом `PTHREAD_CANCELED`. Однако поток может отвергнуть запрос или как-то иначе отреагировать на него. Более подробно мы обсудим эту тему в разделе 12.7. Обратите внимание, что функция `pthread_cancel` не ждет завершения потока. Она просто посыпает запрос.

Поток может назначить некоторую функцию для вызова в момент его завершения примерно так же, как это делается для процессов с помощью функции `atexit` (раздел 7.3), которая регистрирует функции, запускаемые при завершении процесса. Эти функции называют *функциями обработки выхода из потока*. Поток может зарегистрировать несколько таких функций обработки выхода. Обработчики заносятся в стек — это означает, что они будут вызываться в порядке, обратном порядку их регистрации.

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Функция `pthread_cleanup_push` регистрирует функцию `rtn`, которая будет вызвана с аргументом `arg`, когда поток выполнит одно из следующих действий:

- вызовет функцию `pthread_exit`;
- ответит на запрос о принудительном завершении;
- вызовет функцию `pthread_cleanup_pop` с ненулевым аргументом `execute`.

Если аргумент *execute* имеет значение 0, функция обработки выхода из потока вызываться не будет. В любом случае функция *pthread_cleanup_pop* удаляет функцию-обработчик, зарегистрированную последним обращением к функции *pthread_cleanup_push*.

Ограничение, связанное с этими функциями, заключается в том, что они могут быть реализованы в виде макроопределений, и тогда они должны использоваться в паре, в пределах одной области видимости в потоке. Макроопределение функции *pthread_cleanup_push* может включать символ {, и тогда парная ей скобка } будет находиться в макроопределении *pthread_cleanup_pop*.

Пример

В листинге 11.4 показан порядок использования функций обработки выхода из потока. Это во многом надуманный пример, тем не менее он прозрачно иллюстрирует описываемую методику. Обратите внимание: хотя ненулевой аргумент не передается в функцию *pthread_cleanup_pop*, мы по-прежнему вынуждены вызывать функции *pthread_cleanup_push* и *pthread_cleanup_pop* в паре, иначе программа может не скомпилироваться.

Листинг 11.4. Обработчик выхода из потока

```
#include "apue.h"
#include <pthread.h>

void
cleanup(void *arg)
{
    printf("выход: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
    printf("запуск потока 1\n");
    pthread_cleanup_push(cleanup, "поток 1, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 1, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("запуск потока 2\n");
    pthread_cleanup_push(cleanup, "поток 2, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 2, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
```

```
    pthread_exit((void *)2);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_exit(err, "невозможно создать поток 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    printf("код выхода потока 1: %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 2");
    printf("код выхода потока 2: %ld\n", (long)tret);
    exit(0);
}
```

Запуск программы из листинга 11.4 дал нам следующие результаты:

```
$ ./a.out
запуск потока 1
поток 1, регистрация обработчиков закончена
запуск потока 2
поток 2, регистрация обработчиков закончена
выход: поток 2, второй обработчик
выход: поток 2, первый обработчик
код выхода потока 1: 1
код выхода потока 2: 2
```

Из полученных результатов видно, что оба потока нормально запустились и корректно завершились, но функции обработки выхода были вызваны только для второго потока. Отсюда можно сделать вывод, что функции обработки выхода из потока не вызываются, если поток завершается простым возвратом из процедуры запуска. Кроме того, обратите внимание, что функции обработки выхода запускаются в порядке, обратном порядку их регистрации.

Если запустить ту же программу в FreeBSD или Mac OS X, программа завершится аварийно с сохранением файла core. Проблема в том, что в этих системах функция `pthread_cleanup_push` реализована как макрос, сохраняющий некоторые контекстные данные на стеке. Когда поток 1 возвращает управление между вызовами `pthread_cleanup_push` и `pthread_cleanup_pop`, содержимое стека затирается и программа в этих системах использует уже поврежденный к этому моменту контекст, когда пытается вызвать обработчики выхода из потока. Поведение системы при возврате между соответствующими парными вызовами `pthread_cleanup_push` и `pthread_cleanup_pop` не регламентируется стандартом Single UNIX Specification.

Единственный переносимый способ выполнить возврат между вызовами этих двух функций — обратиться к функции `pthread_exit`.

Сейчас вы уже должны обнаружить некоторые черты сходства между функциями управления процессами и функциями управления потоками.

В табл. 11.1 приводится список аналогичных функций.

Таблица 11.1. Сравнение функций управления процессами и потоками

Процессы	Потоки	Описание
<code>fork</code>	<code>pthread_create</code>	Создает новый поток управления
<code>exit</code>	<code>pthread_exit</code>	Завершает существующий поток управления
<code>waitpid</code>	<code>pthread_join</code>	Возвращает код выхода из потока управления
<code>atexit</code>	<code>pthread_cleanup_push</code>	Регистрирует функцию обработки выхода из потока управления
<code>getpid</code>	<code>pthread_self</code>	Возвращает идентификатор потока управления
<code>abort</code>	<code>pthread_cancel</code>	Запрашивает аварийное завершение потока управления

По умолчанию код завершения потока сохраняется, пока для этого потока не будет вызвана функция `pthread_join`. Основная память потока может быть немедленно освобождена по его завершении, если поток был обособлен. Когда поток обособлен, функция `pthread_join` не может использоваться для получения его кода завершения, потому что в этом случае ее поведение не определено. Обособить поток можно с помощью функции `pthread_detach`.

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Как мы увидим в следующей главе, существует возможность создания потока изначально в обособленном состоянии, через изменение атрибутов потока, передаваемых функции `pthread_create`.

11.6. Синхронизация потоков

При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что все потоки будут видеть стабильное представление этих данных. Если каждый из потоков использует переменные, которые не модифицируются в других потоках, проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, здесь также отсутствует проблема сохранения непротиворечивости.

Однако если один поток изменяет значение переменной, читать или изменять которую могут также другие потоки, необходимо синхронизировать доступ к пере-

менной, чтобы гарантировать, что потоки не будут получать неверное значение переменной при одновременном доступе к ней.

Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток прочитает еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинился цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.

На рис. 11.2 приводится пример гипотетической ситуации, когда два потока одновременно выполняют запись и чтение значения одной и той же переменной. В данном примере поток А читает значение переменной и затем записывает в нее новое значение, но операция записи производится за два цикла. Если поток В прочитает значение этой же переменной между двумя циклами записи, он обнаружит переменную в несогласованном состоянии.

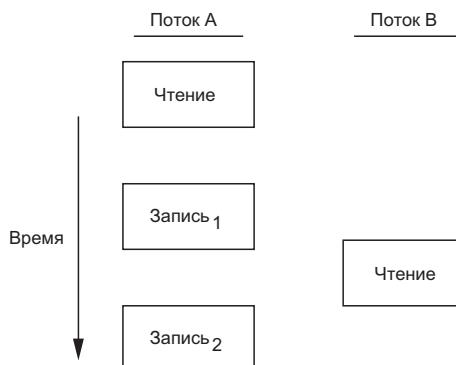


Рис. 11.2. Перемежение циклов доступа к памяти из двух потоков

Для решения этой проблемы потоки должны использовать блокировки, которые позволяют только одному потоку работать с переменной в один момент времени. На рис. 11.3 показана подобная синхронизация. Если поток В должен прочитать значение переменной, он устанавливает блокировку. Аналогично, когда поток А изменяет значение переменной, он также устанавливает блокировку. Благодаря такой организации поток В не сможет прочитать значение переменной, пока поток А не снимет блокировку.

Точно так же следует синхронизировать два или более потока, которые могут попытаться одновременно изменить значение переменной. Рассмотрим случай, когда выполняется увеличение значения переменной на 1 (рис. 11.4). Операцию увеличения (инкремента) обычно можно разбить на три шага.

1. Прочитать значение переменной из памяти в регистр процессора.
2. Увеличить значение в регистре.
3. Записать новое значение из регистра процессора в память.

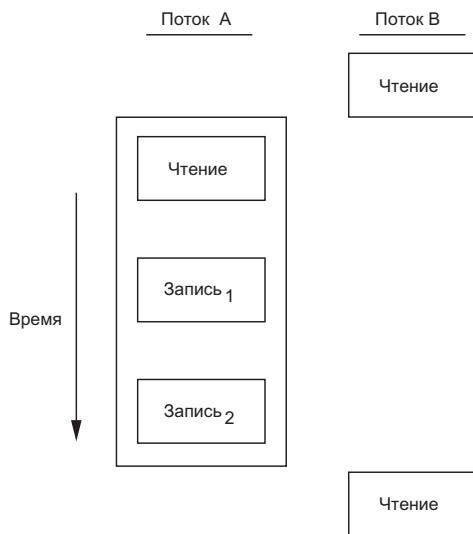


Рис. 11.3. Синхронизированный доступ к памяти из двух потоков

Если два потока попытаются одновременно увеличить значение одной и той же переменной, не согласуя свои действия между собой, результаты могут получиться самые разные. В конечном итоге полученное значение может оказаться на 1 или на 2 больше предыдущего в зависимости от того, какое значение получил второй поток перед началом операции. Если второй поток выполнил шаг 1 до того, как первый выполнил шаг 3, второй поток прочитает то же значение, что и первый поток, увеличит его на 1 и запишет обратно в память, фактически не оказав никакого влияния на значение переменной.

Если изменение переменной производится атомарно, подобная гонка между потоками отсутствует. В предыдущем примере, если увеличение производится за одно обращение к памяти, состояние гонки между потоками не возникает. Если данные постоянно находятся в *непротиворечивом состоянии*, нет необходимости предусматривать дополнительную синхронизацию. Операции являются последовательно непротиворечивыми, если различные потоки не могут получить доступ к данным, когда они находятся в противоречивом состоянии. В современных компьютерных системах доступ к памяти выполняется за несколько тактов шины, а в многопроцессорных системах доступ к шине вообще чередуется между несколькими процессорами, поэтому невозможно гарантировать непротиворечивое состояние данных в любой произвольный момент времени.

В непротиворечивой среде изменения данных можно описать как последовательность операций, выполняемых потоками. Мы можем сказать: «Поток А увеличил значение переменной, затем поток В увеличил значение переменной, в результате значение переменной было увеличено на 2» или: «Поток В увеличил значение переменной, затем поток А увеличил значение переменной, в результате значение переменной было увеличено на 2». Конечный результат не зависит от порядка выполнения потоков.



Рис. 11.4. Два несинхронизированных потока пытаются увеличить значение одной и той же переменной

Помимо особенностей аппаратной архитектуры, состояние гонки может быть вызвано алгоритмом использования переменных в программах. Например, мы можем увеличить значение переменной *i* и затем, основываясь на полученном значении, принять решение о дальнейшем порядке выполнения операций. Комбинация операций, состоящая из увеличения переменной и проверки полученного значения, не является атомарной, и, таким образом, появляется вероятность принятия неверного решения.

11.6.1. Мьютексы

Мы можем защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью взаимоисключений (mutual-exclusion) интерфейса `pthreads`. *Мьютекс* (mutex) — это фактически блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, любой другой поток, который попытается запереть его, будет заблокирован, пока мьютекс не будет отперт. Если в момент отпирания мьютекса сразу несколько потоков будут находиться в заблокированном состоянии, все они будут запущены, и первый, кто успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат запертый мьютекс и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.

Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения соблюдают одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с общими данными, предварительно не ограничив доступ к ним, остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Переменные-мьютексы определяются с типом `pthread_mutex_t`. Прежде чем использовать переменную-мьютекс, следует сначала инициализировать ее, записав значение константы `PTHREAD_MUTEX_INITIALIZER` (только для статически размещаемых мьютексов) или вызвав функцию `pthread_mutex_init`. Если мьютекс размещается в динамической памяти (например, с помощью функции `malloc`), прежде чем освободить занимаемую память, необходимо вызвать функцию `pthread_mutex_destroy`.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Чтобы инициализировать мьютекс со значениями атрибутов по умолчанию, нужно передать `NULL` в аргументе `attr`. Конкретные значения атрибутов мьютексов мы рассмотрим в разделе 12.4.

Запирается мьютекс вызовом функции `pthread_mutex_lock`. Если мьютекс уже заперт,зывающий поток будет заблокирован, пока мьютекс не будет отперт. Мьютекс отпирается вызовом функции `pthread_mutex_unlock`.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Все три возвращают 0 в случае успеха, код ошибки — в случае неудачи

Если поток не должен блокироваться при попытке запереть мьютекс, он может воспользоваться функцией `pthread_mutex_trylock`. Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае `pthread_mutex_trylock` вернет код ошибки `EBUSY`.

Пример

Листинг 11.5 иллюстрирует использование мьютексов для защиты структуры данных. Если более чем один поток работает с данными, размещаемыми динами-

чески, мы можем предусмотреть в структуре данных счетчик ссылок на объект, чтобы освобождать память, только когда все потоки завершат работу с объектом.

Листинг 11.5. Использование мьютексов для защиты структур данных

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(int id) /* размещает объект в динамической памяти */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... продолжение инициализации ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* наращивает счетчик ссылок на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    if (-fp->f_count == 0) { /* последняя ссылка */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Перед увеличением или уменьшением счетчика ссылок и перед его проверкой на равенство нулю мьютекс запирается. При инициализации счетчика ссылок значением 1 в функции `foo_alloc` запирать мьютекс нет необходимости, поскольку пока только поток, который размещает структуру, имеет к ней доступ. Если бы в этой точке структура включалась в некий список, она могла бы быть обнаружена другими потоками, и тогда пришлось бы сначала запереть мьютекс.

Прежде чем приступить к работе с объектом, поток должен увеличить счетчик ссылок на него. По окончании работы с объектом поток должен удалить ссылку. Когда удаляется последняя ссылка, память, занимаемая объектом, освобождается. Данный пример игнорирует возможность обнаружения объектов другими потоками перед вызовом `foo_hold`. Даже если счетчик ссылок равен нулю, для `foo_release` было бы ошибкой пытаться освободить память, занимаемую объектом, так как другой поток мог запереть мьютекс вызовом `foo_hold`. Этой проблемы можно избежать, если перед освобождением памяти убедиться, что объект не может быть найден. Как это сделать, будет показано в примерах, следующих ниже.

11.6.2. Предотвращение тупиковых ситуаций

Поток может попасть в тупиковую ситуацию (deadlock), если попытается дважды захватить один и тот же мьютекс, но есть и менее очевидные способы. Например, тупиковая ситуация может возникнуть, когда в программе используется несколько мьютексов и один поток, удерживая первый мьютекс, пытается запереть второй, в то время как другой поток аналогично удерживает второй мьютекс и пытается запереть первый. В результате ни один из потоков не сможет продолжить работу, поскольку каждый будет ждать освобождения ресурса, захваченного другим потоком. Возникнет тупиковая ситуация.

Тупиковых ситуаций можно избежать, жестко определив порядок захвата ресурсов. Приведем пример. Предположим, что есть два мьютекса, A и B, которые необходимо запереть одновременно. Если все потоки сначала будут запирать мьютекс A, а потом B, тупиковой ситуации с этими мьютексами никогда не возникнет. Аналогично, если все потоки сначала будут запирать мьютекс B, а потом A, тупиковой ситуации с этими мьютексами также никогда не возникнет. Опасность попадания в тупиковую ситуацию возникает, только когда разные потоки могут попытаться запереть мьютессы в разном порядке.

Иногда архитектура приложения не позволяет заранее предопределить порядок захвата мьютексов. Если программа использует достаточно много мьютексов и структур данных, а доступные функции, которые работают с ними, не укладываются в достаточно простую иерархию, придется попробовать иной подход. Например, при невозможности запереть мьютекс можно отпереть захваченные мьютессы и повторить попытку немного позже. В этом случае во избежание блокировки потока можно использовать функцию `pthread_mutex_trylock`. Если мьютекс удалось запереть с помощью `pthread_mutex_trylock`, можно продолжить работу. Если мьютекс запереть не удалось, можно отпереть уже захваченные мьютессы, освободить занятые ресурсы и повторить попытку немного позже.

Пример

В этом примере приводится дополненная версия программы из листинга 11.5 с целью продемонстрировать работу с двумя мьютексами. Во избежание тупиковой ситуации, которая может возникнуть при попытке одновременного захвата обоих ресурсов, во всех потоках используется один и тот же порядок запирания мьютексов.

Второй мьютекс защищает хеш-список структур `foo`. То есть мьютекс `hashlock` защищает хеш-таблицу `fh` и поле связи `f_next` в структуре `foo`. Доступ к остальным полям структуры `foo` производится под защитой мьютекса `f_lock`.

Листинг 11.6. Использование двух мьютексов

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int          f_count;
    pthread_mutex_t f_lock;
    int          f_id;
    struct foo   *f_next; /* защищается мьютексом hashlock */
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(void) /* размещает объект в динамической памяти */
{
    struct foo *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... продолжение инициализации ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* добавляет ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id) /* находит существующий объект */
```

```

{
    struct foo *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    struct foo *tfp;
    int      idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* последняя ссылка */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* необходима повторная проверка условия */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* удалить из списка */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}

```

Сравнив листинги 11.6 и 11.5, можно заметить, что теперь функция размещения объекта в динамической памяти блокирует доступ к хеш-таблице, добавляет в нее новую структуру, а перед снятием блокировки с хеш-таблицы записывает новую структуру. Поскольку новая структура размещается в глобальном списке, ее может обнаружить любой другой поток, и поэтому мы вынуждены записывать ее, пока не будет закончена инициализация структуры.

Функция `foo_find` запирает хеш-таблицу и производит поиск запрошенной структуры. Если таковая будет найдена, мы увеличиваем в ней счетчик ссылок и возвращаем указатель на структуру. Обратите внимание, что здесь мы соблюдаем порядок захвата мьютексов, запирая мьютекс `hashlock` до того, как функция `foo_hold` запрет мьютекс `f_lock`.

Теперь перейдем к функции `foo_rele`, алгоритм работы которой несколько сложнее. Если освобождается последняя ссылка на объект, необходимо отпереть мьютекс `f_lock`, чтобы запереть `hashlock`, поскольку нам необходимо удалить структуру из списка. После этого необходимо запереть мьютекс `f_lock`.

Учитывая, что поток мог быть заблокирован во время повторной попытки захватить мьютессы, мы вынуждены повторить проверку необходимости удаления структуры. Если какой-либо другой поток нашел структуру и нарастил счетчик ссылок в ней, в то время как данный поток был заблокирован в ожидании освобождения мьютекса, мы просто уменьшаем счетчик ссылок, отпираем оба мьютекса и возвращаем управление.

Алгоритм работы с мьютексами получился довольно сложным, поэтому пересмотрим его. Алгоритм заметно упростится, если мьютекс `hashlock` будет защищать еще и счетчик ссылок. Мьютекс `f_lock` будет защищать все остальные поля структуры `foo`. Эти изменения отражены в листинге 11.7.

Листинг 11.7. Упрощенный вариант использования мьютексов

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* защищается мьютексом hashlock */
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next; /* защищается мьютексом hashlock */
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(int id) /* размещает объект в динамической памяти */
{
    struct foo   *fp;
    int           idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(id);
    }
}
```

```

pthread_mutex_lock(&hashlock);
fp->f_next = fh[idx];
fh[idx] = fp;
pthread_mutex_lock(&fp->f_lock);
pthread_mutex_unlock(&hashlock);
/* ... продолжение инициализации ... */
pthread_mutex_unlock(&fp->f_lock);
}
return(fp);
}

void
foo_hold(struct foo *fp) /* добавляет ссылку на объект */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* находит существующий объект */
{
    struct foo *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    struct foo *tfp;
    int      idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* последняя ссылка, удалить из списка */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
    }
    pthread_mutex_unlock(&hashlock);
    pthread_mutex_destroy(&fp->f_lock);
    free(fp);
} else {
    pthread_mutex_unlock(&hashlock);
}
}

```

Обратите внимание, насколько проще стала программа по сравнению с листингом 11.6. Когда мы стали использовать один и тот же мьютекс для защиты хеш-списка и счетчика ссылок, отпала проблема соблюдения порядка захвата мьютексов. При разработке многопоточных приложений достаточно часто приходится идти на подобные компромиссы. Слишком грубая детализация блокировок в конечном итоге приведет к тому, что большинство потоков будут простаивать при попытках запереть один и тот же мьютекс, а преимущества многопоточной архитектуры приложения будут сведены к минимуму. Если детализация блокировок будет слишком мелкой, это существенно усложнит код, а производительность приложения снизится из-за избыточного количества мьютексов. Программист должен найти правильный баланс между производительностью и сложностью алгоритма и при этом выполнить все требования, связанные с захватом ресурсов.

11.6.3. Функция `pthread_mutex_timedlock`

Один из дополнительных примитивов управления мьютексами позволяет ограничить время блокировки потока при попытке захватить мьютекс, запертый другим потоком. Функция `pthread_mutex_timedlock` эквивалентна функции `pthread_mutex_lock`, но по истечении указанного тайм-аута `pthread_mutex_timedlock` вернет код ошибки `ETIMEDOUT`, не запирая мьютекс.

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict tspr);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Под тайм-аутом здесь понимается абсолютное время (в противоположность относительному; то есть мы должны указать момент времени X , когда следует прекратить попытки приобрести блокировку, а не количество секунд Y , в течение которых следует ждать освобождения мьютекса, если он занят). Значение тайм-аута определяется в виде структуры `timespec`, хранящей время в секундах и наносекундах.

Пример

Листинг 11.8 демонстрирует, как использовать функцию `pthread_mutex_timedlock`, чтобы избежать блокировки потока навечно.

Листинг 11.8. Использование функции `pthread_mutex_timedlock`

```
#include "apue.h"
#include <pthread.h>

int
main(void)
{
    int err;
    struct timespec tout;
```

```

struct tm *tmp;
char buf[64];

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
printf("мьютекс заперт\n");
clock_gettime(CLOCK_REALTIME, &tout);
tmp = localtime(&tout.tv_sec);
strftime(buf, sizeof(buf), "%r", tmp);
printf("текущее время: %s\n", buf);
tout.tv_sec += 10; /* 10 секунд, начиная от текущего времени */
/* внимание: это может привести к тупиковой ситуации */
err = pthread_mutex_timedlock(&lock, &tout);
clock_gettime(CLOCK_REALTIME, &tout);
tmp = localtime(&tout.tv_sec);
strftime(buf, sizeof(buf), "%r", tmp);
printf("текущее время: %s\n", buf);
if (err == 0)
    printf("мьютекс снова заперт!\n");
else
    printf("не получилось повторно запереть мьютекс: %s\n", strerror(err));
exit(0);
}

```

Ниже приводится вывод программы из листинга 11.8.

```

$ ./a.out
мьютекс заперт
текущее время: 11:41:58 AM
текущее время: 11:42:08 AM
не получилось повторно запереть мьютекс: Connection timed out

```

Эта программа преднамеренно пытается повторно запереть уже запертый мьютекс, чтобы продемонстрировать работу функции `pthread_mutex_timedlock`. Данную стратегию не рекомендуется использовать на практике, потому что она может служить источником тупиковых ситуаций.

Обратите внимание, что протяженность интервала блокировки может варьироваться в некоторых пределах по следующим причинам: начальный момент времени может быть определен в середине текущей секунды, разрешение системных часов может быть недостаточно точным для поддержки желаемой точности тайм-аута, задержки в планировщике задач могут вызвать увеличение времени ожидания.

Mac OS X 10.6.8 не поддерживает функцию `pthread_mutex_timedlock`, но FreeBSD 8.0, Linux 3.2.0 и Solaris 10 поддерживают ее, однако в Solaris эта функция до сих пор находится в библиотеке реального времени, `librt`. В Solaris 10 имеется также альтернативная функция, которой можно передать относительный тайм-аут.

11.6.4. Блокировки чтения-записи

Блокировки чтения-записи похожи на мьютессы, за исключением того, что они допускают более высокую степень параллелизма. Мьютессы могут иметь всего два состояния, закрытое и открытое, и только один поток может владеть мьютекс-

сом в каждый момент времени. Блокировки чтения-записи могут иметь три состояния: режим блокировки для чтения, режим блокировки для записи и отсутствие блокировки. Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме для записи, все потоки, которые попытаются захватить ее, будут приостановлены, пока блокировка не будет снята. Если блокировка чтения-записи установлена в режиме для чтения, все потоки, которые попытаются захватить ее для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен, пока не будет снята последняя блокировка для чтения. Разные реализации блокировок чтения-записи могут различаться существенно, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи прекрасно подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку владеть блокировкой в этом режиме может только один поток. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут безопасно читать несколько потоков, сумевших получить блокировку для чтения.

Блокировки чтения-записи еще называют совместно-исключающими блокировками (shared-exclusive locks). Когда блокировка чтения-записи установлена в режиме для чтения, говорят, что блокировка находится в режиме совместного использования. Когда блокировка чтения-записи установлена в режиме для записи, говорят, что блокировка находится в режиме исключительного использования.

Как и в случае с мьютексами, блокировки чтения-записи должны инициализироваться перед использованием и разрушаться перед освобождением занимаемой ими памяти.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwLock,
                      const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwLock);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Функция `pthread_rwlock_init` инициализирует блокировку чтения-записи. Если в аргументе `attr` передать пустой указатель, блокировка инициализируется с атрибутами по умолчанию. Атрибуты блокировок чтения-записи мы рассмотрим в разделе 12.4.2.

Стандарт Single UNIX Specification определяет константу `PTHREAD_RWLOCK_INITIALIZER` как расширение XSI, которую можно использовать для инициализации статических блокировок чтения-записи, когда значений по умолчанию для атрибутов вполне достаточно.

Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию `pthread_rwlock_destroy`, чтобы освободить все занимаемые блокировкой ресурсы. Функция `pthread_rwlock_init` размещает все необходимые для блокировки ресурсы, а `pthread_rwlock_destroy` освобождает их. Если освободить память, занимаемую блокировкой чтения-записи без предварительного обращения к функции `pthread_rwlock_destroy`, все ресурсы, занимаемые блокировкой, будут потеряны для системы.

Чтобы установить блокировку в режиме для чтения, необходимо вызвать функцию `pthread_rwlock_rdlock`. Чтобы установить блокировку в режиме для записи, необходимо вызвать функцию `pthread_rwlock_wrlock`. Независимо от режима блокировки чтения-записи, ее снятие выполняется функцией `pthread_rwlock_unlock`.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwLock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwLock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwLock);
```

Все три возвращают 0 в случае успеха, код ошибки — в случае неудачи

Реализации могут ограничивать количество блокировок, установленных в режиме совместного использования, поэтому обязательно нужно проверять значение, возвращаемое функцией `pthread_rwlock_rdlock`. Даже когда функции `pthread_rwlock_wrlock` и `pthread_rwlock_unlock` возвращают код ошибки, нет необходимости проверять возвращаемые значения этих функций, если схема наложения блокировок разработана надлежащим образом. Эти функции могут вернуть код ошибки, только когда блокировка не инициализирована или когда может возникнуть тупиковая ситуация при попытке повторно установить уже установленную блокировку. Однако вы должны помнить, что некоторые реализации могут определять дополнительные коды ошибок.

Стандарт Single UNIX Specification определяет дополнительные версии примитивов для работы с блокировками, которые могут использоваться для проверки состояния блокировки.

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwLock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwLock);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Если блокировка была успешно установлена, эти функции возвращают 0. Иначе возвращается код ошибки EBUSY. Эти функции можно использовать, когда нельзя заранее предопределить порядок установки блокировок, чтобы избежать тупиковых ситуаций, которые мы обсуждали ранее.

Пример

Программа в листинге 11.9 иллюстрирует применение блокировок чтения-записи. Очередь запросов на выполнение заданий защищается единственной блокировкой чтения-записи. Этот пример является одной из возможных реализаций приложения, представленного на рис. 11.1, где множество потоков получают задания, назначаемые им главным потоком.

Листинг 11.9. Использование блокировки чтения-записи

```
#include <stdlib.h>
#include <pthread.h>

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t j_id; /* сообщает, какой поток выполняет это задание */
    /* ... другие поля структуры ... */
};

struct queue {
    struct job     *q_head;
    struct job     *q_tail;
    pthread_rwlock_t q_lock;
};

/*
 * Инициализирует очередь.
 */
int
queue_init(struct queue *qp)
{
    int      err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);
    /* ... продолжение инициализации ... */
    return(0);
}

/*
 * Добавляет задание в начало очереди.
 */
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
```

```

    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp; /* список был пуст */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Добавляет задание в конец очереди.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp; /* список был пуст */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Удаляет задание из очереди.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
        else
            jp->j_next->j_prev = jp->j_prev;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        jp->j_prev->j_next = jp->j_next;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}

/* Находит задание для потока с заданным идентификатором. */

struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)

```

```
    if (pthread_equal(jp->j_id, id))
        break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

В этом примере блокировка чтения-записи очереди устанавливается в режиме для записи, только когда необходимо добавить новое или удалить имеющееся задание. Когда нужно выполнить поиск задания в очереди, мы устанавливаем блокировку в режиме для чтения, допуская возможность поиска заданий несколькими рабочими потоками одновременно. В данном случае использование блокировки чтения-записи дает прирост производительности, только если поиск заданий в очереди выполняется чаще, чем добавление или удаление.

Рабочие потоки извлекают из очереди только те задания, которые соответствуют их идентификаторам. Поскольку сама структура с заданием используется только одним потоком, для организации доступа к ней не требуется дополнительных блокировок.

11.6.5. Блокировки чтения-записи с тайм-аутом

Как и в случае с мьютексами, стандарт Single UNIX Specification определяет функции для приобретения блокировок чтения-записи с тайм-аутом, не позволяющие приложениям заблокироваться навечно при попытке приобрести блокировку. Это функции `pthread_rwlock_timedrdlock` и `pthread_rwlock_timedwrlock`.

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwLock,
                                const struct timespec *restrict tsPtr);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwLock,
                                const struct timespec *restrict tsPtr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Эти функции действуют подобно своим «неограниченным» эквивалентам. В аргументе `tsptr` они принимают указатель на структуру `timespec`, определяющую момент времени, когда следует прекратить попытки приобрести блокировку. Если по истечении тайм-аута не удалось приобрести блокировку, эти функции возвращают код ошибки `ETIMEDOUT`. Как и в функции `pthread_mutex_timedlock`, тайм-аут определяет абсолютный момент времени, а не интервал ожидания.

11.6.6. Переменные состояния

Переменные состояния (condition variables) — еще один механизм синхронизации потоков. Переменные состояния предоставляют потокам своеобразное место встречи. При использовании вместе с мьютексами переменные состояния позволяют потокам ожидать наступления некоторого события, избегая состояния гонки.

Сами переменные состояния защищаются мьютексами. Прежде чем изменить значение такой переменной, поток должен захватить мьютекс. Другие потоки не будут замечать изменений в переменной, пока не попытаются захватить этот мьютекс, потому что для оценки переменной состояния необходимо запереть мьютекс.

Переменная состояния, представленная типом `pthread_cond_t`, должна инициализироваться перед использованием. При статическом размещении переменной можно присвоить значение константы `PTHREAD_COND_INITIALIZER`, но если переменная состояния размещается динамически, ее следует инициализировать вызовом `pthread_cond_init`.

Для уничтожения переменной состояния перед освобождением занимаемой ею памяти используется функция `pthread_cond_destroy`.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Если в аргументе `attr` передать пустой указатель, переменная состояния будет инициализирована значениями атрибутов по умолчанию. Атрибуты переменных состояния мы рассмотрим в разделе 12.4.3.

Функция `pthread_cond_wait` ждет, пока переменная перейдет в истинное состояние. Чтобы ограничить время ожидания заданным интервалом, используется функция `pthread_cond_timedwait`.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict tspr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Мьютекс, передаваемый функции `pthread_cond_wait`, защищает доступ к переменной состояния. Вызывающий поток передает его функции в запертом состоянии, а функция атомарно помещает вызывающий поток в список потоков, ожидающих изменения состояния переменной, и отирает мьютекс. Это исключает вероятность, что переменная изменит состояние между моментом ее проверки и моментом приостановки потока, благодаря чему поток не пропустит наступление ожидаемого события. Когда функция `pthread_cond_wait` возвращает управление, мьютекс снова запирается.

Функция `pthread_cond_timedwait` работает аналогично, но дополнительно дает возможность ограничить время ожидания. Значение аргумента `tpstr` определяет,

как долго поток будет ожидать наступления события. Время тайм-аута задается структурой `timespec`.

Как было показано в листинге 11.8, в этой структуре следует указывать абсолютное время, а не относительное. Например, если потребуется ограничить время ожидания 3 минутами, мы должны сохранить в этой структуре не 3 минуты, а текущее время + 3 минуты.

Для этого можно воспользоваться функцией `clock_gettime` (раздел 6.10), возвращающей текущее время в виде структуры `timespec`. Однако эта функция поддерживается не всеми платформами. Вместо нее можно использовать функцию `gettimeofday`, чтобы получить текущее время в виде структуры `timeval`, и затем преобразовать ее в структуру `timespec`. Чтобы получить абсолютное время для аргумента `tsptr`, можно использовать следующую функцию (предполагается, что продолжительность интервала времени измеряется в минутах):

```
#include <sys/time.h>
#include <stdlib.h>

void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* получить текущее время */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* микросекунды в наносекунды */
    /* добавить величину тайм-аута */
    tsp->tv_sec += minutes * 60;
}
```

Если тайм-аут истечет до появления ожидаемого события, функция `pthread_cond_timedwait` запрет мьютекс и вернет код ошибки `ETIMEDOUT`. Когда функция `pthread_cond_wait` или `pthread_cond_timedwait` завершится успехом, поток должен оценить значение переменной, поскольку к этому моменту другой поток мог изменить его.

Для передачи сообщения о наступлении события существуют две функции. Функция `pthread_cond_signal` возобновит работу одного потока, ожидающего наступления события, а `pthread_cond_broadcast` — всех потоков, ожидающих наступления события.

Для упрощения реализации стандарт POSIX допускает, чтобы функция `pthread_cond_signal` возобновляла работу нескольких потоков.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Когда вызывается функция `pthread_cond_signal`, говорят, что посыпается сигнал о наступлении события. Мы должны сделать все возможное, чтобы сигнал о наступлении события посыпался только после изменения состояния переменной.

Пример

В листинге 11.10 приводится пример синхронизации потоков с помощью переменных состояния и мьютексов.

Листинг 11.10. Пример использования переменных состояния

```
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... другие поля структуры ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* обработка сообщения mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

В данном случае переменная хранит состояние очереди сообщений. Переменная состояния защищена мьютексом, а определение изменения состояния производится в цикле `while`. Чтобы поместить очередное сообщение в очередь, необходимо запереть мьютекс, но чтобы послать сигнал ожидающим потокам, запирать мьютекс не нужно. Такой вариант, когда сигнал посыпается после отпирания мьютекса, будет прекрасно работать, даже если какой-либо поток успеет возобновить работу до передачи сигнала. Поскольку наступление события проверяется в ци-

кле, это не представляет проблемы: поток просто возобновит работу, убедится, что очередь пуста, и опять перейдет в режим ожидания. Если логика программы не допускает подобной гонки, тогда необходимо сначала вызвать `pthread_cond_signal`, а затем отпереть мьютекс.

11.6.7. Циклические блокировки

Циклическая блокировка (spin lock) подобна мьютексу, но блокируемый процесс не приостанавливается, а вращается (spinning) в цикле ожидания, пока не приобретет блокировку. Циклическую блокировку можно использовать в ситуациях, когда блокировка нужна на очень короткий промежуток времени, а накладные расходы на перепланирование потока выполнения выглядят непомерно большими.

Циклические блокировки часто используются как низкоуровневые примитивы для реализации блокировок других типов. В зависимости от архитектуры системы они могут быть реализованы с использованием инструкций «проверил и установил». Несмотря на высокую эффективность, они могут приводить к напрасной тратае вычислительных ресурсов: пока поток вращается в цикле, ожидая, процессор не может заняться чем-то другим. Именно поэтому циклические блокировки должны приобретаться на очень короткие промежутки времени.

Циклические блокировки особенно полезны при использовании невытесняемого ядра (nonpreemptive kernel): помимо поддержки механизма взаимоисключения (mutual exclusion), они блокируют прерывания, поэтому исключается вероятность тупиковой ситуации, когда обработчик прерывания может попытаться приобрести уже запертую циклическую блокировку (прерывания в данном контексте можно рассматривать как одну из разновидностей вытеснения). В ядрах этого типа обработчики прерываний не могут приостанавливаться, поэтому единственные примитивы синхронизации, которые они могут использовать, — это циклические блокировки.

Однако на уровне пользовательского приложения циклические блокировки не так полезны, если только приложение не выполняется с классом планирования в режиме реального времени, не допускающим вытеснения. Пользовательские потоки, выполняющиеся с классом планирования в режиме разделения времени, могут вытесняться после исчерпания выделенного кванта времени или при появлении готового к выполнению потока с более высоким приоритетом. В этих случаях поток, приобретший циклическую блокировку, будет приостановлен и другие потоки, заблокированные на блокировке, продолжат вращаться в цикле ожидания дольше, чем предполагалось.

Многие реализации мьютексов настолько эффективны, что производительность приложений с использованием мьютексов не уступает производительности тех же приложений с циклическими блокировками. На практике некоторые реализации мьютексов вращаются ограниченное время в цикле ожидания, пытаясь приобрести мьютекс, и только когда счетчик циклов ожидания превысит пороговое значение, приостанавливают поток. Такой подход в сочетании с возможностями современных процессоров, позволяющими переключать контекст выполнения

все быстрее и быстрее, делают циклические блокировки пригодными лишь в редких ситуациях.

Интерфейс циклических блокировок похож на интерфейс мьютексов, что позволяет легко заменять одни другими. Инициализация циклической блокировки выполняется вызовом функции `pthread_spin_init`. Чтобы уничтожить циклическую блокировку, следует вызвать функцию `pthread_spin_destroy`.

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

При инициализации циклической блокировки можно указать лишь один атрибут, который имеет смысл, только если платформа поддерживает расширение Thread Process-Shared Synchronization (Синхронизация потоков между процессами, в настоящее время это расширение перенесено в разряд базовых спецификаций Single UNIX Specification; см. табл. 2.5). В аргументе `pshared` передается признак *совместного использования блокировки несколькими процессами* (process-shared), определяющий ее доступность. Если в нем передать значение `PTHREAD_PROCESS_SHARED`, циклическая блокировка будет доступна потокам выполнения, имеющим доступ к памяти, где хранится блокировка, — даже потокам в других процессах. Иначе аргумент `pshared` следует устанавливать в значение `PTHREAD_PROCESS_PRIVATE`, и в этом случае циклическая блокировка будет доступна только потокам процесса, инициализировавшего ее.

Запереть циклическую блокировку можно с помощью `pthread_spin_lock`, которая будет крутиться в цикле ожидания, пока не приобретет блокировку, или `pthread_spin_trylock`, которая вернет код ошибки `EBUSY`, если блокировку нельзя приобрести немедленно. Обратите внимание, что `pthread_spin_trylock` не выполняет цикл ожидания. Независимо от способа, каким была заперта блокировка, ее можно освободить вызовом `pthread_spin_unlock`.

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Все возвращают 0 в случае успеха, код ошибки — в случае неудачи

Обратите внимание: если циклическая блокировка свободна, функция `pthread_spin_lock` может запереть ее, не выполняя цикл ожидания. Стандарты не определяют, что должна делать реализация, если поток попытается приобрести блокировку, которой он уже владеет. В этом случае вызов `pthread_spin_lock` может

вернуть код ошибки EDEADLK (или какой-то другой) или попасть в бесконечный цикл ожидания. Поведение зависит от реализации. Также стандарты не определяют, что должна делать реализация, если поток попытается освободить незапертую блокировку.

Если `pthread_spin_lock` или `pthread_spin_trylock` вернула 0, следовательно, циклическая блокировка была успешно заперта. Необходимо проявлять особую осторожность, чтобы не вызвать какую-нибудь функцию, которая может приостановить поток, пока он удерживает циклическую блокировку. Иначе мы впустую будем тратить процессорное время, увеличивая продолжительность времени, которое другие потоки потратят при попытке приобретения этой циклической блокировки.

11.6.8. Барьеры

Барьеры (barriers) — это механизм синхронизации, который можно использовать для координации действий нескольких потоков, выполняющихся одновременно. Барьер позволяет каждому потоку дождаться момента, когда все сотрудничающие с ним потоки достигнут той же точки, и продолжить работу. Мы уже познакомились с одной из разновидностей барьеров — функцией `pthread_join`, действующей как барьер, позволяя одному потоку дождаться завершения другого.

Однако объекты барьеров более универсальны, чем эта функция. Они дают возможность любому количеству потоков дождаться, пока все потоки завершат обработку, но при этом потоки не обязаны завершаться. Они могут продолжить работу, когда все потоки достигнут барьера.

Инициализировать барьер можно с помощью функции `pthread_barrier_init`, а уничтожить — с помощью функции `pthread_barrier_destroy`.

```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                        const pthread_barrierattr_t *restrict attr,
                        unsigned int count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

При инициализации барьера в аргументе `count` передается количество потоков, которые должны достигнуть барьера, прежде чем всем потокам будет позволено продолжить работу. В аргументе `attr` передаются атрибуты объекта барьера, с которыми мы познакомимся в следующей главе. А пока достаточно знать, что если передать в аргументе `attr` пустой указатель (`NULL`), барьер будет инициализирован значениями атрибутов по умолчанию. Если функция `pthread_barrier_init` выделяет какие-либо ресурсы для барьера, эти ресурсы будут освобождены функцией `pthread_barrier_destroy`.

Чтобы показать, что поток выполнил свое задание и готов ждать, когда другие потоки достигнут барьера, он должен вызывать функцию `pthread_barrier_wait`.

```
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Возвращает 0 или PTHREAD_BARRIER_SERIAL_THREAD в случае успеха, код ошибки — в случае неудачи

Поток, вызвавший `pthread_barrier_wait`, приостанавливается, если количество ожидающих потоков не сравнялось со счетчиком барьера (устанавливается вызовом функции `pthread_barrier_init`). Если поток оказался последним достигшим барьера, вызов `pthread_barrier_wait` возобновит работу всех ожидающих потоков.

В одном потоке (выбранном произвольно) `pthread_barrier_wait` вернет значение `PTHREAD_BARRIER_SERIAL_THREAD`. В остальных она вернет 0. Это дает возможность одному из потоков взять на себя руководящие функции и заняться обработкой результатов, произведенных всеми другими потоками.

Как только количество потоков достигнет счетчика в объекте барьера и все ожидающие потоки разблокируются, барьер можно использовать повторно. Однако изменить счетчик барьера нельзя иначе, как вызвав функцию `pthread_barrier_destroy` и за ней функцию `pthread_barrier_init` с другим значением счетчика.

Пример

Листинг 11.11 демонстрирует, как можно использовать барьер для синхронизации потоков, сотрудничающих над решением общей задачи.

Листинг 11.11. Использование барьера

```
#include "apue.h"
#include <pthread.h>
#include <limits.h>
#include <sys/time.h>

#define NTHR 8           /* количество потоков */
#define NUMNUM 8000000L  /* количество чисел для сортировки */
#define TNUM (NUMNUM/NTHR) /* количество чисел для одного потока */

long nums[NUMNUM];
long snums[NUMNUM];

pthread_barrier_t b;

#ifdef SOLARIS
#define heapsort qsort
#else
extern int heapsort(void *, size_t, size_t,
                    int (*)(const void *, const void *));
#endif

/*
 * Сравнивает два длинных целых (вспомогательная функция для heapsort)
 */
int
```

```
complong(const void *arg1, const void *arg2)
{
    long l1 = *(long *)arg1;
    long l2 = *(long *)arg2;
    if (l1 == l2)
        return 0;
    else if (l1 < l2)
        return -1;
    else
        return 1;
}

/*
 * Рабочий поток, сортирующий фрагмент массива чисел.
 */
void *
thr_fn(void *arg)
{
    long      idx = (long)arg;

    heapsort(&nums[idx], TNUM, sizeof(long), complong);
    pthread_barrier_wait(&b);

    /*
     * Выполнить дополнительные операции при необходимости ...
     */
    return((void *)0);
}

/*
 * Выполняет слияние результатов сортировки фрагментов.
 */
void
merge()
{
    long      idx[NTHR];
    long      i, minidx, sidx, num;

    for (i = 0; i < NTHR; i++)
        idx[i] = i * TNUM;
    for (sidx = 0; sidx < NUMNUM; sidx++) {
        num = LONG_MAX;
        for (i = 0; i < NTHR; i++) {
            if ((idx[i] < (i+1)*TNUM) && (nums[idx[i]] < num)) {
                num = nums[idx[i]];
                minidx = i;
            }
        }
        snums[sidx] = nums[idx[minidx]];
        idx[minidx]++;
    }
}

int
main()
{
    unsigned long  i;
    struct timeval start, end;
    long long      startusec, endusec;
```

```

double      elapsed;
int        err;
pthread_t    tid;

/*
 * Создать начальный массив чисел для сортировки.
 */
srandom(1);
for (i = 0; i < NUMNUM; i++)
    nums[i] = random();

/*
 * Запустить 8 потоков для сортировки массива.
 */
gettimeofday(&start, NULL);
pthread_barrier_init(&b, NULL, NTHR+1);
for (i = 0; i < NTHR; i++) {
    err = pthread_create(&tid, NULL, thr_fn, (void *)(i * TNUM));
    if (err != 0)
        err_exit(err, "невозможно создать поток");
}
pthread_barrier_wait(&b);
merge();
gettimeofday(&end, NULL);

/*
 * Вывести отсортированный массив.
 */
startusec = start.tv_sec * 1000000 + start.tv_usec;
endusec = end.tv_sec * 1000000 + end.tv_usec;
elapsed = (double)(endusec - startusec) / 1000000.0;
printf("продолжительность сортировки (сек.): %.4f\n", elapsed);
for (i = 0; i < NUMNUM; i++)
    printf("%ld\n", snums[i]);
exit(0);
}

```

Этот пример демонстрирует использование барьера в простой ситуации, когда все потоки решают общую задачу. В более сложных ситуациях, после возврата из функции `pthread_barrier_wait`, рабочие потоки могут приступать к решению других задач.

В данном примере мы использовали восемь потоков, чтобы распределить между ними работу по сортировке восьми миллионов чисел. Каждый поток сортирует один миллион чисел, применяя алгоритм пирамидальной сортировки (heapsort) [Knuth, 1998]. Когда все потоки завершат работу, главный поток выполняет объединение результатов.

Нам не потребовалось использовать значение `PTHREAD_BARRIER_SERIAL_THREAD`, возвращаемое функцией `pthread_barrier_wait`, чтобы решить, какой поток будет объединять результаты, потому что эту работу выполняет главный поток.

Именно поэтому мы указали счетчик потоков на единицу больше, чем количество рабочих потоков, — главный поток учитывается как один из ожидающих на барьере.

Если написать программу, выполняющую сортировку 8 миллионов чисел с помощью алгоритма пирамидальной сортировки в единственном потоке, можно за-

метить, насколько быстрее выполняется программа в листинге 11.11. В системе с 8 ядрами однопоточная программа сортирует 8 миллионов чисел за 12,14 секунды. В той же системе программа с 8 потоками, выполняющимися одновременно, и одним потоком, объединяющим результаты, сортирует тот же массив из 8 миллионов чисел за 1,91 секунды — в 6 раз быстрее.

11.7. Подведение итогов

В этой главе мы обсуждали понятие потоков и примитивы POSIX.1 для работы с ними. Мы также коснулись проблемы синхронизации потоков. Были рассмотрены пять фундаментальных механизмов синхронизации — мьютексы, блокировки чтения-записи и переменные состояния, циклические блокировки и барьеры — и их применение для организации доступа к совместно используемым ресурсам.

Упражнения

- 11.1** Измените программу из листинга 11.3 таким образом, чтобы она корректно передавала структуру данных между потоками.
- 11.2** Изучите листинг 11.9 и скажите, какая дополнительная синхронизация должна быть предусмотрена (если она необходима), чтобы позволить главному потоку изменять идентификатор потока в задании. Как это повлияет на функцию `job_remove`?
- 11.3** Примените технику, показанную в листинге 11.10, к программе (рис. 11.1 и листинг 11.9) для реализации функции рабочего потока. Не забудьте дополнить функцию `queue_init` инициализацией переменной состояния и измените функции `job_insert` и `job_append` так, чтобы они посыпали сигналы рабочим потокам. Какие сложности при этом возникнут?
- 11.4** Какую последовательность действий можно считать правильной?
 1. Запереть мьютекс (`pthread_mutex_lock`).
 2. Изменить переменную состояния, защищаемую мьютексом.
 3. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).
 4. Отпереть мьютекс (`pthread_mutex_unlock`).
или
 1. Запереть мьютекс (`pthread_mutex_lock`).
 2. Изменить переменную состояния, защищаемую мьютексом.
 3. Отпереть мьютекс (`pthread_mutex_unlock`).
 4. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).
- 11.5** Какие примитивы синхронизации можно использовать для реализации барьера? Реализуйте функцию `pthread_barrier_wait`.

12

Управление потоками

12.1. Введение

В главе 11 мы рассмотрели основные понятия, связанные с потоками, и вопросы их синхронизации. В этой главе мы обсудим вопросы управления поведением потоков, а также рассмотрим атрибуты потока и объекты синхронизации, которые игнорировали в предыдущей главе, работая со значениями по умолчанию.

Мы также поговорим о том, как скрыть данные потока от других потоков в том же процессе. И закончим главу описанием взаимодействий между некоторыми системными вызовами и потоками.

12.2. Пределы для потоков

В разделе 2.5.4 мы обсуждали функцию `sysconf`. Стандарт Single UNIX Specification определяет ряд пределов, связанных с потоками, которые не были приведены в табл. 2.11. Как и в случае системных пределов, значения пределов потоков можно получить с помощью функции `sysconf`. Эти пределы перечисляются в табл. 12.1.

Подобно другим пределам, значения которых сообщает функция `sysconf`, данные пределы предназначены для повышения переносимости приложений между различными реализациями операционных систем. Например, если приложение требует, чтобы для обработки каждого файла создавалось четыре потока, вероятно, придется ограничить количество обрабатываемых одновременно файлов, чтобы не превысить ограничение системы на количество одновременно работающих потоков.

В табл. 12.2 приводятся значения этих пределов для четырех обсуждаемых в этой книге платформ. Если реализация не определяет константу для передачи функции `sysconf` (имя которой начинается с последовательности символов `_SC_`), в соответствующей колонке указано «Не определено». Если реализация не определяет значение предела, в колонке указывается «Нет ограничения», однако это вовсе не говорит о том, что предел не имеет ограничений.

Обратите внимание, что хотя реализация может и не определять значений этих пределов, это не означает, что их вообще не существует. Это означает лишь, что реализация не дает возможности получить значение предела с помощью функции `sysconf`.

Таблица 12.1. Пределы для потоков и соответствующие значения аргумента name функции sysconf

Имя предела	Описание	Аргумент name
PTHREAD_DESTRUCTOR_ITERATIONS	Максимальное количество попыток системы уничтожить данные потока после его завершения (раздел 12.6)	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	Максимальное количество ключей, которые может создать процесс (раздел 12.6)	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	Минимальное количество байтов, которые можно использовать под стек потока (раздел 12.3)	_SC_THREAD_STACK_MIN
PTHREAD_STACK_MAX	Максимальное количество байтов, которые можно использовать под стек потока (раздел 12.3)	_SC_THREAD_STACK_MAX

Таблица 12.2. Примеры значений пределов для потоков

Предел	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
PTHREAD_DESTRUCTOR_ITERATIONS	4	4	4	Нет ограничения
PTHREAD_KEYS_MAX	256	1024	512	Нет ограничения
PTHREAD_STACK_MIN	2048	16 384	8192	8192
PTHREAD_STACK_MAX	Нет ограничения	Нет ограничения	Нет ограничения	Нет ограничения

12.3. Атрибуты потока

Интерфейс `pthread` позволяет выполнять тонкую настройку поведения потоков и объектов синхронизации, настраивая различные атрибуты, связанные с каждым объектом. В общем случае функции управления этими атрибутами следуют одному и тому же шаблону.

- Каждый из объектов связывается с объектом атрибутов собственного типа (потоки — с атрибутами потоков, мьютексы — с атрибутами мьютексов и т. д.). Объект атрибутов может представлять множество атрибутов. Объект атрибутов непрозрачен для приложения. Это означает, что приложение ничего не должно знать о внутреннем устройстве объекта, что способствует повышению переносимости приложений. Для управления атрибутами приложения должны использовать специализированные функции.
- Функция инициализации устанавливает атрибуты в значения по умолчанию.

3. Уничтожение объектов атрибутов выполняется другими функциями. Если функция инициализации распределяет какие-либо ресурсы, связанные с объектом атрибутов, функция уничтожения освобождает эти ресурсы.
4. Для каждого атрибута существует функция, возвращающая его значение. Так как функция возвращает 0 в случае успеха и код ошибки в случае неудачи, значение атрибута возвращается в области памяти, адрес которой передается в одном из аргументов.
5. Для каждого атрибута существует функция, изменяющая его значение. В данном случае новое значение атрибута передается функции *по значению*.

Во всех примерах главы 11, где вызывалась функция `pthread_create`, мы передавали ей значение `NULL` вместо указателя на структуру `pthread_attr_t`. Структура `pthread_attr_t` используется для изменения значений атрибутов по умолчанию и связывания этих атрибутов с создаваемым потоком. Для инициализации структуры `pthread_attr_t` можно обратиться к функции `pthread_init_attr`. После вызова этой функции структура `pthread_attr_t` будет заполнена значениями атрибутов по умолчанию, которые поддерживает данная реализация.

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Для разрушения структуры `pthread_attr_t` используется функция `pthread_attr_destroy`. Если функция `pthread_attr_init` реализована так, что размещает какие-либо области в динамической памяти, функция `pthread_attr_destroy` освободит их. Кроме того, `pthread_attr_destroy` заполнит структуру ошибочными значениями, чтобы функция `pthread_create` возвращала ошибку при случайном использовании такой структуры.

Атрибуты потока, определяемые стандартом POSIX.1, приводятся в табл. 12.3. Кроме того, стандарт POSIX.1 определяет ряд дополнительных атрибутов для потоков реального времени, но мы не будем обсуждать их здесь. В табл. 12.3 также показано, какие атрибуты поддерживаются нашими четырьмя платформами.

Таблица 12.3. Атрибуты потоков POSIX.1

Атрибут	Описание	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>detachstate</code>	Атрибут обособленности потока	✓	✓	✓	✓
<code>guardsize</code>	Размер резервного буфера в конце стека потока	✓	✓	✓	✓
<code>stackaddr</code>	Самый нижний адрес стека потока	✓	✓	✓	✓
<code>stacksize</code>	Минимальный размер стека потока в байтах	✓	✓	✓	✓

В разделе 11.5 мы упомянули понятие обособленных потоков. Если нас больше не интересует код завершения существующего потока, мы можем обратиться к функции `pthread_detach`, чтобы позволить операционной системе утилизировать ресурсы, занимаемые потоком, после его завершения.

Если заранее известно, что код завершения потока не потребуется, можно сразу же создать и запустить поток в обособленном состоянии, изменив значение атрибута `detachstate` в структуре `pthread_attr_t`. Для этого используется функция `pthread_attr_setdetachstate`, которой передается одно из двух возможных значений — `PTHREAD_CREATE_DETACHED`, чтобы запустить поток в обособленном состоянии, и `PTHREAD_CREATE_JOINABLE`, чтобы запустить поток в нормальном состоянии, в котором приложение сможет получить код завершения потока.

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Чтобы получить текущее состояние атрибута `detachstate`, можно воспользоваться функцией `pthread_attr_getdetachstate`. По адресу, который передается во втором аргументе, функция запишет одно из двух возможных значений: `PTHREAD_CREATE_DETACHED` или `PTHREAD_CREATE_JOINABLE`, в зависимости от значения атрибута в структуре `pthread_attr_t`.

Пример

В листинге 12.1 приводится функция, которую можно использовать для создания потока в обособленном состоянии.

Листинг 12.1. Создание потока в обособленном состоянии

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int            err;
    pthread_t      tid;
    pthread_attr_t attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Обратите внимание, что мы игнорируем значение, возвращаемое функцией `pthread_attr_destroy`. В данном случае мы корректно инициализировали атрибуты потока, поэтому `pthread_attr_destroy` не должна завершаться с ошибкой. Тем не менее если бы эта функция завершилась неудачей, восстановление после такой ошибки было бы достаточно сложным: мы должны были бы разрушить только что созданный поток, который, возможно, уже работает асинхронно по отношению к этой функции. Самое худшее, что может случиться в случае игнорирования возвращаемого значения функции `pthread_attr_destroy`, — это утечка небольшого объема памяти, который, возможно, был распределен функцией `pthread_attr_init`. Но в любом случае, если `pthread_attr_init` завершилась успехом, а `pthread_attr_destroy` — с ошибкой, у нас все равно нет никакой стратегии восстановления после такой ошибки, потому что структура с атрибутами непрозрачна для приложения. Для утилизации структуры определен один-единственный интерфейс `pthread_attr_destroy`, и он потерпел неудачу.

Поддержка атрибутов потоков, связанных со стеком, является необязательной для POSIX-совместимых систем, но обязательна для систем, отвечающих требованиям XSI. Проверить наличие поддержки атрибутов стека для каждого потока можно на этапе компиляции, используя макроопределения `_POSIX_THREAD_ATTR_STACKADDR` и `_POSIX_THREAD_ATTR_STACKSIZE`. Если определен какой-либо из этих символов, поддерживается и соответствующий ему атрибут. Выполнить аналогичную проверку во время выполнения можно также с помощью функции `sysconf`, передав ей символические имена `_SC_THREAD_ATTR_STACKADDR` и `_SC_THREAD_ATTR_STACKSIZE`.

Получить и изменить атрибуты стека потока можно с помощью функций `pthread_attr_getstack` и `pthread_attr_setstack`.

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stackaddr, size_t stacksize);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Виртуальное адресное пространство процесса имеет фиксированный объем. Поскольку однопоточные процессы имеют только один стек, его размер обычно не вызывает проблем. В случае многопоточных приложений одно и то же виртуальное адресное пространство отведено под стеки всех потоков. Если приложение запускает большое количество потоков, иногда приходится уменьшать размер стека, установленный по умолчанию, чтобы суммарный объем стеков не превысил доступный объем виртуального адресного пространства. С другой стороны, если потоки вызывают функции, которые размещают на стеке большое число локальных переменных, или если глубина вызовов функций очень велика, возможно, придется увеличить размер стека.

В случае нехватки виртуального адресного пространства место под стек потока можно выделить с помощью функции `malloc` или `mmap` (раздел 14.8) и затем, посредством функции `pthread_attr_setstack`, изменить местоположение стека создаваемого потока. Адрес стека определяется аргументом `stackaddr`, который представляет наименьший адрес в диапазоне памяти, используемой под стек потока, выровненный по границе в соответствии с аппаратной архитектурой. Разумеется, при этом предполагается, что диапазон виртуальных адресов, используемый функцией `malloc` или `mmap`, отличается от диапазона адресов, занимаемых стеком в текущий момент.

Атрибут `stackaddr` определяет наименьший адрес участка памяти, отведенной под стек. Однако это не обязательно дно (начало) стека. Если для определенной аппаратной архитектуры стек растет от старших адресов к младшим, атрибут `stackaddr` будет определять вершину (конец) стека, а не его дно (начало).

Приложения также могут получать и изменять значение атрибута потока `stacksize` с помощью функций `pthread_attr_getstacksize` и `pthread_attr_setstacksize`.

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                             size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Функция `pthread_attr_setstacksize` удобна, когда необходимо изменить размер стека по умолчанию, но при этом нет желания заниматься распределением памяти для стека. При изменении атрибута `stacksize` новый размер стека не может быть меньше `PTHREAD_STACK_MIN`.

Атрибут `guardsize` управляет размером памяти, расположенной за концом стека, которая служит для предохранения стека от переполнения. Значение по умолчанию для этого атрибута выбирается реализацией, но часто оно равно значению `PAGESIZE`. Можно установить значение атрибута `guardsize` равным 0, запретив тем самым использование предохранительного буфера. Кроме того, если изменить значение атрибута `stackaddr`, система будет предполагать, что мы берем на себя ответственность за распределение памяти под стек, и запретит использование защитного буфера, просто записав значение 0 в атрибут `guardsize`.

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                             size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Если атрибут `guardsize` был изменен, система может округлить его значение до ближайшего целого, кратного размеру страницы. Если указатель стека потока

войдет в пределы предохранительного буфера, приложение получит сообщение об ошибке — вероятно, в виде сигнала.

Стандарт Single UNIX Specification определяет еще целый ряд необязательных атрибутов потоков в виде расширений потоков реального времени, но мы не будем обсуждать их здесь.

Потоки имеют еще несколько атрибутов, не представленных в структуре `pthread_attr_t`, — возможность принудительного завершения и тип принудительного завершения. Мы обсудим их в разделе 12.7.

12.4. Атрибуты синхронизации

Как и потоки, объекты синхронизации потоков также имеют атрибуты. В разделе 11.6.7 говорилось, что циклические блокировки (spin locks) имеют атрибут *process-shared*. В этом разделе мы рассмотрим атрибуты мьютексов, блокировок чтения-записи, переменных состояния и барьеров.

12.4.1. Атрибуты мьютексов

Атрибуты мьютекса определяются структурой `pthread_mutexattr_t`. В главе 11 мы во всех примерах использовали атрибуты со значениями по умолчанию, присваивая значение константы `PTHREAD_MUTEX_INITIALIZER` или вызывая `pthread_mutex_init` с пустым указателем в аргументе, указывающем на структуру с атрибутами мьютекса.

Для инициализации структуры `pthread_mutexattr_t` значениями, отличными от значений по умолчанию, используется функция `pthread_mutexattr_init`, а для ее разрушения — `pthread_mutexattr_destroy`.

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Функция `pthread_mutexattr_init` инициализирует структуру `pthread_mutexattr_t` значениями по умолчанию. Для нас представляют интерес три атрибута: *process-shared*, *robust* и *type*. Согласно стандарту POSIX.1, атрибут *process-shared* является необязательным — если этот атрибут поддерживается заданной платформой, будет определен и символ `_POSIX_THREAD_PROCESS_SHARED`. Проверку во время выполнения можно произвести с помощью функции `sysconf`, передав ей параметр `_SC_THREAD_PROCESS_SHARED`. Хотя POSIX-совместимые системы не обязаны поддерживать этот атрибут, стандарт Single UNIX Specification требует обязательной его поддержки в системах, отвечающих требованиям XSI.

Внутри процесса множество потоков могут иметь доступ к одному и тому же объекту синхронизации. Такое поведение определено по умолчанию, о чем мы уже го-

ворили в главе 11. В этом случае атрибут мьютекса *process-shared* имеет значение `PTHREAD_PROCESS_PRIVATE`.

Как будет показано в главах 14 и 15, существуют определенные механизмы, позволяющие независимым друг от друга процессам отображать одну и ту же область памяти в свои собственные адресные пространства. Доступ к данным, совместно используемым несколькими процессами, обычно требует синхронизации, так же как и доступ к совместно используемым данным из нескольких потоков. Если атрибут *process-shared* установлен в значение `PTHREAD_PROCESS_SHARED`, следовательно, мьютекс размещается в области памяти, общей для нескольких процессов, и может использоваться для их синхронизации.

Получить значение атрибута *process-shared* из структуры `pthread_mutexattr_t` можно с помощью функции `pthread_mutexattr_getpshared`. Чтобы изменить значение этого атрибута, следует использовать функцию `pthread_mutexattr_setpshared`.

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                 int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Атрибут *process-shared* позволяет библиотеке `pthread` выбрать более оптимальную реализацию мьютекса, когда этот атрибут имеет значение `PTHREAD_PROCESS_PRIVATE`, используемое по умолчанию для многопоточных приложений. Тем самым можно гарантировать использование более ресурсоемкой реализации, только когда мьютексы совместно используются несколькими процессами.

Атрибут *robust* имеет значение, когда мьютекс совместно используется несколькими процессами. Он предназначен для решения проблемы восстановления, когда процесс завершается, удерживая мьютекс. В таких ситуациях мьютекс остается запертым и восстановить его — непростая задача. Потоки в других процессах, приостановленные на мьютексе, окажутся заблокированными навечно.

Получить значение атрибута *robust* можно с помощью функции `pthread_mutexattr_getrobust`, а установить — с помощью функции `pthread_mutexattr_setrobust`.

```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict attr,
                                int *restrict robust);

int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                               int robust);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Атрибут `robust` может иметь одно из двух значений. Значение по умолчанию — `PTHREAD_MUTEX_STALLED` — означает, что при завершении процесса, удерживающего мьютекс, никаких специальных действий предприниматься не будет. В этом случае использование мьютекса может привести к непредсказуемым результатам, а приложения, ожидающие на мьютексе, фактически «зависнут». Другим возможным значением является `PTHREAD_MUTEX_ROBUST`. Это значение позволит потоку, заблокированному в вызове `pthread_mutex_lock`, приобрести блокировку, когда другой процесс, удерживающий ее, завершится, но при этом `pthread_mutex_lock` вернет `EOWNERDEAD` вместо 0. Приложения могут использовать это специальное значение как признак, что им необходимо восстановить состояние, защищаемое мьютексом, если это возможно (что это за состояние и как его восстановить, зависит исключительно от приложения). Обратите внимание, что код `EOWNERDEAD` в действительности не является признаком ошибки, потому что вызывающий поток все-таки приобретает блокировку.

При использовании надежных (`robust`) мьютексов изменяется и порядок использования функции `pthread_mutex_lock`, потому что в этом случае необходимо проверять три возвращаемых значения вместо двух: успех без необходимости восстановления, успех с необходимостью восстановления и неудача. Однако при использовании обычных «ненадежных» мьютексов можно продолжать проверять только успех или неудачу.

Из четырех платформ, рассматриваемых в этой книге, только Linux 3.2.0 поддерживает атрибут `robust`. Solaris 10 поддерживает надежные (`robust`) мьютексы только в своей библиотеке Solaris threads library (подробности см. на странице руководства `mutex_init(3C)`). Однако в Solaris 11 надежные мьютексы поддерживаются самой системой.

Если состояние приложения нельзя восстановить, мьютекс окажется в состоянии, непригодном для дальнейшего использования, когда поток освободит его. Чтобы предотвратить это, поток должен вызвать функцию `pthread_mutex_consistent` перед освобождением мьютекса, показав тем самым, что данные, защищаемые мьютексом, находились в непротиворечивом состоянии перед освобождением мьютекса.

```
#include <pthread.h>
int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Если поток освободит мьютекс, не вызвав предварительно `pthread_mutex_consistent`, другие потоки, заблокированные в попытке приобрести мьютекс, получат код ошибки `ENOTRECOVERABLE`. После этого мьютекс уже нельзя будет использовать. Вызов `pthread_mutex_consistent` перед освобождением мьютекса обеспечивает его нормальную работу и возможность дальнейшего использования.

Атрибут `type` позволяет указать тип мьютекса. Стандарт POSIX.1 определяет четыре типа.

PTHREAD_MUTEX_NORMAL Стандартный мьютекс, который не производит дополнительных проверок на наличие ошибок или тупиковых ситуаций.

PTHREAD_MUTEX_ERRORCHECK Мьютексы этого типа производят проверку наличия ошибок.

PTHREAD_MUTEX_RECURSIVE Мьютексы этого типа позволяют одному и тому же потоку многократно запирать мьютекс, не отпирая его. Рекурсивные мьютексы содержат счетчик, хранящий количество запираний мьютекса. Мьютекс будет освобожден, только когда количество отпираний совпадет с количеством запираний. Так, если имеется рекурсивный мьютекс, запертый дважды, и вы отперли его один раз, мьютекс останется заблокированным, пока вы не отопрете его второй раз.

PTHREAD_MUTEX_DEFAULT Мьютексы этого типа могут использоваться для назначения семантики мьютекса по умолчанию. Реализации могут самостоятельно определять, какому из трех предыдущих типов соответствует данный тип мьютекса. Так, например, в Linux 3.2.0 этот тип мьютекса соответствует типу PTHREAD_MUTEX_NORMAL, а в FreeBSD 8.0 — типу PTHREAD_MUTEX_ERRORCHECK.

Поведение мьютексов этих четырех типов показано в табл. 12.4. Колонка «Попытка отпирания другим потоком» соответствует ситуации, когда производится попытка отпирания мьютекса, запертого другим потоком. Колонка «Попытка отпирания незапертого мьютекса» соответствует ситуации, когда поток пытается отпереть незапертый мьютекс, что обычно объясняется ошибкой в алгоритме.

Таблица 12.4. Поведение мьютексов различного типа

Тип	Повторное запирание без отпирания	Попытка отпирания другим потоком	Попытка отпирания незапертого мьютекса
PTHREAD_MUTEX_NORMAL	Тупиковая ситуация	Не определено	Не определено
PTHREAD_MUTEX_ERRORCHECK	Возвращает код ошибки	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_RECURSIVE	Допускается	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_DEFAULT	Не определено	Не определено	Не определено

Получить значение атрибута *type* можно с помощью функции `pthread_mutexattr_gettype`, а изменить — с помощью `pthread_mutexattr_settype`.

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                             int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

В разделе 11.6.6 уже говорилось, что мьютекс используется, чтобы защитить состояние, ассоциированное с переменной состояния. Прежде чем заблокировать поток, функции `pthread_cond_wait` и `pthread_cond_timedwait` отпирают мьютекс, ассоциированный с переменной состояния. Это позволяет другим потокам запирать мьютекс, изменять состояние, отпирать мьютекс и подавать сигнал об изменении состояния. Поскольку перед изменением состояния мьютекс должен быть захвачен, было бы неправильно использовать для этой цели рекурсивные мьютексы. Если рекурсивный мьютекс был заперт несколько раз, а затем передан функции `pthread_cond_wait`, изменение состояния не будет замечено, потому что, отпирая мьютекс, функция `pthread_cond_wait` не освобождает его.

Рекурсивные мьютексы удобны, когда необходимо адаптировать существующие функции для работы в многопоточной среде, но нельзя изменять прототипы функций, чтобы сохранить совместимость с существующим программным обеспечением. Однако использование рекурсивных блокировок имеет свои особенности, поэтому их следует применять, только когда нет иного выхода.

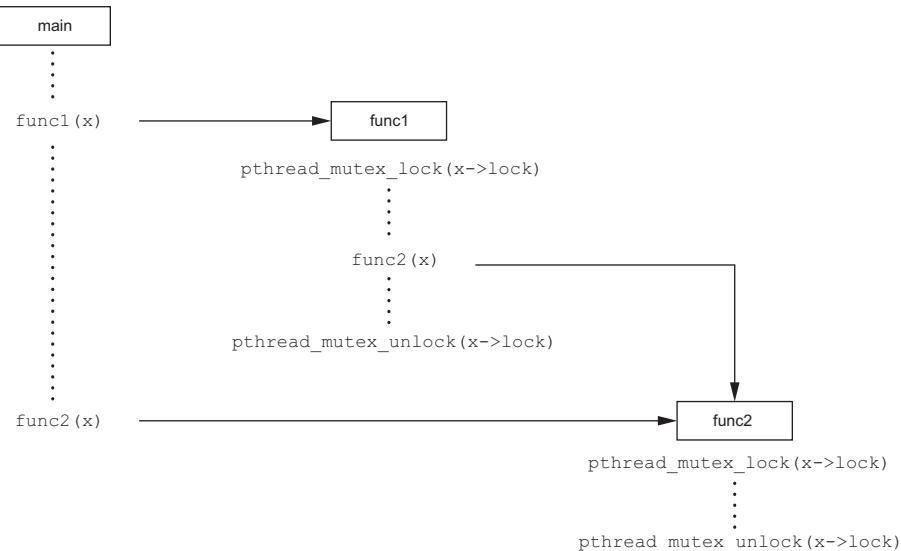
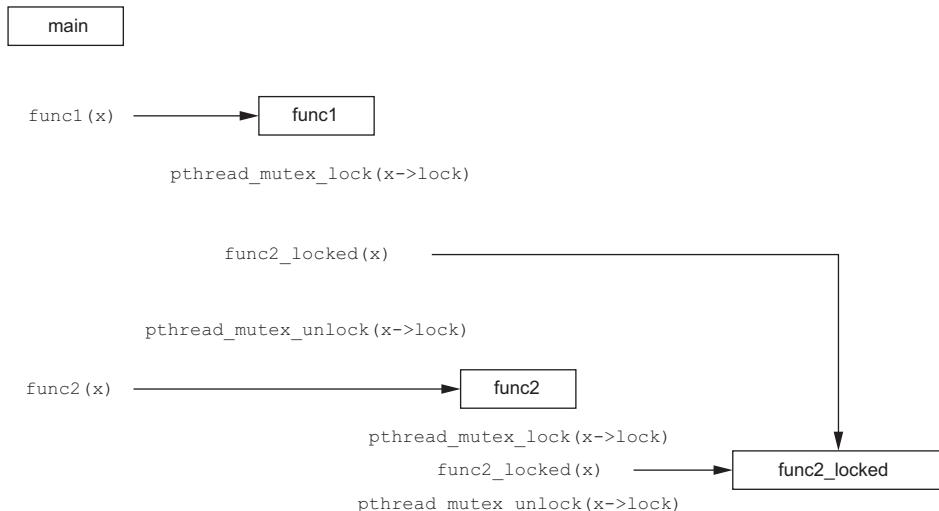
Пример

На рис. 12.1 изображена ситуация, когда использование рекурсивного мьютекса, казалось бы, является решением проблемы адаптации функций для работы в многопоточной среде. Допустим, что `func1` и `func2` — существующие библиотечные функции, и их прототипы не могут быть изменены из-за необходимости сохранения совместимости с программным обеспечением, которое не может подвергаться изменениям.

Чтобы сохранить прототипы функций без изменения, мы внедряем мьютекс в структуру данных, адрес которой (`x`) передается функциям в виде аргумента. Это возможно, только если размещением структуры в памяти занимается отдельная библиотечная функция, благодаря которой приложение ничего не знает о размере структуры (логично предположить, что в результате внедрения мьютекса в структуру ее размер увеличится).

Возможно, что структура изначально была определена с дополнительным зарезервированным объемом, и это позволит без труда добавить в нее поле с мьютексом. К сожалению, большинство программистов лишены дара предсказывать будущее, поэтому такая практика распространена не очень широко.

Если обе функции должны выполнять некоторые действия со структурой и существует вероятность, что они будут вызываться одновременно из нескольких потоков, то `func1` и `func2` должны запирать мьютекс перед выполнением действий с данными. Если функция `func1` должна вызывать `func2`, при использовании нерекурсивного мьютекса приложение легко может попасть в тупиковую ситуацию. Нерекурсивный мьютекс можно было бы использовать, освобождая его перед вызовом `func2` из `func1` и запирая вновь после возврата из `func2`, но при таком подходе появляется некий интервал времени, в течение которого другой поток может захватить мьютекс и внести нежелательные изменения в данные. Такое решение может быть неприемлемо в зависимости от того, какие данные защищены мьютексом.

**Рис. 12.1.** Использование рекурсивной блокировки**Рис. 12.2.** Отказ от использования рекурсивной блокировки

На рис. 12.2 показан альтернативный вариант решения той же проблемы — без использования рекурсивного мьютекса. Мы можем оставить прототипы функций `func1` и `func2` неизменными и отказаться от использования рекурсивного мьютекса за счет реализации скрытой от приложений версии функции `func2` — `func2_locked`. Перед вызовом функции `func2_locked` мьютекс, внедренный в структуру

данных, которая передается в аргументе, должен быть заперт. Тело `func2_locked` представляет собой копию прежней `func2`, а сама `func2` теперь просто запирает мьютекс, вызывает `func2_locked` и затем отпирает мьютекс.

Если бы не было требования неизменности прототипов библиотечных функций, можно было бы добавить в каждую из функций дополнительный аргумент, указывающий, был ли заблокирован доступ к структуре в вызывающей функции. Однако прототипы библиотечных функций лучше оставлять без изменения, если это возможно, чем засорять код особенностями реализации.

Стратегия использования заблокированных и незаблокированных версий функций обычно применима только в простых ситуациях. В сложных случаях (например, когда библиотечная функция вызывает функцию, расположенную за пределами библиотеки, которая, в свою очередь, может обращаться к библиотечным функциям) не остается ничего другого, как полагаться на рекурсивные блокировки.

Пример

Программа в листинге 12.2 иллюстрирует еще один случай, когда необходимо использовать рекурсивный мьютекс. Здесь у нас имеется функция `timeout`, которая позволяет запланировать запуск другой функции на определенное время. Допустим, что поток не является дорогостоящим ресурсом, тогда для каждого из запланированных тайм-аутов можно запускать отдельный поток. Он будет ожидать указанного момента времени и затем вызывать запрошеннную функцию.

Проблема возникает, когда приложение не может создать новый поток или когда запрошенный момент запуска функции уже прошел. В таких случаях мы просто вызываем требуемую функцию в текущем контексте. Поскольку функция все время пытается установить одну и ту же блокировку, при использовании нерекурсивной блокировки может возникнуть тупиковая ситуация.

Листинг 12.2. Использование рекурсивного мьютекса

```
#include "apue.h"
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

extern int makethread(void * (*)(void *), void *);

struct to_info {
    void          (*to_fn)(void *); /* функция */
    void          *to_arg;           /* аргумент */
    struct timespec to_wait;       /* время запуска */
};

#define SECTONSEC 1000000000 /* наносекунд в секунде */

#if !defined(CLOCK_REALTIME) || defined(BSD)
#define clock_nanosleep(ID, FL, REQ, REM) nanosleep((REQ), (REM))
#endif

#ifndef CLOCK_REALTIME
#define CLOCK_REALTIME 0
```

```
#define USECTONSEC 1000 /* микросекунд в микросекунде */

void
clock_gettime(int id, struct timespec *tsp)
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    tsp->tv_sec = tv.tv_sec;
    tsp->tv_nsec = tv.tv_usec * USECTONSEC;
}

#endif

void *
timeout_helper(void *arg)
{
    struct to_info *tip;

    tip = (struct to_info *)arg;
    clock_nanosleep(CLOCK_REALTIME, 0, &tip->to_wait, NULL);
    (*tip->to_fn)(tip->to_arg);
    free(arg);
    return(0);
}

void
timeout(const struct timespec *when, void (*func)(void *), void *arg)
{
    struct timespec now;
    struct to_info *tip;
    int err;

    clock_gettime(CLOCK_REALTIME, &now);
    if ((when->tv_sec > now.tv_sec) ||
        (when->tv_sec == now.tv_sec && when->tv_nsec > now.tv_nsec)) {
        tip = malloc(sizeof(struct to_info));
        if (tip != NULL) {
            tip->to_fn = func;
            tip->to_arg = arg;
            tip->to_wait.tv_sec = when->tv_sec - now.tv_sec;
            if (when->tv_nsec >= now.tv_nsec) {
                tip->to_wait.tv_nsec = when->tv_nsec - now.tv_nsec;
            } else {
                tip->to_wait.tv_sec--;
                tip->to_wait.tv_nsec = SECTONSEC - now.tv_nsec +
                    when->tv_nsec;
            }
            err = makethread(timeout_helper, (void *)tip);
            if (err == 0)
                return;
            else
                free(tip);
        }
    }

/*
 * В эту точку управление переходит, если (а) when <= now,
 * или (б) вызов функции malloc терпит неудачу,
 * или (в) невозможно создать новый поток,
 */
```

```

    * поэтому мы просто вызываем требуемую функцию.
    */
    (*func)(arg);
}

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void
retry(void *arg)
{
    pthread_mutex_lock(&mutex);

    /* выполнить действия, предусмотренные функцией ... */

    pthread_mutex_unlock(&mutex);
}

int
main(void)
{
    int             err, condition, arg;
    struct timespec when;

    if ((err = pthread_mutexattr_init(&attr)) != 0)
        err_exit(err, "ошибка вызова функции pthread_mutexattr_init");
    if ((err = pthread_mutexattr_settype(&attr,
                                         PTHREAD_MUTEX_RECURSIVE)) != 0)
        err_exit(err, "невозможно установить рекурсивный тип мьютекса");
    if ((err = pthread_mutex_init(&mutex, &attr)) != 0)
        err_exit(err, "невозможно создать рекурсивный мьютекс");

    /* продолжить обработку ... */

    pthread_mutex_lock(&mutex);

    /*
     * Убедиться, что переменная состояния находится под защитой блокировки,
     * чтобы обеспечить атомарность проверки и вызова функции timeout.
     */
    if (condition) {
        /* рассчитать время запуска функции "when" */
        clock_gettime(CLOCK_REALTIME, &when);
        when.tv_sec += 10; /* через 10 секунд от текущего момента */
        timeout(&when, retry, (void *)((unsigned long)arg));
    }

    /* продолжить обработку ... */

    pthread_mutex_unlock(&mutex);

    exit(0);
}

```

Для создания потоков в обособленном состоянии мы воспользовались функцией `makethread` из листинга 12.1. Нам необходимо запланировать запуск функции на будущее, но мы не желаем ждать завершения потока.

Для задержки можно было бы воспользоваться функцией `sleep`, но она может отмерять интервалы времени с точностью лишь до секунды. Если нужна задержка на промежуток времени, отличный от целого числа секунд, следует использовать функцию `nanosleep` или `clock_nanosleep`, каждая из которых обеспечивает более высокое разрешение времени.

Для случая использования программы в системах, не определяющих константу `CLOCK_REALTIME`, мы определили свою версию `clock_nanosleep`, реализовав ее на основе `nanosleep`. FreeBSD 8.0 определяет этот символ для поддержки `clock_gettime` и `clock_settime`, но она не поддерживает `clock_nanosleep` (в настоящее время эта функция поддерживается только в Linux 3.2.0 и Solaris 10).

Кроме того, для использования программы в системах, не определяющих константу `CLOCK_REALTIME`, мы реализовали свою версию `clock_gettime`, которая вызывает `gettimeofday` и преобразует микросекунды в наносекунды.

Функция, вызывающая `timeout`, должна удерживать мьютекс на время проверки условия и планирования функции `retry`, чтобы обеспечить атомарность этих двух операций. Функция `retry` пытается запереть тот же самый мьютекс. Если бы в программе использовался нерекурсивный мьютекс, прямой вызов `retry` из функции `timeout` приводил бы к тупиковой ситуации.

12.4.2. Атрибуты блокировок чтения-записи

Блокировки чтения-записи, подобно мьютексам, также имеют атрибуты. Для инициализации структуры `pthread_rwlockattr_t` используется функция `pthread_rwlockattr_init`, а для ее разрушения — функция `pthread_rwlockattr_destroy`.

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Единственный атрибут, поддерживаемый блокировками чтения-записи, — это атрибут *process-shared*, полностью идентичный аналогичному атрибуту мьютексов. Как и в случае с мьютексами, для обслуживания атрибута *process-shared* блокировок чтения-записи используется пара функций: `pthread_rwlockattr_getpshared` и `pthread_rwlockattr_setpshared`.

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                  int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Хотя стандарт POSIX определяет всего один атрибут для блокировок чтения-записи, реализации могут свободно добавлять собственные нестандартные атрибуты.

12.4.3. Атрибуты переменных состояния

Стандарт Single UNIX Specification в настоящее время определяет два атрибута для переменных состояния: *process-shared* и *clock*. Как и для других объектов атрибутов, для инициализации и разрушения атрибутов переменных состояния существует своя пара функций.

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Атрибут *process-shared* обеспечивает то же поведение, что и в других объектах синхронизации. Он управляет доступностью переменных состояния из других процессов. Получить текущее значение атрибута *process-shared* можно с помощью функции *pthread_condattr_getpshared*, а изменить — с помощью *pthread_condattr_setpshared*.

```
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Атрибут *clock* управляет выбором часов для вычисления аргумента тайм-аута (*tsptr*) функции *pthread_cond_timedwait*. Допустимыми значениями являются идентификаторы часов, перечисленные в табл. 6.7. Получить идентификатор часов, который будет использоваться функцией *pthread_cond_timedwait* для переменной состояния с объектом атрибутов *pthread_condattr_t*, можно с помощью *pthread_condattr_getclock*, а изменить — с помощью *pthread_condattr_setclock*.

```
#include <pthread.h>
int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                             clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr, clockid_t clock_id);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Как ни странно, но стандарт Single UNIX Specification не определяет атрибут *clock* для остальных объектов синхронизации, имеющих функцию ожидания с таймом.

12.4.4. Атрибуты барьеров

Барьеры тоже имеют атрибуты. Инициализировать структуру атрибутов барьера можно с помощью функции `pthread_barrierattr_init`, а разрушить — с помощью функции `pthread_barrierattr_destroy`.

```
#include <pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

В настоящее время для барьеров определен единственный атрибут *process-shared*, который управляет доступностью барьеров из других процессов. Как и в случаях с другими объектами атрибутов, у нас имеется одна функция для получения значения атрибута (`pthread_barrierattr_getpshared`) и одна функция для его изменения (`pthread_barrierattr_setpshared`).

```
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
                                   int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

Обе возвращают 0 в случае успеха, код ошибки — в случае неудачи

Атрибут *process-shared* может иметь либо значение `PTHREAD_PROCESS_SHARED` (барьер доступен другим процессам), либо значение `PTHREAD_PROCESS_PRIVATE` (барьер доступен только потокам в процессе, инициализировавшем его).

12.5. Реентерабельность

В разделе 10.6 обсуждались обработчики сигналов и реентерабельные функции. Потоки в чем-то похожи на обработчики сигналов, когда дело касается реентерабельности. Как и в случае с обработчиками сигналов, в многопоточных приложениях вполне вероятна ситуация, когда одну и ту же функцию одновременно вызывают несколько потоков.

Функции, которые можно безопасно вызывать одновременно из нескольких потоков, называются *безопасными в многопоточной среде*, или *потокобезопасными (threadsafe)*. Все функции, определяемые стандартом Single UNIX Specification,

являются потокобезопасными, за исключением перечисленных в табл. 12.5. Кроме того, функции `ctermid` и `tmpnam` не гарантируют безопасность в многопоточной среде, если им в аргументе передается пустой указатель. Аналогично, функции `wcrtomb` и `wcsrtombs` не гарантируют безопасность в многопоточной среде, если им в аргументе `mbstate_t` передается пустой указатель.

Таблица 12.5. Функции, безопасность которых в многопоточной среде не гарантировается стандартом POSIX.1

<code>basename</code>	<code>getchar_unlocked</code>	<code>getservent</code>	<code>putc_unlocked</code>
<code>catgets</code>	<code>getdate</code>	<code>getutxent</code>	<code>putchar_unlocked</code>
<code>crypt</code>	<code>getenv</code>	<code>getutxid</code>	<code>putenv</code>
<code>dbm_clearerr</code>	<code>getgrent</code>	<code>getutxline</code>	<code>pututxline</code>
<code>dbm_close</code>	<code>getgrgid</code>	<code>gmtime</code>	<code>rand</code>
<code>dbm_delete</code>	<code>getgrnam</code>	<code>hcreate</code>	<code>readdir</code>
<code>dbm_error</code>	<code>gethostent</code>	<code>hdestroy</code>	<code>setenv</code>
<code>dbm_fetch</code>	<code>getlogin</code>	<code>hsearch</code>	<code>setgrent</code>
<code>dbm_firstkey</code>	<code>getnetbyaddr</code>	<code>inet_ntoa</code>	<code>setkey</code>
<code>dbm_nextkey</code>	<code>getnetbyname</code>	<code>l64a</code>	<code>setpwent</code>
<code>dbm_open</code>	<code>getnetent</code>	<code>lgamma</code>	<code>setutxent</code>
<code>dbm_store</code>	<code> getopt</code>	<code>lgammaf</code>	<code>strerror</code>
<code>dirname</code>	<code>getprotobynam</code>	<code>lgammal</code>	<code>strsignal</code>
<code>dlerror</code>	<code>getprotobynumber</code>	<code>localeconv</code>	<code>strtok</code>
<code>drand48</code>	<code>getprotoent</code>	<code>localtime</code>	<code>system</code>
<code>encrypt</code>	<code>getpwent</code>	<code>lrand48</code>	<code>ttyname</code>
<code>endgrent</code>	<code>getpwnam</code>	<code>mrand48</code>	<code>unsetenv</code>
<code>endpwent</code>	<code>getpwuid</code>	<code>nftw</code>	<code>wcstombs</code>
<code>endutxent</code>	<code>getservbyname</code>	<code>nl_langinfo</code>	<code>wctomb</code>
<code>getc_unlocked</code>	<code>getservbyport</code>	<code>ptsname</code>	

Реализации, которые поддерживают потокобезопасные функции, определяют в заголовочном файле `<unistd.h>` символ `_POSIX_THREAD_SAFE_FUNCTIONS`. Кроме того, для проверки поддержки безопасных функций во время выполнения можно вызывать функцию `sysconf` с аргументом `_SC_THREAD_SAFE_FUNCTIONS`. До выхода версии 4 стандарта Single UNIX Specification все реализации, отвечающие требованиям XSI, обязаны были обеспечить поддержку потокобезопасных функций. Однако в версии SUSv4 поддержка потокобезопасных функций стала обязательной для всех POSIX-совместимых систем.

Кроме потокобезопасных функций, реализации предоставляют альтернативные потокобезопасные версии некоторых небезопасных функций POSIX.1. Эти без-

опасные версии перечислены в табл. 12.6. Функции в табл. 12.6 называются подобно их небезопасным аналогам, но с добавлением символов `_r` в конце имени, что указывает на их реентерабельность. Многие функции не являются безопасными, потому что возвращают результаты в буфере, размещенном статически. Они делаются безопасными за счет изменения интерфейса — для этого нужно, чтобы вызывающая программа предоставила свой буфер для результатов.

Таблица 12.6. Альтернативные версии функций, безопасных в многопоточной среде

<code>getgrgid_r</code>	<code>localtime_r</code>
<code>getgrnam_r</code>	<code>readdir_r</code>
<code>getlogin_r</code>	<code>strerror_r</code>
<code>getpwnam_r</code>	<code>strtok_r</code>
<code>getpwuid_r</code>	<code>ttynname_r</code>
<code>gmtime_r</code>	

Если функция является реентерабельной по отношению к потокам, ее называют потокобезопасной. Но это не говорит о том, что функция реентерабельна по отношению к обработчикам сигналов. Если функцию можно безопасно вызывать из обработчиков асинхронных сигналов, такая функция называется *безопасной в контексте обработки асинхронных сигналов*. Функции, безопасные по отношению к обработчикам сигналов, перечислялись в табл. 10.3 при обсуждении реентерабельных функций (раздел 10.6).

В дополнение к функциям, перечисленным в табл. 12.6, стандарт POSIX.1 определяет еще несколько функций, поддерживающих безопасный способ управления объектами `FILE` в многопоточной среде. Чтобы заблокировать доступ к определенному объекту `FILE`, можно использовать функции `flockfile` и `ftrylockfile`. Эта блокировка является рекурсивной: ее можно повторно установить, не опасаясь попасть в тупиковую ситуацию. Стандарт не оговаривает точную реализацию таких блокировок, но требует, чтобы все функции стандартной библиотеки ввода/вывода, работающие с объектом `FILE`, вели себя так, будто обращаются к функциям `flockfile` и `funlockfile`.

```
#include <stdio.h>
int ftrylockfile(FILE *fp);
void flockfile(FILE *fp);
void funlockfile(FILE *fp);
```

Возвращает 0 в случае успеха, ненулевое значение —
при невозможности установки блокировки

Хотя функции стандартной библиотеки ввода/вывода могут быть реализованы, как потокобезопасные (в смысле безопасности их собственных внутренних струк-

тур данных), тем не менее блокировку доступа лучше выполнять в самом приложении. Это позволит приложениям производить серии вызовов функций в виде атомарных последовательностей. Разумеется, при обслуживании многочисленных объектов `FILE` следует остерегаться потенциальных тупиковых ситуаций и очень тщательно продумывать порядок захвата блокировок.

Если функции стандартной библиотеки ввода/вывода устанавливают свои собственные блокировки, можно столкнуться с серьезным снижением производительности при выполнении посимвольного ввода/вывода. В этой ситуации блокировка устанавливается и снимается для каждого прочитанного или записанного символа. Чтобы избежать этой проблемы, библиотека предоставляет версии функций посимвольного ввода/вывода, которые не устанавливают блокировку.

```
#include <stdio.h>
int getchar_unlocked(void);
int getc_unlocked(FILE *fp);
```

Обе возвращают следующий символ в случае успеха,
EOF — в случае ошибки

```
int putchar_unlocked(int c);
int putc_unlocked(int c, FILE *fp);
```

Обе возвращают значение аргумента *c* в случае успеха,
EOF — в случае ошибки

Эти четыре функции должны окружаться вызовами `flockfile` (или `ftrylockfile`) и `funlockfile`. Иначе можно получить непредсказуемые результаты (так же, как и в случае несинхронизированного доступа к данным из нескольких потоков выполнения).

После установки блокировки на объект `FILE` и до ее снятия можно производить вызовы функций ввода/вывода. Накладные расходы на установку и снятие блокировки могут в значительной степени компенсироваться объемом прочитанных или записанных данных.

Пример

В листинге 12.3 приводится пример возможной реализации функции `getenv` (раздел 7.9). Эта версия не является реентерабельной. Если произойдет одновременное обращение к функции из двух потоков, они получат неверные данные, потому что возвращаемая строка сохраняется в статическом буфере, совместно используемом всеми потоками, вызывающими функцию `getenv`.

Листинг 12.3. Нереентерабельная версия функции `getenv`

```
#include <limits.h>
#include <string.h>

#define MAXSTRINGSZ 4096

static char envbuf[MAXSTRINGSZ];
```

```

extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {
            strncpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-1);
            return(envbuf);
        }
    }
    return(NULL);
}

```

В листинге 12.4 показана реентерабельная версия функции `getenv` с именем `getenv_r`. Она использует функцию `pthread_once` (описывается в разделе 12.6), чтобы гарантировать, что в ходе выполнения процесса функция `thread_init` будет вызвана единственный раз, независимо от количества потоков выполнения, попытавшихся одновременно вызвать `getenv_r`. Подробнее о функции `pthread_once` рассказывается в разделе 12.6.

Листинг 12.4. Реентерабельная (потокобезопасная) версия функции `getenv`

```

#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

extern char **environ;

pthread_mutex_t env_mutex;

static pthread_once_t init_done = PTHREAD_ONCE_INIT;

static void
thread_init(void)
{
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}

int
getenv_r(const char *name, char *buf, int buflen)
{
    int i, len, olen;

    pthread_once(&init_done, thread_init);
    len = strlen(name);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {

```

```

olen = strlen(&environ[i][len+1]);
if (olen >= buflen) {
    pthread_mutex_unlock(&env_mutex);
    return(ENOSPC);
}
strcpy(buf, &environ[i][len+1]);
pthread_mutex_unlock(&env_mutex);
return(0);
}
pthread_mutex_unlock(&env_mutex);
return(ENOENT);
}

```

Чтобы `getenv_r` стала реентерабельной, мы изменили ее интерфейс, и теперь вызывающая программа должна передать ей свой собственный буфер. В результате каждый поток будет использовать отдельный буфер, что исключит возможность наложения одних данных на другие. Но этого недостаточно, чтобы сделать функцию `getenv_r` безопасной в многопоточной среде. Чтобы сделать ее безопасной, нужно запретить возможность изменения среды окружения на время, пока выполняется поиск запрошенной строки. Для организации доступа к списку переменных окружения из функций `getenv_r` и `putenv` можно использовать мьютекс. Также можно использовать блокировки чтения-записи, чтобы разрешить одновременный вызов функции `getenv_r` из нескольких потоков, но, скорее всего, это не принесет существенной выгоды по двум причинам. Во-первых, объем среды окружения обычно не очень велик, и поэтому мьютекс во время поиска будет засириваться на достаточно короткий промежуток времени. Во-вторых, вызовы функций `putenv` и `getenv` производятся очень редко, поэтому, повышая производительность этих двух функций, мы не увеличиваем производительность всего приложения.

Если мы сделаем функцию `getenv_r` безопасной в многопоточной среде, это вовсе не означает, что она станет безопасной в контексте обработчиков сигналов. При использовании нерекурсивного мьютекса есть риск возникновения тупиковой ситуации, если произойдет вызов `getenv_r` из обработчика сигнала. Если сигнал доставлен в тот момент, когда поток находился внутри `getenv_r` и мьютекс `env_mutex` уже был заперт, повторная попытка запереть мьютекс будет заблокирована, что приведет поток к тупиковой ситуации. Поэтому, чтобы воспрепятствовать изменению данных из других потоков и предотвратить тупиковые ситуации в обработчиках сигналов, необходимо использовать рекурсивные мьютексы. Но тут есть еще одна проблема — функции библиотеки `pthreads` не гарантируют безопасность в контексте обработки асинхронных сигналов, то есть мы не можем их использовать, чтобы сделать безопасными другие функции.

12.6. Локальные данные потоков

Локальные данные потока — это механизм хранения и поиска данных, связанных только с конкретным потоком выполнения. Локальные данные нужны, чтобы каждый поток мог обладать некоторым набором данных, принадлежащих ему одному, и не беспокоиться о синхронизации при работе с ними.

К созданию модели совместного использования ресурсов и атрибутов в многопоточных приложениях были приложены усилия многих опытных специалистов. Тогда зачем нам могут понадобиться интерфейсы, препятствующие использованию этой модели? Тому есть две причины.

Во-первых, иногда необходимо сохранять некоторые данные, специфичные для конкретного потока. Поскольку нет никакой гарантии, что идентификаторы потока представлены небольшими последовательными целыми числами, мы не можем просто завести массив с данными для каждого потока, который индексируется идентификатором потока. Но даже если бы это было возможно, все равно не было бы никаких гарантий, что данные одного потока не перемешаются с данными другого потока.

Вторая причина: механизм организации локальных данных потока предоставляет возможность адаптации интерфейсов процессов к многопоточной среде. Типичный пример такой адаптации — переменная `errno` (раздел 1.7). Старые интерфейсы (которые были определены еще до появления понятия потоков) рассматривают `errno` как целочисленную переменную с глобальной областью видимости в пределах процесса. Системные вызовы и библиотечные функции в случае неудачи записывают в эту переменную код ошибки. Чтобы позволить потокам использовать те же самые системные вызовы и библиотечные функции, переменная `errno` была переопределена как локальная переменная потока. Поэтому теперь, когда поток вызывает функцию, изменяющую значение `errno`, он уже не оказывает влияния на другие потоки в процессе.

Не забывайте, что все потоки в процессе имеют доступ ко всему адресному пространству процесса. И нет никакого способа предотвратить доступ к данным одного потока из другого, за исключением использования регистров процессора. Это утверждение истинно даже для локальных данных потока. Несмотря на то что реализация, в принципе, не может воспрепятствовать доступу к данным, все же существуют функции для работы с локальными данными потока, которые способствуют продвижению модели с раздельными данными потоков.

Перед размещением локальных данных потока необходимо создать ключ, который будет идентифицировать данные. Этот ключ будет использоваться для получения доступа к локальным данным потока. Создается такой ключ вызовом функции `pthread_key_create`.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Созданный ключ сохраняется по адресу `keyp`. Один и тот же ключ может использоваться различными потоками в процессе, но каждый поток будет ассоциировать с ключом отдельный набор локальных данных. После создания ключа адрес локальных данных для каждого потока устанавливается равным `NULL`.

Кроме того, функция `pthread_key_create` может связать с созданным ключом функцию-деструктор. Если адрес локальных данных при завершении потока име-

ет ненулевое значение, вызывается функция-деструктор, которой в аргументе передается адрес области с локальными данными потока. Если в аргументе *destroyctor* передается пустой указатель, это означает, что для данного ключа не предусматривается вызов деструктора. Когда поток завершает работу вызовом функции `pthread_exit` или возвращает управление из запускающей процедуры, вызывается деструктор. Но если поток вызывает функцию `exit`, `_exit`, `_Exit`, `abort` или завершает работу аварийно, деструктор не вызывается.

Как правило, для выделения памяти под локальные данные потоки используют функцию `malloc`. Функция-деструктор обычно освобождает эту память. Если поток завершит работу без освобождения памяти, эта область памяти будет потеряна для процесса.

Поток может создать несколько ключей для своих данных. Каждый ключ может быть ассоциирован с деструктором. Это могут быть отдельные деструкторы для каждого из ключей или, наоборот, все ключи могут быть ассоциированы с одной и той же функцией-деструктором. Каждая реализация операционной системы может накладывать свои ограничения на количество ключей, создаваемых процессом (`PTHREAD_KEYS_MAX` в табл. 12.1).

Порядок вызова деструктора при завершении потока зависит от реализации. В деструкторе допускается вызывать функции, которые могут создавать новые локальные данные потока и ассоциировать их с ключом. После вызова всех деструкторов система проверяет, не сохранились ли какие-либо непустые указатели на локальные данные потока, и если таковые будут обнаружены, деструкторы будут вызваны снова. Этот процесс будет повторяться снова и снова, пока не будут обнулены все указатели на локальные данные или не будет достигнуто максимально возможное количество итераций `PTHREAD_DESTRUCTOR_ITERATIONS` (см. табл. 12.1).

Разорвать связь ключа с локальными данными для всех потоков можно, вызвав функцию `pthread_key_delete`.

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t *key);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Обратите внимание, что вызов `pthread_key_delete` не приводит к вызову деструктора, ассоциированного с ключом. Чтобы освободить память, занимаемую локальными данными потока, мы должны предусмотреть все необходимые действия в самом приложении.

Размещая новый ключ, следует побеспокоиться, чтобы он не изменился в процессе инициализации из другого потока. Код, подобный приведенному ниже, может привести к тому, что функция `pthread_key_create` будет вызвана одновременно из нескольких потоков:

```
void destructor(void *);
pthread_key_t key;
```

```
int init_done = 0;

int
threadfunc(void *arg)
{
    if (!init_done) {
        init_done = 1;
        err = pthread_key_create(&key, destructor);
    }
    ...
}
```

В зависимости от того, как система планирует выполнение потоков, одни потоки могут увидеть одно значение ключа, другие — другое. Решение проблемы заключается в использовании функции `pthread_once`.

```
#include <pthread.h>

pthread_once_t initflag = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Параметр `initflag` должен быть глобальной или статической переменной, инициализированной значением `PTHREAD_ONCE_INIT`.

Система гарантирует, что функция инициализации `initfn` будет вызвана всего один раз при самом первом обращении к `pthread_once`, независимо от того, сколько раз вызывается функция `pthread_once`. Соответственно правильный способ создания ключа выглядит так:

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;

void
thread_init(void)
{
    err = pthread_key_create(&key, destructor);
}

int
threadfunc(void *arg)
{
    pthread_once(&init_done, thread_init);
    ...
}
```

После создания ключ можно ассоциировать с локальными данными потока вызовом функции `pthread_setspecific`. Чтобы по заданному ключу получить адрес области памяти с локальными данными потока, следует вызвать `pthread_getspecific`.

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);

    Возвращает указатель на область памяти с локальными данными или
    NULL, если ключ не ассоциирован с локальными данными

int pthread_setspecific(pthread_key_t key, const void *value);

    Возвращает 0 в случае успеха, код ошибки — в случае неудачи
```

Если с ключом не ассоциированы локальные данные потока, функция `pthread_getspecific` вернет NULL. Это обстоятельство можно использовать, чтобы определить, следует ли вызывать функцию `pthread_setspecific`.

Пример

В листинге 12.3 приводился пример возможной реализации функции `getenv`. Затем, в листинге 12.4, мы показали потокобезопасный вариант этой же функции. Но что делать, если нельзя изменить прикладную программу, чтобы она пользовалась новой версией функции? В подобной ситуации можно задействовать локальные данные потока, в которых будет храниться буфер для возвращаемой строки. Такой подход представлен в листинге 12.5.

Листинг 12.5. Потокобезопасная версия функции `getenv`

```
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>

#define MAXSTRINGSZ 4096

static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;

extern char **environ;

static void
thread_init(void)
{
    pthread_key_create(&key, free);
}

char *
getenv(const char *name)
{
    int     i, len;
    char   *envbuf;

    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *)pthread_getspecific(key);
    if (envbuf == NULL) {
        envbuf = malloc(MAXSTRINGSZ);
```

```

    if (envbuf == NULL) {
        pthread_mutex_unlock(&env_mutex);
        return(NULL);
    }
    pthread_setspecific(key, envbuf);
}
len = strlen(name);
for (i = 0; environ[i] != NULL; i++) {
    if ((strncmp(name, environ[i], len) == 0) &&
        (environ[i][len] == '=')) {
        strcpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-1);
        pthread_mutex_unlock(&env_mutex);
        return(envbuf);
    }
}
pthread_mutex_unlock(&env_mutex);
return(NULL);
}

```

Здесь мы использовали функцию `pthread_once`, чтобы гарантировать единственность ключа, который будет ассоциироваться с локальными данными потоков. Если функция `pthread_getspecific` возвращает пустой указатель, нужно разместить в памяти буфер и связать его с полученным ключом. В противном случае используется буфер, возвращаемый функцией `pthread_getspecific`. В деструкторе мы вызываем функцию `free`, которая освобождает память, выделенную функцией `malloc`. Деструктор будет вызван, только если поток связал указатель на локальные данные с ключом и этот указатель не является пустым.

Обратите внимание: хотя эта версия функции `getenv` является потокобезопасной, она не безопасна в контексте обработки асинхронных сигналов. Даже если сделать мьютекс рекурсивным, это не гарантирует ее безопасное использование в обработчиках сигналов, потому что она вызывает функцию `malloc`, которая сама не является безопасной в контексте обработки сигналов.

12.7. Принудительное завершение потоков

Потоки имеют два атрибута, которые не входят в состав структуры `pthread_attr_t` — атрибут *возможности принудительного завершения потока* (*cancelability state*) и атрибут *типа принудительного завершения* (*cancelability type*). Эти атрибуты определяют поведение потока в ответ на вызов функции `pthread_cancel` (раздел 11.5).

Атрибут *cancelability state* может иметь значение `PTHREAD_CANCEL_ENABLE` или `PTHREAD_CANCEL_DISABLE`. Поток может изменить значение этого атрибута вызовом функции `pthread_setcancelstate`.

```

#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);

```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

В одной атомарной операции функция `pthread_setcancelstate` изменяет значение атрибута *cancelability state* в соответствии со значением аргумента *state* и сохраняет прежнее значение атрибута по адресу, который передается в аргументе *oldstate*.

В разделе 11.5 уже говорилось, что функция `pthread_cancel` не ждет, пока поток завершит работу. По умолчанию поток продолжает работу после вызова этой функции, пока не достигнет *точки выхода*. Точка выхода — это место, где поток может обнаружить запрос на принудительное завершение и откликнуться на него. Стандарт POSIX.1 назначает точками выхода функции, перечисленные в табл. 12.7.

Таблица 12.7. Точки выхода, определяемые стандартом POSIX.1

accept	mq_timedsend	pthread_join	sendto
aio_suspend	msgrcv	pthread_testcancel	sigsuspend
clock_nanosleep	msgsnd	pwrite	sigtimedwait
close	msync	read	sigwait
connect	nanosleep	readv	sigwaitinfo
creat	open	recv	sleep
fcntl	openat	recvfrom	system
fdatasync	pause	recvmsg	tcdrain
fsync	poll	select	wait
lockf	pread	sem_timedwait	waitid
mq_receive	pselect	sem_wait	waitpid
mq_send	pthread_cond_timedwait	send	write
mq_timedreceive	pthread_cond_wait	sendmsg	writev

В момент запуска потока значение его атрибута *cancelability state* устанавливается равным `PTHREAD_CANCEL_ENABLE`. Если поток установит значение этого атрибута равным `PTHREAD_CANCEL_DISABLE`, вызов функции `pthread_cancel` не будет приводить к завершению потока. Вместо этого запрос на принудительное завершение встает в режим ожидания. Когда поток опять разрешит возможность принудительного завершения, он откликнется на ожидающий запрос в ближайшей точке выхода.

В дополнение к функциям, перечисленным в табл. 12.7, стандарт POSIX.1 определяет еще ряд функций (табл. 12.8), которые могут служить точками выхода.

Таблица 12.8. Дополнительные точки выхода, определяемые стандартом POSIX.1

access	fseeko	getwchar	putwc
catclose	fsetpos	glob	putwchar
catgets	fstat	iconv_close	readdir
catopen	fstatat	iconv_open	readdir_r

chmod	fstell	ioctl	readlink
chown	ftello	link	readlinkat
closedir	futimens	linkat	remove
closelog	fwprintf	lio_listio	rename
ctermid	fwrite	localtime	renameat
dbm_close	fwscanf	localtime_r	rewind
dbm_delete	getaddrinfo	lockf	rewinddir
dbm_fetch	getc	lseek	scandir
dbm_nextkey	getc_unlocked	lstat	scanf
dbm_open	getchar	mkdir	seekdir
dbm_store	getchar_unlocked	mkdirat	semop
dlclose	getcwd	mkdtemp	setrent
dlopen	getdate	mkfifo	sethostent
dprintf	getdelim	mkfifoat	setnetent
endgrent	getgrent	mknod	setprotoent
endhostent	getgrgid	mknodat	setpwent
endnetent	getgrgid_r	mkstemp	setservent
endprotoent	getgrnam	mktme	setutxent
endpwent	getgrnam_r	nftw	stat
endservent	gethostent	opendir	strerror
endutxent	gethostid	openlog	strerror_r
faccessat	gethostname	pathconf	strftime
fchmod	getline	pclose	symlink
fchmodat	getlogin	perror	symlinkat
fchown	getlogin_r	popen	sync
fchownat	getnameinfo	posix_fadvise	syslog
fclose	getnetbyaddr	posix_fallocate	tmpfile
fcntl	getnetbyname	posix_madvise	ttynname
fflush	getnetent	posix_openpt	ttynname_r
fgetc	getopt	posix_spawn	tzset
fgetpos	getprotobynam	posix_spawnp	ungetc
fgets	getprotobyname	posix_typed_mem_open	ungetwc
fgetwc	getprotoent	printf	unlink
fgetws	getpwent	psiginfo	unlinkat
fmtmsg	getpwnam	psignal	utimensat

fopen	getpwnam_r	pthread_rwlock_rdlock	utimes
fpathconf	getpwuid	pthread_rwlock_timedrdlock	vdprintf
fprintf	getpwuid_r	pthread_rwlock_timedwrlock	vfprintf
fputc	getservbyname	pthread_rwlock_wrlock	vfwprintf
fputs	getservbyport	putc	vprintf
fputwc	getservent	putc_unlocked	vwprintf
fputws	getutxent	putchar	wcsftime
fread	getutxid	putchar_unlocked	wordexp
freopen	getutxline	puts	wprintf
fscanf	getwc	pututxline	wscanf
fseek			

Некоторые функции из табл. 12.8, например выполняющие операции с каталогами сообщений и многобайтными символами, не обсуждаются в данной книге.

Если приложение не обращается к функциям, перечисленным в табл. 12.7 и 12.8, достаточно продолжительное время (например, при выполнении объемных вычислений), можно определить свою точку выхода с помощью функции `pthread_testcancel`.

```
#include <pthread.h>

void pthread_testcancel(void);
```

Функция `pthread_testcancel` проверяет наличие ожидающего запроса на принудительное завершение и, если таковой имеется и при этом атрибут *cancelability state* разрешает принудительное завершение, поток завершит свою работу. Но если возможность принудительного завершения потока запрещена, вызов функции `pthread_testcancel` не оказывает никакого влияния.

По умолчанию для потока устанавливается тип принудительного завершения, известный как *отложенный выход*. После вызова функции `pthread_cancel` поток не завершается немедленно, он продолжает работу, пока не достигнет ближайшей точки выхода. Изменить тип принудительного завершения можно с помощью функции `pthread_setcanceltype`.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Функция `pthread_setcanceltype` устанавливает значение атрибута в соответствии с аргументом *type* (либо `PTHREAD_CANCEL_DEFERRED`, либо `PTHREAD_CANCEL_ASYNCHRONOUS`) и возвращает предыдущее значение атрибута в переменной, на которую указывает аргумент *oldtype*.

Асинхронное завершение потока отличается от отложенного тем, что позволяет принудительно завершить поток в любой момент. В этом случае поток завершится, даже если он не достиг точки выхода.

12.8. Потоки и сигналы

Взаимодействие с сигналами может быть весьма сложным даже в однопоточных приложениях. Наличие нескольких потоков еще больше усложняет дело.

Каждый поток имеет собственную маску сигналов, но диспозиция сигналов одна для всех потоков в процессе. Это означает, что каждый отдельно взятый поток может заблокировать доставку сигнала, но когда поток назначает определенное действие для сигнала, оно становится общим для всех потоков. То есть если один поток установил диспозицию сигнала так, чтобы он игнорировался, другой поток может отменить это действие, установив диспозицию сигнала в значение по умолчанию или назначив обработчик сигнала.

Сигналы доставляются только одному потоку в процессе. Если сигнал связан с аппаратной ошибкой или истечением таймера, он доставляется потоку, который стал причиной этого сигнала. Однако остальные сигналы доставляются любому произвольному потоку.

В разделе 10.12 рассказывалось, как использовать функцию `sigprocmask` для блокировки сигналов. Поведение функции `sigprocmask` в многопоточной среде не определено. Вместо нее потоки должны использовать функцию `pthread_sigmask`.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Функция `pthread_sigmask` идентична функции `sigprocmask`, за исключением того, что предназначена для работы в многопоточной среде и в случае ошибки возвращает не значение `-1` с записью кода ошибки в `errno`, а выдает код ошибки. Напомню, что аргумент `set` должен содержать набор сигналов для изменения маски сигналов. Аргумент `how` может иметь одно из трех значений: `SIG_BLOCK` — чтобы добавить набор сигналов в маску, `SIG_SETMASK` — чтобы заменить маску сигналов потока данным набором, или `SIG_UNBLOCK` — чтобы удалить указанный набор сигналов из маски. Если в аргументе `oset` передается непустой указатель, прежняя маска сигналов потока сохраняется в структуре `sigset_t` по адресу `oset`. Поток может получить текущую маску сигналов, передав в аргументе `set` пустой указатель `NULL`, а в аргументе `oset` — адрес структуры `sigset_t`. В этом случае аргумент `how` игнорируется.

Поток может приостановиться в ожидании доставки сигнала, вызвав функцию `sigwait`.

```
#include <signal.h>
int sigwait(const sigset_t *restrict set, int *restrict signop);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Аргумент *set* определяет набор сигналов, доставка которых ожидается. По возвращении из функции по адресу *signop* будет записан номер доставленного сигнала. Если какой-либо сигнал, входящий в набор *set*, ожидал обработки к моменту вызова *sigwait*, функция вернет управление немедленно. Перед возвратом управления *sigwait* удалит этот сигнал из набора сигналов, ожидающих обработки. Если реализация поддерживает очереди сигналов и в очереди находится несколько экземпляров одного и того же сигнала, *sigwait* удалит из очереди только один экземпляр, остальные останутся в очереди, ожидая обработки.

Во избежание ошибочной реакции на сигнал поток должен заблокировать ожидаемые сигналы перед вызовом *sigwait*. Функция *sigwait* атомарно разблокирует сигналы и перейдет в режим ожидания, пока один из сигналов не будет доставлен. Перед возвратом управления *sigwait* восстановит маску сигналов потока. Если сигнал не будет заблокирован к моменту вызова функции, возникнет промежуток времени, когда сигнал может быть доставлен потоку еще до того, как он выполнит вызов *sigwait*.

Преимущество использования функции *sigwait* заключается в том, что она позволяет упростить обработку сигналов и обрабатывать асинхронные сигналы на синхронный манер. Чтобы воспрепятствовать прерыванию выполнения потока по сигналу, можно добавить требуемые сигналы к маске сигналов каждого потока. Благодаря этому можно выделить отдельные потоки, которые будут заниматься только обработкой сигналов. Эти специально выделенные потоки могут обращаться к любым функциям, которые нельзя использовать в обработчиках сигналов, потому что в этой ситуации функции будут вызываться в контексте обычного потока, а не из традиционного обработчика сигнала, прерывающего работу потока. Если сразу несколько потоков окажутся заблокированными в функции *sigwait* в ожидании одного и того же сигнала, только в одном из них функция *sigwait* вернет управление, когда сигнал будет доставлен процессу. Если сигнал перехватывается процессом (например, когда процесс установил обработчик сигнала с помощью функции *sigaction*) и при этом поток, обратившийся к функции *sigwait*, ожидает доставки того же сигнала, принятие решения о способе доставки сигнала оставляется на усмотрение реализации.

Операционная система в этом случае может либо вызвать установленный обработчик сигнала, либо позволить функции *sigwait* вернуть управление в поток, но не то и другое вместе. Для посылки сигнала процессу используется функция *kill* (раздел 10.9). Для посылки сигнала потоку используется функция *pthread_kill*.

```
#include <signal.h>
int pthread_kill(pthread_t thread, int signo);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Передав в аргументе `signo` значение 0, можно проверить существование потока. Если действие по умолчанию для сигнала заключается в завершении процесса, передача такого сигнала потоку приведет к завершению всего процесса.

Обратите внимание, что таймеры являются ресурсами процесса, и все потоки совместно используют один и тот же набор таймеров. Следовательно, в многопоточном приложении невозможно использовать таймеры в одном потоке, не оказывая влияния на другие (это тема упражнения 12.6).

Пример

В программе в листинге 10.16 мы приостанавливали работу процесса, пока обработчик сигнала не установит флаг, который указывает, что процесс должен завершить работу. Единственными потоками управления в этой программе были главный поток программы и обработчик сигнала, поэтому блокировка сигнала служила надежным средством, не позволяющим пропустить получение сигнала и изменение флага. В многопоточном приложении мы вынуждены защищать доступ к флагу с помощью мьютекса, как показано в листинге 12.6.

Листинг 12.6. Синхронная обработка сигнала

```
#include "apue.h"
#include <pthread.h>

int      quitflag; /* поток записывает сюда ненулевое значение */
sigset_t mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "ошибка вызова функции sigwait");
        switch (signo) {
        case SIGINT:
            printf("\nпрерывание\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("получен непредвиденный сигнал %d\n", signo);
            exit(1);
        }
    }
}
```

```
}

int
main(void)
{
    int          err;
    sigset_t     oldmask;
    pthread_t    tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "невозможно создать поток");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /*
     * Сигнал SIGQUIT был перехвачен и к настоящему моменту
     * опять заблокирован.
     */
    quitflag = 0;

    /* Восстановить маску сигналов, в которой SIGQUIT разблокирован. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
    exit(0);
}
```

Вместо того чтобы обрабатывать сигнал в функции-обработчике, прерывающей выполнение главного потока, мы создали для этого отдельный поток. Изменение флага `quitflag` производится под защитой мьютекса, чтобы главный поток не смог пропустить изменение значения флага, когда поток-обработчик вызывает функцию `pthread_cond_signal`. Тот же самый мьютекс используется в главном потоке для контроля состояния флага, мы атомарно освобождаем его и ожидаем наступления события.

Обратите внимание, что в самом начале главный поток программы блокирует сигналы **SIGINT** и **SIGQUIT**. Когда создается поток, который будет обрабатывать доставку сигналов, он наследует текущую маску сигналов. Поскольку функция **sigwait** разблокирует сигналы, только один поток сможет получить их. Это позволяет при написании программы не беспокоиться о том, что главный поток может быть прерван этими сигналами.

Запустив эту программу, мы получим результаты, похожие на те, что дала нам программа из листинга 10.16:

прерывание
^?
прерывание
^\\$

и еще раз
а теперь введем символ завершения

12.9. Потоки и fork

Когда поток вызывает функцию `fork`, создается копия всего адресного пространства процесса. В разделе 8.3 уже рассказывалось о технике копирования при записи. Дочерний процесс — это процесс, совершенно отдельный от родительского, и пока ни один из них не изменяет никаких данных, они могут совместно использовать одно и то же адресное пространство.

Наследуя адресное пространство, дочерний процесс наследует и состояние каждого из мьютексов, блокировок чтения-записи и переменных состояния. Если родительский процесс состоит более чем из одного потока, то дочерний процесс долженбросить состояние блокировки, если он не собирается немедленно вызвать функцию `exec`.

Внутри дочернего процесса существует только один поток — копия потока в родительском процессе, вызвавшего функцию `fork`. Если в родительском процессе поток владел какими-либо блокировками, этими же блокировками будет владеть и дочерний процесс. Проблема состоит в том, что дочерний процесс не имеет копий других потоков, которые также могут удерживать блокировки, поэтому у дочернего процесса нет возможности узнать, какие блокировки установлены и какие из них следует освободить.

Этой проблемы можно избежать, если дочерний процесс сразу же после выхода из функции `fork` вызовет `exec`. В этом случае старое адресное пространство исчезает, и потому состояние имеющихся блокировок не имеет никакого значения. Однако это не всегда возможно: бывает так, что дочерний процесс должен продолжить обработку данных, поэтому приходится использовать иную стратегию.

Во избежание проблем противоречивости состояния в многопоточных процессах стандарт POSIX.1 требует, чтобы в промежутке времени между возвратом из `fork` и вызовом `exec` дочерний процесс вызывал только функции, безопасные для использования в обработчиках асинхронных сигналов. Это ограничивает набор операций, доступных дочернему процессу до вызова функции `exec`, но не решает проблему с состоянием блокировок в дочернем процессе.

Чтобыбросить состояние блокировок, доставшихся в наследство от родительского процесса, можно установить обработчик операции ветвления вызовом функции `pthread_atfork`.

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void (*child)(void));
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

С помощью `pthread_atfork` можно установить до трех функций, которые служат для сброса блокировок. Функция *prepare* вызывается в родительском процессе перед созданием дочернего процесса вызовом `fork`. Этот обработчик должен установить все блокировки, которые имеются в родительском процессе.

Функция *parent* вызывается в контексте родительского процесса после создания дочернего процесса, но до того, как `fork` вернет управление. Цель этой функции – снять все блокировки, установленные в функции *prepare*. Функция *child* вызывается в контексте дочернего процесса до того, как `fork` вернет управление. Подобно функции *parent*, функция *child* также должна освободить все блокировки, установленные в функции *prepare*.

Обратите внимание, что здесь не происходит двойного снятия блокировок, установленных один раз, как может показаться на первый взгляд. Когда создается адресное пространство дочернего процесса, в нем находятся копии всех блокировок, определенных в родительском процессе. Поскольку обработчик *prepare* установил все блокировки, содержимое памяти в родительском и в дочернем процессах будет полностью идентично. Когда родитель и потомок разблокируют свои «копии» блокировок, для дочернего процесса выделяется новая область памяти и содержимое памяти родительского процесса копируется в адресное пространство потомка (копирование при записи). То есть ситуация выглядит так, будто родительский процесс заблокировал свои копии блокировок, а дочерний процесс – свои. И родительский и дочерний процессы снимают блокировки, расположенные в различных адресных пространствах, как если бы выполнялась следующая последовательность действий.

1. Родительский процесс установил все блокировки.
2. Дочерний процесс установил все блокировки.
3. Родительский процесс освободил все блокировки.
4. Дочерний процесс освободил все блокировки.

Мы можем вызвать функцию `pthread_atfork` много раз, чтобы установить несколько наборов обработчиков процедуры ветвления. Если потребность в каком-либо из трех обработчиков отсутствует, в соответствующем аргументе можно передать пустой указатель. Когда назначается несколько наборов обработчиков, порядок их вызова изменяется. Функции *parent* и *child* вызываются в том порядке, в каком они были зарегистрированы, тогда как функции *prepare* вызываются в противоположном порядке. Это позволяет различным модулям устанавливать свои обработчики процедуры ветвления и сохранять при этом иерархию блокировок.

Например, предположим, что модуль А вызывает функции из модуля В и при этом каждый модуль имеет собственный набор блокировок. Если в соответствии с алгоритмом модуль А должен установить блокировки раньше модуля В, модуль В должен первым установить обработчик процедуры ветвления. После того как родительский процесс вызвал функцию `fork`, действия будут развиваться в следующей последовательности, если предположить, что дочерний процесс первым получит управление.

1. Будет вызвана функция *prepare* из модуля А, которая установит все блокировки модуля А.

2. Будет вызвана функция *prepare* из модуля В, которая установит все блокировки модуля В.
3. Будет создан дочерний процесс.
4. Функция *child* из модуля В освободит все блокировки модуля В в дочернем процессе.
5. Функция *child* из модуля А освободит все блокировки модуля А в дочернем процессе.
6. Функция *fork* вернет управление в дочернем процессе.
7. Функция *parent* из модуля В освободит все блокировки модуля В в родительском процессе.
8. Функция *parent* из модуля А освободит все блокировки модуля А в родительском процессе.
9. Функция *fork* вернет управление в родительском процессе.

Если обработчики процедуры ветвления предназначены для сброса блокировок, как тогда сбросить переменные состояния? В некоторых реализациях не требуется сбрасывать переменные состояния. Однако в реализациях, использующих блокировки как составную часть переменных состояния, эти блокировки необходимо сбрасывать. Проблема в том, что не существует интерфейсов, которые позволяют сделать это. Если в структуре переменной состояния присутствует блокировка, мы не сможем воспользоваться этой переменной после вызова функции *fork*, потому что не существует достаточно переносимого способа сбросить эту блокировку. С другой стороны, если для защиты переменных состояния реализация использует глобальные блокировки в процессе, операционная система сама сбросит их в функции *fork*. Однако прикладные программы не должны полагаться на подобное поведение реализации.

Пример

Программа в листинге 12.7 демонстрирует использование функции *pthread_atfork* и обработчиков процедуры ветвления.

Листинг 12.7. Пример использования функции *pthread_atfork*

```
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    int err;

    printf("подготовка блокировок...\n");
    if ((err = pthread_mutex_lock(&lock1)) != 0)
        err_cont(err, "ошибка захвата блокировки lock1 в функции prepare");
    if ((err = pthread_mutex_lock(&lock2)) != 0)
```

```

        err_cont(err, "ошибка захвата блокировки lock2 в функции prepare");
    }

void
parent(void)
{
    int err;

    printf("родитель снимает блокировки...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "ошибка освобождения блокировки lock1 в parent");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "ошибка освобождения блокировки lock2 в parent");
}

void
child(void)
{
    int err;

    printf("потомок снимает блокировки...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "ошибка освобождения блокировки lock1 в child");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "ошибка освобождения блокировки lock2 в child");
}

void *
thr_fn(void *arg)
{
    printf("поток запущен...\n");
    pause();
    return(0);
}

int
main(void)
{
    int          err;
    pid_t        pid;
    pthread_t    tid;

    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "ошибка установки обработчика процедуры ветвления");
    if ((err = pthread_create(&tid, NULL, thr_fn, 0)) != 0)
        err_exit(err, "невозможно создать поток");

    sleep(2);
    printf("родительский процесс вызывает fork...\n");

    if ((pid = fork()) < 0)
        err_quit("ошибка вызова функции fork");
    else if (pid == 0) /* дочерний процесс */
        printf("функция fork вернула управление в дочерний процесс\n");
    else /* родительский процесс */
        printf("функция fork вернула управление в родительский процесс\n");
    exit(0);
}

```

Здесь мы определили два мьютекса, `lock1` и `lock2`. Функция `prepare` захватывает их оба, функция `child` освобождает их в контексте дочернего процесса, а `parent` – в контексте родительского процесса.

Запуск программы дал следующие результаты:

```
$ ./a.out
поток запущен...
родительский процесс вызывает fork...
подготовка блокировок...
потомок снимает блокировки...
функция fork вернула управление в дочерний процесс
родитель снимает блокировки...
функция fork вернула управление в родительский процесс
```

Как видите, функция `prepare` вызывается после вызова `fork`, функция `child` запускается перед выходом из функции `fork` в дочерний процесс, а функция `parent` – перед выходом из функции `fork` в родительский процесс.

Хотя механизм `pthread_atfork` предназначен специально, чтобы обеспечить не-противоречивость блокировок после вызова `fork`, он имеет некоторые недостатки, ограничивающие возможность его применения:

- Отсутствует возможность повторной инициализации сложных объектов синхронизации, таких как переменные состояния и барьеры.
- Некоторые реализации мьютексов с проверкой ошибок генерируют ошибки при попытке разблокировать мьютесксы в дочернем обработчике ветвления, которые были заблокированы в родительском процессе.
- Рекурсивные мьютесксы нельзя освободить в дочернем обработчике ветвления из-за невозможности определить, сколько раз они были захвачены.
- Так как дочерним процессам позволено вызывать только функции, которые безопасно использовать в контексте обработчиков асинхронных сигналов, дочерний обработчик ветвления не должен даже пытаться сбрасывать объекты синхронизации, потому что ни одна из функций, используемых для этого, не является безопасной. Фактически проблема состоит в том, что к моменту вызова `fork` в одном из потоков объект синхронизации может находиться в промежуточном состоянии, а объекты синхронизации, как известно, нельзя освободить, пока они не окажутся в непротиворечивом состоянии.
- Если приложение вызовет `fork` в обработчике сигнала (что вполне допустимо, потому что функция `fork` является безопасной в контексте обработки сигналов), обработчики ветвления, зарегистрированные с помощью `pthread_atfork`, смогут вызывать только безопасные функции, иначе результат может оказаться непредсказуемым.

12.10. Потоки и операции ввода/вывода

В разделе 3.11 мы говорили о функциях `pread` и `pwrite`. Эти функции удобно использовать в многопоточной среде, потому что все потоки в процессе совместно используют одни и те же файловые дескрипторы.

Рассмотрим два потока, которые одновременно работают с одним и тем же файловым дескриптором.

Поток А

```
lseek(fd, 300, SEEK_SET);  
read(fd, buf1, 100);
```

Поток В

```
lseek(fd, 700, SEEK_SET);  
read(fd, buf2, 100);
```

Если поток А вызовет `lseek`, а затем поток В также вызовет `lseek` до того, как поток А успеет вызвать функцию `read`, оба потока прочитают одну и ту же запись. Понятно, что это совсем не то, что нам нужно.

Чтобы решить эту проблему, можно использовать функцию `pread`, которая устанавливает текущую позицию файла и производит чтение данных атомарно.

Поток А

```
pread(fd, buf1, 100, 300);
```

Поток В

```
pread(fd, buf2, 100, 700);
```

Используя функцию `pread`, можно быть уверенным, что поток А прочитает запись, начиная со смещения 300, а поток В — со смещения 700. Для решения аналогичной проблемы, связанной с записью в один и тот же файл, можно использовать функцию `pwrite`.

12.11. Подведение итогов

Потоки в системе UNIX предоставляют альтернативную модель разбиения крупных задач на подзадачи. Потоки следуют модели совместного использования одних и тех же данных, что, в свою очередь, порождает специфические проблемы синхронизации. В этой главе мы рассмотрели вопросы, связанные с настройкой поведения потоков, и примитивы синхронизации. Мы обсудили вопрос реентерабельности относительно потоков. А также увидели, как потоки взаимодействуют с некоторыми системными вызовами.

Упражнения

- 12.1 Запустите программу из листинга 12.7 в Linux, перенаправив вывод в файл. Объясните полученные результаты.
- 12.2 Реализуйте функцию `putenv_r` — реентерабельную версию функции `putenv`. Убедитесь, что ваша версия функции безопасна в контексте обработки асинхронных сигналов и в многопоточной среде.
- 12.3 Возможно ли функцию `getenv` из листинга 12.5 сделать безопасной в контексте обработки сигналов, блокируя доставку сигнала в начале функции и восстанавливая предыдущую маску сигналов перед возвратом из нее? Объясните почему.
- 12.4 Напишите программу для проверки версии функции `getenv` из листинга 12.5. Скомпилируйте и запустите программу в FreeBSD. Объясните, что произошло.

- 12.5** Если существует возможность создавать многочисленные потоки для решения разнообразных задач в рамках одного процесса, объясните, зачем в этих условиях может понадобиться функция `fork`.
- 12.6** Измените реализацию программы из листинга 10.21 так, чтобы она стала безопасной в многопоточной среде, не используя функцию `nanosleep` или `clock_nanosleep`.
- 12.7** Возможно ли в дочернем процессе после возврата из функции `fork` безопасно переинициализировать переменные состояния путем их разрушения функцией `pthread_cond_destroy` и последующей инициализацией вызовом `pthread_cond_init`?
- 12.8** Функцию `timeout` из листинга 12.2 можно существенно упростить. Объясните как.

13 Процессы-демоны

13.1. Введение

Демоны — это долгоживущие процессы. Зачастую они запускаются во время загрузки системы и завершают работу вместе с ней. Так как они не имеют управляющего терминала, говорят, что они работают в фоновом режиме. В системе UNIX демоны решают множество повседневных задач.

В этой главе мы рассмотрим структуру процессов-демонов и покажем, как они создаются. Так как демоны не имеют управляющего терминала, нам необходимо выяснить, как демон может вывести сообщение об ошибке, если что-то идет не так, как надо.

Обсуждение истории термина «демон» применительно к компьютерным системам вы найдете в [Raymond, 1996].

13.2. Характеристики демонов

Рассмотрим некоторые наиболее распространенные системные демоны и их связь с группами процессов, управляющими терминалами и сессиями, описанными в главе 9. Команда `ps(1)` выводит информацию о процессах в системе. Эта команда имеет множество параметров, дополнительную информацию о них вы найдете в справочном руководстве. Запустим команду

```
ps -axj
```

в BSD-системе и будем использовать полученную от нее информацию при дальнейшем обсуждении. Ключ `-a` используется для вывода процессов, которыми владеют другие пользователи, ключ `-x` — для вывода процессов, не имеющих управляющего терминала, и ключ `-j` — для вывода дополнительных сведений, имеющих отношение к заданиям: идентификатора сессии, идентификатора группы процессов, управляющего терминала и идентификатора группы процессов терминала.

Для систем, основанных на System V, аналогичная команда выглядит как `ps -efj`. (В целях безопасности некоторые версии UNIX не допускают просмотра с помощью команды `ps` процессов, принадлежащих другим пользователям.) Вывод команды `ps` выглядит примерно так:

UID	PID	PPID	PGID	SID	TTY	CMD
root	1	0	1	1	?	/sbin/init
root	2	0	0	0	?	[kthreadd]
root	3	2	0	0	?	[ksoftirqd/0]
root	6	2	0	0	?	[migration/0]
root	7	2	0	0	?	[watchdog/0]
root	21	2	0	0	?	[cpuset]
root	22	2	0	0	?	[khelper]
root	26	2	0	0	?	[sync_supers]
root	27	2	0	0	?	[bdi-default]
root	29	2	0	0	?	[kblockd]
root	35	2	0	0	?	[kswapd0]
root	49	2	0	0	?	[scsi_eh_0]
root	256	2	0	0	?	[jbd2/sda5-8]
root	257	2	0	0	?	[ext4-dio-unwrit]
syslog	847	1	843	843	?	rsyslogd -c5
root	906	1	906	906	?	/usr/sbin/cupsd -F
root	1037	1	1037	1037	?	/usr/sbin/inetd
root	1067	1	1067	1067	?	cron
daemon	1068	1	1068	1068	?	atd
root	8196	1	8196	8196	?	/usr/sbin/sshd -D
root	13047	2	0	0	?	[kworker/1:0]
root	14596	2	0	0	?	[flush-8:0]
root	26464	1	26464	26464	?	rpcbind -w
statd	28490	1	28490	28490	?	rpc.statd -L
root	28553	2	0	0	?	[rpciod]
root	28554	2	0	0	?	[nfsiod]
root	28561	1	28561	28561	?	rpc.idmapd
root	28761	2	0	0	?	[lockd]
root	28764	2	0	0	?	[nfsd]
root	28775	1	28775	28775	?	/usr/sbin/rpc.mountd --manage-gids

Из данного примера мы убрали несколько колонок, которые не представляют для нас особого интереса, такие как накопленное процессорное время. Здесь показаны следующие колонки, слева направо: идентификатор пользователя (**UID**), идентификатор процесса (**PID**), идентификатор родительского процесса (**PPID**), идентификатор группы процессов (**PGID**), идентификатор сеанса (**SID**), имя терминала (**TTY**) и строка команды (**CMD**).

Система, в которой была запущена эта команда (Linux 3.2.0), поддерживает понятие идентификатора сеанса, который мы упоминали при обсуждении функции `setsid` в разделе 9.5. Идентификатор сеанса — это просто идентификатор процесса лидера сеанса. Однако в системах, основанных на BSD, будет выведен адрес структуры `session`, соответствующей группе процессов, которой принадлежит данный процесс (раздел 9.11).

Перечень системных процессов, который вы увидите, во многом зависит от реализации операционной системы. Обычно это будут процессы с идентификатором родительского процесса 0, запускаемые ядром в процессе загрузки системы. (Исключением является процесс `init`, так как это команда уровня пользователя, которая запускается ядром во время загрузки.) Процессы ядра — это особые процессы, они существуют все время, пока работает система. Эти процессы обладают привилегиями суперпользователя и не имеют ни управляющего терминала, ни строки команды.

В этом примере вывода команды `ps` демоны ядра можно распознать по именам, заключенным в квадратные скобки. Для создания процессов ядра данная версия

Linux использует специальный процесс ядра, `kthreadd`, поэтому родителем всех остальных демонов ядра оказывается процесс `kthreadd`. Для каждого компонента ядра, который должен выполнять операции в контексте процесса, но не был вызван из пользовательского процесса, обычно создается собственный процесс — демон ядра. Например, в Linux:

- Имеется демон `kswapd`, также известный как демон выгрузки страниц памяти. Он обеспечивает поддержку подсистемы виртуальной памяти, с течением времени записывая измененные страницы памяти на диск, чтобы их можно было освободить.
- Демон `flush` выталкивает измененные страницы на диск, когда объем доступной памяти падает до установленного минимального предела. Он также выталкивает измененные страницы памяти на диск через регулярные интервалы времени, чтобы уменьшить потерю данных в случае краха системы. В системе может одновременно выполняться несколько демонов `flush` — по одному для каждого устройства. В примере выше присутствует только один демон `flush` с именем `flush-8:0`, где 8 — это старший номер устройства, а 0 — младший.
- Демон `sync_supers` периодически выталкивает на диск метаданные файловой системы.
- Демон `jbd` обеспечивает поддержку журнала в файловой системе `ext4`.

Процесс с идентификатором 1 — это обычно процесс `init` (`launchd` в Mac OS X), о чем уже говорилось в разделе 8.2. Это системный демон, который, кроме всего прочего, отвечает за запуск различных системных служб на различных уровнях загрузки. Как правило, эти службы также реализованы в виде демонов.

Демон `rpcbind` осуществляет преобразование числовых идентификаторов служб RPC (Remote Procedure Call — удаленный вызов процедур) в номера сетевых портов. Демон `rsyslogd` может использоваться программами для вывода сообщений в системный журнал, куда затем сможет заглянуть администратор. Сообщения могут выводиться в консоль, а также записываться в файл. (Более подробно механизм журналирования `syslogd` рассматривается в разделе 13.4.)

В разделе 9.3 мы уже говорили о демоне `inetd`. Этот демон ожидает поступления из сети запросов к различным сетевым серверам. Демоны `nfsd`, `nfsiod`, `lockd`, `rpciod`, `rpc.idmapd`, `rpc.statd` и `rpc.mountd` обеспечивают поддержку сетевой файловой системы (Network File System, NFS). Обратите внимание, что первые четыре из них являются демонами ядра, а последние три — демонами уровня пользователя.

Демон `cron` производит запуск команд через регулярные интервалы времени. Он обрабатывает различные задания системного администрирования, запуская их через заданные промежутки времени. Демон `atd` напоминает демон `cron` и дает пользователям возможность запускать задания в определенные моменты времени, но запускает задания однократно. Демон `cupsd` — это сервер печати, он обслуживает запросы к принтеру. Демон `sshd` обеспечивает удаленный доступ к системе и выполнение в защищенном режиме.

Обратите внимание, что большинство демонов обладают привилегиями суперпользователя (`root`). Ни один из демонов не имеет управляющего терминала — вместо имени терминала стоит знак вопроса. Демоны ядра запускаются без

управляющего терминала. Отсутствие управляющего терминала у демонов пользовательского уровня, вероятно, результат вызова функции `setsid`. Все демоны пользовательского уровня являются лидерами групп и лидерами сеансов, а также единственными процессами в своих группах процессов и сеансах (исключение составляет `rsyslogd`). И наконец, обратите внимание, что родителем для большинства демонов является процесс `init`.

13.3. Правила программирования демонов

Во избежание нежелательных взаимодействий при программировании демонов следует придерживаться определенных правил. Сначала мы перечислим эти правила, а затем продемонстрируем функцию `daemonize`, которая их реализует.

1. Вызывать функцию `umask`, чтобыбросить маску режима создания файлов в значение 0. Мaska, наследуемая от запускающего процесса, может маскировать некоторые биты прав доступа. Если предполагается, что процесс-демон будет создавать файлы, может потребоваться установка определенных битов прав доступа. Например, если демон создает файлы с правом на чтение и на запись для группы, маска режима создания файла, которая выключает любой из этих битов, будет препятствовать этому. С другой стороны, если демон вызывает библиотечные функции, создающие файлы, имеет смысл установить более ограничивающую маску (например, 007), так как библиотечные функции могут не принимать аргумент с битами прав доступа.
2. Вызывать функцию `fork` и завершить родительский процесс. Для чего это делается? Во-первых, если демон запущен как обычная команда оболочки, завершив родительский процесс, мы заставим командную оболочку думать, что команда выполнилась. Во-вторых, дочерний процесс наследует идентификатор группы процессов от родителя, но получает свой идентификатор процесса; тем самым гарантируется, что дочерний процесс не будет являться лидером группы, а это необходимое условие для вызова функции `setsid`, который будет произведен далее.
3. Создать новый сеанс, обратившись к функции `setsid`. При этом (вспомните раздел 9.5) процесс становится (а) лидером нового сеанса, (б) лидером новой группы процессов и (в) лишается управляющего терминала.

Для систем, основанных на System V, некоторые специалисты рекомендуют в этой точке повторно вызвать функцию `fork` и завершить родительский процесс, чтобы второй потомок продолжал работу в качестве демона. Такой прием гарантирует, что демон не будет являться лидером сеанса, и это препятствует получению управляющего терминала в System V (раздел 9.6). Как вариант, чтобы избежать обретения управляющего терминала, при любом открытии терминального устройства следует указывать флаг `O_NOSTTY`.

4. Сделать корневой каталог текущим рабочим каталогом. Текущий рабочий каталог, унаследованный от родительского процесса, может находиться на смонтированной файловой системе. Поскольку демон, как правило, существует все

время, пока система не перезагрузится, в подобной ситуации, когда рабочий каталог демона находится в смонтированной файловой системе, ее невозможно будет отмонтировать. Как вариант, некоторые демоны могут устанавливать собственный текущий рабочий каталог, в котором они производят все необходимые действия. Например, демоны печати в качестве текущего рабочего каталога часто выбирают буферный каталог, куда помещаются задания для печати.

5. Закрыть все ненужные файловые дескрипторы. Это предотвращает удержание в открытом состоянии некоторых дескрипторов, унаследованных от родительского процесса (командной оболочки или другого процесса). С помощью нашей функции `open_max` (листинг 2.4) или с помощью функции `getrlimit` (раздел 7.11) можно определить максимально возможный номер дескриптора и закрыть все дескрипторы вплоть до этого номера.
6. Некоторые демоны открывают файловые дескрипторы с номерами 0, 1 и 2 на устройстве `/dev/null`, — таким образом, любые библиотечные функции, которые пытаются читать со стандартного устройства ввода или писать в стандартное устройство вывода или сообщений об ошибках, не будут оказывать никакого влияния. Поскольку демон не связан ни с одним терминальным устройством, он не сможет взаимодействовать с пользователем в интерактивном режиме. Даже если демон запущен в интерактивном сеансе, он все равно переходит в фоновый режим, и начальный сеанс может завершиться без воздействия на процесс-демон. С этого же терминала в систему могут входить другие пользователи, и демон не должен выводить какую-либо информацию на терминал, да и пользователи не ждут, что их ввод с терминала будет прочитан демоном.

Пример

В листинге 13.1 приводится функция, которую может вызывать приложение, желающее стать демоном.

Листинг 13.1. Инициализация процесса-демона

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int             i, fd0, fd1, fd2;
    pid_t           pid;
    struct rlimit   rl;
    struct sigaction sa;

    /*
     * Сбросить маску режима создания файла.
     */
    umask(0);
```

```
/*
 * Получить максимально возможный номер дескриптора файла.
 */
if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
    err_quit("%s: невозможно получить максимальный номер дескриптора ", cmd);

/*
 * Стать лидером нового сеанса, чтобы утратить управляющий терминал.
 */
if ((pid = fork()) < 0)
    err_quit("%s: ошибка вызова функции fork", cmd);
else if (pid != 0) /* родительский процесс */
    exit(0);
setsid();

/*
 * Обеспечить невозможность обретения управляющего терминала в будущем.
 */
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: невозможно игнорировать сигнал SIGHUP", cmd);
if ((pid = fork()) < 0)
    err_quit("%s: ошибка вызова функции fork", cmd);
else if (pid != 0) /* родительский процесс */
    exit(0);

/*
 * Назначить корневой каталог текущим рабочим каталогом,
 * чтобы впоследствии можно было отмонтировать файловую систему.
 */
if (chdir("/") < 0)
    err_quit("%s: невозможно сделать текущим рабочим каталогом /", cmd);

/*
 * Закрыть все открытые файловые дескрипторы.
 */
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
close(i);

/*
 * Присоединить файловые дескрипторы 0, 1 и 2 к /dev/null.
 */
fd0 = open("/dev/null", O_RDWR);
fd1 = dup(0);
fd2 = dup(0);

/*
 * Инициализировать файл журнала.
 */
openlog(cmd, LOG_CONS, LOG_DAEMON);
if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
    syslog(LOG_ERR, "ошибочные файловые дескрипторы %d %d %d",
           fd0, fd1, fd2);
    exit(1);
}
```

Если функцию `daemonize` вызвать из программы, которая затем приостанавливает работу, мы сможем проверить состояние демона с помощью команды `ps`:

```
$ ./a.out
$ ps -efj
UID      PID  PPID  PGID   SID   TTY  CMD
sar  13800      1  13799 13799    ?  ./a.out
$ ps -efj | grep 13799
sar  13800      1  13799 13799    ?  ./a.out
```

С помощью команды `ps` можно также убедиться, что в системе нет активного процесса с идентификатором 13799. Это означает, что наш демон относится к осиротевшей группе процессов (раздел 9.10) и не является лидером сеанса, а поэтому не имеет возможности обрести управляющий терминал. Это результат второго вызова функции `fork` в функции `daemonize`. Как видите, наш демон инициализирован правильно.

13.4. Журналирование ошибок

Одна из проблем, присущих демонам, связана с обслуживанием сообщений об ошибках. Демон не может просто выводить сообщения в стандартное устройство вывода сообщений об ошибках, поскольку не имеет управляющего терминала. Мы не можем требовать от демона, чтобы он выводил сообщения в консоль, поскольку на большинстве рабочих станций в консоли запускается многооконная система. Мы также не можем требовать, чтобы демон хранил свои сообщения в отдельном файле. Это стало бы источником постоянной головной боли для системного администратора, который будет вынужден запоминать, в какой файл каждый демон пишет свои сообщения. Необходим некий централизованный механизм регистрации сообщений об ошибках.

Механизм `sysLog` для BSD-систем был разработан в Беркли и получил широкое распространение начиная с 4.2BSD. Большинство систем, производных от BSD, поддерживают `sysLog`. До появления SVR4 OS System V не имела централизованного механизма регистрации сообщений об ошибках. Функция `sysLog` была включена в стандарт Single UNIX Specification как расширение XSI.

Механизм `syslog` для BSD-систем широко используется начиная с 4.2BSD. Большинство демонов используют именно этот механизм. На рис. 13.1 показана его структура.

Существует три способа регистрации сообщений.

1. Процедуры ядра могут обращаться к функции `log`. Эти сообщения доступны любому пользовательскому процессу, который может открыть и прочитать устройство `/dev/klog`. Мы не будем рассматривать эту функцию, поскольку не собираемся писать процедуры ядра.
2. Большинство пользовательских процессов (демонов) для регистрации сообщений вызывают функцию `syslog(3)`. Порядок работы с ней мы рассмотрим позже. Эта функция отправляет сообщения через сокет домена UNIX — `/dev/log`.

3. Пользовательский процесс, выполняющийся на данном или на другом компьютере, соединенном с данным компьютером сетью TCP/IP, может отправлять сообщения по протоколу UDP на порт 514. Обратите внимание, что функция `syslog` никогда не генерируетдейтограммы UDP – данная функциональность требует, чтобы сама программа поддерживала сетевые взаимодействия.

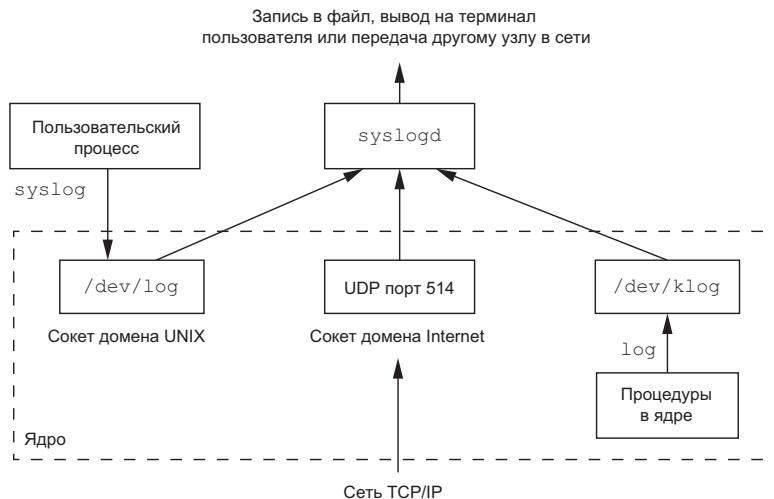


Рис. 13.1. Механизм `syslog` для BSD-систем

За дополнительной информацией о сокетах домена UNIX обращайтесь к [Stevens, Fenner, and Rudoff, 2004].

Обычно демон `syslogd` понимает все три способа регистрации сообщений. На запуске этот демон читает конфигурационный файл (как правило, `/etc/syslog.conf`), в котором определяется, куда должны передаваться сообщения различных классов. Например, срочные сообщения могут выводиться в консоль системного администратора (если он находится в системе), тогда как предупреждения могут записываться в файл.

В нашем случае взаимодействие с этим механизмом осуществляется посредством функции `syslog`.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

Возвращает предыдущее значение маски приоритета журналируемых сообщений

Функцию `openlog` можно не вызывать. Если перед первым обращением к функции `syslog` функция `openlog` не вызывалась, она будет вызвана автоматически. Вызывать функцию `closelog` также необязательно — она просто закрывает файловый дескриптор, который использовался для взаимодействия с демоном `syslogd`.

Функция `openlog` позволяет определить в аргументе *ident* строку идентификации, которая обычно содержит имя программы (например, `cron` или `inetd`). Аргумент *option* — это битовая маска, которая определяет различные способы вывода сообщений. В табл. 13.1 приводятся значения, которые могут быть включены в маску. В столбце XSI отмечены те из них, которые стандарт Single UNIX Specification включает в определение функции `openlog`.

Возможные значения аргумента *facility* приводятся в табл. 13.2. Обратите внимание, что стандарт Single UNIX Specification определяет лишь часть значений, обычно доступных в конкретной системе. Аргумент *facility* позволяет определить, как должны обрабатываться сообщения из разных источников. Если программа не вызывает функцию `openlog` или передает ей в аргументе *facility* значение 0, указать источник сообщения можно с помощью функции `syslog`, определив его как часть аргумента *priority*.

Функция `syslog` вызывается для передачи сообщения. В аргументе *priority* передается комбинация из значения для аргумента *facility* (табл. 13.2) и уровня важности сообщения (табл. 13.3). Уровни важности перечислены в табл. 13.3 в порядке убывания, от высшего к низшему.

Таблица 13.1. Возможные значения, которые могут быть включены в аргумент *option* функции `openlog`

option	XSI	Описание
<code>LOG_CONS</code>	✓	Если сообщение не может быть передано через сокет домена UNIX, оно будет выведено в консоль
<code>LOG_NDELAY</code>	✓	Сразу открыть сокет домена UNIX для взаимодействия с демоном <code>syslogd</code> , не дожидаясь, пока будет отправлено первое сообщение. Как правило, сокет открывается, только когда отправлено первое сообщение
<code>LOG_NOWAIT</code>	✓	Не ждать завершения дочерних процессов, которые могли быть созданы в процессе регистрации сообщения. Это предотвращает возникновение конфликтов для тех приложений, которые перехватывают сигнал <code>SIGCHLD</code> , так как приложение уже могло получить код завершения дочернего процесса к моменту, когда <code>syslog</code> вызвал функцию <code>wait</code>
<code>LOG_ODELAY</code>	✓	Отложить установление соединения с демоном <code>syslogd</code> до появления первого сообщения
<code>LOG_PERROR</code>	✓	Вывести сообщение на стандартное устройство вывода сообщений об ошибках и дополнительно передать его демону <code>syslogd</code> (эта возможность недоступна в Solaris)
<code>LOG_PID</code>	✓	Записывать идентификатор процесса вместе с текстом сообщения. Эта возможность предназначена для демонов, которые порождают дочерние процессы для обработки различных запросов (в противоположность демонам, таким как <code>syslogd</code> , которые никогда не вызывают функцию <code>fork</code>)

Аргумент *format* и все последующие аргументы передаются функции `vfprintf` для создания строки сообщения. Символы `%m` в строке формата заменяются сообщением об ошибке (`strerror`), которое соответствует значению переменной `errno`. Функция `setlogmask` может использоваться для установки маски приоритетов сообщений процесса. Эта функция возвращает предыдущее значение маски. Если маска приоритетов установлена, сообщения, уровень приоритета которых не содержится в маске, не будут журналироваться. Обратите внимание: из вышесказанного следует, что если маска имеет значение 0, журналироваться будут все сообщения.

Во многих системах имеется также программа `logger(1)`, которая может передавать сообщения механизму `syslog`. Некоторые реализации позволяют передавать программе необязательные аргументы, в которых указывается источник сообщения (*facility*), уровень важности (*level*) и строка идентификации (*ident*), хотя стандарт System UNIX Specification не определяет дополнительных аргументов. Команда `logger` предназначена для использования в сценариях на языке командной оболочки, которые выполняются в неинтерактивном режиме и нуждаются в механизме журналирования сообщений.

Таблица 13.2. Возможные значения аргумента facility функции openlog

facility	XSI	Описание
<code>LOG_AUDIT</code>		Средства контроля
<code>LOG_AUTH</code>		Программы авторизации: <code>login</code> , <code>su</code> , <code>getty</code> , ...
<code>LOG_AUTHPRIV</code>		То же самое, что и <code>LOG_AUTH</code> , но журналирование идет в файл с ограниченными правами доступа
<code>LOG_CONSOLE</code>		Сообщения, выводимые в <code>/dev/console</code>
<code>LOG_CRON</code>		<code>cron</code> и <code>at</code>
<code>LOG_DAEMON</code>		Системные демоны: <code>inetd</code> , <code>routed</code> , ...
<code>LOG_FTP</code>		Демон FTP (<code>ftpd</code>)
<code>LOG_KERN</code>		Сообщения, сгенерированные ядром
<code>LOG_LOCAL0</code>	✓	Определяется пользователем
<code>LOG_LOCAL1</code>	✓	Определяется пользователем
<code>LOG_LOCAL2</code>	✓	Определяется пользователем
<code>LOG_LOCAL3</code>	✓	Определяется пользователем
<code>LOG_LOCAL4</code>	✓	Определяется пользователем
<code>LOG_LOCAL5</code>	✓	Определяется пользователем
<code>LOG_LOCAL6</code>	✓	Определяется пользователем
<code>LOG_LOCAL7</code>	✓	Определяется пользователем
<code>LOG_LPR</code>		Система печати: <code>lpd</code> , <code>lpc</code> , ...
<code>LOG_MAIL</code>		Система электронной почты

Таблица 13.2 (окончание)

facility	XSI	Описание
LOG_NEWS		Система новостей Usenet
LOG_NTP		Протокол системы точного времени
LOG_SECURITY		Подсистема безопасности
LOG_SYSLOG		Сам демон <code>syslogd</code>
LOG_USER	✓	Сообщения от других пользовательских процессов (по умолчанию)
LOG_UUCP		Система UUCP

Таблица 13.3. Уровни серьезности сообщений (в порядке убывания)

Уровень	Описание
LOG_EMERG	Аварийная ситуация (система остановлена, наивысший приоритет)
LOG_ALERT	Требуется немедленное вмешательство
LOG_CRIT	Критическая ситуация (например, ошибка жесткого диска)
LOG_ERR	Ошибка
LOG_WARNING	Предупреждение
LOG_NOTICE	Обычная ситуация, которая не является ошибочной, но, возможно, требующая специальных действий
LOG_INFO	Информационное сообщение
LOG_DEBUG	Отладочное сообщение (низший приоритет)

Пример

В (гипотетическом) демоне печати можно встретить следующие строки:

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

Обращение к функции `openlog` устанавливает строку идентификации с именем программы, указывает, что идентификатор процесса обязательно должен добавляться к сообщению, и оговаривает, что источником сообщений будет демон системы печати. В вызове функции `syslog` указан уровень важности сообщения и само сообщение. Если опустить вызов функции `openlog`, вызов `syslog` мог бы выглядеть так:

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Здесь в аргументе *priority* мы объединили ссылку на источник сообщения и уровень важности сообщения.

Кроме функции `syslog` многие платформы поддерживают ее разновидность, которая принимает дополнительные аргументы в виде списка переменной длины.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

Все четыре платформы, обсуждаемые в данной книге, поддерживают функцию `vsysLog`, но она не входит в состав стандарта Single UNIX Specification. Обратите внимание: чтобы сделать эту функцию доступной в своем приложении, может потребоваться определить дополнительный символ, такой как `__BSD_VISIBLE` в FreeBSD или `__USE_BSD` в Linux.

Большинство реализаций `syslogd` для сокращения времени обработки запросов от приложений помещают поступившие сообщения в очередь. Если в это время демону поступит два одинаковых сообщения, в журнал будет записано только одно. Но в конец такого сообщения демоном будет добавлена строка примерно такого содержания: «last message repeated N times» (последнее сообщение было повторено N раз).

13.5. Демоны в единственном экземпляре

Некоторые демоны реализованы так, что допускают одновременную работу только одной своей копии. Причиной такого поведения может служить, например, требование монопольного владения каким-либо ресурсом. Так, если бы демон `cron` допускал одновременную работу нескольких своих копий, каждая из них пыталась бы по достижении запланированного времени запустить одну и ту же операцию, что наверняка привело бы к ошибке.

Если демон требует наличия доступа к устройству, некоторые действия по предотвращению открытия устройства несколькими программами может выполнить драйвер устройства. Это ограничит количество одновременно работающих экземпляров демона до одного. Однако если не предполагается обращения демона к подобным устройствам, то нам самим придется выполнить всю необходимую работу по наложению ограничений.

Одним из основных механизмов, обеспечивающих ограничение количества одновременно работающих копий демона, являются блокировки файлов и записей. (Блокировки файлов и записей в файлах мы рассмотрим в разделе 14.3.) Если каждый из демонов создаст файл и попытается установить для этого файла блокировку для записи, система разрешит установить только одну такую блокировку. Все последующие попытки установить блокировку для записи будут терпеть неудачу, сообщая остальным копиям демона, что демон уже запущен. Блокировки файлов и записей — это удобный механизм взаимного исключения. Если демон установит блокировку для целого файла, она будет автоматически снята по завершении демона. Это упрощает процедуру восстановления после ошибок, поскольку снимает необходимость удаления блокировки, оставшейся от предыдущей копии демона.

Пример

Функция в листинге 13.2 демонстрирует использование блокировок файлов и записей, чтобы обеспечить запуск единственного экземпляра демона.

Листинг 13.2. Функция, которая гарантирует запуск только одной копии демона

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int      fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "невозможно открыть %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "невозможно установить блокировку на %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Каждая копия демона будет пытаться создать файл и записать в него свой идентификатор процесса. Это поможет системному администратору идентифицировать процесс. Если файл уже заблокирован, функция `lockfile` завершится неудачей с кодом ошибки `EACCES` или `EAGAIN` в переменной `errno` и в вызывающую программу вернет значение 1, указывающее, что демон уже запущен. Иначе функция усекает размер файла до нуля, записывает в него идентификатор процесса и возвращает значение 0.

Усечение размера файла необходимо, потому что идентификатор процесса предыдущей копии демона, представленный в виде строки, мог иметь большую длину. Предположим, например, что ранее запускавшаяся копия демона имела иденти-

фикатор процесса 12345, а текущая копия имеет идентификатор процесса 9999. То есть когда этот демон запишет свой идентификатор, в файле окажется строка 99995. Операция усечения файла удаляет информацию, которая относится к предыдущей копии демона.

13.6. Соглашения для демонов

В системе UNIX демоны придерживаются следующих соглашений.

- Если демон использует файл блокировки, этот файл помещается в каталог `/var/run`. Однако, чтобы создать файл в этом каталоге, демон должен обладать привилегиями суперпользователя. Имя файла обычно имеет вид `name.pid`, где `name` — имя демона или службы. Например, демон `cron` создает файл блокировки с именем `/var/run/crond.pid`.
- Если демон поддерживает определение дополнительных настроек, они обычно хранятся в каталоге `/etc`. Имя конфигурационного файла, как правило, имеет вид `name.conf`, где `name` — имя демона или службы. Например, конфигурационный файл демона `syslogd` называется `/etc/syslog.conf`.
- Демоны могут запускаться из командной строки, но чаще запуск демонов производится из сценариев инициализации системы (`/etc/rc*` или `/etc/init.d/*`). Если после завершения демон должен автоматически перезапускаться, мы можем указать на это процессу `init`, добавив запись `respawn` в файл `/etc/inittab` (предполагается, что система использует команду `init` в стиле System V).
- Если демон имеет конфигурационный файл, настройки из него читаются демоном во время запуска, и затем он обычно не обращается к этому файлу. Если в конфигурационный файл были внесены изменения, демон пришлось бы останавливать и перезапускать снова, чтобы новые настройки вступили в силу. Во избежание этого некоторые демоны устанавливают обработчики сигнала `SIGHUP`, в которых производится чтение конфигурационного файла и перенастройка демона. Поскольку демоны не имеют управляющего терминала и являются либо лидерами сеансов без управляющего терминала, либо членами осиротевших групп процессов, у них нет причин ожидать сигнала `SIGHUP`. Поэтому он может использоваться для других целей.

Пример

Программа в листинге 13.3 демонстрирует один из способов заставить демон перечитать файл конфигурации. Программа использует функцию `sigwait` и отдельный поток для обработки сигналов, как описано в разделе 12.8.

Листинг 13.3. Пример демона, который перечитывает конфигурационный файл по сигналу

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>

sigset(SIGPOLL, handle_sigpoll);
```

```
extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "ошибка вызова функции sigwait");
            exit(1);
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Чтение конфигурационного файла");
            reread();
            break;
        case SIGTERM:
            syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
            exit(0);
        default:
            syslog(LOG_INFO, "получен непредвиденный сигнал %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int             err;
    pthread_t       tid;
    char           *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);

    /*
     * Убедиться, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Восстановить действие по умолчанию для сигнала SIGHUP
    
```

```

    * и заблокировать все сигналы.
 */
sa.sa_handler = SIG_DFL;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: невозможно восстановить действие SIG_DFL для SIGHUP");
sigfillset(&mask);
if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
    err_exit(err, "ошибка выполнения операции SIG_BLOCK");

/*
 * Создать поток для обработки SIGHUP и SIGTERM.
 */
err = pthread_create(&tid, NULL, thr_fn, 0);
if (err != 0)
    err_exit(err, "невозможно создать поток");

/*
 * Остальная часть программы-демона.
 */
/* ... */
exit(0);
}

```

Для перехода в режим демона программа использует функцию `daemonize` из листинга 13.1. После возврата из нее вызывается функция `already_running` из листинга 13.2, которая проверяет наличие других запущенных копий демона. В этой точке сигнал `SIGHUP` все еще игнорируется, поэтому мы должны переустановить его диспозицию в значение по умолчанию, иначе функция `sigwait` никогда не сможет получить его.

Далее блокируются все сигналы, поскольку это рекомендуется для многопоточных программ, и создается поток, который будет заниматься обработкой сигналов. Поток обслуживает только сигналы `SIGHUP` и `SIGTERM`. При получении сигнала `SIGHUP` функция `reread` перечитывает файл конфигурации, а при получении сигнала `SIGTERM` поток записывает сообщение в журнал и завершает работу процесса.

В табл. 10.1 указано, что по умолчанию сигналы `SIGHUP` и `SIGTERM` завершают процесс. Поскольку эти сигналы заблокированы, демон не будет завершаться, если получит один из них. Вместо этого поток, вызывая `sigwait`, будет получать номера доставленных сигналов.

Пример

Программа в листинге 13.4 показывает, как демон может перехватить сигнал `SIGHUP` и выполнить повторное чтение конфигурационного файла, не используя для этого отдельный поток.

Листинг 13.4. Альтернативная реализация демона, который перечитывает конфигурационный файл по сигналу

```

#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

```

```
void
reread(void)
{
    /* ... */
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Чтение конфигурационного файла");
    reread();
}

int
main(int argc, char *argv[])
{
    char                  *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);

    /*
     * Убедиться, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Установить обработчики сигналов.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "невозможно перехватить сигнал SIGTERM: %s",
               strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
```

```
if (sigaction(SIGHUP, &sa, NULL) < 0) {
    syslog(LOG_ERR, "невозможно перехватить сигнал SIGHUP: %s",
           strerror(errno));
    exit(1);
}

/*
 * Остальная часть программы-демона.
 */
/* ... */
exit(0);
}
```

После инициализации демона устанавливаются обработчики сигналов **SIGHUP** и **SIGTERM**. У нас есть два варианта обработки сигнала **SIGHUP**: прочитать конфигурационный файл в функции-обработчике или просто установить в обработчике специальный флаг, а все необходимые действия выполнять в основном потоке программы.

13.7. Модель клиент-сервер

Наиболее часто процессы-демоны используются в качестве серверных процессов. На рис. 13.1 показан пример взаимодействия с сервером **syslogd**, который получает сообщения от приложений (клиентов) посредством сокета домена UNIX.

Вообще, под сервером подразумевается некий процесс, который ожидает запросов на предоставление определенных услуг клиентам. Так, на рис. 13.1 сервер **syslogd** предоставляет услуги журнализирования сообщений об ошибках.

Показанное на рис. 13.1 взаимодействие между сервером и клиентом носит односторонний характер. Клиент отсылает сообщения серверу, но ничего от него не получает. В последующих главах мы увидим множество примеров двустороннего взаимодействия сервера и клиента, когда клиент посыпает запрос серверу, а сервер возвращает клиенту ответ.

Серверы часто обеспечивают обслуживание клиентов, запуская другие программы с помощью **fork** и **exec**. Такие серверы нередко открывают множество файловых дескрипторов: конечных точек взаимодействий, конфигурационных файлов, файлов журналов и др. В лучшем случае будет просто небрежностью оставлять дескрипторы открытыми в дочернем процессе, потому что они, скорее всего, не будут использоваться в программе, запускаемой потомком, особенно если эта программа никак не связана с сервером, в худшем — это может привести к проблемам с безопасностью: запускаемая программа может попытаться выполнить какие-нибудь злонамеренные действия, например изменить конфигурационный файл сервера или обманным путем получить от клиента важную информацию.

Самое простое решение этой проблемы: установить флаг закрытия при вызове функции **exec** (close-on-exec) для всех файловых дескрипторов, которые не требуются запускаемой программе. В листинге 13.5 приводится функция, которую можно использовать в серверном процессе для этих целей.

Листинг 13.5. Установка флага закрытия при вызове exec

```
#include "apue.h"
#include <fcntl.h>

int
set_cloexec(int fd)
{
    int     val;

    if ((val = fcntl(fd, F_GETFD, 0)) < 0)
        return(-1);

    val |= FD_CLOEXEC; /* установить флаг закрытия при вызове exec */

    return(fcntl(fd, F_SETFD, val));
}
```

13.8. Подведение итогов

Время работы процессов-демонов в большинстве случаев совпадает с временем работы самой системы. При разработке программ, которые будут работать как демоны, необходимо понимать и учитывать взаимоотношения между процессами, которые были описаны в главе 9. В этой главе мы разработали функцию, которую можно вызывать из процесса для корректного перехода в режим демона.

Мы также обсудили способы журналирования сообщений об ошибках демонов, поскольку они, как правило, не имеют управляющего терминала. Мы рассмотрели ряд соглашений, которым должны следовать демоны в большинстве версий UNIX, и показали примеры реализации этих соглашений.

Упражнения

- 13.1 Исходя из рис. 13.1 можно предположить, что при инициализации механизма `syslog` (прямым обращением к функции `openlog` или при первом обращении к функции `syslog`) он открывает специальный файл устройства `/dev/log`. Что произойдет, если пользовательский процесс (демон) вызовет функцию `chroot` перед вызовом `openlog`?
- 13.2 Вспомните пример вывода программы `ps` в разделе 13.2. Единственным демоном уровня пользователя, не являющимся лидером сеанса, является процесс `rsyslogd`. Объясните, почему демон `syslogd` не является лидером сеанса.
- 13.3 Перечислите все демоны в вашей системе и укажите их функциональное назначение.
- 13.4 Напишите программу, которая вызывает функцию `daemonize` из листинга 13.1. После вызова этой функции определите имя пользователя с помощью `getlogin` (раздел 8.15), чтобы узнать, не изменился ли пользователь процесса после перехода в режим демона. Выведите полученные результаты в файл.

14

Расширенные операции ввода/вывода

14.1. Введение

В этой главе мы обсудим большое количество тем и функций, которые объединяются под общим термином *расширенные операции ввода/вывода*: неблокирующий ввод/вывод, блокировка записей, мультиплексирование операций ввода/вывода (функции `select` и `poll`), асинхронный ввод/вывод, функции `readv` и `writev` и ввод/вывод для файлов, отображаемых в память (`mmap`). Нам необходимо рассмотреть эти темы, прежде чем мы перейдем к обсуждению межпроцессных взаимодействий в главах 15 и 17 и в многочисленных примерах в последующих главах.

14.2. Неблокирующий ввод/вывод

В разделе 10.5 мы говорили, что системные вызовы подразделяются на две категории: «медленные» и все остальные. Медленными называют такие системные вызовы, которые могут заблокировать процесс «навечно». В эту категорию входят:

- Операция чтения, которая может «навечно» заблокировать вызывающий процесс, если в файлах определенных типов (каналы, терминальные устройства, сетевые устройства) отсутствуют данные, доступные для чтения.
- Операция записи также может «навечно» заблокировать вызывающий процесс, если данные не могут быть немедленно записаны в файлы тех же типов (отсутствует место в канале, переполнено сетевое соединение и т. п.).
- Операция открытия может заблокировать вызывающий процесс, пока не будут соблюдены некоторые условия для файлов определенных типов (например, открытие терминального устройства не может быть произведено, пока не будет установлено соединение между модемами, или открытие именованного канала FIFO только на запись будет заблокировано, пока не появится другой процесс, который откроет этот канал на чтение).
- Операции чтения и записи над файлами, для которых установлена принудительная блокировка записей.
- Некоторые операции `ioctl`.
- Некоторые функции, относящиеся к механизму межпроцессных взаимодействий (глава 15).

Мы также говорили, что системные вызовы, связанные с дисковыми операциями ввода/вывода, не относятся к категории медленных, хотя операция чтения с диска или записи на диск может на какое-то время заблокировать вызывающий процесс.

Неблокирующий режим ввода/вывода позволяет запускать такие операции, как `open`, `read` или `write`, не опасаясь, что они заблокируют процесс. Если запрошенная операция не может быть выполнена немедленно, системный вызов тут же вернет управление вызывающему процессу с признаком ошибки, сообщающим, что операция может быть заблокирована.

Существует два способа указать, что для заданного дескриптора файла должны использоваться неблокирующие операции ввода/вывода.

1. Если для получения дескриптора вызывается функция `open`, можно указать флаг `O_NONBLOCK` (раздел 3.3).
2. Чтобы включить флаг `O_NONBLOCK` для уже открытого дескриптора, следует воспользоваться функцией `fcntl` (раздел 3.14). В листинге 3.5 приводится функция, с помощью которой можно установить любой флаг дескриптора файла.

В ранних версиях System V для выбора неблокирующего режима операций ввода/вывода использовался флаг `O_NDELAY`. В этих версиях при отсутствии доступных для чтения данных функция `read` возвращала значение 0. Поскольку это противоречит принятому в UNIX соглашению, в соответствии с которым функция `read` возвращает 0 по достижении конца файла, стандарт POSIX.1 определил флаг неблокирующего режима с другим именем и с другой семантикой. В старых версиях System V, когда функция `read` возвращала значение 0, нельзя было определить, то ли это системный вызов вернул управление, потому что мог быть заблокирован, то ли действительно был достигнут конец файла. Стандарт POSIX.1 требует, чтобы в неблокирующем режиме при отсутствии доступных для чтения данных функция `read` возвращала признак ошибки –1 и код ошибки `EAGAIN` в переменной `errno`. Некоторые версии UNIX, происходящие от System V, поддерживают оба флага – и устаревший `O_NDELAY`, и определяемый стандартом POSIX.1 `O_NONBLOCK`, но в данной книге мы будем использовать только ту функциональность, которая определяется стандартом POSIX.1. Флаг `O_NDELAY` поддерживается лишь для сохранения обратной совместимости и не должен использоваться в новых приложениях.

В 4.3BSD появился флаг `FNDELAY` функции `fcntl` с несколько иной семантикой. Он воздействовал не только на флаги состояния файла в его дескрипторе – изменялись также флаги терминального устройства или сокета, что оказывало влияние на всех пользователей терминала или сокета, а не только на пользователей, совместно использующих одну и ту же запись в таблице файлов (в 4.3BSD неблокирующий режим ввода/вывода мог назначаться только терминальным устройствам или сокетам). Кроме того, в 4.3BSD в вызывающую программу возвращалось значение `EWOLDBLOCK`, если операция с дескриптором в неблокирующем режиме не могла быть завершена. Современные BSD-системы поддерживают флаг `O_NONBLOCK` и определяют константу `EWOLDBLOCK` с тем же значением, что и `EAGAIN`. Эти системы предоставляют семантику неблокирующего режима, совместимую со стандартом POSIX.1: изменения флагов состояния файла оказывают влияние на всех пользователей одной и той же записи в таблице файлов, но не затрагивают режимы работы с одним и тем же устройством, если для доступа к нему используются различные записи в таблице файлов (рис. 3.1 и 3.3).

Пример

Рассмотрим пример ввода/вывода в неблокирующем режиме. Программа в листинге 14.1 читает 500 000 байт со стандартного ввода и пытается вывести их в стандартный вывод. Стандартное устройство вывода предварительно переводится в неблокирующий режим. Вывод результатов каждой операции записи производится в стандартное устройство вывода сообщений об ошибках. Функция `clr_f1` очень похожа на функцию `set_f1` из листинга 3.5. Она просто сбрасывает один или более флагов.

Листинг 14.1. Вывод большого количества данных в неблокирующем режиме

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int      ntwrite, nwrite;
    char    *ptr;

    ntwrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "прочитано %d байт\n", ntwrite);

    set_f1 STDOUT_FILENO, O_NONBLOCK); /* установить неблокирующий режим */

    ptr = buf;
    while (ntwrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntwrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntwrite -= nwrite;
        }
    }

    clr_f1(STDOUT_FILENO, O_NONBLOCK); /* выход из неблокирующего режима */

    exit(0);
}
```

Мы предполагаем, что функция `write` отработает всего один раз, если стандартный вывод перенаправить в обычный файл:

```
$ ls -l /etc/services          проверим размер файла
-rw-r--r-- 1 root    677959 Jun 23 2009 /etc/services
$ ./a.out < /etc/services > temp.file для начала попробуем с обычным файлом
прочитано 500000 байт
nwrite = 500000, errno = 0           единственный вызов write
$ ls -l temp.file                 проверим размер получившегося файла
-rw-rw-r-- 1 sar     500000 Apr 1 13:03 temp.file
```

Но если в качестве устройства вывода будет использоваться терминал, мы предполагаем, что функция `write` будет иногда возвращать меньшее значение счетчика, а иногда — признак ошибки. Вот что мы получили в этом случае:

```
$ ./a.out < /etc/services 2>stderr.out вывод в терминал

$ cat stderr.out
прочитано 500000 байт
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = -1, errno = 35
nwrite = 1002, errno = 0
nwrite = 1004, errno = 0
nwrite = 1003, errno = 0
nwrite = 1003, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                                61 такая ошибка
. . .
nwrite = 1006, errno = 0
nwrite = 1004, errno = 0
nwrite = 1005, errno = 0
nwrite = 1006, errno = 0
nwrite = -1, errno = 35                                108 таких ошибок
. . .
nwrite = 1006, errno = 0
nwrite = 1005, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                                681 такая ошибка
. . .
и так далее...
nwrite = 347, errno = 0
```

В данной системе число 35 соответствует коду ошибки `EAGAIN`. Объем данных, принимаемых терминалом за одно обращение, варьируется от системы к системе. Результаты также зависят от того, как был произведен вход в систему — с консоли, с удаленного терминала или через сетевое соединение, которое использует псевдотерминал. Если на вашем терминале работает многооконная система, значит, вы также работаете через устройство псевдотерминала.

В этом примере программа произвела более 9000 вызовов функции `write`, хотя фактически вся работа была выполнена 500 вызовами. Остальные только возвращали признак ошибки. Такой тип цикла называется *опросом* (polling), и в многопользовательских системах он понапрасну расходует процессорное время. В разделе 14.4 мы рассмотрим более эффективный подход к работе с дескрипторами в неблокирующем режиме.

Иногда удается избежать применения неблокирующих операций ввода/вывода за счет использования потоков (глава 11). В этом случае можно позволить заблокировать один поток, если другие потоки смогут продолжать работу. Иногда такой подход упрощает архитектуру приложения, как будет показано в главе 21; однако в некоторых случаях проблемы, связанные с необходимостью синхронизации потоков, могут свести на нет все преимущества многопоточной модели.

14.3. Блокировка записей

Что произойдет, если два пользователя попытаются одновременно редактировать один файл? В большинстве версий UNIX окончательное содержимое файла будет определяться последней операцией записи. Однако в некоторых приложениях, таких как системы управления базами данных, процесс должен убедиться, что только он пишет в файл. С этой целью коммерческие версии UNIX предоставляют возможность блокировки отдельных записей в файле. (В главе 20 мы напишем библиотеку функций для работы с базой данных, которая использует блокировки записей.)

Термин *блокировка записи* обычно используется для описания функциональной возможности, которая позволяет одному процессу предотвратить изменение участка файла другим процессом, пока первый процесс читает или изменяет эту часть файла. Использование понятия «запись» для системы UNIX не совсем корректно, поскольку ядро UNIX не имеет представления ни о структуре файла, ни о записях в файле. Более правильный термин — *блокировка диапазона байтов*, поскольку на самом деле подразумевается некий диапазон байтов в файле (возможно, даже весь файл), доступ к которому заблокирован.

Предыстория

Ранние версии UNIX часто подвергались критике за то, что не могли использоваться для построения систем управления базами данных из-за отсутствия механизма, позволяющего блокировать доступ к отдельным участкам файлов. Позднее в различных семействах UNIX появилась поддержка механизма блокировки записей (в различных видах, разумеется).

Ранние версии из Беркли поддерживали только функцию `flock`, с помощью которой можно заблокировать файл целиком, но не отдельный его участок.

Механизм блокировки записей впервые появился в функции `fcntl` в System V Release 3. Функция `lockf` была реализована поверх `fcntl` и являла собой упрощенный интерфейс к последней. Эти функции позволяли вызывающей программе блокировать доступ к произвольному диапазону байтов, начиная от единственного байта и заканчивая всем файлом.

Стандарт POSIX.1 выбрал для стандартизации подход на основе `fcntl`. В табл. 14.1 перечислены различные формы блокировок записи, предоставляемые разными системами. Обратите внимание: стандарт Single UNIX Specification включает функцию `lockf` как расширение XSI.

Таблица 14.1. Способы блокировки записей, поддерживаемые различными версиями UNIX

Система	Рекомендательные	Принудительные	<code>fcntl</code>	<code>lockf</code>	<code>flock</code>
SUS	✓		✓	XSI	
FreeBSD 8.0	✓		✓	✓	✓
Linux 3.2.0	✓	✓	✓	✓	✓
Mac OS X 10.6.8	✓		✓	✓	✓
Solaris 10	✓	✓	✓	✓	✓

Различия между рекомендательными и принудительными блокировками рассматриваются далее в этом же разделе. В этой книге обсуждаются только блокировки на основе функции `fcntl`.

Механизм блокировки записей изначально был добавлен в Version 7 Джоном Бассом (John Bass) в 1980 году. Системный вызов в ядре получил название Locking. Эта функция представляла механизм принудительных блокировок, который прошел через множество версий System III. Xenix включала эту функцию, и некоторые производные от System V системы для архитектуры Intel, такие как OpenServer 5, продолжают поддерживать ее в библиотеке совместимости с Xenix.

Блокировка записей на основе функции `fcntl`

Приведем еще раз прототип функции `fcntl` из раздела 3.14.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* struct flock *flockptr */);
```

Возвращаемое значение зависит от аргумента *cmd* (см. ниже) в случае успеха, *-1* — в случае ошибки

При использовании этой функции для блокировки записей в аргументе *cmd* передаются значения `F_GETLK`, `F_SETLK` или `F_SETLKW`. Третий аргумент (который обозначен как *flockptr*) — это указатель на структуру `flock`.

```
struct flock {
    short l_type; /* F_RDLCK, F_WRLCK или F_UNLCK */
    short l_whence; /* SEEK_SET, SEEK_CUR или SEEK_END */
    off_t l_start; /* смещение в байтах относительно l_whence */
    off_t l_len; /* длина области в байтах; 0 означает: до конца файла */
    pid_t l_pid; /* возвращается при использовании команды F_GETLK */
};
```

Эта структура определяет:

- Тип блокировки: `F_RDLCK` (блокировка для совместного чтения), `F_WRLCK` (исключительная блокировка для записи) или `F_UNLCK` (снятие блокировки).
 - Начало блокируемой или разблокируемой области (*l_start* и *l_whence*).
 - Размер области в байтах (*l_len*).
 - Идентификатор процесса, удерживающего блокировку, которая может заблокировать текущий процесс (возвращается только для операции `F_GETLK`).
- Существует целый ряд правил, используемых для определения участка файла, который должен быть заблокирован или разблокирован.
- Два элемента структуры, которые определяют начало участка, похожи на последние два аргумента функции `lseek` (раздел 3.6). Поле *l_whence* может принимать значения `SEEK_SET`, `SEEK_CUR` или `SEEK_END`.
 - Блокируемый участок может начинаться и заканчиваться за текущим концом файла, но не может начинаться или заканчиваться перед началом файла.

- Если в поле `l_len` указано значение 0, это означает, что блокировка распространяется до конца файла. Это дает возможность заблокировать область, которая может начинаться в любом месте файла и продолжаться до конца файла, включая все данные, которые могут быть дописаны позже. (Мы не будем строить предположения по поводу того, сколько именно байтов может быть добавлено в конец файла.)
- Чтобы заблокировать весь файл, нужно установить в поля `l_start` и `l_whence` значения, соответствующие началу файла, а в поле `l_len` — значение 0. (Существует несколько способов указать начало файла, но в большинстве приложений для этого в поле `l_start` записывается 0, а в поле `l_whence` — значение `SEEK_SET`.)

Мы уже упоминали два типа блокировок — совместно используемые блокировки для чтения (в поле `l_type` записывается значение `F_RDLCK`) и исключительные блокировки для записи (`F_WRLCK`). Основное различие между этими двумя типами заключается в том, что любое количество процессов могут установить совместно используемую блокировку для чтения заданного байта, но только один сможет установить исключительную блокировку для записи в заданный байт. Кроме того, если для байта установлена хотя бы одна блокировка для чтения, никакая блокировка для записи на него уже установлена быть не может, и наоборот, если на заданный байт установлена исключительная блокировка для записи, на него уже не может быть установлена блокировка для чтения. Правила совместимости типов блокировок приводятся в табл. 14.2.

Таблица 14.2. Совместимость различных типов блокировок

		Запрос на	
		блокировку для чтения	блокировку для записи
К настоящему моменту область	Не заблокирована	OK	OK
	Уже имеет одну или более блокиро- вок для чтения	OK	Отклоняется
	Уже имеет одну блокировку для записи	Отклоняется	Отклоняется

Правило совместимости блокировок применяется к запросам, поступающим от различных процессов, а не ко множеству запросов, производимых одним процессом. Если процесс уже обладает одной блокировкой на некоторый участок файла и пытается установить блокировку на тот же участок, существующая блокировка будет замещена новой блокировкой. То есть если процесс установил блокировку для записи на диапазон с 16-го по 32-й байт и затем попытается установить блокировку для чтения на тот же диапазон, его запрос будет удовлетворен (при условии, что никакой другой процесс не пытается заблокировать тот же участок файла) и блокировка для записи будет замещена блокировкой для чтения.

Чтобы установить блокировку для чтения, дескриптор файла должен быть открыт для чтения. Чтобы установить блокировку для записи, дескриптор файла должен быть открыт для записи.

Теперь мы можем описать три команды функции `fcntl`, имеющие отношение к блокировкам.

F_GETLK Определяет, будет ли попытка установить блокировку заблокирована некоторой другой блокировкой. Если к моменту выполнения команды уже существует блокировка, которая может помешать установить новую блокировку, по адресу `flockptr` записывается информация о существующей блокировке. Если таких блокировок не существует, содержимое структуры `flockptr` не изменяется, за исключением поля `l_type`, в которое записывается значение `F_UNLCK`.

F_SETLK Установить блокировку, определение которой находится по адресу `flockptr`. При попытке установить блокировку, несовместимую с существующей (в соответствии с правилами в табл. 14.2), функция `fcntl` сразу же вернет управление с кодом ошибки `EAGAIN` или `EACCES` в переменной `errno`.

Хотя стандарт POSIX.1 допускает возврат любого кода ошибки, тем не менее все четыре реализации, обсуждаемые в данной книге, возвращают код ошибки EAGAIN, если запрос на установку блокировки не может быть удовлетворен.

Эта команда также используется для снятия блокировки. В этом случае в поле `l_type` структуры `flockptr` указывается значение `F_UNLCK`.

F_SETLKW Эта команда является блокирующей версией команды `F_SETLK`. (В данном случае `W` означает «*wait*» — «ждать».) Если запрос на установку блокировки не может быть немедленно удовлетворен из-за того, что другой процесс уже установил блокировку, несовместимую с данной, на диапазон, часть которого попадает в запрошенный диапазон, работа вызывающего процесса приостанавливается. Процесс возобновит работу, когда появится возможность установить блокировку или когда ожидание будет прервано сигналом.

Вы должны понимать, что проверка возможности установки блокировки (`F_GETLK`) с последующей попыткой установки блокировки (`F_SETLK` или `F_SETLKW`) не является атомарной операцией. Нельзя гарантировать, что между двумя вызовами `fcntl` управление не будет передано другому процессу, который пожелает установить ту же самую блокировку. Если необходимо предотвратить блокирование процесса в ожидании возможности получения блокировки, следует использовать команду `F_SETLK` и должным образом обрабатывать возвращаемый функцией результат.

Обратите внимание: стандарт POSIX.1 не определяет, что может произойти, когда один процесс уже установил блокировку для чтения на некоторый диапазон в файле, затем второй пробует установить блокировку для записи на тот же диапазон, а потом третий пытается установить на тот же диапазон блокировку для чтения. Если третьему процессу будет позволено установить блокировку для чтения на диапазон, который уже заблокирован блокировкой для чтения, реализация может «подвесить» процесс, который ожидает блокировки для записи. Это означает, что по мере поступления новых запросов на получение блокировки для чтения ожидание блокировки для записи может растянуться на неопределенное время. Если запросы на получение блокировки для чтения поступают беспорядочно и достаточно часто, процесс, ожидающий блокировки для записи, может оставаться в состоянии ожидания достаточно длительное время.

Во время установки или снятия блокировки система соединяет вместе или разбивает смежные области в соответствии с характеристиками выполняемой операции. Если, например, заблокировать байты 100–199, а затем разблокировать байт 150, ядро будет обслуживать два заблокированных диапазона: байты 100–149 и байты 151–199. Эта ситуация изображена на рис. 14.1.



Рис. 14.1. Схема блокировки диапазона байтов

Если затем установить блокировку на 150-й байт, система объединит смежные области в одну — с 100-го по 199-й байт. В этом случае конечный результат будет соответствовать первой диаграмме, приведенной на рис. 14.1, то есть той, с которой мы начали.

Пример — запрос на установку и снятие блокировки

Чтобы избавить себя от необходимости всякий раз размещать и заполнять структуру `flock`, мы написали функцию `lock_reg`, которая выполняет все необходимые действия (листинг 14.2).

Листинг 14.2. Функция наложения и снятия блокировки на участок файла

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;   /* смещение в байтах относительно l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* количество байтов (0 - до конца файла) */

    return(fcntl(fd, cmd, &lock));
}
```

Поскольку в большинстве случаев функция будет вызываться для наложения или снятия блокировки (команда `F_GETLK` используется редко), мы обычно используем один из следующих пяти макросов, определенных в заголовочном файле `apue.h` (приложение B).

```
#define read_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

Мы преднамеренно определили порядок первых трех аргументов так, чтобы он соответствовал порядку аргументов функции `lseek`.

Пример — проверка возможности наложения блокировки

В листинге 14.3 приводится исходный код функции `lock_test`, которую мы будем использовать при проверке возможности наложения блокировки.

Листинг 14.3. Функция проверки возможности наложения блокировки

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;      /* F_RDLCK или F_WRLCK */
    lock.l_start = offset;   /* смещение в байтах относительно l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* количество байтов (0 - до конца файла) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);           /* ложь, заданная область не заблокирована */
                           /* другим процессом */
    return(lock.l_pid); /* истина, вернуть pid владельца блокировки */
}
```

Если уже существует блокировка, которая может заблокировать выполнение запроса с заданными параметрами, эта функция возвращает идентификатор процесса, владеющего блокировкой. Иначе возвращается 0 (ложь). Мы обычно будем вызывать эту функцию из следующих двух макросов (определенных в файле `apue.h`).

```
#define is_read_lockable(fd, offset, whence, len) \
    (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
    (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Обратите внимание, что функция `lock_test` не может использоваться процессом, чтобы проверить, является ли он владельцем блокировки заданного участка файла. Определение команды `F_GETLK` гласит, что возвращаемая информация относится к существующей блокировке, которая может помешать наложению новой блокировки. Так как команды `F_SETLK` и `F_SETLKW` всегда заменяют существующую блокировку, если ее владельцем является вызывающий процесс, мы никогда не сможем заблокировать процесс на своей собственной блокировке. То есть команда `F_GETLK` никогда не будет сообщать о наличии блокировки, если эта блокировка принадлежит вызывающему процессу.

Пример — тупиковая ситуация

Тупиковая ситуация (или ситуация взаимоблокировки) возникает, когда каждый из двух процессов ожидает освобождения ресурса, заблокированного другим процессом. Опасность тупиковой ситуации возникает, если процесс владеет блокировкой на некоторый участок файла и, пытаясь наложить блокировку на другой участок, приостанавливается в ожидании снятия с этого участка блокировки, установленной другим процессом.

В листинге 14.4 приводится пример такой тупиковой ситуации. Дочерний процесс блокирует доступ к байту 0, а родительский процесс — к байту 1. После этого каждый из процессов пытается заблокировать байт, уже заблокированный другим процессом. Для синхронизации родительского и дочернего процессов мы использовали процедуры из раздела 8.9 (`TELL_xxx` и `WAIT_xxx`), дающие возможность каждому процессу дождаться другого процесса, чтобы наложить нужную блокировку.

Листинг 14.4. Пример выявления тупиковой ситуации

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: ошибка вызова функции writew_lock", name);
    printf("%s: установлена блокировка на байт %lld\n",
           name, (long long)offset);
}

int
main(void)
{
    int      fd;
    pid_t   pid;

    /*
     * Создать файл и записать в него два байта.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("ошибка вызова функции creat");
    if (write(fd, "ab", 2) != 2)
        err_sys("ошибка вызова функции write");

    TELL_WAIT();
```

```

if ((pid = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) { /* потомок */
    lockabyte ("потомок", fd, 0);
    TELL_PARENT(getppid());
    WAIT_PARENT();
    lockabyte("потомок", fd, 1);
} else { /* родитель */
    lockabyte("родитель", fd, 1);
    TELL_CHILD(pid);
    WAIT_CHILD();
    lockabyte("родитель", fd, 0);
}
exit(0);
}

```

Запустив программу из листинга 14.4, получили:

```

$ ./a.out
родитель: установлена блокировка на байт 1
потомок: установлена блокировка на байт 0
потомок: ошибка вызова функции writew_lock: Resource deadlock avoided
родитель: установлена блокировка на байт 0

```

Когда ядро обнаруживает наличие тупиковой ситуации, оно возвращает одному из процессов признак ошибки. В данном случае признак ошибки был возвращен дочернему процессу. В одних системах признак ошибки всегда получает дочерний процесс, в других — родительский, в третьих признак ошибки возвращается обоим процессам.

Правила наследования блокировок

Наследование и снятие блокировок записей в файле производится в соответствии со следующими тремя правилами.

- Блокировки ассоциируются с процессом и с файлом. Это проявляется в следующем. Во-первых, по завершении процесса все его блокировки освобождаются. Во-вторых, когда закрывается дескриптор, освобождаются все блокировки, связанные с файлом, на который ссылается заданный дескриптор. Это означает, что если программа выполняет код

```

fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = dup(fd1);
close(fd2);

```

после закрытия дескриптора `fd2` блокировка, установленная для дескриптора `fd1`, освобождается. То же произойдет, если заменить вызов функции `dup` вызовом `open`, как в следующем фрагменте:

```

fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = open(pathname, ...)
close(fd2);

```

где открывается тот же файл, но с другим дескриптором.

2. Блокировки никогда не наследуются дочерним процессом, созданным функцией `fork`. То есть если процесс установил блокировку, а затем вызвал функцию `fork`, с точки зрения блокировок, установленных родительским процессом, дочерний процесс будет рассматриваться как совершенно другой процесс. Потомок должен будет вызывать функцию `fcntl`, чтобы установить собственные блокировки для любых дескрипторов, унаследованных от родителя. Такое положение вещей имеет определенный смысл, так как предотвращает возможность записи в один и тот же участок файла из нескольких процессов. Если бы дочерний процесс наследовал родительские блокировки, они оба смогли бы одновременно писать в один и тот же файл.
3. Блокировки наследуются новыми программами при вызове функции `exec`. Однако если для дескриптора установлен флаг `close-on-exec` (закрыть при вызове `exec`), все блокировки, связанные с данным файлом, освобождаются при закрытии дескриптора в функции `exec`.

Реализация в FreeBSD

Давайте поближе познакомимся со структурами данных, которые используются в FreeBSD. Это поможет вам лучше понять правило 1, которое утверждает, что блокировки связаны с процессом и с файлом.

Рассмотрим следующий фрагмент программы (не принимая во внимание случаи, когда обращения к функциям завершаются неудачей):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1); /* родитель устанавливает блокировку */
/* для записи на байт с номером 0 */
if ((pid = fork()) > 0) {           /* родительский процесс */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* потомок устанавливает блокировку */
    /* для чтения на байт с номером 1 */
}
pause();
```

На рис. 14.2 показано состояние структур данных после того, как оба процесса вызовут функцию `pause`.

Ранее мы уже показывали состояние структур данных после вызова функций `open`, `fork` и `dup` (рис. 3.3 и 8.1). Единственное, что изменилось здесь, — появились структуры `lockf`, которые связаны со структурой виртуального узла. Каждая структура `lockf` описывает отдельную область в файле (которая определяется началом и длиной), заблокированную конкретным процессом. Здесь показаны две такие структуры: одна создана вызовом `write_lock` из родительского процесса, а вторая — вызовом `read_lock` из дочернего. Каждая структура содержит соответствующий идентификатор процесса.

При закрытии любого из трех дескрипторов в родительском процессе — `fd1`, `fd2` или `fd3` — блокировка для записи снимается. Когда закрывается какой-либо из этих дескрипторов, ядро обходит связанный список блокировок для соответству-

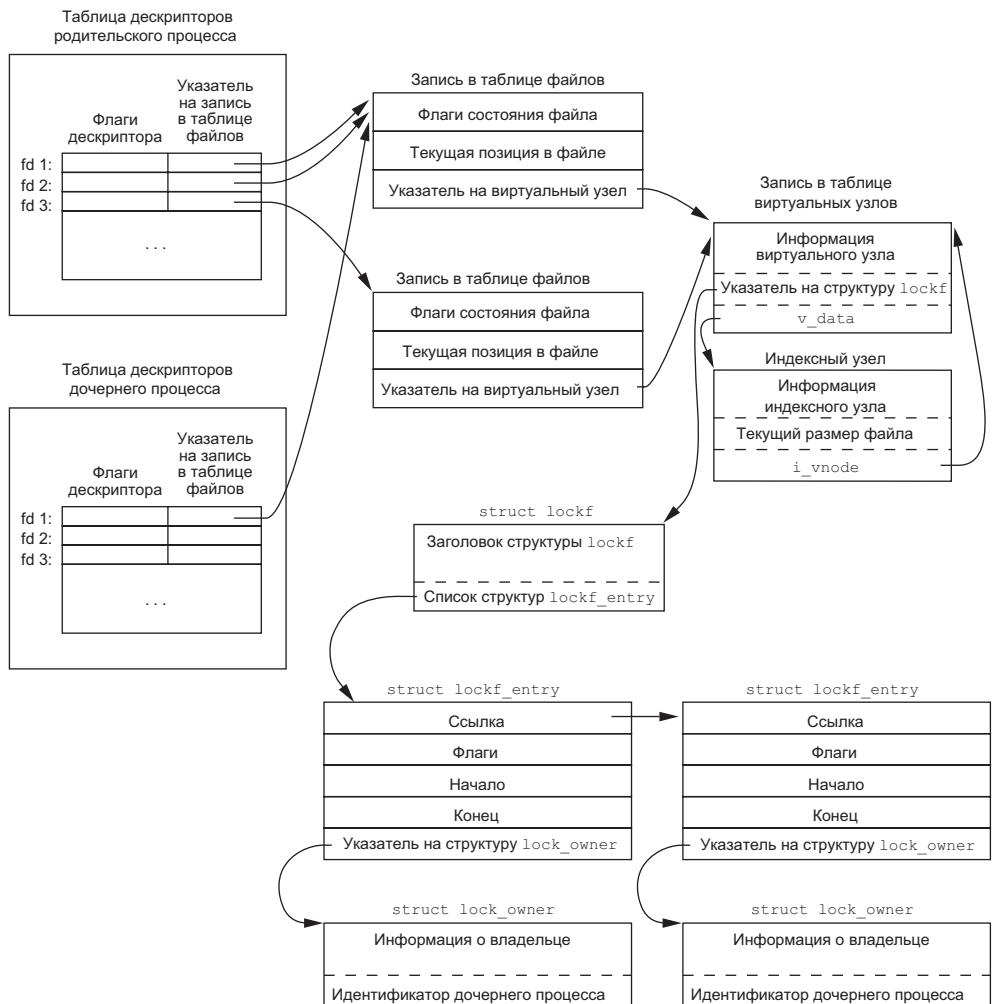


Рис. 14.2. Структуры данных, связанные с блокировками записей в файле, в FreeBSD

ющего индексного узла и освобождает все блокировки, установленные вызывающим процессом. Ядро не имеет возможности определить, какой дескриптор использован родительским процессом для установки блокировки.

Пример

В программе из листинга 13.2 мы видели, как демон может использовать блокировку файла, чтобы обеспечить запуск единственного экземпляра программы. В листинге 14.5 приводится реализация функции `lockfile`, которая использовалась демоном, чтобы установить блокировку для записи.

Листинг 14.5. Установка блокировки для записи на весь файл

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Как вариант мы могли бы определить функцию `lockfile` в терминах функции `write_lock`:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

Блокировки в конце файла

С особой осторожностью следует подходить к установке блокировок, когда начало области задается относительно конца файла. В большинстве реализаций значения поля `l_whence` (`SEEK_CUR` и `SEEK_END`) преобразуются в абсолютное смещение с использованием значений поля `l_start` и текущей позиции или текущей длины файла. Однако зачастую возникает необходимость указывать начало области относительно текущей позиции или текущей длины файла, потому что мы не можем вызывать функцию `fstat` для получения значения текущей позиции в файле, так как не владеем блокировкой. (В этот момент у других процессов появляется шанс вклиниваться между вызовами функции `fstat` и функции, которая устанавливает блокировку, и изменить длину файла.)

Рассмотрим следующую последовательность действий:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

Этот код может делать совсем не то, что вы от него ожидаете. Здесь устанавливается блокировка для записи, начиная от текущего конца файла и дальше, включая данные, которые могут быть добавлены в конец файла позже.

Предположим, что текущая позиция находится в конце файла, тогда первый вызов `write` добавит один байт в конец файла, и этот байт будет заблокирован. Следующая затем операция снятия блокировки разблокирует все данные, которые могут быть добавлены в конец файла позже, но оставит текущий последний байт заблокированным. Когда будет выполнена вторая операция записи, размер файла увеличится еще на один байт, и этот байт не будет заблокирован. Состояние блокировок для данной последовательности действий показано на рис. 14.3.



Рис. 14.3. Схема заблокированных участков файла

Когда на участок файла устанавливается блокировка, ядро преобразует указанное смещение в абсолютное смещение относительно начала файла. Кроме смещения относительно начала файла (`SEEK_SET`), функция `fcntl` позволяет указать смещение относительно текущей позиции в файле (`SEEK_CUR`) или относительно конца файла (`SEEK_END`). Ядро вынуждено запоминать положение блокировок в представлении, не зависящем от текущей позиции или конца файла, потому что текущая позиция или размер файла могут измениться, но эти изменения не должны влиять на положение блокировки.

Чтобы удалить блокировку байта, добавленного первой операцией записи, мы должны были бы указать значение `-1` в качестве длины участка. Отрицательное значение длины соответствует участку, расположенному перед заданным смещением.

Рекомендательные и принудительные блокировки

Рассмотрим библиотеку процедур, обеспечивающих доступ к базе данных. Если все функции в библиотеке используют возможность блокировки записей в файле непротиворечивым способом, мы можем сказать, что любое множество процессов, использующих для доступа к базе данных эти функции, является кооперативными процессами (сотрудничающими друг с другом). Для данных функций вполне подходит рекомендательный тип блокировок, при условии, что только эти функции используются для доступа к базе данных. Но рекомендательные блокировки не могут предотвратить возможность записи в файлы базы данных из других процессов, которые имеют право на запись в эти файлы. Такой «жульничавший» процесс можно назвать некооперативным (несотрудничающим), так как он не использует общепринятые методы (библиотека функций) для доступа к базе данных.

Принудительные блокировки вынуждают ядро проверять каждую операцию `open`, `read` и `write` на предмет противоречия блокировкам, связанным с файлом. Принудительные блокировки иногда называют блокировками *форсированного режима*.

В табл. 14.1 мы видели, что Linux 3.2.0 и Solaris 10 поддерживают принудительные блокировки, а FreeBSD 8.0 и Mac OS X 10.6.8 – нет. Механизм принудительных блокировок не является частью стандарта *Single UNIX Specification*. При желании использовать принудительные блокировки в Linux вам придется сделать это на уровне файловой системы, для чего необходимо использовать параметр *-o mand* команды *mount*.

Применение принудительных блокировок к отдельным файлам разрешается включением бита set-group-ID и выключением group-execute (листинг 4.4). Поскольку установка бита set-group-ID теряет смысл при сброшенном бите group-execute, разработчики SVR3 выбрали именно такой способ указать, что файл должен подвергаться принудительной, а не рекомендательной блокировке.

Что произойдет, если процесс попытается выполнить операцию чтения или записи в файл, для которого разрешена принудительная блокировка и указанная часть файла как раз находится под защитой блокировки для чтения или для записи, установленной другим процессом? Ответ на этот вопрос зависит от типа операции (чтение или запись), типа блокировки, установленной другим процессом (для чтения или для записи), и от того, был ли открыт дескриптор файла в неблокирующем режиме. В табл. 14.3 приводится восемь различных вариантов ответа на этот вопрос.

Таблица 14.3. Воздействие принудительных блокировок на операции чтения/записи из других процессов

Тип блокировки, установленной другим процессом	Дескриптор в блокирующем режиме		Дескриптор в неблокирующем режиме	
	read	write	read	write
Для чтения	OK	Блокируется	OK	EAGAIN
Для записи	Блокируется	Блокируется	EAGAIN	EAGAIN

Кроме операций *read* и *write*, указанных в табл. 14.3, принудительные блокировки могут также оказывать влияние на операцию открытия файла другим процессом. Обычно вызов функции *open* завершается успехом, даже если открываемый файл находится под защитой принудительной блокировки. В этом случае последующие операции чтения и записи будут выполняться в соответствии с правилами из табл. 14.3. Но если открываемый файл находится под защитой принудительной блокировки (неважно, для чтения или для записи) и функции *open* передается флаг *O_TRUNC* или *O_CREAT*, в этом случае операция открытия файла будет завершаться неудачей с кодом ошибки *EAGAIN* и управление из функции *open* будет немедленно возвращено вызывающему процессу, независимо от наличия флага *O_NONBLOCK*.

Только Solaris трактует использование флага *O_CREAT* в данной ситуации как ошибку. Linux допускает указывать этот флаг при открытии файла, на который установлена принудительная блокировка. То, что функция *open* возвращает признак ошибки при использовании флага *O_TRUNC*, вполне оправданно, потому что файл не может быть усечен, если он находится под защитой блокировки для чтения или для записи, установленной другим процессом. Генерировать ошибку для флага *O_CREAT* не имеет большого смысла,

поскольку этот флаг говорит о том, что файл должен быть создан, только если он не существует. Однако файл должен существовать, если другой процесс смог установить на него блокировку.

Изучение конфликтов между функцией `open` и блокировками может привести к неожиданным результатам. При разработке упражнений для этого раздела мы запускали тестовую программу, которая открывала файл (с разрешенным режимом принудительной блокировки), устанавливала блокировку для чтения на весь файл и затем приостанавливалась на некоторое время. (В табл. 14.3 показано, что блокировка для чтения должна предотвратить возможность записи в этот файл.) Пока программа находилась в режиме ожидания, было отмечено следующее поведение стандартных программ UNIX.

- Этот файл можно было редактировать с помощью программы `ed`, и результаты записывались на диск! Получалось так, что принудительная блокировка вообще не оказывала никакого эффекта. С помощью системного вызова `trace`, который поддерживается некоторыми версиями UNIX, удалось выяснить, что редактор `ed` записывает обновленное содержимое во временный файл, удаляет оригинальный файл и затем переименовывает временный файл, называя его именем оригинального файла. Обязательная блокировка не оказывает влияния на функцию `unlink`, в результате чего подобное оказалось возможным.

В FreeBSD 8.0 и Solaris 10 системный вызов `trace` используется командой `truss(1)`. В Linux 3.2.0 ту же роль играет команда `strace(1)`. В Mac OS X 10.6.8 для трассировки системных вызовов, производимых процессом, предоставляется команда `dtruss(1m)`.

- Редактор `vi` не способен редактировать такой файл. Он мог прочитать содержимое файла, но при попытке сохранить его получал код ошибки `EAGAIN`. При попытках добавить в файл новые данные функция `write` блокировалась. Впрочем, мы предвидели такое поведение редактора `vi`.
- При использовании операторов перенаправления `>` и `>>` командной оболочки Korn shell для записи или добавления данных в файл мы получили ошибку «`cannot create`» («невозможно создать»).
- При использовании тех же самых операторов перенаправления в Bourne shell мы получали ошибку только в случае оператора `>`, выполнение же оператора `>>` просто блокировалось до момента снятия блокировки. (Различия в действиях оператора перенаправления `>>` для Korn shell и Bourne shell объясняются тем, что в Korn shell этот оператор вызывает функцию `open` с флагами `O_CREAT` и `O_APPEND`, а мы уже упоминали, что использование флага `O_CREAT` в подобной ситуации расценивается как ошибка. В командной оболочке Bourne shell функция `open` вызывается без флага `O_CREAT`, если запрошенный файл уже существует, поэтому обращение к функции `open` завершается успехом, а последующее обращение к функции `write` блокируется системой.)

Результаты могут различаться в разных версиях операционной системы. Этот пример показывает, насколько осторожно следует подходить к использованию принудительных блокировок. Кроме того, пример с редактором `ed` показывает, что обойти принудительные блокировки не составляет особого труда.

Принудительные блокировки могут использоваться злонамеренным пользователем, чтобы ограничить доступ к некоторому общедоступному файлу только режимом чтения, установив на него принудительную блокировку для чтения. Такой прием не позволит никому изменить содержимое файла. (Разумеется, файл при этом должен быть доступен для установки принудительной блокировки, для чего пользователь должен иметь право на изменение прав доступа к файлу.) Представьте себе файл базы данных, доступный для чтения всем и для которого установлена принудительная блокировка. Если злоумышленник сможет установить принудительную блокировку для чтения на весь файл, никакой процесс не сможет записать в файл новые данные.

Пример

Программа в листинге 14.6 определяет, поддерживает ли система принудительные блокировки.

Листинг 14.6. Определяет, поддерживает ли система принудительные блокировки

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int             fd;
    pid_t          pid;
    char           buf[5];
    struct stat    statbuf;

    if (argc != 2) {
        fprintf(stderr, "Использование: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("ошибка вызова функции open");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("ошибка вызова функции write");

    /* включить бит set-group-ID и выключить бит GROUP-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("ошибка вызова функции fstat");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("ошибка вызова функции fchmod");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        /* установить блокировку для записи на весь файл */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("ошибка вызова функции write_lock");

        TELL_CHILD(pid);
    }
}
```

```

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("ошибка вызова функции waitpid");
    } else { /* дочерний процесс */
        WAIT_PARENT(); /* дождаться, пока предок установит блокировку */

        set_f1(fd, O_NONBLOCK);

        /*
         * Прежде всего, посмотрим, возможно ли установить
         * другую блокировку на уже заблокированную область.
         */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1) /* не ждать */
            err_sys("потомок: вызов read_lock завершился успехом");
        printf("вызов read_lock для заблокированного региона вернул код %d\n",
               errno);

        /* теперь попробуем читать из файла под принудительной блокировкой */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("ошибка вызова функции lseek");
        if (read(fd, buf, 2) < 0)
            err_ret("ошибка чтения (принуд. блокировка сработала)");
        else
            printf("данные прочитаны (принуд. блокировка не сработала),
                   buf = %2.2s\n", buf);
    }
    exit(0);
}

```

Эта программа создает файл и разрешает установку на него принудительных блокировок. После этого программа делится на два процесса. Родительский процесс устанавливает блокировку для записи на весь файл. Дочерний процесс устанавливает для дескриптора неблокирующий режим и затем пытается установить на файл блокировку для чтения, ожидая получить ошибку. Это позволит нам увидеть, возвращает ли система код ошибки `EACCES` или `EAGAIN`. После этого дочерний процесс переходит в начало файла и предпринимает попытку чтения из него. Если система поддерживает принудительные блокировки, функция `read` должна вернуть признак ошибки с кодом `EACCES` или `EAGAIN` (поскольку дескриптор находится в неблокирующем режиме). Иначе функция `read` вернет данные, которые удалось прочитать. Запуск этой программы в Solaris 10 (которая поддерживает принудительные блокировки) дал следующие результаты:

```
$ ./a.out temp.lock
вызов read_lock для заблокированного региона вернул код 11
ошибка чтения (принуд. блокировка сработала): Resource temporarily unavailable
```

Если заглянуть в заголовочные файлы системы или в страницу справочного руководства `intro(2)`, мы увидим, что коду 11 соответствует ошибка `EAGAIN`. В FreeBSD 8.0 были получены следующие результаты:

```
$ ./a.out temp.lock
вызов read_lock для заблокированного региона вернул код 35
данные прочитаны (принуд. блокировка не сработала), buf = ab
```

Коду 35 соответствует ошибка `EAGAIN`. Принудительные блокировки не поддерживаются.

Пример

А теперь вернемся к главному вопросу этого раздела: что случится, если два пользователя одновременно попытаются редактировать один и тот же файл? Обычные текстовые редакторы в UNIX не используют механизм блокировки записей, следовательно, ответ на этот вопрос остается прежним: результат будет соответствовать тому, что запишет в файл последний процесс.

Некоторые версии редактора *vi* используют рекомендательные блокировки записи в файле. Даже если мы будем пользоваться одной из таких версий *vi*, это все равно не сможет предотвратить использование других редакторов, которые ничего не знают о рекомендательных блокировках.

Если система поддерживает механизм принудительных блокировок, мы можем изменить свой любимый редактор так, чтобы он пользовался ими (при наличии исходных текстов). Если исходные тексты редактора недоступны, мы могли бы попробовать написать программу, реализующую интерфейс к редактору *vi*. Предполагается, что программа сразу же должна вызывать функцию *fork*, после которой родительский процесс просто становится в ожидание завершения потомка. Дочерний процесс должен открыть указанный файл, разрешить для него установку принудительных блокировок, установить блокировку для записи на весь файл и затем запустить редактор *vi*. Пока работает редактор, файл будет находиться под защитой принудительной блокировки, вследствие чего никто из пользователей не сможет изменить его. По завершении работы редактора родительский процесс получит управление от функции *wait* и завершится сам.

Подобную программу можно написать достаточно быстро, но она не будет работать. Проблема в том, что большинство известных редакторов читают содержимое входного файла и закрывают его. Когда дескриптор, связанный с файлом, закрывается, освобождается и блокировка. Это означает, что когда редактор закрывает файл после чтения его содержимого, блокировка снимается. И нет никакой возможности предотвратить снятие блокировки.

В главе 20 мы будем использовать механизм блокировки записей в библиотеке для работы с базой данных, чтобы обеспечить параллельный доступ к ней из нескольких процессов. Мы также проведем ряд тестов на производительность, чтобы увидеть, какой эффект оказывают блокировки записей на производительность процесса.

14.4. Мультиплексирование ввода/вывода

При чтении из одного дескриптора и записи в другой можно в цикле использовать блокирующие операции ввода/вывода — например, так:

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("ошибка вызова функции write");
```

Мы уже много раз встречали такую форму блокирующего ввода/вывода. А что делать, если нужно читать из двух дескрипторов? В этом случае нельзя исполь-

зователь блокирующую операцию чтения для любого из них, так как данные могут появиться в одном дескрипторе, в то время как процесс заблокирован в ожидании появления данных в другом. Для решения этой проблемы существуют различные приемы.

Давайте рассмотрим структуру программы `telnet(1)`. Эта программа читает данные с терминала (стандартный ввод) и записывает их в сетевое соединение, и в обратном порядке — читает из сетевого соединения и выводит на терминал (стандартный вывод). На другом конце сетевого соединения демон `telnetd` читает то, что мы ввели с терминала, и передает это командной оболочке. Вывод, полученный в результате запуска команды, отправляется обратно через команду `telnet` и отображается на нашем терминале. Схема этих действий изображена на рис. 14.4.



Рис. 14.4. Схема работы команды `telnet`

Процесс `telnet` имеет два дескриптора для ввода и два для вывода. Эта программа не может использовать блокирующие операции чтения для какого-либо из дескрипторов ввода, так как заранее неизвестно, в каком из них имеются готовые для чтения данные.

Один из вариантов решения этой проблемы — разделить процесс на две части (с помощью функции `fork`), каждая из которых будет обслуживать одно направление передачи данных. Схема такого решения показана на рис. 14.5. (Примерно так была реализована команда `cu(1)` из пакета `uucp` в System V.)

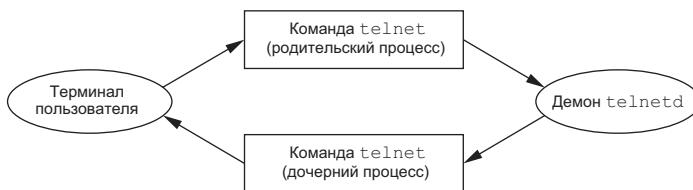


Рис. 14.5. Реализация программы `telnet` в виде двух процессов

Используя схему с двумя процессами, мы можем позволить каждому из них производить блокирующую операцию чтения. Но здесь появляется другая проблема, связанная с завершением работы. Если дочерний процесс получит признак конца файла (сетевое соединение закрыто со стороны демона `telnetd`), он завершится, а родительский процесс будет извещен об этом сигналом `SIGCHLD`. Но если первым завершится родительский процесс (пользователь введет с терминала признак конца файла), он должен велеть потомку завершиться. Для этого можно использовать сигнал (например, `SIGUSR1`), но это усложнит программу.

Вместо схемы с двумя процессами можно использовать схему с двумя потоками выполнения. Это поможет избежать сложностей, связанных с завершением, но потребует введения синхронизации между потоками, в результате сложность программы может не только не уменьшиться, но увеличиться еще больше.

Мы могли бы использовать неблокирующие операции ввода/вывода, установив для обоих дескрипторов неблокирующий режим, и попытаться прочитать данные из первого дескриптора функцией `read`. Если данные присутствуют, мы сможем получить их и обработать. Если данных нет, функция `read` сразу же вернет управление. Затем то же самое проделаем со вторым дескриптором. После этого можно подождать некоторое время (возможно, несколько секунд) и повторить попытку чтения из первого дескриптора. Циклы такого рода называются *опросом* (*polling*). Основная проблема такого решения — напрасный расход процессорного времени. Большую часть времени данные для чтения отсутствуют, и обращения к системному вызову `read` будут производиться вхолостую. Кроме того, мы должны решить, как долго ждать перед началом нового цикла. Несмотря на то что такой подход пригоден для любой системы, которая поддерживает неблокирующие операции ввода/вывода, в многозадачных системах его следует избегать.

Еще одно решение связано с операциями асинхронного ввода/вывода. Чтобы воспользоваться ими, мы должны сообщить ядру о необходимости посыпать процессу сигнал, когда дескриптор будет готов для ввода/вывода. С этим решением связаны две проблемы. Во-первых, не все системы поддерживают эту функциональность (ранее в стандарте Single UNIX Specification она относилась к разряду необязательных, но стала обязательной начиная с SUSv4). В System V для этих целей предусмотрен сигнал `SIGPOLL`, но он посыпается ядром, только если дескриптор ссылается на устройство STREAMS. В BSD есть похожий сигнал `SIGIO`, но и он имеет примерно такие же ограничения — сигнал посыпается, только если дескриптор ссылается на терминальное устройство или сетевое соединение.

Во-вторых, при использовании такой методики процесс может назначить сигнал (`SIGPOLL` или `SIGIO`) лишь для одного дескриптора. Если мы разрешим доставку сигнала для двух дескрипторов (в данном примере речь идет о чтении из двух дескрипторов), получив его, мы не сможем сказать, какой из дескрипторов готов к выполнению операции чтения. Хотя определение интерфейса асинхронного ввода/вывода в стандарте POSIX.1 позволяет выбирать, какой сигнал использовать для передачи извещений, количество их все равно меньше количества дескрипторов файлов, которые можно открыть. Чтобы проверить готовность дескрипторов, придется перевести каждый из них в неблокирующий режим и попытаться прочитать данные из обоих. Краткое описание асинхронного ввода/вывода приводится в разделе 14.5.

Наилучшим решением является *мультиплексирование ввода/вывода*. Для этого необходимо создать список дескрипторов, представляющих определенный интерес (обычно список состоит более чем из одного дескриптора), и вызвать функцию, которая не вернет управление, пока один из дескрипторов не будет готов к выполнению операции ввода/вывода. По возвращении из функции мы получим информацию о том, какие дескрипторы готовы для ввода/вывода.

Стандарт POSIX указывает, что для добавления всех определений, необходимых для обращения к функции `select`, программа должна подключать заголовочный файл `<sys/select.h>`. Устаревшие системы требуют подключения заголовочных файлов `<sys/types.h>`, `<sys/time.h>` и `<unistd.h>`.

Возможность мультиплексирования ввода/вывода с помощью функции `select` появилась в 4.2BSD. Эта функция всегда могла работать с любыми дескрипторами, хотя основное ее предназначение — работа с дескрипторами терминалов и сетевых соединений. В SVR3, с появлением механизма STREAMS, была добавлена функция `poll`. Однако изначально она могла работать только с устройствами STREAMS. Начиная с версии SVR4 в нее была добавлена поддержка любых типов дескрипторов.

14.4.1. Функции `select` и `pselect`

Функция `select` позволяет производить мультиплексирование ввода/вывода на любой POSIX-совместимой платформе. Аргументы, которые передаются функции `select`, сообщают ядру:

- список интересующих дескрипторов;
- какие состояния каждого из дескрипторов нас интересуют (готовность к чтению, готовность к записи, наличие исключительной ситуации);
- как долго ожидать изменения состояния дескриптора (не ограничивать время ожидания, определить некоторый интервал времени или вообще не ждать).

По возвращении из функции ядро сообщает:

- общее количество дескрипторов, перешедших в требуемое состояние;
- какие из дескрипторов готовы для чтения, какие для записи и для каких была обнаружена исключительная ситуация.

Обладая этой информацией, можно производить соответствующие операции ввода/вывода (обычно чтение или запись), заранее зная, что они не будут заблокированы.

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

int select(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
           fd_set *restrict exceptfds, struct timeval *restrict tvptr);
```

Возвращает количество дескрипторов, готовых к выполнению операции, 0 — в случае истечения тайм-аута, -1 — в случае ошибки

Для начала рассмотрим последний аргумент. Он определяет продолжительность времени ожидания в секундах и микросекундах (раздел 4.20). Всего возможны три различных состояния этого аргумента.

tvptr == NULL

Время ожидания не ограничено. Бесконечное ожидание может быть прервано при перехвате сигнала. Возврат из функции возможен, только когда хотя бы один из дескрипторов будет готов к выполнению операции или когда будет

перехвачен сигнал. В последнем случае функция `select` возвращает значение `-1` с кодом ошибки `EINTR` в `errno`.

`tvptr->tv_sec == 0 && tvptr->tv_usec == 0`

Вообще не ждать. В этом случае просто производится проверка всех указанных дескрипторов и управление тут же возвращается в вызывающую программу. Это один из способов запросить информацию об изменении состояния для целой группы дескрипторов, не блокируя процесс в функции `select`.

`tvptra->tv_sec != 0 || tvptr->tv_usec != 0`

Ждать не более заданного количества секунд и микросекунд. Возврат из функции возможен, когда хотя бы один из дескрипторов будет готов к выполнению операции или когда истечет время тайм-аута. Если по истечении тайм-аута ни один из дескрипторов не будет готов к выполнению операции, функция вернет значение `0`. (Если система не поддерживает измерение времени с точностью до микросекунд, значение поля `tvptra->tv_usec` округляется до ближайшего поддерживаемого значения.) Как и в первом случае, ожидание может быть прервано перехваченным сигналом.

Стандарт POSIX.1 позволяет реализациям изменять значения полей структуры `timeval`, поэтому после возврата из функции `select` нельзя полагаться на то, что структура будет содержать значения, которые были записаны перед вызовом `select`. FreeBSD 8.0, Mac OS X 10.6.8 и Solaris 10 оставляют эту структуру без изменений, а в Linux 3.2.0 в случае возврата до истечения тайм-аута в этой структуре возвращается оставшееся время.

Второй, третий и четвертый аргументы — `readfds`, `writefd`s и `exceptfds` — представляют собой указатели на наборы дескрипторов. Эти три набора определяют, какие дескрипторы нас интересуют и в каких состояниях (готовность к чтению, к записи или наличие исключительной ситуации). Для хранения набора дескрипторов предусмотрен тип данных `fd_set`. Этот тип данных выбирается реализацией так, чтобы он мог хранить один бит для каждого возможного дескриптора. Его можно рассматривать как большой массив битов (рис. 14.6).



Рис. 14.6. Определение наборов дескрипторов для функции `select`

Единственное, что можно сделать с переменными типа `fd_set`, — присвоить значение одной переменной этого типа другой переменной того же типа или передать переменную одной из следующих функций.

```
#include <sys/select.h>
int FD_ISSET(int fd, fd_set *fdset);
```

Возвращает ненулевое значение, если дескриптор *fd* включен в набор,
0 — в противном случае

```
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Эти функции могут быть реализованы в виде макроопределений. Функция **FD_ZERO** сбрасывает все биты в наборе *fd_set*. Функция **FD_SET** «взводит» один бит. Функция **FD_CLR** сбрасывает один бит. И наконец, с помощью функции **FD_ISSET** можно проверить состояние конкретного бита.

После объявления набора дескрипторов необходимобросить в нем все биты с помощью функции **FD_ZERO**, а затем установить биты для интересующих нас дескрипторов — например, так:

```
fd_set    rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

После возврата из функции **select** необходимо с помощью функции **FD_ISSET** проверить, какие биты в наборе остались установленными:

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```

В любом (или во всех) из трех описанных аргументов (указатели на наборы дескрипторов) допускается передавать пустой указатель. Если во всех трех аргументах передать **NULL**, тогда в нашем распоряжении появится таймер с более высоким разрешением, чем предоставляемый функцией **sleep**. (В разделе 10.19 мы говорили, что функция **sleep** приостанавливает выполнение процесса на целое число секунд. С помощью функции **select** можно отмерять временные интервалы продолжительностью менее одной секунды — фактическая точность зависит от системных часов.) В упражнении 14.5 как раз говорится о таком применении функции.

Имя первого аргумента функции **select** — *maxfdp1* — происходит от выражения «maximum file descriptor plus 1» (максимальный номер дескриптора плюс 1). В качестве значения этого аргумента берется максимальный номер дескриптора, который нас интересует, увеличенный на единицу. Можно было бы просто передать в этом аргументе значение константы **FD_SETSIZE** из заголовочного файла **<sys/select.h>**. Эта константа определяет максимально возможный номер дескриптора (часто 1024), но это значение слишком велико для большинства программ. В действительности большинство программ используют от 3 до 10 дескрипторов. (Некоторым программам требуется гораздо больше дескрипторов, но это нетипично для приложений UNIX.) Указав максимальный номер интересующего де-

скриптора, мы можем предотвратить просмотр ядром сотен неиспользуемых дескрипторов в трех наборах в поисках установленных битов.

В качестве примера на рис. 14.7 показаны два набора дескрипторов, созданные следующим фрагментом программы:

```
fd_set    readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

Максимальный номер дескриптора необходимо увеличивать на единицу по той причине, что нумерация дескрипторов начинается с 0, а первый аргумент функции на самом деле является счетчиком дескрипторов, которые необходимо проверять (начиная с дескриптора 0).

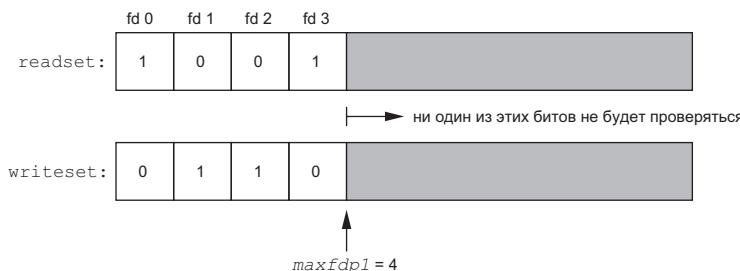


Рис. 14.7. Пример наборов дескрипторов для функции select

Функция `select` может возвращать три разных значения:

1. Возвращаемое значение `-1` свидетельствует об ошибке. Она может произойти, например, в случае перехвата сигнала, когда ни один из дескрипторов еще не готов для выполнения операции. В этой ситуации ни один из наборов дескрипторов не модифицируется.
2. Возвращаемое значение `0` свидетельствует о том, что ни один из дескрипторов не готов к выполнению операции. Это может произойти, если тайм-аут истек и ни один из дескрипторов не стал готов для выполнения операции. Когда это происходит, все биты в наборах сбрасываются в ноль.
3. Положительное возвращаемое значение показывает количество дескрипторов, готовых к выполнению операции ввода/вывода. Это значение представляет сумму готовых дескрипторов из всех трех наборов. То есть если один и тот же дескриптор готов как для чтения, так и для записи, в возвращаемом значении он будет посчитан дважды. «Взведенными» остаются только те биты в наборах, которые соответствуют дескрипторам, готовым к выполнению операций ввода/вывода.

Теперь уточним смысл понятия «готов».

- Дескриптор из набора *readfds* считается готовым, если вызов функции *read* для этого дескриптора не будет заблокирован.
- Дескриптор из набора *writelfds* считается готовым, если вызов функции *write* для этого дескриптора не будет заблокирован.
- Дескриптор из набора *exceptfds* считается готовым, если для данного дескриптора существует исключительная ситуация, ожидающая обработки. В настоящее время под исключительной ситуацией понимается либо поступление экстренных (out-of-band) данных через сетевое соединение, либо некоторые определенные события, возникающие на псевдотерминале, работающем в пакетном режиме. (Описание этих событий вы найдете в [Stevens, 1990; раздел 15.10].)
- Для обычных файлов всегда возвращается признак готовности к чтению, к записи и наличию исключительной ситуации.

Важно понимать, что режим дескриптора (блокирующий или неблокирующий) не оказывает никакого влияния на то, будет ли заблокирован вызов функции *select*. То есть если у нас имеется дескриптор, открытый в неблокирующем режиме для чтения, и мы вызываем функцию *select* с тайм-аутом в 5 секунд, *select* заблокирует процесс на 5 секунд. Аналогично, если не ограничить время тайм-аута, функция *select* заблокирует процесс, пока не поступят ожидаемые данные или не будет перехвачен какой-либо сигнал.

Если дескриптор достигнет конца файла, функция *select* будет рассматривать его как готовый для чтения. После этого вызов функции *read* вернет нам 0, что в UNIX расценивается как признак конца файла. (Многие неправильно полагают, что функция *select* расценивает признак конца файла как исключительную ситуацию.)

Стандартом POSIX.1 определена разновидность функции *select* — функция *pselect*.

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

int pselect(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writelfds,
            fd_set *restrict exceptfds, const struct timespec *restrict tspr,
            const sigset_t *restrict sigmask);
```

Возвращает количество готовых дескрипторов, 0 — в случае тайм-аута,
-1 — в случае ошибки

Функция *pselect* идентична функции *select*, со следующими исключениями:

- Значение тайм-аута для *select* задается в виде структуры *timeval*, а для *pselect* — в виде структуры *timespec*. (Описание структуры *timespec* приводится в разделе 4.2.) Вместо секунд и микросекунд структура *timespec* представляет время в секундах и наносекундах. Это позволяет задавать время тайм-аута с более высокой точностью на платформах, поддерживающих такой уровень точности измерения временных интервалов.

- Аргумент, в котором передается значение тайм-аута, объявлен со спецификатором `const`. Это гарантирует, что содержимое структуры не изменится в вызове функции `pselect`.
- Функция `pselect` имеет дополнительный аргумент — маску сигналов. Если в аргументе `sigmask` передать пустой указатель, функция `pselect` будет вести себя по отношению к сигналам так же, как функция `select`. Иначе `sigmask` указывает на маску сигналов, которая будет автоматически установлена при вызове функции `pselect`. По возвращении из функции предыдущая маска сигналов будет восстановлена.

14.4.2. Функция `poll`

Функция `poll` напоминает функцию `select`, но ее программный интерфейс существенно отличается. Поскольку `poll` изначально появилась в System V, она тесно связана с механизмом STREAMS, хотя и допускает использование с любыми типами дескрипторов.

```
#include <poll.h>
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

Возвращает количество готовых дескрипторов, 0 — в случае тайм-аута, -1 — в случае ошибки

Вместо того чтобы строить наборы дескрипторов для проверки трех возможных условий (готовность к чтению, готовность к записи, наличие исключительной ситуации), как это делается для функции `select`, при использовании `poll` нужно создать массив структур `pollfd`, в котором каждый элемент соответствует определенному дескриптору и проверяемому условию:

```
struct pollfd {
    int fd;          /* номер дескриптора или число <0, */
                    /* если номер дескриптора игнорируется */
    short events;   /* интересующие события для заданного дескриптора */
    short revents;  /* произошедшие события для заданного дескриптора */
};
```

Количество элементов в массиве `fdarray` определяется аргументом `nfds`.

Традиционно существовали некоторые различия в том, как объявлялся аргумент `nfds`. В SVR3 количество элементов в массиве определялось как `unsigned Long`, что кажется излишним. В справочном руководстве к SVR4 [AT&T 1990d] второй аргумент в прототипе функции `poll` имел тип `size_t`. (Элементарные системные типы данных были приведены в табл. 2.17.) Но фактический прототип в заголовочном файле `<poll.h>` по-прежнему определял тип второго аргумента как `unsigned Long`. Стандарт Single UNIX Specification определил новый тип — `nfds_t`, что позволяет реализациям выбирать для него соответствующий тип данных и скрывать детали реализации от приложений. Обратите внимание, что этот тип должен быть достаточно большим, чтобы хранить

целое число, так как возвращаемое значение — это количество элементов в массиве, для которых возникли ожидаемые события.

Документ *SVID* (определение интерфейса *System V*), соответствующий *SVR4* [AT&T, 1989], определяет первый аргумент функции *poll* как *struct pollfd fdarray[]*, тогда как справочное руководство *SVR4* [AT&T, 1990d] указывает, что этот аргумент определяется как *struct pollfd *fdarray*. В языке C эти объявления эквивалентны. Однако мы будем использовать первое объявление, чтобы напомнить еще раз, что аргумент *fdarray* указывает на массив структур, а не на отдельную структуру.

Чтобы сообщить ядру, какие события нас интересуют, мы должны записать в поле *events* для каждого элемента массива одно или более значений, перечень которых приводится в табл. 14.4. По возвращении из функции *poll* ядро указывает в поле *revents*, какие события произошли для каждого из дескрипторов. (Обратите внимание: функция *poll* не изменяет значение поля *events*. Это отличает ее от функции *select*, которая модифицирует значения входных аргументов, чтобы указать на готовые дескрипторы.)

Таблица 14.4. Значения флагов *events* и *revents* для функции *poll*

Имя	events	revents	Описание
POLLIN	✓	✓	Данные, помимо высокоприоритетных, доступны для чтения без блокировки (эквивалент POLLRDNORM POLLRBAND)
POLLRDNORM	✓	✓	Обычные данные (с приоритетом 0) доступны для чтения без блокировки
POLLRBAND	✓	✓	Данные с ненулевым приоритетом доступны для чтения без блокировки
POLLPRI	✓	✓	Высокоприоритетные данные доступны для чтения без блокировки
POLLOUT	✓	✓	Обычные данные можно записать без блокировки
POLLWRNORM	✓	✓	То же, что и POLLOUT
POLLWRBAND	✓	✓	Данные с ненулевым приоритетом можно записать без блокировки
POLLERR		✓	Возникла ошибка
POLLHUP		✓	Обрыв связи
POLLNVAL		✓	Дескриптор не соответствует открытому файлу

Первые четыре строки в табл. 14.4 проверяют готовность дескриптора для чтения, следующие три — готовность для записи и последние три — наличие исключительной ситуации. Последние три значения в табл. 14.4 всегда возвращаются ядром в поле *revents*, когда возникают соответствующие события, даже если они не были указаны в поле *events*.

Имена событий, оканчивающиеся на *BAND*, соответствуют полосам приоритетов в *STREAMS*. За дополнительной информацией о механизме *STREAMS* и полосах приоритетов обращайтесь к [Rago, 1993].

Когда для дескриптора обнаруживается обрыв связи (`POLLHUP`), мы уже не сможем ничего записать в него. Однако дескриптор еще может содержать данные, доступные для чтения.

Последний аргумент функции `poll` определяет, как долго ожидать наступления указанных событий. Как и в случае с функцией `select`, здесь возможны три значения аргумента.

timeout == -1

Время ожидания не ограничено. (В некоторых системах для этих целей в заголовочном файле `<stropts.h>` определена константа `INFTIM` со значением `-1`.) Управление будет возвращено в вызывающую программу, если хотя бы один из дескрипторов произойдет ожидаемое событие или если процесс перехватит какой-либо сигнал. В последнем случае функция вернет значение `-1` и код ошибки `EINTR` в переменной `errno`.

timeout == 0

Не ждать. В этом случае просто производится проверка всех указанных дескрипторов, и управление сразу же возвращается в вызывающую программу. Это один из способов запросить информацию об изменении состояния целой группы дескрипторов, не блокируя процесс в функции `poll`.

timeout > 0

Ждать не более *timeout* миллисекунд. Управление будет возвращено в вызывающую программу, когда хотя бы один из дескрипторов будет готов или когда истечет время тайм-аута. Если время тайм-аута истечет раньше, функция вернет 0. (Если система не поддерживает измерение временных интервалов с точностью до миллисекунды, значение *timeout* округляется до ближайшего поддерживаемого значения.)

Важно понимать различие между обрывом связи и признаком конца файла. Если после ввода данных с терминала введен символ конца файла, будет установлен флаг `POLLIN`, и благодаря этому мы сможем прочитать этот символ (функция `read` вернет значение 0). При этом флаг `POLLHUP` не будет выставлен в поле `revents`. Если во время чтения данных через modem происходит разрыв соединения, для дескриптора выставляется флаг `POLLHUP`.

Как и в случае с функцией `select`, неблокирующий режим дескриптора вовсе не определяет, будет ли блокироваться функция `poll`.

Прерываемость функций `poll` и `select`

Когда в 4.2BSD появилась возможность автоматического перезапуска прерванных системных вызовов (раздел 10.5), для функции `select` такая возможность не была предусмотрена. Это положение дел сохраняется в большинстве систем, даже если указывается флаг `SA_RESTART`. Но в SVR4, если флаг `SA_RESTART` указан, даже функции `select` и `poll` перезапускаются автоматически. Чтобы воспрепятствовать такому поведению, которое может обернуться неприятными последствиями при переносе программного обеспечения на системы, происходящие от SVR4, мы всегда используем функцию `signal_intr` (листинг 10.13), если сигнал может прервать работу системного вызова `select` или `poll`.

Ни одна из реализаций, рассматриваемых в данной книге, не предусматривает перезапуска системных вызовов `select` и `poll` при получении сигнала, даже если установлен флаг `SA_RESTART`.

14.5. Асинхронный ввод/вывод

Функции `select` и `poll`, описанные в предыдущем разделе, представляют синхронную форму уведомления. Система ничего не сообщает о произошедших событиях, пока мы явно не спросим ее об этом (вызовом `select` или `poll`). В главе 10 мы видели, что сигналы являются асинхронной формой уведомления о происходящих событиях. Все системы, производные от BSD или System V, предоставляют возможность выполнения асинхронных операций ввода/вывода, используя сигналы (`SIGPOLL` — в System V и `SIGIO` — в BSD) для извещения процессов о том, что с дескриптором произведены некоторые действия. Как упоминалось в предыдущем разделе, эти формы асинхронного ввода/вывода имеют ограничения: они могут применяться не ко всем типам файлов и позволяют использовать единственный сигнал. Если разрешить асинхронный режим работы для нескольких дескрипторов, при получении сигнала мы не сможем сказать, какому дескриптору он соответствует.

В версии 4 стандарта Single UNIX Specification обобщенный механизм асинхронного ввода/вывода был перемещен из расширений реального времени в базовые спецификации. Этот механизм устраниет массу ограничений, существующих в обсуждаемом устаревшем механизме асинхронного ввода/вывода.

Прежде чем перейти к знакомству с разными способами использования асинхронного ввода/вывода, необходимо обсудить затраты. Использование асинхронного ввода/вывода усложняет организацию приложения, которое теперь должно выполнять различные операции параллельно. Возможно, проще было бы использовать несколько потоков выполнения и организовать выполнение операций с применением синхронной модели в асинхронных потоках.

Используя асинхронные интерфейсы ввода/вывода, определяемые стандартом POSIX, мы вводим дополнительные сложности:

- Нам придется предусмотреть обработку трех источников ошибок для каждой асинхронной операции: один связан с запуском операции, другой — с результатом операции и третий — с функцией, применяемой для определения состояния асинхронной операции.
- Как будет показано ниже, использование интерфейсов связано с необходимостью выполнения дополнительных подготовительных операций и соблюдения большего количества правил обработки в сравнении с их обычными аналогами.

В действительности мы не можем называть неасинхронные функции ввода/вывода «синхронными», потому что, несмотря на синхронность по отношению к потоку выполнения программы, они не являются синхронными по отношению к вводу/выводу. Вспомните обсуждение синхронной операции записи в главе 3. Мы говорили, что операция записи считается «синхронной», если записываемые данные сохраняются в устройстве до возврата из функции `write`. Мы также не можем отделять обычные функции ввода/вывода от асинхронных, называя обычные функции «стандартными», потому что возникает

путаница с функциями из стандартной библиотеки ввода/вывода. Чтобы избежать путаницы, в этом разделе мы будем называть функции `read` и `write` «обычными» функциями ввода/вывода.

- Обработка ошибок может оказаться весьма непростым делом. Например, как обработать ситуацию, когда запущено несколько асинхронных операций записи и одна из них потерпела неудачу? Если эти операции как-то связаны между собой, может потребоваться отменить те, что завершились успехом.

14.5.1. Асинхронный вывод в System V

В System V механизмы асинхронного ввода/вывода являются составной частью системы STREAMS и применимы только к устройствам и каналам STREAMS. Для задач асинхронного ввода/вывода в System V используется сигнал `SIGPOLL`. Чтобы установить асинхронный режим ввода/вывода для устройства STREAMS, нужно вызвать функцию `ioctl` и передать ей во втором аргументе (*request*) значение `I_SETSIG`. Третий аргумент функции в этом случае формируется из констант, перечисленных в табл. 14.5. Все эти константы определяются в заголовочном файле `<stropts.h>`.

Интерфейсы, имеющие отношение к механизму STREAMS, в стандарте SUSv4 были отмечены как устаревшие, поэтому мы не будем рассматривать их. За дополнительной информацией о механизме STREAMS и полосах приоритетов обращайтесь к [Rago, 1993].

Кроме указания с помощью функции `ioctl` условий, при которых должен генерироваться сигнал `SIGPOLL`, мы также должны установить обработчик этого сигнала. В табл. 10.1 указывается, что по умолчанию сигнал `SIGPOLL` завершает процесс, поэтому обработчик необходимо установить до вызова функции `ioctl`.

Таблица 14.5. Условия, при которых генерируется сигнал `SIGPOLL`

Константа	Описание
<code>S_INPUT</code>	Можно прочитать данные (кроме высокоприоритетных) без блокировки
<code>S_RDNORM</code>	Можно прочитать обычные данные без блокировки
<code>S_RDBAND</code>	Можно прочитать приоритетные данные без блокировки
<code>S_BANDURG</code>	Эта константа в паре с <code>S_RDBAND</code> указывает, что когда появляется возможность прочитать приоритетные данные без блокировки, вместо сигнала <code>SIGPOLL</code> должен генерироваться сигнал <code>SIGURG</code>
<code>S_HIPRI</code>	Можно прочитать высокоприоритетные данные без блокировки
<code>S_OUTPUT</code>	Можно записать обычные данные без блокировки
<code>S_WRNORM</code>	То же, что и <code>S_OUTPUT</code>
<code>S_WRBAND</code>	Можно записать приоритетные данные без блокировки
<code>S_MSG</code>	Сообщение, породившее сигнал <code>SIGPOLL</code> , достигло головы потока
<code>S_ERROR</code>	Ошибка потока
<code>S_HANGUP</code>	Зависание потока

14.5.2. Асинхронный ввод/вывод в BSD

Асинхронный ввод/вывод в системах, производных от BSD, строится на комбинации сигналов **SIGIO** и **SIGURG**. Первый из них — общий для всех операций асинхронного ввода/вывода, а второй используется для извещения процесса о прибытии экстренных данных через сетевое соединение.

Чтобы подготовиться к принятию сигнала **SIGIO**, необходимо выполнить следующие действия.

1. Установить обработчик сигнала **SIGIO** вызовом **signal** или **sigaction**.
2. Назначить идентификатор процесса или идентификатор группы процессов, которым будет посыпаться сигнал для дескриптора, вызвав функцию **fcntl** с командой **F_SETOWN** (раздел 3.14).
3. Разрешить асинхронный режим работы для дескриптора, вызвав функцию **fcntl** с командой **F_SETFL**, чтобы установить флаг состояния файла **O_ASYNC** (табл. 3.3).

Шаг 3 можно выполнить, только если дескриптор ссылается на терминальное устройство или сетевое соединение, что само по себе является фундаментальным ограничением механизма асинхронного ввода/вывода в BSD.

Чтобы организовать получение сигнала **SIGURG**, достаточно выполнить только действия 1 и 2. Этот сигнал генерируется лишь для дескрипторов, ссылающихся на сетевые соединения, поддерживающие прием экстренных данных.

14.5.3. Асинхронный ввод/вывод в POSIX

Интерфейсы асинхронного ввода/вывода, определяемые стандартом POSIX, дают непротиворечивый способ асинхронного ввода/вывода, независимо от типов файлов. Эти интерфейсы заимствованы из предварительного стандарта реального времени, который включался как расширение в стандарт Single UNIX Specification. В версии Single UNIX Specification 4 эти интерфейсы перенесены в раздел базовых спецификаций, поэтому в настоящее время они обязательно должны поддерживаться всеми платформами.

Функции асинхронного ввода/вывода используют для описания асинхронных операций управляющие блоки AIO. Структура **aiocb** определяет управляющий блок AIO. Она содержит по меньшей мере следующие поля (реализации могут включать в структуру дополнительные поля):

```
struct aiocb {
    int           aio_fildes;      /* дескриптор файла */
    off_t         aio_offset;      /* смещение в файле */
    volatile void *aio_buf;        /* буфер ввода/вывода */
    size_t        aio_nbytes;      /* количество байтов */
    int           aio_reqpri;     /* приоритет */
    struct sigevent aio_sigevent;  /* информация о сигнале */
    int           aio_lio_opcode;   /* операция для списка запросов */
};
```

Поле `aio_fildes` — это дескриптор открытого файла, к которому применяется операция чтения или записи. Чтение или запись начинаются со смещения, определяемого полем `aio_offset`. При выполнении операции чтения данные копируются в буфер, начинающийся с адреса, определяемого полем `aio_buf`. При выполнении операции записи данные копируются из этого буфера. Поле `aio_nbytes` определяет количество байтов, которые нужно прочитать или записать.

Обратите внимание на необходимость явно указывать смещение при выполнении асинхронных операций ввода/вывода. Асинхронные функции ввода/вывода не оказывают влияния на позицию в файле, поддерживаемую операционной системой. В этом нет никакой проблемы, если в процессе никогда не смешиваются асинхронные функции ввода/вывода с их обычными аналогами. Отметьте также, что при записи в файл, открытый для записи в конец (с флагом `O_APPEND`), с использованием асинхронного интерфейса поле `aio_offset` в управляющем блоке AIO игнорируется системой.

Другие поля никак не связаны с обычными функциями ввода/вывода. Поле `aio_reqprio` дает приложению возможность подсказать системе, в каком порядке должны выполняться асинхронные операции. Однако система дает лишь ограниченный контроль над порядком выполнения запросов, поэтому нет никаких гарантий, что значение этого поля будет учитываться системой в полной мере. Поле `aio_lio_opcode` используется только в списках запросов на асинхронный ввод/вывод, о которых рассказывается чуть ниже. Поле `aio_sigevent` определяет способ извещения приложения о завершении ввода/вывода. Этот способ описывается структурой `sigevent`.

```
struct sigevent {
    int           sigev_notify;          /* тип извещения */
    int           sigev_signo;          /* номер сигнала */
    union sigval  sigev_value;          /* аргумент обработчика */
    void (*sigev_notify_function)(union sigval); /* функция-обработчик */
    pthread_attr_t *sigev_notify_attributes; /* атрибуты обработки */
};
```

Поле `sigev_notify` определяет тип извещения. Оно может принимать одно из следующих трех значений:

SIGEV_NONE Процесс не извещается о выполнении асинхронной операции ввода/вывода.

SIGEV_SIGNAL По завершении асинхронной операции ввода/вывода процессу посыпается сигнал, указанный в поле `sigev_signo`. Если приложение предусматривает обработку сигнала и установило флаг `SA_SIGINFO` при регистрации обработчика сигнала, сигнал будет поставлен в очередь (если реализация поддерживает такую возможность). Обработчик сигнала получит структуру `siginfo`, поле `si_value` которой будет хранить значение поля `sigev_value` (опять же, если был установлен флаг `SA_SIGINFO`).

SIGEV_THREAD По завершении асинхронной операции ввода/вывода будет вызвана функция, определяемая полем `sigev_notify_function`. Этой функции будет передан единственный аргумент — значение поля `sigev_value`. Функция вызывается в отдельном потоке, выполняющемся в обособленном состоянии, если в поле

`sigev_notify_attributes` не указан адрес структуры альтернативных атрибутов потока.

Чтобы выполнить асинхронный ввод/вывод, необходимо инициализировать управляющий блок АІО и вызвать функцию `aio_read`, чтобы выполнить чтение, или `aio_write`, чтобы выполнить запись.

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);
int aio_write(struct aiocb *aiocb);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Когда эти функции возвращают признак успеха, это означает, что запрошенные асинхронные операции ввода/вывода поставлены в очередь для обработки. Возвращаемое значение не имеет никакого отношения к результату фактической операции ввода/вывода. Пока операция ожидает обработки, необходимо обеспечить надежное хранение управляющего блока АІО и буфера с данными — занимаемая ими память должна оставаться доступной и не может использоваться для других нужд, пока операция ввода/вывода не будет выполнена.

Чтобы произвести принудительное выполнение всех ожидающих операций записи, можно создать управляющий блок АІО и передать его функции `aio_fsync`.

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocb);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Поле `aio_fildes` в управляющем блоке АІО должно определять файл, для которого требуется обеспечить принудительное выполнение асинхронных операций записи. Если в аргументе `op` передать `O_DSYNC`, функция `aio_fsync` будет действовать подобно `fdatasync`. Иначе, если в аргументе `op` передать `O_SYNC`, она будет действовать подобно функции `fsync`.

Подобно функциям `aio_read` и `aio_write`, `aio_fsync` возвращает управление сразу, как только запрос на выполнение синхронизации будет поставлен в очередь. Данные не будут сохранены, пока операция синхронизации не завершится. Управляющий блок АІО также определяет способ извещения приложения, как и при вызове функций `aio_read` и `aio_write`.

Чтобы определить состояние асинхронной операции чтения, записи или синхронизации, можно вызвать функцию `aio_error`.

```
#include <aio.h>

int aio_error(const struct aiocb *aiocb);
```

Возвращаемое значение описывается ниже

Возвращаемое значение может иметь одно из следующих четырех значений:

0 Асинхронная операция завершилась успехом. Чтобы получить возвращаемое значение операции, следует вызвать функцию `aio_return`.

-1 Вызов `aio_error` завершился ошибкой. Код ошибки можно получить из переменной `errno`.

EINPROGRESS Асинхронная операция чтения, записи или синхронизации все еще ожидает выполнения.

любое другое Любое другое значение соответствует коду ошибки, возникшей при выполнении асинхронной операции.

Если асинхронная операция завершилась успехом, можно вызвать `aio_return`, чтобы получить возвращаемое значение операции.

```
#include <aio.h>
ssize_t aio_return(const struct aiocb *aiocb);
```

Возвращаемое значение описывается ниже

Пока асинхронная операция не завершится, не следует вызывать функцию `aio_return`, потому что возвращаемый ею результат в этом случае не определен. Также не следует вызывать `aio_return` более одного раза для каждой асинхронной операции ввода/вывода. После первого вызова этой функции система может освободить память, занимаемую записью с возвращаемым значением операции ввода/вывода.

Если во время вызова `aio_return` произойдет ошибка, она вернет `-1` и сохранит в переменной `errno` код ошибки. Иначе она вернет результат асинхронной операции — то, что вернула бы функция `read`, `write` или `fsync` в случае успеха.

Асинхронный ввод/вывод обычно используется, когда требуется выполнить некоторые другие действия и нежелательно блокировать процесс в ожидании завершения обычных операций ввода/вывода. Но, закончив обработку и обнаружив, что асинхронная операция все еще не выполнена, мы можем вызвать функцию `aio_suspend`, чтобы приостановить выполнение процесса до завершения ввода/вывода.

```
#include <aio.h>
int aio_suspend(const struct aiocb *const list[], int nent,
                const struct timespec *timeout);
```

Возвращает 0 в случае успеха, `-1` — в случае ошибки

Функция `aio_suspend` возвращает управление в одном из трех случаев. Если `aio_suspend` прервет сигнал, она вернет `-1` с кодом ошибки `EINTR` в `errno`. Если до завершения любой операции ввода/вывода истечет тайм-аут, указанный в необязательном аргументе `timeout`, `aio_suspend` вернет `-1` с кодом ошибки `EAGAIN` в `errno` (если в `timeout` передать пустой указатель, функция заблокирует выполнение без

ограничения по времени). Как только любая операция ввода/вывода завершится, `aio_suspend` вернет 0. Если к моменту вызова `aio_suspend` все асинхронные операции ввода/вывода завершатся, она вернет управление немедленно.

В аргументе *list* передается указатель на массив управляющих блоков AIO, а в аргументе *next* – количество элементов в массиве. Пустые указатели в массиве просто пропускаются, но другие элементы должны указывать на управляющие блоки AIO, использовавшиеся для запуска асинхронных операций ввода/вывода.

Если имеется ожидающая асинхронная операция ввода/вывода, выполнения которой мы больше не можем ждать, можно попробовать отменить ее вызовом функции `aio_cancel`.

```
#include <aio.h>
int aio_cancel(int fd, struct aiocb *aiocb);
```

Возвращаемое значение описывается ниже

В аргументе *fd* передается дескриптор файла, для которого выполняется попытка отменить асинхронные операции ввода/вывода. Если в аргументе *aiocb* передать NULL, система попытается отменить все асинхронные операции ввода/вывода для указанного файла. Иначе система попробует отменить единственную асинхронную операцию, описанную управляющим блоком AIO. Мы говорим, что система «попытается» отменить операции, потому что нет никаких гарантий, что она сможет отменить уже выполняющиеся операции.

Функция `aio_cancel` может вернуть одно из четырех значений:

AIO_ALLDONE Все операции завершились до попытки отменить их.

AIO_CANCELED Все запрошенные операции были отменены.

AIO_NOTCANCELED По меньшей мере одна из запрошенных операций не может быть отменена.

-1 Ошибка вызова `aio_cancel`. Код ошибки хранится в `errno`.

Если асинхронная операция ввода/вывода была успешно отменена, вызов функции `aio_error` с соответствующим управляющим блоком AIO вернет код ошибки `ECANCELED`. Если операцию невозможно отменить, соответствующий управляющий блок AIO останется без изменений.

В состав интерфейса асинхронного ввода/вывода включена еще одна функция, которую, впрочем, можно использовать как в асинхронном, так и в синхронном режиме. Функция `lio_listio` посыпает множество запросов ввода/вывода, описываемых списком управляющих блоков AIO.

```
#include <aio.h>
int lio_listio(int mode, struct aiocb *restrict const list[restrict],
               int nent, struct sigevent *restrict sigev);
```

Возвращает 0 в случае успеха, -1 – в случае ошибки

Аргумент *mode* определяет, должны ли операции ввода/вывода выполняться в асинхронном режиме. Если передать в нем значение `LIO_WAIT`, функция `lio_listio` вернет управление, только когда будут выполнены все операции ввода/вывода, перечисленные в списке. В этом случае аргумент *sigev* игнорируется. Если передать в аргументе *mode* значение `LIO_NOWAIT`, функция `lio_listio` вернет управление сразу, как только поставит все запросы в очередь. Процесс будет извещен о выполнении всех операций ввода/вывода, как определено аргументом *sigev*. Если извещать процесс не требуется, в аргументе *sigev* можно передать `NULL`. Обратите внимание, что сами блоки управления AIO могут требовать передачи извещений для отдельных операций. Асинхронное извещение, определяемое аргументом *sigev*, является дополнительным и передается программе только после выполнения всех операций ввода/вывода.

В аргументе *list* передается указатель на список управляющих блоков AIO, определяющих операции ввода/вывода. Аргумент *next* определяет количество элементов в списке (в массиве). Список управляющих блоков AIO может содержать пустые указатели, такие элементы будут игнорироваться.

В каждом управляющем блоке AIO поле `aio_lio_opcode` должно определять тип операции (`LIO_READ` — для чтения, `LIO_WRITE` — для записи или `LIO_NOP` — для отсутствующей операции); операции с типом `LIO_NOP` игнорируются. Операции чтения выполняются, как если бы соответствующие блоки управления AIO передавались функции `aio_read`. Аналогично, операции записи выполняются, как если бы соответствующие блоки управления AIO передавались функции `aio_write`.

Реализации могут ограничивать количество асинхронных операций ввода/вывода. Эти пределы, перечисленные в табл. 14.6, не могут изменяться во время выполнения.

Таблица 14.6. Неизменяемые во время выполнения пределы для асинхронных операций ввода/вывода, предусмотрываемые стандартом POSIX.1

Имя	Описание	Минимально допустимое значение
<code>AIO_LISTIO_MAX</code>	Максимальное количество операций ввода/вывода в одном списке	<code>_POSIX_AIO_LISTIO_MAX (2)</code>
<code>AIO_MAX</code>	Максимальное количество ожидающих асинхронных операций ввода/вывода	<code>_POSIX_AIO_MAX (1)</code>
<code>AIO_PRIO_DELTA_MAX</code>	Максимальное значение, на которое процесс может уменьшать свой приоритет асинхронных операций ввода/вывода	0

Узнать значение `AIO_LISTIO_MAX` можно вызовом функции `sysconf`, передав `_SC_LISTIO_MAX` в аргументе *name*. Аналогично, узнать значение `AIO_MAX` можно вызовом функции `sysconf`, передав `_SC_AIO_MAX` в аргументе *name*, а узнать значение `AIO_PRIO_DELTA_MAX` можно вызовом функции `sysconf`, передав `_SC_AIO_PRIO_DELTA_MAX` в аргументе *name*.

Первоначально асинхронный интерфейс ввода/вывода в стандарте POSIX разрабатывался для приложений реального времени, с целью дать возможность избежать блокирования при выполнении ввода/вывода. Теперь рассмотрим пример использования этого интерфейса.

Пример

Мы не будем обсуждать в этой книге приемы создания приложений реального времени, но так как асинхронный интерфейс ввода/вывода POSIX в настоящее время входит в раздел базовых спецификаций стандарта Single UNIX Specification, посмотрим, как можно его использовать. Для сравнения функций асинхронного ввода/вывода с их обычными аналогами реализуем преобразование файлов из одного формата в другой.

Программа в листинге 14.7 преобразует содержимое файла с помощью алгоритма ROT-13, применявшегося популярной в 1980-е годы системой новостей USENET для шифрования текста, который мог бы содержать оскорблени или чересчур пикантные шутки. Алгоритм циклически сдвигает коды символов от «а» до «z» и от «A» до «Z» на 13 позиций, а все остальные оставляет без изменений.

Листинг 14.7. Преобразование файла с применением алгоритма ROT13

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>

#define BSZ 4096

unsigned char buf[BSZ];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= 'n')
            c -= 13;
        else if (c >= 'a')
            c += 13;
        else if (c >= 'N')
            c -= 13;
        else
            c += 13;
    }
    return(c);
}

int
main(int argc, char* argv[])
{
    int      ifd, ofd, i, n, nw;

    if (argc != 3)
        err_quit("Использование: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("невозможно открыть %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
```

```

        err_sys("невозможно создать %s", argv[2]);

    while ((n = read(ifd, buf, BSZ)) > 0) {
        for (i = 0; i < n; i++)
            buf[i] = translate(buf[i]);
        if ((nw = write(ofd, buf, n)) != n) {
            if (nw < 0)
                err_sys("ошибка вызова write");
            else
                err_quit("записано меньше, чем запрошено (%d/%d)", nw, n);
        }
    }
    fsync(ofd);
    exit(0);
}

```

Часть программы, выполняющая ввод/вывод, достаточно проста: она читает блок данных из входного файла, преобразует его и затем записывает блок в выходной файл. Этот процесс повторяется, пока не будет достигнут конец входного файла и `read` вернет ноль. Программа в листинге 14.8 демонстрирует решение той же задачи с использованием функций асинхронного ввода/вывода.

Листинг 14.8. Преобразование файла с применением алгоритма ROT13 и асинхронного ввода/вывода

```

#include "apue.h"
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>

#define BSZ 4096
#define NBUF 8

enum rwop {
    UNUSED = 0,
    READ_PENDING = 1,
    WRITE_PENDING = 2
};

struct buf {
    enum rwop      op;
    int           last;
    struct aiocb  aiocb;
    unsigned char data[BSZ];
};

struct buf bufs[NBUF];

unsigned char
translate(unsigned char c)
{
    /* реализация не изменилась */
}

int
main(int argc, char* argv[])
{
    int             ifd, ofd, i, j, n, err, numop;

```

```

struct stat          sbuf;
const struct aiocb *aiolist[NBUF];
off_t                off = 0;

if (argc != 3)
    err_quit("Использование: rot13 infile outfile");
if ((ifd = open(argv[1], O_RDONLY)) < 0)
    err_sys("невозможно открыть %s", argv[1]);
if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
    err_sys("невозможно создать %s", argv[2]);
if (fstat(ifd, &sbuf) < 0)
    err_sys("ошибка вызова fstat");

/* инициализировать буферы */
for (i = 0; i < NBUF; i++) {
    bufs[i].op = UNUSED;
    bufs[i].aiocb.aio_buf = bufs[i].data;
    bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
    aiolist[i] = NULL;
}

numop = 0;
for (;;) {
    for (i = 0; i < NBUF; i++) {
        switch (bufs[i].op) {
        case UNUSED:
            /*
             * Прочитать данные из входного файла, если конец еще не достигнут.
             */
            if (off < sbuf.st_size) {
                bufs[i].op = READ_PENDING;
                bufs[i].aiocb.aio_fildes = ifd;
                bufs[i].aiocb.aio_offset = off;
                off += BSZ;
                if (off >= sbuf.st_size)
                    bufs[i].last = 1;
                bufs[i].aiocb.aio_nbytes = BSZ;
                if (aio_read(&bufs[i].aiocb) < 0)
                    err_sys("ошибка вызова aio_read");
                aiolist[i] = &bufs[i].aiocb;
                numop++;
            }
            break;
        case READ_PENDING:
            if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
                continue;
            if (err != 0) {
                if (err == -1)
                    err_sys("ошибка вызова aio_error");
                else
                    err_exit(err, "ошибка вызова read");
            }
        /*
         * Чтение закончено, преобразовать буфер и записать его.
         */
            if ((n = aio_return(&bufs[i].aiocb)) < 0)
                err_sys("ошибка вызова aio_return");
            if (n != BSZ && !bufs[i].last)

```

```

        err_quit("прочитано меньше, чем запрошено (%d/%d)", n, BSZ);
    for (j = 0; j < n; j++)
        bufs[i].data[j] = translate(bufs[i].data[j]);
    bufs[i].op = WRITE_PENDING;
    bufs[i].aiocb.aio_fildes = ofd;
    bufs[i].aiocb.aio_nbytes = n;
    if (aio_write(&bufs[i].aiocb) < 0)
        err_sys("ошибка вызова aio_write");
    /* оставить в списке aiolist */
    break;
case WRITE_PENDING:
    if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
        continue;
    if (err != 0) {
        if (err == -1)
            err_sys("ошибка вызова aio_error");
        else
            err_exit(err, "ошибка записи");
    }
    /*
     * Запись завершена, пометить буфер как неиспользуемый.
     */
    if ((n = aio_return(&bufs[i].aiocb)) < 0)
        err_sys("ошибка вызова aio_return");
    if (n != bufs[i].aiocb.aio_nbytes)
        err_quit("записано меньше, чем запрошено (%d/%d)", n, BSZ);
    aiolist[i] = NULL;
    bufs[i].op = UNUSED;
    numop--;
    break;
}
}
if (numop == 0) {
    if (off >= sbuf.st_size)
        break;
} else {
    if (aio_suspend(aiolist, NBUF, NULL) < 0)
        err_sys("ошибка вызова aio_suspend");
}
}

bufs[0].aiocb.aio_fildes = ofd;
if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
    err_sys("ошибка вызова aio_fsync");
exit(0);
}

```

Обратите внимание, что в этой программе используется восемь буферов, благодаря чему можно одновременно запускать до восьми асинхронных операций ввода/вывода. Как ни странно, но это может ухудшить производительность — если операции чтения будут выполняться не по порядку, это может свести на нет действие алгоритма опережающего чтения, используемого операционной системой.

Перед проверкой значения, возвращаемого операцией необходимо убедиться, что она завершилась. Когда `aio_error` возвращает значение, отличное от `EINPROGRESS` или `-1`, можно быть уверенными, что операция завершилась. Исключая эти значения, если `aio_error` вернула нечто отличное от нуля, можно быть уверенными, что

операция завершилась неудачей. После проверки этих условий можно безопасно вызвать `aio_return` и получить значение, которое вернула операция ввода/вывода. Пока нам есть что делать, мы можем продолжать запускать асинхронные операции ввода/вывода. При появлении свободного управляющего блока AIO можно запустить асинхронную операцию чтения. По завершении чтения очередного буфера программа преобразует его содержимое и запускает асинхронную операцию записи. Когда все управляющие блоки AIO оказываются заняты, программа переходит в режим ожидания завершения какой-нибудь операции вызовом `aio_suspend`. Когда блок записывается в выходной файл, мы оставляем то же смещение, которое было определено для операции чтения из входного файла. По этой причине порядок выполнения операций записи не имеет значения. Данная стратегия работает только потому, что каждому символу во входном файле соответствует символ в выходном файле; мы не добавляем и не удаляем символы перед записью в выходной файл. (Это знание поможет вам выполнить упражнение 14.8.)

В этом примере не использовались асинхронные извещения, потому что синхронная модель проще в использовании. Если бы нам потребовалось произвести какие-то другие действия, пока выполняются асинхронные операции ввода/вывода, их можно было бы вставить в цикл `for`. Если бы нам потребовалось предотвратить задержки в преобразовании файла, обусловленные этими дополнительными действиями, мы могли бы использовать одну из форм асинхронных извещений. При одновременном решении нескольких задач следует определить, какие из них приоритетнее, и только потом переходить к определению организации программы.

14.6. Функции `readv` и `writev`

Функции `readv` и `writev` предназначены для чтения и записи данных нескольких несмежных буферов одним обращением к функции. Эти операции называются *чтением вразброс и записью со слиянием*.

```
#include <sys/uio.h>
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Обе возвращают количество прочитанных или записанных байтов,
–1 — в случае ошибки

Второй аргумент в обеих функциях — указатель на массив структур `iovec`:

```
struct iovec {
    void    *iov_base; /* адрес начала буфера */
    size_t   iov_len;  /* размер буфера */
};
```

Количество элементов в массиве `iov` определяется аргументом `iovcnt` и ограничивается значением `IOV_MAX` (табл. 2.10). На рис. 14.8 показаны взаимоотношения между аргументами этих двух функций и структурой `iovec`.

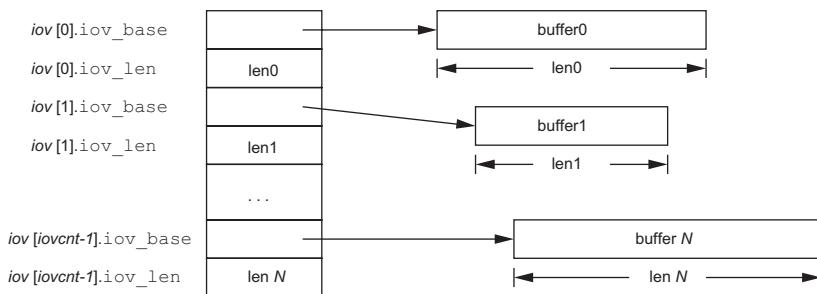


Рис. 14.8. Структура iovec для функций readv и writev

Функция `writev` производит запись данных из буферов в порядке следования элементов в массиве — `iov[0], iov[1] ... iov[iovcnt-1]` — и возвращает общее количество записанных байтов, которое обычно совпадает с суммой размеров всех буферов. Функция `readv` разбрасывает данные по буферам в том же порядке, всегда до конца заполняя один буфер, прежде чем перейти к заполнению следующего. Она возвращает общее количество прочитанных байтов. Если достигнут конец файла, функция `readv` возвращает значение 0.

Эти две функции впервые появились в 4.2BSD и позднее были добавлены в SVR4. Они определяются стандартом Single UNIX Specification как расширения XSI.

Пример

В разделе 20.8, в функции `_db_writeidx`, нам потребуется записать в файл последовательно два буфера. Второй буфер передается в функцию из вызывающей программы в виде аргумента, а первый создается внутри функции — он содержит длину второго буфера и смещение записи с данными от начала файла. Сделать это можно тремя способами.

1. Дважды вызвать функцию `write` — по разу для каждого буфера.
2. Разместить в динамической памяти общий буфер достаточного объема, скопировать в него оба буфера и затем одним вызовом функции `write` записать его в файл.
3. Записать оба буфера одним вызовом функции `writev`.

В разделе 20.8 мы используем функцию `writev`, но было бы любопытно сравнить все три способа.

В табл. 14.7 показаны результаты сравнения только что описанных методов.

Тестовая программа, с помощью которой проводились измерения, выводила 100-байтный заголовок и 200 байт данных. Запись выполнялась 1 048 576 раз, в результате был получен файл размером 300 Мбайт. Данная тестовая программа предусматривала запись по всем трем методикам, приведенным в табл. 14.7. Измерение времени производилось с помощью функции `times` (раздел 8.17), которая вызывалась до и после выполнения цикла записи. Все три значения времени (пользовательское, системное и общее время) приводятся в секундах.

Таблица 14.7. Результаты сравнения производительности функции `writev` с другими способами

Метод	Linux (Intel x86)			Mac OS X (Intel x86)		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
Две операции записи	0,06	2,04	2,13	0,85	8,33	13,83
Создание общего буфера и запись одним вызовом <code>write</code>	0,03	1,13	1,16	0,70	4,87	9,25
Запись вызовом <code>writev</code>	0,04	1,21	1,26	0,43	5,34	9,24

Как и следовало ожидать, при двойном обращении к функции `write` системное время выполнения больше, чем в случае одного вызова функции `write` или `writev`. Это совпадает с результатами, приведенными в табл. 3.2.

Далее, обратите внимание, что процессорное время (сумма пользовательского и системного времени) при копировании буферов и единственном вызове `write` меньше, чем при использовании функции `writev`. В случае с единственным вызовом функции `write` выполняется копирование буферов в промежуточный буфер в пространстве пользователя, и затем, при вызове функции `write`, ядро копирует его в свой внутренний буфер. При использовании функции `writev` мы выигрываем в объеме копирования, потому что здесь необходимо только скопировать данные во внутренний буфер. Однако фиксированная стоимость копирования таких небольших объемов данных сводит на нет все остальные преимущества функции `writev`. При увеличении объема копируемых данных вариант на основе функции `writev` будет выглядеть более привлекательно.

Вас не должна смущать такая большая разница в производительности Linux и Mac OS X. Дело в том, что эти два компьютера слишком сильно отличаются друг от друга: они собраны на процессорах разных поколений, имеют разные объемы оперативной памяти и жесткие диски с разным быстродействием. Чтобы сравнение различных операционных систем было корректным, они должны работать на одинаковой аппаратуре.

Вывод: всегда старайтесь делать как можно меньше системных вызовов. Если объемы данных невелики, методика с единственным вызовом `write` может оказаться менее дорогостоящей по сравнению с методикой на основе `writev`. Иногда, однако, повышение производительности не оправдывает усложнения программы, связанного с необходимостью управления промежуточными буферами.

14.7. Функции `readn` и `writen`

Именованные и неименованные каналы и некоторые другие устройства, а именно терминалы, сетевые соединения, обладают следующими двумя свойствами.

1. Функция `read` может вернуть меньшее количество байтов, чем было запрошено, хотя конец файла не достигнут. Это не является ошибкой, и мы можем продолжать чтение из устройства.

2. Функция `write` также может вернуть значение меньшее, чем мы указали. Это может произойти, например, из-за ограничений, накладываемых модулями в исходящем потоке данных. Такое поведение также не следует расценивать как ошибку, а оставшиеся данные можно записать повторным обращением к `write`. (Обычно подобное случается только при использовании неблокирующего режима для дескрипторов или в результате перехвата сигнала.)

Такое никогда не происходит при работе с дисковыми файлами, за исключением случаев переполнения файловой системы или достижения предела выделенной квоты на дисковое пространство, когда система не в состоянии записать весь требуемый объем данных.

Вообще, при работе с терминалами, сетевыми соединениями или каналами всегда необходимо учитывать эти особенности. Чтобы прочитать или записать определенное количество байтов, можно воспользоваться следующими двумя функциями. Они сами позаботятся об обслуживании ситуаций, когда операции чтения или записи выполняются лишь частично: они будут вызывать функции `read` или `write` столько раз, сколько потребуется для чтения или записи заданного количества байтов.

```
#include "apue.h"  
  
ssize_t readn(int fd, void *buf, size_t nbytes);  
  
ssize_t writen(int fd, void *buf, size_t nbytes);
```

Обе возвращают количество прочитанных или записанных байтов,
-1 – в случае ошибки

Мы даем описание этих двух функций, потому что они используются, например, в процедурах обработки ошибок, которые будут встречаться в дальнейших примерах. Функции `readn` и `writen` не являются частью какого-либо стандарта.

Функцию `writen` всегда можно использовать для типов файлов, о которых мы говорили выше, но функция `readn` должна вызываться, только когда заранее известно, что из данного файла можно прочитать заданное количество байтов. В листинге 14.9 показаны реализации функций `readn` и `writen`, которые будут использоваться в последующих примерах.

Обратите внимание, что в случае ошибки в процессе чтения или записи данных вместо признака ошибки возвращается количество переданных данных. Аналогично, если в процессе чтения достигнут конец файла, функция возвращает количество байтов, скопированных в буфер, предоставленный вызывающей программой, если некоторый объем данных удалось прочитать, но при этом он не равен запрошенному объему данных.

Листинг 14.9. Функции readn и writen

```
#include "apue.h"  
  
ssize_t /* Читает n байт из дескриптора */  
readn(int fd, void *ptr, size_t n)
```

```

{
    size_t      nleft;
    ssize_t     nread;

    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1);/* ошибка, вернуть -1 */
            else
                break; /* ошибка, вернуть количество прочитанных байтов */
        } else if (nread == 0) {
            break; /* конец файла */
        }
        nleft -= nread;
        ptr += nread;
    }
    return(n - nleft); /* возвращаемое значение >= 0 */
}

ssize_t          /* Записывает n байт в дескриптор */
written(int fd, const void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;

    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1);/* ошибка, вернуть -1 */
            else
                break; /* ошибка, вернуть количество записанных байтов */
        } else if (nwritten == 0) {
            break;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n - nleft); /* возвращаемое значение >= 0 */
}

```

14.8. Операции ввода/вывода с отображаемой памятью

Операции ввода/вывода с отображаемой памятью (mapped memory) позволяют отображать дисковые файлы в участки памяти так, что при выборке данных из памяти производится чтение соответствующих байтов из файла. Аналогично, при записи данных в отображенную память автоматически производится запись соответствующих байтов в файл. Это дает возможность производить ввод/вывод без использования функций `read` и `write`.

Операции ввода/вывода с отображаемой памятью уже много лет используются для организации работы с виртуальной памятью. В 1981 году в 4.1BSD появился другой вариант

ввода/вывода с отображаемой памятью — с использованием функций `vread` и `vwrite`. Позднее, в 4.2BSD, эти функции были удалены, их должна была заменить функция `mmap`, однако она не вошла в состав 4.2BSD (по причинам, которые описаны в разделе 2.5 [McKusick et al., 1996]). Одна из реализаций `mmap` приводится в [Gingell, Moran, and Shannon, 1987]. В версии 4 стандарта Single UNIX Specification функция `mmap` была включена в состав базовых спецификаций и теперь является обязательной для реализации во всех POSIX-совместимых системах.

Чтобы воспользоваться этой возможностью, нужно сообщить ядру о необходимости отобразить заданный файл в память. Делается это с помощью функции `mmap`.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);
```

Возвращает адрес начала области отображаемой памяти в случае успеха, `MAP_FAILED` — в случае ошибки

В аргументе `addr` можно указать желаемый адрес начала участка отображенной памяти. Обычно в этом аргументе передается 0, что позволяет системе самой выбрать начальный адрес. Возвращаемое значение функции является адресом начала отображенной памяти.

Через аргумент `fd` передается дескриптор отображаемого файла. Прежде чем отобразить файл в адресное пространство, необходимо открыть его. В аргументе `len` передается количество байтов, которые надо отобразить в память, а в аргументе `off` — смещение отображаемого участка от начала файла. (Далее будут описаны некоторые ограничения, существующие для аргумента `off`.)

Аргумент `prot` определяет степень защищенности отображенного участка.

Таблица 14.8. Защита области отображенной памяти

prot	Описание
<code>PROT_READ</code>	Область памяти доступна для чтения
<code>PROT_WRITE</code>	Область памяти доступна для записи
<code>PROT_EXEC</code>	Область памяти доступна для выполнения
<code>PROT_NONE</code>	Область памяти недоступна

Степень защищенности может указать как `PROT_NONE` или как объединение по ИЛИ (OR) любой комбинации из `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`. Для области памяти нельзя использовать степень защищенности, которая дает больше прав доступа, чем позволяет режим, в котором открыт файл. Например, нельзя указать значение `PROT_WRITE`, если файл открыт только для чтения.

Прежде чем перейти к описанию аргумента `flag`, рассмотрим рис. 14.9, где показан файл, отображенный в память. (Типичная организация памяти процесса изображена на рис. 7.3.) На данном рисунке «адрес начала» соответствует значению, возвращаемому функцией `mmap`. Область отображенной памяти показана где-то

между областью динамической памяти и стеком, но это зависит от конкретной реализации.

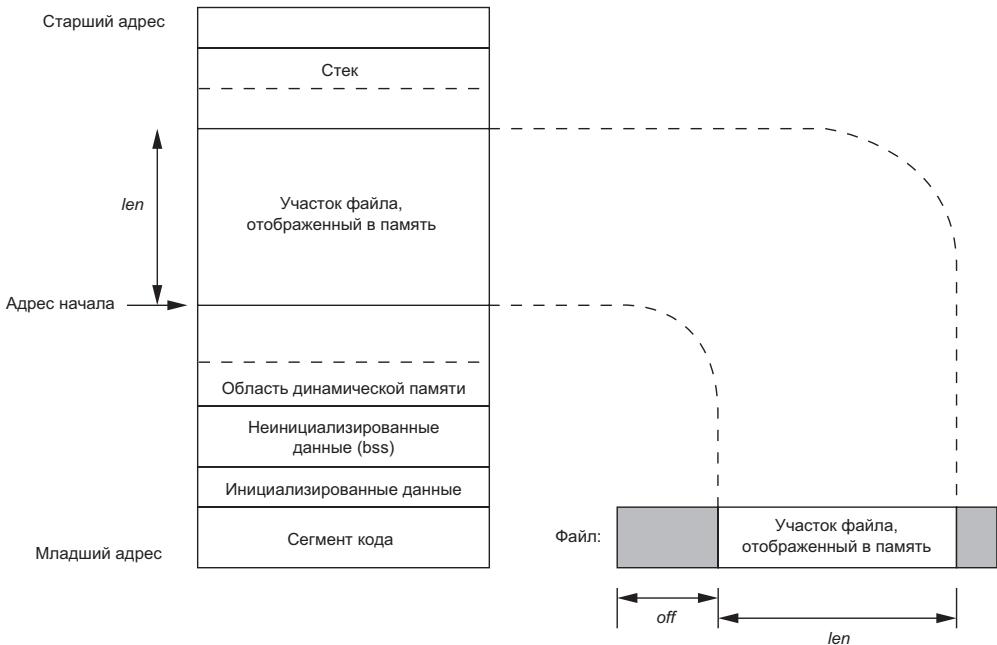


Рис. 14.9. Пример отображенного в память файла

Аргумент *flag* оказывает воздействие на различные атрибуты области отображеной памяти.

MAP_FIXED Возвращаемое значение должно быть равно значению аргумента *addr*. Применять этот флаг не рекомендуется, так как он снижает переносимость приложения. Если при использовании ненулевого значения в аргументе *addr* этот флаг не указывается, ядро расценивает значение аргумента *addr* как желаемый адрес, но не дает никакой гарантии, что отображенная память будет размещена, начиная с этого адреса. Максимальная переносимость достигается при указании значения 0 в аргументе *addr*.

Поддержка флага MAP_FIXED является необязательной в POSIX-совместимых системах, но обязательна в XSI-совместимых.

MAP_SHARED Флаг определяет характер операций над областью отображенной памяти. Если указать этот флаг, все операции записи в область отображенной памяти будут приводить к модификации самого файла, то есть операции записи в память будут эквивалентны вызову функции `write` для файла. Допускается одновременная установка только одного флага — либо этого, либо следующего.

MAP_PRIVATE Этот флаг говорит о том, что все операции записи в область отображенной памяти будут приводить к созданию скрытой копии файла, отобра-

женного в память. Изменения в памяти не будут влиять на содержимое самого файла. (Флаг используется отладчиками, чтобы отобразить сегмент кода из файла программы в память и позволить пользователю модифицировать инструкции. Все модификации будут производиться только в памяти процесса и не затронут оригинальный файл программы.)

Каждая реализация поддерживает дополнительные, специфичные для нее значения `MAP_xxx`. За дополнительной информацией обращайтесь к странице справочного руководства по функции `mmap(2)`.

Значения аргументов *off* и *addr* (если указан флаг `MAP_FIXED`) должны быть кратны размеру страницы виртуальной памяти. Это значение можно получить вызовом `sysconf` (раздел 2.5.4) с аргументом `_SC_PAGESIZE` или `_SC_PAGE_SIZE`. Поскольку чаще всего в аргументах *off* и *addr* передается значение 0, это требование не представляет большой проблемы.

Это требование обычно выдвигается конкретными реализациями системы. Хотя стандарт Single UNIX Specification больше не требует, чтобы это условие удовлетворялось, все платформы, описываемые в этой книге, кроме FreeBSD 8.0, выдвигают это требование. OS FreeBSD 8.0 позволяет использовать любой адрес и величину смещения, при условии, что они имеют одинаковое выравнивание.

Поскольку смещение начала отображаемого участка файла привязано к размеру страницы виртуальной памяти, что случится, если длина отображаемого участка будет не кратна размеру страницы? Представим себе, что размер файла составляет 12 байт, а размер страницы виртуальной памяти — 512 байт. В этом случае система выделит область отображенной памяти размером 512 байт, но в последние 500 байт этой области будут записаны нули. Мы можем изменять последние 500 байт, но эти изменения не будут отражаться на содержимом файла. То есть с помощью функции `mmap` невозможно добавить новые данные в конец файла. Для этого необходимо сначала увеличить размер файла, как показано в листинге 14.10.

Для работы с отображенными областями памяти обычно используются два сигнала. Сигнал `SIGSEGV`, как правило, указывает на попытку обращения к недоступной области памяти. Этот сигнал также может быть генерирован при попытке записи в память, которая была определена как доступная только для чтения. Сигнал `SIGBUS` может указывать на попытку обратиться к части отображенной области, которая не имеет смысла к моменту обращения. Например, предположим, что мы отобразили в память весь файл целиком, но прежде чем мы смогли приступить к операциям с отображенной областью памяти, файл был усечен некоторым другим процессом. Тогда, если попытаться обратиться к части файла, которая была усечена, мы получим сигнал `SIGBUS`.

Области отображенной памяти наследуются дочерними процессами через функцию `fork` (поскольку отображенная память является частью адресного пространства родительского процесса), но по той же причине отображенная память не наследуется новыми программами через функцию `exec`.

Изменить права доступа к отображенной памяти можно с помощью функции `mprotect`.

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

В аргументе *prot* могут передаваться те же значения, что в аргументе *prot* функции *mmap* (табл. 14.8). Аргумент *addr* должен быть целым числом, кратным размеру страницы виртуальной памяти.

При изменении страниц памяти, отображенных в адресное пространство процесса с флагом **MAP_SHARED**, изменения не будут записываться в файл немедленно. Вместо этого решение о записи измененных страниц принимается демонами ядра, которые учитывают (а) текущую нагрузку на систему и (б) значение конфигурационных параметров, способствующих снижению вероятности потери данных в случае краха системы. Когда данные записываются обратно на диск, запись выполняется блоками с размерами, кратными размеру страницы виртуальной памяти. То есть если изменить только один байт в странице памяти, в файл будет записана страница целиком.

Если страницы в разделяемой области отображенной памяти были изменены, их можно сбросить в файл с помощью функции *msync*. Функция *msync* напоминает функцию *fsync* (раздел 3.13), но предназначена для работы с областями отображенной памяти.

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Если область отображенной памяти создана с флагом **MAP_PRIVATE**, содержимое отображаемого файла не изменится. Как и в других функциях обслуживания отображенной памяти, аргумент *addr* должен содержать адрес, кратный размеру страницы виртуальной памяти.

Аргумент *flags* позволяет до некоторой степени управлять порядком сбросывания памяти в файл. Чтобы просто запланировать запись данных, можно передать в этом аргументе значение **MS_ASYNC**. Если необходимо дождаться, пока данные запишутся полностью, нужно указать флаг **MS_SYNC**. Аргумент должен содержать одно из двух значений: **MS_ASYNC** или **MS_SYNC**.

Необязательный флаг **MS_INVALIDATE** требует аннулировать все изменения, произведенные в памяти, и синхронизировать ее содержимое с содержимым отображаемого объекта (файла). Некоторые реализации аннулируют все измененные страницы в указанном диапазоне, но это совершенно необязательно.

Функция msync определяется стандартом Single UNIX Specification как расширение XSI. Поэтому все системы UNIX должны поддерживать ее.

Область отображенной памяти автоматически удаляется по завершении процесса или в результате вызова функции `munmap`. Закрытие файлового дескриптора не приводит к удалению этой области.

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Функция `munmap` не оказывает влияния на отображаемый объект, то есть вызов функции `munmap` не приводит к записи области отображенной памяти в файл. Обновление файла на диске при внесении изменений в область отображенной памяти, созданной с флагом `MAP_SHARED`, производится ядром автоматически. Все изменения, внесенные в область отображенной памяти, созданной с флагом `MAP_PRIVATE`, после вызова функции `munmap` будут утеряны.

Пример

Программа в листинге 14.10 копирует файл (подобно команде `cp(1)`), используя для этого операции ввода/вывода с отображаемой памятью.

Листинг 14.10. Копирование файла с использованием операций ввода/вывода с отображаемой памятью

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define COPYINCR (1024*1024*1024) /* 1 Гбайт */

int
main(int argc, char *argv[])
{
    int          fdin, fdout;
    void        *src, *dst;
    size_t      copysz;
    struct stat sbuf;
    off_t       fsz = 0;

    if (argc != 3)
        err_quit("Использование: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("невозможно открыть %s для чтения", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                      FILE_MODE)) < 0)
        err_sys("невозможно создать %s для записи", argv[2]);

    if (fstat(fdin, &sbuf) < 0) /* определить размер входного файла */
        err_sys("ошибка вызова fstat");

    /* установить размер выходного файла */
    if (ftruncate(fdout, sbuf.st_size) < 0)
        err_sys("ошибка вызова ftruncate");

    while (fsz < sbuf.st_size) {
```

```

if ((sbuf.st_size - fsz) > COPYINCR)
    copysz = COPYINCR;
else
    copysz = sbuf.st_size - fsz;

if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
                fdin, fsz)) == MAP_FAILED)
    err_sys("ошибка вызова mmap для входного файла");
if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
                MAP_SHARED, fdout, fsz)) == MAP_FAILED)
    err_sys("ошибка вызова mmap для выходного файла");

memcpy(dst, src, copysz); /* сделать копию файла */
munmap(src, copysz);
munmap(dst, copysz);
fsz += copysz;
}
exit(0);
}

```

Сначала мы открываем оба файла и затем с помощью `fstat` получаем размер исходного файла. Этот размер необходим для вызова функции `mmap`, а также для того, чтобы установить размер выходного файла. Затем вызывается функция `ftruncate`, чтобы установить размер выходного файла. Если не установить размер выходного файла, вызов функции `mmap` завершится успехом, но при первой же попытке обратиться к отображенной памяти мы получим сигнал `SIGBUS`.

Затем дважды вызывается функция `mmap` для отображения обоих файлов в память и, наконец, производится копирование содержимого входного буфера в выходной буфер с помощью функции `memcpy`. Копирование выполняется фрагментами, размер которых не превышает 1 Гбайт, чтобы ограничить объем используемой памяти (может случиться, что исходный файл будет настолько велик, что не уместится в памяти целиком). Перед отображением следующих фрагментов файлов мы освобождаем предыдущие.

В момент выборки данных из входного буфера (`src`) ядро автоматически производит чтение данных из исходного файла. При сохранении данных в выходной буфер (`dst`) данные автоматически записываются в выходной файл.

Точный момент времени, когда данные записываются в файл, зависит от алгоритма обслуживания страниц виртуальной памяти. В некоторых системах запись измененных страниц производится отдельным демоном через продолжительные промежутки времени. Если необходимо, чтобы данные сразу же были записаны на диск, вызывайте перед выходом из программы функцию `msync` с флагом `MS_SYNC`.

А теперь сравним производительность копирования файла через отображение в память и копирования с помощью функций `read` и `write` (с размером буфера 8192 байт). Результаты приводятся в табл. 14.9. Размер копируемого файла составлял 300 Мбайт, результаты даны в секундах.

В обеих системах процессорное время (сумма пользовательского и системного времени) практически одинаково для обоих вариантов копирования. В Solaris копирование с использованием связки `mmap/memcpy` заняло больше пользовательского времени и меньше системного, чем копирование с использованием связки

Таблица 14.9. Результаты измерения производительности копирования файла

Метод	Linux 3.2.0 (Intel x86)			Solaris 10 (SPARC)		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
read/write	0,01	0,54	5,67	0,29	10,60	43,67
mmap/memcpy	0,08	0,65	22,54	1,89	8,56	38,42

read/write. В Linux были получены похожие результаты для пользовательского времени, но системное время для комбинации `read/write` получилось несколько лучше, чем для связки `mmap/memcpy`. Обе системы выполняли одни и те же операции, но делали это по-разному.

Основное отличие заключается в том, что при использовании `read` и `write` выполняется намного больше системных вызовов и больше операций копирования, чем при использовании `mmap` и `memcpy`. При использовании функций `read/write` данные копируются из буфера ядра в буфер приложения (`read`) и затем обратно в буфер ядра (`write`). При использовании функций `mmap/memcpy` данные копируются напрямую из одного буфера ядра, отображенного в адресное пространство процесса, в другой буфер ядра, также отображенный в адресное пространство процесса. Это копирование возникает как результат обработки ошибки отсутствия страницы, когда выполняется попытка сослаться на еще отсутствующую страницу (здесь ошибка возникает только один раз для каждой страницы — при попытке чтения и при попытке записи). Если накладные расходы на обращения к системным вызовам и дополнительное копирование отличаются от накладных расходов на обработку ошибки обращения к отсутствующей странице, тогда один подход будет показывать лучшие результаты, чем другой.

В Linux 3.2.0 две версии программы показывают огромную разницу общего времени выполнения: версия на основе функций `read` и `write` выполняется в четыре раза быстрее, чем версия на основе функций `mmap` и `memcpy`. Однако в Solaris 10 версия на основе `mmap` и `memcpy` оказалась быстрее версии на основе `read` и `write`. Но если процессорное время осталось практически тем же, чем обусловлена такая большая разница общего времени? Одной из причин может быть более длительное ожидание завершения ввода/вывода в одной из версий. Время ожидания не включается в процессорное время. Другая причина может заключаться в том, что некоторые системные операции не относятся на счет приложения, такие как запись страниц на диск демоном. Когда нам требуется разместить страницы для чтения и записи, эти системные демоны помогают сделать их доступными. Если страницы записываются в произвольном порядке, а не друг за другом, для их записи на диск потребуется больше времени, и нам дольше придется ждать, когда страницы станут доступны для повторного использования.

В зависимости от системы операции ввода/вывода с отображаемой памятью могут производиться быстрее при копировании одного обычного файла в другой. Такой способ копирования невозможен для некоторых типов устройств (таких, как сетевые или терминальные устройства), и, кроме того, нужно проявлять осторожность, если размер файла после отображения может быть изменен. Однако

некоторые приложения могут извлечь определенные выгоды из операций ввода/вывода с отображаемой памятью, так как их использование зачастую упрощает алгоритмы, поскольку вместо использования функций `read` и `write` мы манипулируем объектом в памяти. Один из примеров, когда подобные операции ввода/вывода могут быть полезны, — работа с буфером изображения, который связан с растровым дисплеем.

В [Kreiger, Stumm, and Unrau, 1992] описывается альтернатива стандартной библиотеке ввода/вывода (глава 5), построенная на операциях ввода/вывода с отображаемой памятью.

Мы еще вернемся к вводу/выводу с отображаемой памятью в разделе 15.9, где рассмотрим пример использования общей памяти для взаимодействия процессов.

14.9. Подведение итогов

В этой главе мы описали многочисленные дополнительные функции ввода/вывода, большинство из которых будут использоваться в примерах к следующим главам:

- Неблокирующий ввод/вывод — операции ввода/вывода, которые не могут заблокировать процесс.
- Блокировки записей (более подробно мы будем их рассматривать на примере реализации библиотеки для работы с базой данных в главе 20).
- Мультиплексирование ввода/вывода — функции `select` и `poll` (мы часто будем использовать их в последующих примерах).
- Асинхронный ввод/вывод.
- Функции `readv` и `writev` (которые также будут использоваться в последующих примерах).
- Операции ввода/вывода с отображаемой памятью (`mmap`).

Упражнения

- 14.1 Напишите тестовую программу, демонстрирующую поведение вашей системы в ситуации, когда процесс пытается получить блокировку для записи на участок файла, на который уже установлена блокировка для чтения, и при этом продолжают поступать запросы на установку блокировки для чтения. Будет ли подвешен процесс, запрошивший блокировку для записи, процессами, которые устанавливают блокировки для чтения?
- 14.2 Просмотрите заголовочные файлы вашей системы и исследуйте реализацию функции `select` и четырех макросов `FD_`.
- 14.3 В системных заголовочных файлах обычно определено ограничение на количество дескрипторов, которое может храниться в типе `fd_set`. Предположим, что нам необходимо увеличить этот предел до 2048 дескрипторов. Как это сделать?

- 14.4** Сравните функции для работы с наборами сигналов (раздел 10.11) с функциями для работы с наборами дескрипторов `fd_set`. А также сравните реализацию тех и других в вашей системе.
- 14.5** Реализуйте функцию `sleep_us`, похожую на `sleep`, но приостанавливающую работу процесса на заданное количество микросекунд. Используйте для выполнения задержки функцию `select` или `poll`. Сравните эту функцию с функцией `usleep` систем BSD.
- 14.6** Можно ли реализовать функции `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` из листинга 10.17, используя рекомендательные блокировки записей вместо сигналов? Если да, напишите программу и проверьте ее.
- 14.7** Определите емкость неименованного канала, используя для этого неблокирующую операцию записи. Сравните полученное значение с константой `PIPE_BUF` из главы 2.
- 14.8** Перепишите программу из листинга 14.8, превратив ее в фильтр: она должна читать данные со стандартного ввода и записывать их в стандартный вывод, но используйте для этого асинхронные функции ввода/вывода. Что понадобится изменить, чтобы обеспечить корректную работу? Имейте в виду, что результаты должны быть одинаковыми, независимо от того, к чему подключен стандартный вывод — к терминалу, каналу или к обычному файлу.
- 14.9** Вспомните табл. 14.7. Определите для своей системы объем данных, при котором функция `writev` будет работать быстрее, чем `write`.
- 14.10** Запустите программу из листинга 14.10, скопируйте файл и посмотрите, изменилось ли время последнего обращения к исходному файлу.
- 14.11** В программе из листинга 14.10 попробуйте закрыть дескриптор исходного файла сразу после вызова функции `mmap`, чтобы убедиться, что закрытие дескриптора не оказывает влияния на операции ввода/вывода с отображаемой памятью.

15

Межпроцессные взаимодействия

15.1. Введение

В главе 8 мы описали примитивы управления процессами и увидели, как создать несколько процессов. Но единственный способ обмена информацией между этими процессами заключался в передаче открытых файловых дескрипторов через функции `fork` или `exec` либо через файловую систему. Теперь мы рассмотрим другие способы взаимодействия процессов друг с другом — механизмы IPC (Interprocess Communication), или механизмы межпроцессных взаимодействий.

В прошлом механизмы IPC в UNIX представляли собой смесь самых разных концепций, лишь немногие из которых были переносимы между различными реализациями. Благодаря усилиям по стандартизации, предпринятым POSIX и The Open Group (ранее X/Open), ситуация значительно улучшилась, но некоторые различия все еще существуют. В табл. 15.1 приводится список различных форм IPC, которые поддерживаются всеми четырьмя платформами, обсуждаемыми в этой книге.

Обратите внимание, что стандарт Single UNIX Specification (колонка SUS) разрешает реализациям поддерживать дуплексные неименованные каналы, но обязательной является лишь поддержка полудуплексных неименованных каналов. В реализациях, которые поддерживают дуплексные каналы, по-прежнему корректно работают приложения, написанные для реализаций, поддерживающих только полудуплексные каналы. В табл. 15.1 мы используем обозначение «дуплекс», чтобы выделить реализации, которые поддерживают полудуплексные каналы через использование дуплексных каналов.

В табл. 15.1 поддержка базовых функциональных возможностей обозначена галочкой (✓). Для случая с дуплексными каналами, если эта функциональность может предоставляться через сокеты домена UNIX (раздел 17.2), в соответствующих ячейках таблицы указана аббревиатура UDS (UNIX domain socket). Некоторые реализации поддерживают как базовую функциональность, так и сокеты домена UNIX, поэтому в этих ячейках указаны и галочка, и аббревиатура UDS.

Интерфейсы IPC, ранее стандартизованные как часть расширений реального времени POSIX.1, были включены в состав расширений стандарта Single UNIX Specification. Кроме того, в SUSv4 поддержка семафоров была перенесена из расширений в базовые спецификации.

Таблица 15.1. Перечень механизмов IPC, доступных в UNIX

Тип IPC	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Полудуплексные неименованные каналы	✓	дуплекс	✓	✓	дуплекс
Именованные каналы	✓	✓	✓	✓	✓
Дуплексные неименованные каналы	Допускается	✓, UDS	UDS	UDS	✓, UDS
Именованные дуплексные каналы	Устаревший	UDS	UDS	UDS	✓, UDS
Очереди сообщений XSI	XSI	✓	✓	✓	✓
Семафоры XSI	XSI	✓	✓	✓	✓
Разделяемая память XSI	XSI	✓	✓	✓	✓
Очереди сообщений (реального времени)	Расширение MSG	✓	✓		✓
Семафоры	✓	✓	✓	✓	✓
Разделяемая память (реального времени)	Расширение SHM	✓	✓	✓	✓
Сокеты	✓	✓	✓	✓	✓
STREAMS	Устаревший				✓

Именованные дуплексные каналы были реализованы на основе монтируемых каналов STREAMS, но отмечены как устаревшие в стандарте Single UNIX Specification.

Поддержка STREAMS в Linux доступна в виде пакета «Linux FastSTREAMS» из проекта OpenSS7, однако этот пакет давно не обновлялся. В описании к последней доступной версии пакета, датированной 2008 годом, утверждается, что она поддерживает ядра Linux до версии 2.6.26.

Первые десять видов IPC из табл. 15.1 обычно предназначены для взаимодействий между процессами, работающими на одном компьютере. Последние два — сокеты и STREAMS — единственные формы IPC, которые повсеместно используются для организации взаимодействий между процессами, работающими на разных компьютерах, объединенных в сеть.

Мы разделили обсуждение механизмов межпроцессных взаимодействий на три главы. В этой главе мы рассмотрим классические формы IPC: именованные и неименованные каналы, очереди сообщений, семафоры и разделяемую (общую) память. В следующей главе обсудим механизмы взаимодействий через сеть с помощью сокетов. И в главе 17 расскажем о некоторых расширенных возможностях IPC.

15.2. Неименованные каналы

Неименованные каналы (pipes, далее для краткости просто каналы) — это старейшая форма организации взаимодействий между процессами, предоставляемая операционными системами UNIX. Каналы имеют два ограничения:

1. Исторически они являются полудуплексными (то есть данные могут передаваться по ним только в одном направлении). Некоторые современные системы предоставляют дуплексные каналы, но для сохранения переносимости приложений никогда не следует пользоваться этой возможностью.
2. Каналы могут использоваться только для организации взаимодействий между процессами, которые имеют общего предка. Обычно канал создается родительским процессом, который затем вызывает функцию `fork`, после чего канал может использоваться для общения между родительским и дочерним процессами.

Далее (в разделе 15.5) мы увидим, что именованные каналы не имеют второго ограничения, а сокеты домена UNIX (unix domain sockets, раздел 17.2) — обоих ограничений.

Несмотря на указанные ограничения, полудуплексные каналы по-прежнему являются одной из наиболее широко используемых форм IPC. Каждый раз, когда вы вводите в командной строке последовательность команд, объединенных в конвейер, оболочка создает отдельный процесс для каждой команды и связывает с помощью канала стандартный вывод предыдущей команды со стандартным вводом следующей.

Неименованный канал создается с помощью функции `pipe`.

```
#include <unistd.h>
int pipe(int fd[2]);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Через аргумент `fd` возвращаются два файловых дескриптора: `fd[0]` открыт для чтения, а `fd[1]` — для записи. Данные, выводимые в `fd[1]`, становятся входными данными для `fd[0]`.

В ОС 4.3BSD и 4.4BSD каналы реализованы с использованием сокетов домена UNIX. Даже несмотря на то что сокеты по своей природе являются дуплексными, эти операционные системы ограничивают сокеты, используемые для организации каналов, так что они могут передавать информацию только в одном направлении.

Стандарт POSIX.1 разрешает реализациям поддерживать дуплексные каналы. В таких реализациях дескрипторы `fd[0]` и `fd[1]` открываются как для чтения, так и для записи.

На рис. 15.1 изображены два примера использования полудуплексных каналов. Слева показан случай, когда канал обоими концами связан с одним и тем же процессом. Справа демонстрируется случай обмена данными между двумя процессами через ядро.

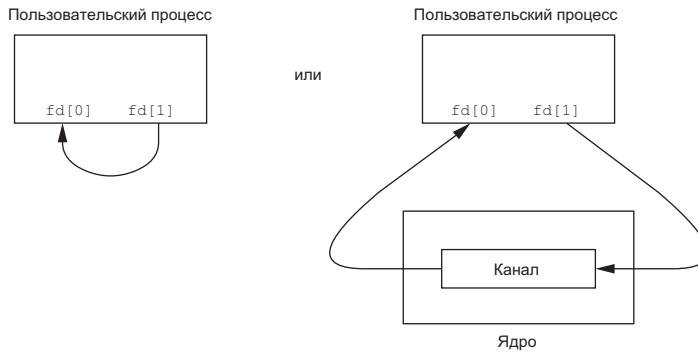


Рис. 15.1. Два примера использования полуудплексных каналов

Функция `fstat` (раздел 4.2) для дескриптора любого конца канала возвращает тип файла FIFO. Убедиться, что дескриптор соответствует каналу, можно с помощью макроса `S_ISFIFO`.

Стандарт POSIX.1 утверждает, что значение поля `st_size` структуры `stat` не определено для каналов. Но в большинстве систем вызов функции `fstat` для дескриптора, открытого на чтение, возвращает в поле `st_size` количество байтов в канале, доступных для чтения. Однако это поведение не должно использоваться при разработке переносимых приложений.

Канал, обоими концами связанный с одним и тем же процессом, бесполезен. Обычно процесс, вызывающий функцию `pipe`, затем обращается к функции `fork`, создавая, таким образом, канал для передачи данных от родительского процесса к дочернему или наоборот. Этот сценарий показан на рис. 15.2.

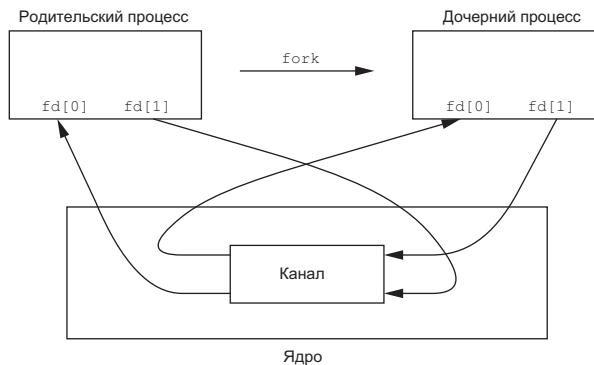


Рис. 15.2. Полудуплексные каналы после вызова функции `fork`

Порядок действий, следующих за вызовом `fork`, зависит от направления передачи данных. Если данные должны передаваться от родительского процесса дочернему, родитель закрывает дескриптор, открытый на чтение (`fd[0]`), а потомок закрывает дескриптор, открытый на запись (`fd[1]`). На рис. 15.3 показано окончательное состояние дескрипторов.

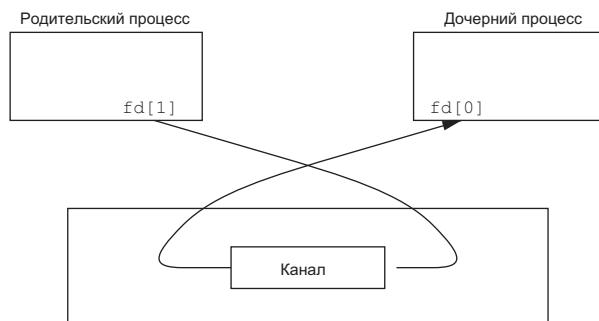


Рис. 15.3. Канал от родительского процесса к дочернему

Чтобы организовать передачу в обратном направлении, родительский процесс закрывает `fd[1]`, а дочерний процесс — `fd[0]`.

Когда один из концов канала закрывается, в силу вступают следующие два правила.

1. Если попытаться прочитать данные из канала после закрытия дескриптора для записи, функция `read` вернет 0, чтобы сообщить о достижении конца файла после того, как все данные будут прочитаны. (Технически признак конца файла не будет сгенерирован, пока не будут закрыты все дескрипторы, открытые для записи в канал. Такое возможно при создании дубликатов дескрипторов, благодаря чему сразу несколько процессов могут производить запись в канал. Однако обычно у канала имеется один дескриптор, открытый для чтения, и один дескриптор, открытый для записи. Когда в следующем разделе мы перейдем к изучению именованных каналов, то увидим, что зачастую в один именованный канал могут писать сразу несколько процессов.)
2. Если попытаться выполнить запись в канал после закрытия дескриптора для чтения, будет сгенерирован сигнал `SIGPIPE`. Если приложение игнорирует этот сигнал или перехватывает его и возвращает управление из обработчика сигнала нормальным образом, функция `write` вернет значение `-1` и код ошибки `EPIPE` в переменной `errno`.

При записи данных в канал (именованный или неименованный) размер буфера канала в ядре определяется константой `PIPE_BUF`. Если в канал записывается количество байтов, не превышающее `PIPE_BUF`, эти данные не будут перемежаться данными, записываемыми в канал другими процессами. Если попытаться одним вызовом `write` записать больше, чем `PIPE_BUF` байт, записанные данные могут быть перемешаны с данными, поступившими от других процессов. Определить значение `PIPE_BUF` можно с помощью функции `pathconf` или `fpathconf` (табл. 2.12).

Пример

Программа в листинге 15.1 создает канал между родительским и дочерним процессами и передает данные по этому каналу.

Листинг 15.1. Передача данных от родительского процесса к дочернему через канал

```
#include "apue.h"

int
main(void)
{
    int      n;
    int      fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        close(fd[0]);
        write(fd[1], "привет, МИР\n", 11);
    } else { /* дочерний процесс */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Обратите внимание, что направление передачи данных по каналу здесь соответствует изображенному на рис. 15.3.

В этом примере мы работали с дескрипторами канала напрямую, используя функции `write` и `read`. Но гораздо интереснее было бы продублировать тот или иной дескриптор на стандартный ввод или стандартный вывод. После этого дочерний процесс мог бы запустить программу, которая получает данные из стандартного ввода (из созданного нами канала) или производит запись в стандартный вывод (в канал).

Пример

Рассмотрим программу, которая должна выводить данные постранично. Вместо того чтобы заново придумывать алгоритм постраничного вывода, который уже реализован некоторыми утилитами UNIX, мы попробуем воспользоваться программой постраничного просмотра, которую предпочитает пользователь. Чтобы не создавать временный файл для хранения результатов и не вызывать функцию `system` для отображения содержимого этого файла, мы воспользуемся каналом, по которому сразу будем отправлять данные программе постраничного просмотра. Для этого сначала создадим канал, с помощью `fork` запустим дочерний процесс, переустановим дескриптор канала, открытый для чтения, на стандартный ввод и затем вызовом `exec` запустим программу постраничного просмотра. Листинг 15.2 показывает, как это можно сделать. (В этом примере программа принимает аргумент командной строки с именем файла для вывода. Но часто бывает, что данные, которые нужно вывести в терминал, уже находятся в памяти.)

Листинг 15.2. Передача файла программе постраничного просмотра

```

#include "apue.h"
#include <sys/wait.h>

/* программа постраничного просмотра по умолчанию */

#define DEF_PAGER "/bin/more"

int
main(int argc, char *argv[])
{
    int      n;
    int      fd[2];
    pid_t    pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("Использование: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        close(fd[0]);      /* закрыть дескриптор для чтения */

        /* родительский процесс копирует argv[1] в канал */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("ошибка записи в канал");
        }
        if (ferror(fp))
            err_sys("ошибка вызова функции fgets");

        close(fd[1]);      /* закрыть дескриптор для записи */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("ошибка вызова функции waitpid");

        exit(0);
    } else {                  /* дочерний процесс */
        close(fd[1]);      /* закрыть дескриптор для записи */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("ошибка переназначения канала на stdin");
            close(fd[0]); /* уже не нужен после вызова dup2 */
        }

        /* определить аргументы для execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;          /* перейти за последний слеш */
    }
}

```

```

    else
        argv0 = pager; /* в имени программы нет слеша */

    if (exec1(pager, argv0, (char *)0) < 0)
        err_sys("ошибка запуска программы %s", pager);
    }
    exit(0);
}

```

Перед вызовом функции `fork` создается канал. После вызова `fork` родительский процесс закрывает дескриптор канала, открытый для чтения, а дочерний процесс — дескриптор, открытый для записи. После этого дочерний процесс вызывает функцию `dup2` и переназначает конец канала, открытый для чтения, на стандартный ввод.

Дублируя один дескриптор в другой (`fd[0]` — на стандартный ввод в дочернем процессе), необходимо убедиться в том, что номер дескриптора не совпадает с тем, который нам нужен. Если бы это был дескриптор с нужным нам номером, в результате вызова функций `dup2` и `close` единственная копия дескриптора была бы закрыта. (Поведение функции `dup2`, когда оба ее аргумента равны, обсуждалось в разделе 3.12). В этой программе, если бы стандартный ввод не был открыт командной оболочкой, функция `fopen`, вызываемая в самом начале, все равно открыла бы для файла дескриптор с номером 0 — наименьшим неиспользуемым номером дескриптора, поэтому `fd[0]` никогда не должен совпадать с дескриптором стандартного ввода. Однако всякий раз, обращаясь к функциям `dup2` и `close`, дублируя один дескриптор в другой, в качестве меры предосторожности мы будем сначала сравнивать эти дескрипторы.

Обратите внимание, как используется переменная окружения `PAGER`, чтобы получить имя программы постраничного вывода, предпочтаемой пользователем. Если таковая не определена, мы запускаем программу по умолчанию. Это наиболее распространенное правило использования переменных окружения.

Пример

Давайте вспомним функции `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT` и `WAIT_CHILD` из раздела 8.9. В листинге 10.17 была показана реализация этих функций на основе сигналов. В листинге 15.3 приводится реализация этих же функций, но уже на основе каналов.

Листинг 15.3. Процедуры синхронизации родительского и дочернего процессов

```
#include "apue.h"

static int pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("ошибка вызова функции pipe");
}

void
```

```

TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");

    if (c != 'p')
        err_quit("WAIT_PARENT: получены некорректные данные");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("ошибка вызова функции write");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("ошибка вызова функции read");

    if (c != 'c')
        err_quit("WAIT_CHILD: получены некорректные данные");
}

```

Перед вызовом `fork` создаются два канала, как показано на рис. 15.4. Вызовом `TELL_CHILD` родительский процесс записывает в канал символ «р», а дочерний процесс вызовом `TELL_PARENT` записывает символ «с». Функции `WAIT_xxx` блокируют-ся в системном вызове `read` до получения одиночного символа.

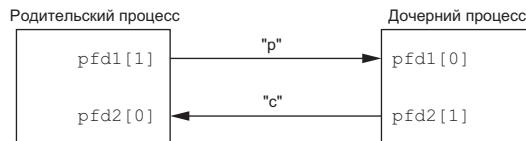


Рис. 15.4. Использование каналов для синхронизации родительского и дочернего процессов

Обратите внимание, что в этой реализации каждый канал имеет два открытых для чтения дескриптора. Кроме того что дочерний процесс может читать из дескриптора `pfd1[0]`, родительский процесс также может читать из этого канала. Но в данном случае это не имеет никакого значения, потому что родительский процесс не пытается читать из него.

15.3. Функции `popen` и `pclose`

Поскольку чаще всего канал создается для взаимодействия с другим процессом, чтобы получать от него или отправлять ему данные, стандартная библиотека ввода/вывода традиционно поддерживает функции `pclose` и `popen`. Эти две функции берут на себя всю рутинную работу, которую мы до сих пор выполняли самостоятельно: создание канала, создание дочернего процесса, закрытие неиспользуемых дескрипторов канала, запуск команды и ожидание завершения команды.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Возвращает указатель на структуру FILE в случае успеха,
NULL – в случае ошибки

```
int pclose(FILE *fp);
```

Возвращает код завершения *cmdstring*, -1 – в случае ошибки

Функция `popen` посредством `fork` и `exec` запускает команду *cmdstring* и возвращает указатель на объект `FILE`. Если в аргументе *type* передается значение "r", указатель на файл будет связан со стандартным выводом *cmdstring* (рис. 15.5).



Рис. 15.5. Результат выполнения инструкции `fp = popen(cmdstring, «r»)`

Если в аргументе *type* передается значение "w", указатель на файл будет связан со стандартным вводом *cmdstring* (рис. 15.6).



Рис. 15.6. Результат выполнения инструкции `fp = popen(cmdstring, «w»)`

Чтобы запомнить правила назначения второго аргумента функции `popen`, вспомните функцию `fopen`, которая возвращает объект `FILE`, открытый для чтения, если аргумент *type* имеет значение "r", и для записи, если аргумент *type* имеет значение "w".

Функция `pclose` закрывает поток ввода/вывода, ожидает завершения команды и возвращает код завершения командного интерпретатора, запущенного для выполнения команды *cmdstring*. (Код завершения рассматривался в разделе 8.6. Функция `system`, описанная в разделе 8.13, также возвращает код завершения.) Если командный интерпретатор не смог запуститься, функция `pclose` вернет код завершения, как если бы командная оболочка вызвала `exit(127)`.

Команда *cmdstring* запускается интерпретатором Bourne shell как

```
sh -c cmdstring
```

Это означает, что командная оболочка производит интерпретацию всех специальных символов, которые встречаются в строке *cmdstring*, что позволяет, например, выполнить команду

```
fp = popen("ls *.c", "r");
```

или

```
fp = popen("cmd 2>&1", "r");
```

Пример

Теперь попробуем переписать программу из листинга 15.2 так, чтобы она использовала функцию *popen*. Текст новой программы приводится в листинге 15.4.

Листинг 15.4. Передача файла программе постраничного просмотра с использованием функции *popen*

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER "${PAGER:-more}" /* либо значение переменной окружения, */
                           /* либо значение по умолчанию */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE   *fpin, *fpout;

    if (argc != 2)
        err_quit("Использование: a.out <полное_имя_файла>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("невозможно открыть %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("ошибка вызова функции popen");

    /* передать argv[1] программе постраничного просмотра */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("ошибка записи в канал");
    }
    if (ferror(fpin))
        err_sys("ошибка вызова функции fgets");
    if (pclose(fpout) == -1)
        err_sys("ошибка вызова функции pclose");

    exit(0);
}
```

Использование функции *popen* позволило значительно уменьшить размер программы.

Команда `${PAGER:-more}` говорит о том, что следует использовать значение переменной окружения *PAGER*, если она определена и содержит непустую строку, или строку *more*.

Пример — функции popen и pclose

В листинге 15.5 приводится наша реализация функций `popen` и `pclose`.

Листинг 15.5. Функции popen и pclose

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Указатель на массив, размещаемый во время выполнения.
 */
static pid_t      *childpid = NULL;

/*
 * Будет получено из нашей функции open_max(), листинг 2.4.
 */
static int        maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int      i;
    int      pfd[2];
    pid_t    pid;
    FILE    *fp;

    /* допустимы только "r" или "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;
        return(NULL);
    }

    if (childpid == NULL) { /* самый первый вызов функции */
        /* разместить массив идентификаторов потомков, заполненный нулями */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL); /* значение errno будет установлено функцией pipe() */
    if (pfd[0] >= maxfd || pfd[1] >= maxfd) {
        close(pfd[0]);
        close(pfd[1]);
        errno = EMFILE;
        return(NULL);
    }

    if ((pid = fork()) < 0) {
        return(NULL); /* значение errno будет установлено функцией fork() */
    } else if (pid == 0) { /* дочерний процесс */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
        }
    }
}
```

```

        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }

/* закрыть все дескрипторы в childpid[] */
for (i = 0; i < maxfd; i++)
    if (childpid[i] > 0)
        close(i);

execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
_exit(127);
}

/* родительский процесс... */
if (*type == 'r') {
    close(pfd[1]);
    if ((fp = fdopen(pfd[0], type)) == NULL)
        return(NULL);
} else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return(NULL);
}

childpid[fileno(fp)] = pid; /* запомнить pid потомка для данного fd */
return(fp);
}

int
pclose(FILE *fp)
{
    int      fd, stat;
    pid_t    pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1); /* функция popen() никогда не вызывалась */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
        errno = EINVAL;
        return(-1); /* недопустимый дескриптор файла */
    }

    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1); /* fp не был открыт функцией popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* от waitpid получен код ошибки, отличный от EINTR */

    return(stat); /* вернуть код завершения потомка */
}

```

В принципе, функция `ropen` похожа на тот код, который мы использовали ранее в этой главе, однако существует ряд моментов, которые необходимо принять во внимание. Прежде всего, каждый раз, когда вызывается функция `ropen`, нужно запоминать идентификатор дочернего процесса и дескриптор файла либо указатель на объект `FILE`. Мы решили сохранять идентификатор дочернего процесса в массиве `childpid`, который индексируется номерами файловых дескрипторов. Благодаря этому функция `pclose`, получая указатель на объект `FILE`, сможет восстановить по нему номер дескриптора файла с помощью функции `fileno` и затем извлечь из массива идентификатор дочернего процесса, чтобы передать его функции `waitpid`. Поскольку данный процесс может вызывать функцию `ropen` много раз, при первом обращении к функции `ropen` мы размещаем в динамической памяти массив `childpid` максимального размера.

Обратите внимание, что функция `open_max` из листинга 2.4 может вернуть предполагаемое максимально допустимое количество открытых файлов, если это значение не определено системой. Следует проявлять особую осторожность и избегать использования файловых дескрипторов, номера которых больше (или равны) значения, возвращаемого функцией `open_max`. В функции `ropen`, если значение, возвращаемое вызовом `open_max`, окажется слишком маленьким, мы закрываем дескрипторы канала, устанавливаем в переменной `errno` код ошибки `EMFILE` (чтобы сообщить, что открыто слишком много файлов) и возвращаем `-1`. В функции `pclose`, если файловый дескриптор, соответствующий аргументу `fp`, оказывается больше ожидаемого, мы устанавливаем в переменной `errno` код ошибки `EINVAL` и возвращаем `-1`.

Вызовы функций `pipe` и `fork` и создание дубликатов дескрипторов для каждого процесса производятся практически так же, как мы это делали раньше.

Стандарт POSIX.1 требует, чтобы в дочернем процессе функция `ropen` закрывала все потоки, открытые предыдущими обращениями к ней. Для этого в дочернем процессе выполняется обход массива `childpid` и закрытие всех открытых дескрипторов.

Что случится, если процесс, вызывающий `pclose`, установил обработчик сигнала `SIGCHLD`? В этом случае функция `waitpid` вернет код ошибки `EINTR`. Так как мы допускаем, что вызывающий процесс может перехватывать сигнал `SIGCHLD` (или любой другой сигнал, в результате чего может быть прервано выполнение системного вызова `waitpid`), мы просто вызываем функцию `waitpid` еще раз, если ее выполнение прервано в результате перехвата сигнала.

Обратите внимание: приложение может самостоятельно вызвать функцию `waitpid` и получить код завершения дочернего процесса, созданного функцией `ropen`. В этом случае функция `waitpid`, вызываемая из `pclose`, обнаружит отсутствие дочернего процесса и вернет значение `-1` с кодом ошибки `ECHILD` в `errno`. Такое поведение регламентируется стандартом POSIX.1.

Некоторые ранние версии `pclose` возвращали код ошибки `EINTR`, если выполнение функции `wait` было прервано перехваченным сигналом. Кроме того, в некоторых ранних версиях `pclose` игнорировались или блокировались сигналы `SIGINT`, `SIGQUIT` и `SIGHUP`. В стандарте POSIX.1 такое поведение считается недопустимым.

Обратите внимание, что функция `popen` никогда не должна вызываться из программ, для которых установлен бит `set-user-ID` или `set-group-ID`. Выполнение команды функцией `popen` эквивалентно выполнению инструкции

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

которая запускает командный интерпретатор и команду *command* с окружением, унаследованным от вызывающей программы. В этом случае злоумышленник, манипулируя значениями переменных окружения, получает возможность запускать произвольные команды с повышенными привилегиями.

Функция `popen` особенно удобна для организации взаимодействия с простыми фильтрами, предназначенными для преобразования входных или выходных данных запускаемой команды. Это относится к случаям, когда программа сама выстраивает конвейер команд.

Пример

Рассмотрим пример программы, которая выводит в стандартный вывод приглашение и читает введенную строку из стандартного ввода. С помощью функции `popen` можно поместить некоторую программу между основным приложением и его стандартным вводом, чтобы выполнить первичную обработку входных данных. На рис. 15.7 показано, как взаимодействуют процессы в такой ситуации.

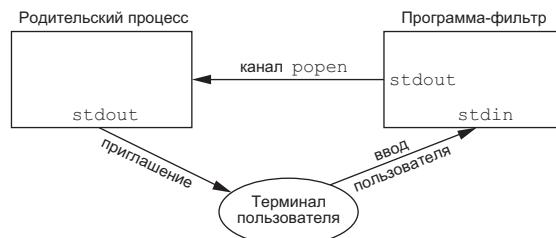


Рис. 15.7. Первичная обработка входных данных с помощью функции `popen`

В качестве первичной обработки может выполняться, например, автоматическое дополнение имен файлов или предоставление истории команд (сохранение ранее введенных команд).

В листинге 15.6 показан пример подобного простого фильтра. Этот фильтр копирует данные со стандартного ввода на стандартный вывод, преобразуя символы верхнего регистра в нижний. В следующем разделе, когда мы будем рассказывать о сопроцессах, вы узнаете, почему мы вставили вызов функции `fflush` после вывода символа перевода строки.

Листинг 15.6. Фильтр для преобразования символов верхнего регистра

в нижний регистр

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

```
int
```

```

main(void)
{
    int      c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("ошибка вывода символа");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}

```

Мы скомпилировали этот фильтр в выполняемый файл `myuc1c`, который будет вызываться функцией `popen` из программы в листинге 15.7.

Листинг 15.7. Вызов фильтра преобразования регистра символов при чтении данных

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char      line[MAXLINE];
    FILE     *fpin;

    if ((fpin = popen("myuc1c", "r")) == NULL)
        err_sys("ошибка вызова функции popen");

    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* чтение из канала */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("ошибка вызова функции fputs");
    }
    if (pclose(fpin) == -1)
        err_sys("ошибка вызова функции pclose");
    putchar('\n');
    exit(0);
}

```

Вызов `fflush` после вывода строки приглашения необходим, так как стандартный вывод обычно буферизуется построчно, а строка приглашения не включает символ перевода строки.

15.4. Сопроцессы

В системе UNIX фильтрами называются программы, которые читают входные данные со стандартного ввода и выводят результаты на стандартный вывод. Как правило, фильтры используются при конвейерной обработке данных. Фильтр

становится сопроцессом, если одна и та же программа предоставляет данные для фильтра и получает его вывод.

Командная оболочка Korn shell поддерживает возможность запуска сопроцессов (см. [Bolsky and Korn, 1995]). Командные оболочки Bourne shell, Bourne-again shell и C shell такой возможности не имеют. Обычно сопроцесс запускается из командной оболочки в фоновом режиме, и его стандартный ввод и стандартный вывод соединены с другой программой посредством каналов. Несмотря на то что синтаксис команд оболочки, необходимых для запуска сопроцесса и соединения его ввода и вывода с другим процессом, весьма запутан (за подробностями обращайтесь к [Bolsky and Korn, 1995], с. 62–63), работа с сопроцессами достаточно удобна из программ на языке С.

Учитывая одностороннюю природу каналов, для организации взаимодействия с сопроцессом необходимо создать два односторонних канала: один — к стандартному вводу сопроцесса и другой — от его стандартного вывода. После этого можно записать данные на стандартный ввод сопроцесса, обработать их и прочитать результат с его стандартного вывода.

Пример

Рассмотрим пример сопроцесса. Основной процесс создает два канала: один связан со стандартным вводом сопроцесса, а второй — с его стандартным выводом. Эта ситуация изображена на рис. 15.8.



Рис. 15.8. Запись в стандартный ввод и чтение из стандартного вывода сопроцесса

Программа в листинге 15.8 — простейший сопроцесс, который принимает два числа со стандартного ввода, вычисляет сумму и выводит ее в стандартный вывод. (Сопроцессам обычно доверяют более серьезные задачи. Это во многом искусственный пример, он придуман лишь с целью показать работу механизмов взаимодействия между процессами.)

Листинг 15.8. Простейший фильтр, который складывает два числа

```
#include "apue.h"

int
main(void)
{
    int      n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0; /* завершить строку нулевым символом */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
        }
    }
}
```

```

        n = strlen(line);
        if (write(STDOUT_FILENO, line, n) != n)
            err_sys("ошибка вызова функции write");
    } else {
        if (write(STDOUT_FILENO, "неверные аргументы\n", 19) != 19)
            err_sys("ошибка вызова функции write");
    }
}
exit(0);
}

```

Мы скомпилировали эту программу в выполняемый файл **add2**.

Программа в листинге 15.9 читает два числа со стандартного ввода и вызывает сопроцесс **add2**. Значение, полученное от сопроцесса, выводится в стандартный вывод.

Листинг 15.9. Программа, использующая фильтр add2

```

#include "apue.h"

static void sig_pipe(int); /* обработчик сигнала */

int
main(void)
{
    int      n, fd1[2], fd2[2];
    pid_t    pid;
    char     line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("ошибка вызова функции signal");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("ошибка вызова функции pipe");

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid > 0) { /* родительский процесс */
        close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("ошибка записи в канал");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("ошибка чтения из канала");
            if (n == 0) {
                err_msg("канал был закрыт в дочернем процессе");
                break;
            }
            line[n] = 0; /* добавить завершающий нулевой символ */
            if (fputs(line, stdout) == EOF)
                err_sys("ошибка вызова функции fputs");
        }

        if (ferror(stdin))
            err_sys("ошибка получения данных со стандартного ввода");
        exit(0);
    }
}

```

```

} else { /* дочерний процесс */
    close(fd1[1]);
    close(fd2[0]);
    if (fd1[0] != STDIN_FILENO) {
        if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("ошибка вызова функции dup2 для stdin");
        close(fd1[0]);
    }

    if (fd2[1] != STDOUT_FILENO) {
        if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("ошибка вызова функции dup2 для stdout");
        close(fd2[1]);
    }

    if (execl("./add2", "add2", (char *)0) < 0)
        err_sys("ошибка вызова функции execl");
}
exit(0);
}

static void
sig_pipe(int signo)
{
    printf("перехвачен сигнал SIGPIPE\n");
    exit(1);
}

```

Здесь создаются два канала; дочерний и родительский процессы закрывают ненужные дескрипторы каналов. Мы должны использовать два канала: один — в качестве стандартного ввода сопроцесса и второй — в качестве его стандартного вывода. Затем, перед вызовом `execl`, дочерний процесс вызывает функцию `dup2`, чтобы перенести дескрипторы каналов на свои стандартные устройства ввода и вывода.

Если скомпилировать и запустить программу из листинга 15.9, она будет работать так, как мы и ожидали. Если в то время, когда основная программа ждет ввода двух чисел, завершить сопроцесс `add2` командой `kill`, при попытке выполнить запись в канал, для которого отсутствует читающий процесс, основная программа получит сигнал `SIGPIPE` (упражнение 15.4).

Пример

В примере сопроцесса `add2` (листинг 15.8) мы намеренно использовали низкоуровневые операции ввода/вывода (системные вызовы UNIX) `read` и `write`. А может ли сопроцесс использовать стандартную библиотеку ввода/вывода? Такая версия сопроцесса приводится в листинге 15.10.

Листинг 15.10. Простейший фильтр, складывающий два числа и реализованный с применением стандартной библиотеки ввода/вывода

```
#include "apue.h"

int
main(void)
{
    int      int1, int2;
    char    line[MAXLINE];
```

```

while (fgets(line, MAXLINE, stdin) != NULL) {
    if (sscanf(line, "%d%d", &int1, &int2) == 2) {
        if (printf("%d\n", int1 + int2) == EOF)
            err_sys("ошибка вызова функции printf");
    } else {
        if (printf("неверные аргументы\n") == EOF)
            err_sys("ошибка вызова функции printf");
    }
}
exit(0);
}

```

Если теперь попытаться вызвать этот новый сопроцесс из программы в листинге 15.9, она перестанет работать. Проблема в том, что стандартная библиотека по умолчанию буферизует операции ввода/вывода. При первом обращении к функции `fgets` в программе из листинга 15.10 стандартная библиотека ввода/вывода размещает буфер и выбирает режим буферизации. Поскольку стандартный ввод является каналом, библиотека по умолчанию выбирает режим полной буферизации. То же происходит со стандартным выводом. Пока программа `add2` ожидает поступления данных со стандартного ввода, основная программа (из листинга 15.9) ожидает поступления данных из канала. В результате возникает тупиковая ситуация.

Можно изменить программу из листинга 15.10, добавив перед циклом `while` следующие строки:

```

if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("ошибка вызова функции setvbuf");

```

Эти строки заставят функцию `fgets` вернуть управление после записи строки символов в канал, а функцию `printf` — вызвать `fflush` после вывода символа перевода строки (подробное описание режимов буферизации в стандартной библиотеке ввода/вывода приводилось в разделе 5.4). Добавив явные вызовы функции `setvbuf`, мы исправим ошибку в программе из листинга 15.10.

Иная методика требуется, если нет возможности изменить программу, к стандартному выводу которой мы присоединяем канал. Например, при использовании в нашей программе в качестве сопроцесса программы `awk(1)` (вместо `add2`) следующий код не будет работать:

```

#! /bin/awk -f
{ print $1 + $2 }

```

Причина опять кроется в буферизации операций ввода/вывода. Но на этот раз мы не можем изменить программу `awk` (если, конечно, не имеем исходных текстов этой программы). У нас нет возможности внести изменения в выполняемый файл, чтобы изменить режим буферизации по умолчанию.

Решить проблему можно, если заставить сопроцесс (в данном случае `awk`) думать, что его стандартный ввод и стандартный вывод соединены с терминалом. Это заставит функции стандартной библиотеки ввода/вывода в сопроцессе установить

режим построчной буферизации для двух потоков ввода/вывода, как если бы функция `setvbuf` была вызвана явно. Для этого в главе 19 мы будем использовать псевдотерминалы.

15.5. FIFO

Каналы FIFO (First In First Out — первым пришел, первым ушел) иногда еще называют именованными каналами. Неименованные каналы можно использовать только для взаимодействия процессов, имеющих общего предка, создавшего каналы. С помощью каналов FIFO можно организовать взаимодействие между процессами, которые не связаны «родственными узами».

В главе 4 мы видели, что FIFO — это особый тип файлов. Определенный код в поле `st_mode` структуры `stat` (раздел 4.2) указывает, что файл является каналом FIFO. Проверить файл на принадлежность к типу FIFO можно с помощью макропса `S_ISFIFO`.

Создание канала FIFO очень похоже на создание обычного файла. Канал с полным именем `pathname` действительно создается в пределах файловой системы.

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

int mkfifo(const char *pathname, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Аргумент `mode` функции `mkfifo` имеет тот же смысл, что и в функции `open` (раздел 3.3). Правила назначения пользователя и группы владельца FIFO совпадают с описанными в разделе 4.6.

Функция `mkfifoat` действует подобно функции `mkfifo`, но ее можно использовать для создания FIFO в каталоге, путь `path` к которому откладывается относительно каталога, представленного дескриптором `fd`. Как и для других функций `*at`, возможны три варианта.

- Если параметр `path` является строкой абсолютного пути, параметр `fd` игнорируется и функция `mkfifoat` действует подобно функции `mkfifo`.
- Если параметр `path` является строкой относительного пути и в параметре `fd` передается допустимый файловый дескриптор открытого каталога, относительный путь откладывается от данного каталога.
- Если параметр `path` является строкой относительного пути и в параметре `fd` указано специальное значение `AT_FDCWD`, относительный путь откладывается от текущего рабочего каталога и `mkfifoat` действует подобно `mkfifo`.

После создания канала FIFO с помощью функции `mkfifo` или `mkfifoat` можно открыть его функцией `open`. Все обычные функции ввода/вывода (`close`, `read`, `write`, `unlink` и др.) могут работать с каналами FIFO.

Приложения могут создавать каналы FIFO с помощью функции `mknod` или `mknodat`. Поскольку изначально стандарт POSIX.1 не включал в себя функцию `mknod`, специально для этого стандарта была придумана функция `mkfifo`. Сейчас функции `mknod` и `mknodat` включены в стандарт в виде расширения XSI.

Стандарт POSIX.1 также включает команду `mkfifo(1)`. Все четыре платформы, обсуждаемые в этой книге, поддерживают данную команду. Она позволяет создавать каналы FIFO из командной оболочки, чтобы затем использовать их для перенаправления ввода/вывода.

При открытии FIFO оказывает влияние флаг `O_NONBLOCK`.

- В обычной ситуации (без флага `O_NONBLOCK`) операция открытия FIFO только для чтения будет заблокирована, пока другой процесс не откроет канал для записи. Аналогично, операция открытия только для записи будет заблокирована, пока другой процесс не откроет канал для чтения.
- Если флаг `O_NONBLOCK` указан, при попытке открыть канал только для чтения функция `open` сразу же вернет управление. Но при попытке открыть канал только для записи функция `open` вернет значение `-1` и код ошибки `ENXIO` в переменной `errno`, если канал не был открыт другим процессом для чтения.

Как и в случае с неименованными каналами, если попытаться выполнить запись в FIFO, который не был открыт для чтения, процесс получит сигнал `SIGPIPE`. Когда последний пишущий в FIFO процесс закроет канал, читающий процесс получит признак конца файла.

Нередко запись данных в канал FIFO выполняется из нескольких процессов. Это означает, что необходимо побеспокоиться об атомарности операции записи, чтобы избежать смешивания данных, поступающих от разных процессов. Максимальный объем данных, который можно атомарно записать в канал FIFO, определяется, как и для неименованных каналов, константой `PIPE_BUF`.

Каналы FIFO применяются в двух случаях.

1. Каналы FIFO используются командными оболочками для передачи данных от одного конвейера команд другому без создания временных файлов для хранения промежуточных данных.
2. Каналы FIFO используются для организации взаимодействий типа клиент-сервер.

Каждый из этих случаев мы рассмотрим на конкретных примерах.

Пример — использование FIFO для дублирования вывода

Каналы FIFO можно использовать для дублирования данных, передаваемых между сериями команд оболочки. Это (как и неименованные каналы) помогает избежать создания временных файлов для хранения промежуточных данных. Но если неименованные каналы могут служить исключительно для линейного объединения процессов в конвейер, то каналы FIFO, благодаря наличию имени, могут использоваться для нелинейного объединения.

Рассмотрим ситуацию, когда необходимо обработать отфильтрованные данные дважды, изображенную на рис. 15.9.

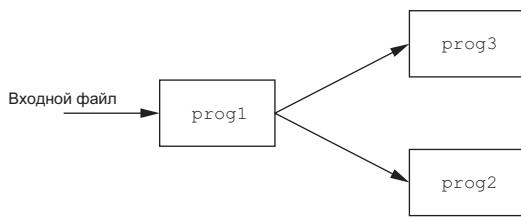


Рис. 15.9. Ситуация, когда необходимо обработать отфильтрованные данные дважды

С помощью канала FIFO и программы `tee(1)` можно реализовать эту процедуру без использования временного файла. (Программа `tee` копирует данные из стандартного ввода в стандартный вывод и в файл, заданный в командной строке.)

```

mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
  
```

Эта последовательность команд создает канал FIFO, после чего запускает в фоновом режиме программу `prog3`, которая читает данные из канала. Затем запускается фильтр `prog1`, вывод которого через команду `tee` поступает в канал и на вход программы `prog2`. На рис. 15.10 показана схема взаимодействия процессов в этой ситуации.

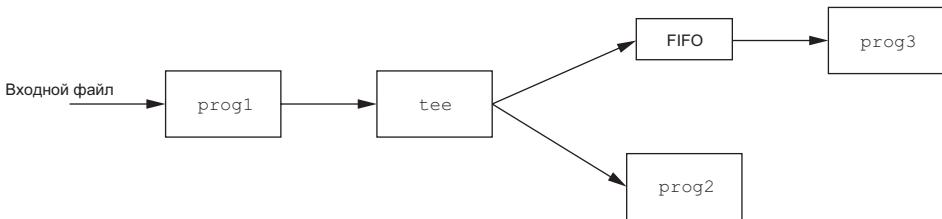


Рис. 15.10. Использование канала FIFO и команды `tee` для обработки потока данных двумя процессами

Пример — взаимодействия типа клиент-сервер

Еще одна область применения каналов FIFO — передача данных между клиентом и сервером. Если имеется сервер, который обслуживает многочисленные клиентские приложения, каждый клиент может посылать запросы через канал FIFO с известным именем, заранее созданный сервером. (Имеется в виду, что полное имя канала FIFO заранее известно всем клиентам, которые нуждаются в услугах сервера.) Схема такого взаимодействия показана на рис. 15.11.

Поскольку в этой ситуации писать в канал FIFO могут сразу несколько процессов, необходимо, чтобы размеры запросов не превышали величины `PIPE_BUF`. Это позволит избежать смешивания данных, записываемых различными процессами.

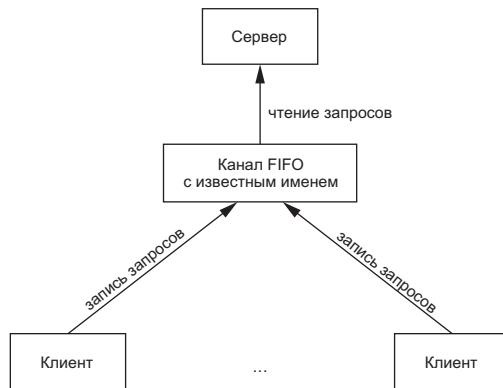


Рис. 15.11. Обмен данными между клиентами и сервером с помощью канала FIFO

При использовании каналов FIFO для организации взаимодействий такого типа возникает проблема с отправкой ответа сервера клиенту. Для этого не может использоваться единственный канал FIFO, поскольку клиенты не смогут отличить ответ сервера на свой запрос от ответов на запросы других клиентов. Одно из решений состоит в том, чтобы каждый клиент отсыпал вместе с запросом идентификатор процесса. Тогда сервер мог бы создавать каналы FIFO для связи с каждым клиентом, генерируя имя канала на основе идентификатора процесса клиента. Например, сервер может создавать каналы FIFO с именами `/tmp/serv1.XXXXX`, где `XXXXX` – идентификатор процесса клиента. Схема такого взаимодействия показана на рис. 15.12.



Рис. 15.12. Организация взаимодействий типа клиент/сервер с помощью каналов FIFO

Такая схема вполне работоспособна, хотя сервер не сможет обнаружить ситуацию аварийного завершения клиента. Это приводит к тому, что каналы FIFO, созданные для взаимодействия с конкретными клиентами, остаются в файловой

системе. Кроме того, сервер должен предусматривать обработку сигнала SIGPIPE, поскольку клиент может послать запрос и завершить работу, не дожидаясь ответа и оставляя свой канал FIFO с одним пишущим процессом (сервером), но без читающего процесса.

В ситуации, изображенной на рис. 15.12, если сервер откроет канал FIFO с заранее предопределенным именем только для чтения, то всякий раз, когда количество клиентов будет достигать 0, сервер будет получать из канала признак конца файла. Чтобы этого не происходило, сервер обычно открывает канал FIFO как для чтения, так и записи (упражнение 15.10).

15.6. XSI IPC

Три типа IPC, которые называются XSI IPC, — очереди сообщений, семафоры и разделяемая память — имеют много общего. В этом разделе мы рассмотрим характеристики, общие для всех трех типов взаимодействий, а в следующих разделах — функции, специфичные для каждого из них.

Функции XSI IPC целиком основаны на функциях System V IPC. Эти три типа взаимодействий появились в 70-х годах во внутренней версии UNIX AT&T, которая называлась Columbus UNIX. Позднее эти механизмы IPC были добавлены в System V. Их часто критикуют за то, что они используют свою собственную схему именования объектов, а не файловую систему.

15.6.1. Идентификаторы и ключи

Каждой структуре IPC (очереди сообщений, семафору или сегменту разделяемой памяти) в ядре соответствует неотрицательное целое число — *идентификатор*. Например, чтобы отправить сообщение в очередь или получить его, достаточно знать лишь идентификатор очереди. В отличие от дескрипторов файлов, идентификаторы IPC не являются маленькими целыми числами. Каждый раз, когда создается какая-либо структура IPC, идентификатор, присваиваемый этой структуре, все время увеличивается, пока не достигнет максимально возможного значения для целых чисел, после чего сбрасывается в ноль.

Идентификатор — это внутреннее имя объекта IPC. Процессам же необходим механизм внешних имен, чтобы организовать взаимодействие через определенный объект IPC. Для этого каждому объекту IPC ставится в соответствие ключ, который выступает в роли внешнего имени.

Всякий раз, когда создается структура IPC (`msgget`, `semget` или `shmget`), обязательно должен быть указан ключ. Тип данных ключа является одним из основных системных типов данных — `key_t`, который часто определяется в заголовочном файле `<sys/types.h>` как длинное целое со знаком. Ядро преобразует этот ключ во внутренний идентификатор.

Существуют разные способы организовать взаимодействие между клиентом и сервером через структуру IPC.

1. Сервер может создать новую структуру IPC с ключом `IPC_PRIVATE` и записать куда-нибудь (например, в файл) полученный идентификатор, чтобы сделать его доступным для клиента. Ключ `IPC_PRIVATE` гарантирует, что сервер создаст совершенно новую структуру IPC. Недостаток этого метода заключается в применении операций с файловой системой — сервер должен записать идентификатор в файл, а клиент — прочитать его из файла.

Ключ `IPC_PRIVATE` также может использоваться для организации взаимодействия родительского и дочернего процессов. Родительский процесс создает новую структуру IPC с ключом `IPC_PRIVATE`, а полученный в результате идентификатор останется доступным для потомка после вызова функции `fork`. Дочерний процесс может передать полученный идентификатор новой программе в качестве аргумента функции `exec`.

2. Клиент и сервер могут договориться об использовании предопределенного ключа, задав его, например, в общем заголовочном файле. После этого сервер может создавать структуру IPC с заданным ключом. Но такое решение также не лишено недостатков — есть вероятность, что в системе уже существует некоторая структура IPC с таким ключом. В этом случае функции создания структуры (`msgget`, `semget` или `shmget`) будут возвращать признак ошибки. Сервер должен правильно обработать ошибочную ситуацию, удалить существующую структуру IPC и попытаться создать ее снова.
3. Клиент и сервер могут договориться об использовании некоторого полного имени файла и идентификатора проекта (идентификатор проекта — это символ с кодом от 0 до 255), на основе которых с помощью функции `ftok` можно получить значение ключа. После этого полученный ключ может использоваться точно так же, как и в предыдущем случае.

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

key_t ftok(const char *path, int id);
```

Возвращает ключ в случае успеха, (`key_t`) -1 — в случае ошибки

Аргумент `path` должен содержать имя существующего файла. В создании ключа используются только 8 младших бит аргумента `id`.

Обычно при создании ключа функцией `ftok` используются значения полей `st_dev` и `st_ino` структуры `stat` (раздел 4.2), соответствующей файлу с заданным именем, в комбинации с идентификатором проекта. Если имена файлов различаются, функция `ftok` обычно возвращает разные ключи. Однако поскольку номера индексных узлов и ключи часто хранятся в виде длинных целых чисел со знаком, при создании ключа может происходить некоторая потеря информации. Это означает, что существует вероятность при использовании разных имен файлов получить одинаковые ключи, если в обоих случаях использовались одинаковые значения идентификатора проекта.

Все три функции `get` (`msgget`, `semget` и `shmget`) имеют одинаковые аргументы: `key` и `flag`. Новая структура IPC создается (обычно сервером) в том случае, если в ар-

гументе *key* передается значение `IPC_PRIVATE` или заданный ключ не соответствует какой-либо существующей структуре IPC данного типа и в аргументе *flag* передается флаг `IPC_CREAT`. Чтобы получить ссылку на существующую очередь (обычно со стороны клиента), в аргументе *key* нужно передать значение, совпадающее с ключом, использовавшимся при создании этой очереди, а флаг `IPC_CREAT` не должен быть установлен.

Обратите внимание, что при использовании флага `IPC_PRIVATE` получить ссылку на существующую очередь невозможно, так как с помощью этого специального ключа всегда создается новая очередь. Чтобы иметь возможность обращаться к существующей очереди, созданной с ключом `IPC_PRIVATE`, мы должны узнать связанный с ней идентификатор и затем использовать его во всех остальных функциях работы с объектом IPC (таких, как `msgsnd` или `msgrcv`) в обход функции `get`.

Если нужно создать новую структуру IPC, а не получить ссылку на существующую, следует в аргументе *flag* вместе с флагом `IPC_CREAT` указать флаг `IPC_EXCL`. В результате, если данная структура IPC уже существует, функция вернет признак ошибки с кодом `EEXIST`. (Очень напоминает правила определения флагов `O_CREAT` и `O_EXCL` в вызове функции `open`.)

15.6.2. Структура прав доступа

С каждой структурой IPC механизм XSI IPC связывает структуру `ipc_perm`. Эта структура определяет права доступа к объекту и его владельца. Она содержит как минимум следующие поля:

```
struct ipc_perm {
    uid_t uid; /* эффективный идентификатор пользователя владельца */
    gid_t gid; /* эффективный идентификатор группы владельца */
    uid_t cuid; /* эффективный идентификатор пользователя создателя */
    gid_t cgid; /* эффективный идентификатор группы создателя */
    mode_t mode; /* режим доступа */
    ...
};
```

Каждая реализация включает в эту структуру дополнительные поля. Полное определение структуры в своей системе вы найдете в заголовочном файле `<sys/ipc.h>`.

Все поля структуры инициализируются при создании структуры IPC. Позднее можно изменить состояние полей `uid`, `gid` и `mode` с помощью функции `msgctl`, `semctl` или `shmctl`. Чтобы иметь возможность изменять эти значения, процесс должен обладать правами создателя структуры или правами суперпользователя. Изменение этих полей похоже на вызов функции `chown` или `chmod` для обычного файла.

Значения поля `mode` напоминают значения, которые мы уже видели в табл. 4.5, за исключением права на выполнение. Кроме того, применительно к очередям сообщений и разделяемой памяти используются термины «право на чтение» и «право на запись», а применительно к семафорам — «право на чтение» и «право на изменение». В табл. 15.2 приводится список различных прав доступа к каждой из структур IPC.

Таблица 15.2. Права доступа XSI IPC

Право доступа	Бит
user-read — доступно пользователю для чтения	0400
user-write — доступно пользователю для записи (изменения)	0200
group-read — доступно группе для чтения	0040
group-write — доступно группе для записи (изменения)	0020
other-read — доступно остальным для чтения	0004
other-write — доступно остальным для записи (изменения)	0002

Некоторые реализации определяют символические константы для каждого бита прав доступа, однако имена этих констант не стандартизированы в Single UNIX Specification.

15.6.3. Конфигурируемые пределы

Мы можем столкнуться со встроенными пределами для всех трех форм XSI IPC. Большинство из них можно изменять при конфигурировании ядра. Мы будем рассматривать эти пределы при обсуждении каждой из трех форм IPC.

Каждая из платформ предоставляет собственный способ получения и изменения конкретных пределов. В FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 имеется команда sysctl, с помощью которой можно просмотреть и изменить конфигурационные параметры ядра. В Solaris 10 для изменения конфигурационных параметров ядра используется команда prctl.

В Linux можно просмотреть пределы, связанные с IPC, запустив команду ipcs -l. В FreeBSD и Mac OS X ей соответствует команда ipcs -t. В Solaris конфигурируемые параметры можно просмотреть, запустив команду sysdef -i.

15.6.4. Преимущества и недостатки

Фундаментальная проблема всех форм XSI IPC заключается в том, что структуры IPC привязаны к системе в целом, а не к конкретному процессу и не имеют счетчика ссылок. Например, если мы создадим очередь сообщений, поместим в нее некоторое сообщение и завершим процесс, эта очередь не будет удалена. Сообщение останется в системе, пока не будет прочитано или удалено каким-либо процессом с помощью функции msgrecv или msgctl, командой ipcrm(1) или пока система не будет перезагружена. Сравните это с неименованными каналами, которые удаляются автоматически, когда завершается последний процесс, имеющий ссылку на этот канал. В случае с FIFO имя канала остается в системе, пока явно не будет удалено, но данные удаляются из канала FIFO автоматически, когда завершается последний процесс, имеющий ссылку на этот канал.

Другая проблема, связанная с механизмами XSI IPC, заключается в том, что они не имеют имен в файловой системе. Мы не можем получить к ним доступ или изменить их свойства с помощью функций, описанных в главах 3 и 4. Для поддержки этих объектов IPC в ядро была добавлена почти дюжина новых системных вы-

зовов (`msgget`, `semop`, `shmat` и др.). Мы не можем получить список существующих объектов IPC с помощью команды `ls`, удалить их с помощью `rm` и изменить права доступа к ним с помощью `chmod`. Вместо них должны использоваться две новые команды — `ipcs(1)` и `ipcfrm(1)`.

Так как эти формы IPC не используют файловые дескрипторы, к ним нельзя применить функции мультиплексирования ввода/вывода (`select` и `poll`). Это осложняет одновременную работу с более чем одной структурой IPC или использование какой-либо из этих структур совместно с файлами или устройствами ввода/вывода. Например, мы не можем организовать на стороне сервера ожидание сообщения, которое может поступить по одной из двух очередей, не применяя ту или иную форму цикла активного ожидания.

Краткий обзор системы диалоговой обработки запросов, построенной на основе System V IPC, приводится в [Andrade, Carges, and Kovach, 1989]. Авторы этой книги утверждают, что принцип именования, используемый System V IPC (идентификаторы), является преимуществом, а не недостатком, как мы говорили выше, потому что идентификаторы позволяют процессам посыпать сообщения в очередь, используя всего одну функцию (`msgsnd`), тогда как другие формы IPC требуют вызова трех функций: `open`, `write` и `close`. Однако в действительности это не так: чтобы избежать использования ключа и вызова `msgget`, клиенты должны каким-то способом получить идентификатор очереди на сервере. Значение идентификатора, присвоенного конкретной очереди, зависит от количества существующих очередей сообщений и от того, сколько раз создавались новые очереди с момента последней перезагрузки ядра. Это динамическое значение — его невозможно предугадать или предопределить в заголовочном файле. Как мы уже говорили в разделе 15.6.1, в простейшем случае сервер должен записать идентификатор очереди в файл, который может быть прочитан клиентами.

Среди других преимуществ очередей сообщений, на которые указывают авторы упоминавшейся выше книги, можно назвать надежность, управление ходом выполнения, ориентированность на отдельные записи и возможность извлекать сообщения не только в порядке их помещения в очередь. В табл. 15.3 приводятся некоторые сравнительные характеристики различных форм IPC.

Таблица 15.3. Сравнение некоторых характеристик различных форм IPC

Тип IPC	Ориентированность на установление соединения	Надежность	Управление ходом выполнения	Записи	Типы сообщений или свойства
Очереди сообщений	Да	Да	Да	Да	Да
STREAMS	Да	Да	Да	Да	Да
Сокеты домена UNIX, ориентированные на потоки	Да	Да	Да	Нет	Нет
Сокеты домена UNIX, ориентированные на дейтаграммы	Нет	Да	Нет	Да	Нет
FIFO (не STREAMS)	Да	Да	Да	Нет	Нет

(Сокеты, ориентированные на потоки и на дейтаграммы, будут описаны в главе 16. Сокеты домена UNIX будут описаны в разделе 17.3.) Под понятием «ориентированность на установление соединения» подразумевается необходимость предварительного вызова некоторой функции открытия механизма IPC. Как было сказано ранее, мы считаем, что очереди сообщений ориентированы на установление соединения, поскольку для получения идентификатора очереди должны быть предварительно выполнены некоторые действия. Так как область применения всех этих механизмов IPC ограничивается одним хостом, их можно отнести к разряду надежных. Возможность потери сообщений возникает, если сообщения передаются через сеть. Под «управлением ходом выполнения» подразумевается, что передающий процесс может быть приостановлен, если в приемном буфере недостаточно места или принимающий процесс в данное время не может принять сообщение. Когда появится возможность принять сообщение от передающего процесса, его работа будет возобновлена автоматически.

Одна из характеристик, которую мы не упомянули в табл. 15.3, — это возможность автоматически создавать уникальное соединение между сервером и каждым из клиентов. В главе 17 мы увидим, что сокеты, ориентированные на потоки, поддерживают такую возможность. В следующих трех разделах подробно описываются все три формы XSI IPC.

15.7. Очереди сообщений

Очередь сообщений — это связный список сообщений, который хранится в памяти ядра и идентифицируется идентификатором очереди сообщений. Далее мы будем называть очередь сообщений просто *очередью*, а ее идентификатор — *идентификатором очереди*.

Стандарт Single UNIX Specification включает определение альтернативной реализации механизма очередей сообщений, унаследованное из расширений реального времени POSIX. Но мы не будем рассматривать его в этой книге.

Создание новой очереди или открытие существующей производится с помощью функции `msgget`. Новые сообщения добавляются в конец очереди функцией `msgsnd`. Каждое сообщение содержит тип (положительное длинное целое число), неотрицательное значение длины и собственно данные (объем которых определяется длиной сообщения). Все эти значения передаются функции `msgsnd` при добавлении сообщения в очередь. Сообщения могут извлекаться из очереди не только в порядке «первым пришел — первым ушел», но и на основе типа сообщения.

С каждой очередью связывается структура `msgid_desc`:

```

time_t          msg_stime; /* время последнего вызова msgsnd()*/
time_t          msg_rtime; /* время последнего вызова msgrcv()*/
time_t          msg_ctime; /* время последнего изменения */
...
};

};

```

Эта структура определяет текущее состояние очереди. Поля структуры, перечисленные здесь, определяются стандартом Single UNIX Specification. Реализации, как правило, включают в эту структуру дополнительные поля, которые не включаются в стандарт.

В табл. 15.4 перечислены системные пределы, имеющие отношение к очередям сообщений. Значение «Производное» указывается там, где предел является производным от других пределов. Например, максимальное количество сообщений в Linux зависит от максимального количества очередей и максимального объема данных, которые могут быть помещены в очередь. Максимальное количество очередей, в свою очередь, зависит от объема доступной оперативной памяти в системе. Обратите внимание, что максимальный размер очереди в байтах, как следствие, ограничивает максимальный размер сообщения, которое можно поместить в очередь.

Таблица 15.4. Системные пределы, связанные с очередями сообщений

Описание	Типичные значения			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Максимальный размер сообщения	16 384	8192	16 384	Производ- ное
Максимальный размер очереди в байтах (то есть сумма всех сообщений в очереди)	2048	16 384	2048	65 536
Максимальное количество очередей сообщений в системе	40	Производ- ное	40	128
Максимальное количество сообщений в системе	40	Производ- ное	40	8192

Обычно при работе с очередями прежде всего вызывается функция `msgget`, которая открывает существующую или создает новую очередь.

```

#include <sys/msg.h>

int msgget(key_t key, int flag);

```

Возвращает идентификатор очереди в случае успеха,
–1 — в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новая очередь, а когда открывается существующая. При создании новой очереди инициализируются следующие поля структуры `msqid_ds`.

- Структура `msqid_ds` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе `flag`. Значения для каждого конкретного права доступа приведены в табл. 15.2.
- В поля `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime` и `msg_rtime` записывается значение 0.
- В поле `msg_ctime` записывается текущее время.
- В поле `msg_qbytes` записывается значение соответствующего системного предела.

В случае успеха `msgget` возвращает неотрицательный идентификатор очереди. Это значение может использоваться в других трех функциях для работы с очередями сообщений.

Функция `msgctl` производит различные операции над очередью. Она и аналогичные ей функции для семафоров и разделяемой памяти (`semctl` и `shmctl`) являются аналогами функции `ioctl` для механизмов XSI IPC.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Возвращает 0 в случае успеха, -1 – в случае ошибки

Аргумент `cmd` представляет код операции над очередью, определяемой аргументом `msqid`.

IPC_STAT Получить структуру `msqid_ds` данной очереди и сохранить ее по адресу `buf`.

IPC_SET Скопировать из `buf` в структуру `msqid_ds`, которая связана с очередью, значения следующих полей: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` и `msg_qbytes`. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением `msg_perm.cuid` или `msg_perm.uid` либо если процесс обладает привилегиями суперпользователя. Значение поля `msg_qbytes` может увеличить только суперпользователь.

IPC_RMID Удалить очередь сообщений и все данные, которые в ней имеются. Удаление очереди происходит немедленно. Все процессы, которые продолжают использовать очередь, получат код ошибки `EIDRM` при первой же попытке обращения к ней. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением `msg_perm.cuid` или `msg_perm.uid` либо если процесс обладает привилегиями суперпользователя.

Позже мы увидим, что те же команды (`IPC_STAT`, `IPC_SET` и `IPC_RMID`) используются также для управления семафорами и сегментами разделяемой памяти.

Помещение данных в очередь сообщений производится вызовом функции `msgsnd`.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Возвращает 0 в случае успеха, -1 – в случае ошибки

Как уже упоминалось, каждое сообщение состоит из значения, определяющего тип сообщения, поля длины сообщения (*nbytes*) и собственно данных (объем которых равен значению поля длины сообщения). Сообщения всегда помещаются в конец очереди.

Через аргумент *ptr* передается указатель на длинное целое со знаком, которое содержит положительное значение, определяющее тип сообщения, за которым сразу же размещаются данные сообщения. (Считается, что сообщение не имеет данных, если в аргументе *nbytes* передается значение 0.) Если максимальный размер отправляемых сообщений составляет 512 байт, можно определить следующую структуру:

```
struct mymesg {
    long mtype;      /* тип сообщения – положительное число */
    char mtext[512]; /* данные сообщения, объем которых равен nbytes */
};
```

В этом случае в аргументе *ptr* можно передать указатель на структуру *mymesg*. Тип сообщения может использоваться принимающим процессом для извлечения сообщений в порядке, отличном от порядка помещения сообщений в очередь.

*Некоторые платформы поддерживают как 32-разрядную, так и 64-разрядную реализацию. Это сказывается на размере длинных целых чисел и указателей. Например, в 64-разрядной реализации Solaris на аппаратной архитектуре SPARC допускается существование 32- и 64-разрядных приложений. Если 32-разрядное приложение попытается передать приведенную выше структуру 64-разрядному приложению через неименованный канал или сокет, возникнут проблемы, поскольку размер длинного целого для 32-разрядных приложений составляет 4 байта, а для 64-разрядных приложений – 8 байт. Это означает, что 32-разрядное приложение будет считать, что поле *mtext* отстоит на 4 байта от начала структуры, тогда как 64-разрядное приложение – что оно отстоит от начала структуры на 8 байт. В этой ситуации часть поля *mtype* 64-разрядного приложения будет расценена 32-разрядным приложением как часть поля *mtext*, а первые 4 байта поля *mtext* 32-разрядного приложения будут расценены 64-разрядным приложением как часть поля *mtype*.*

Эта проблема отсутствует в механизме очередей сообщений XSI. Solaris реализует 32- и 64-разрядные версии системных вызовов IPC с различными точками входа. Системные вызовы заранее предусматривают возможность корректного обмена данными между 32- и 64-разрядными приложениями и правильно интерпретируют размер поля типа сообщения. Единственная проблема, которая может возникнуть, – потеря информации о типе, когда 64-разрядное приложение посылает сообщение 32-разрядному приложению, поскольку 8-байтовое поле типа сообщения нельзя без потерь уместить в 4-байтовое поле, используемое в 32-разрядных приложениях. В этом случае 32-разрядное приложение будет получать усеченное значение типа.

В аргументе *flag* можно указать значение **IPC_NOWAIT**. Это аналог флага **O_NONBLOCK**, который используется для определения неблокирующего режима операций файлового ввода/вывода (раздел 14.2). Если очередь сообщений заполнена до отказа (количество сообщений в очереди или общее количество байтов в очереди достигло системного предела), при указании флага **IPC_NOWAIT** функция **msgsnd** будет сразу же возвращать управление с кодом ошибки **EAGAIN**. Если флаг **IPC_NOWAIT**

не указан, процесс заблокируется, пока в очереди не освободится место, пока очередь не будет удалена из системы или пока не будет перехвачен какой-либо сигнал и обработчик вернет управление. Во втором случае будет возвращен код ошибки `EIDRM` (`identifier removed` — идентификатор удален), а в последнем — код ошибки `EINTR`.

Обратите внимание, как неудачно обрабатывается ситуация удаления очереди. Поскольку для очередей сообщений не поддерживается счетчик ссылок (как для открытых файлов), удаление очереди просто приводит к появлению ошибок при последующих попытках выполнить какие-либо действия с ней. В случае семафоров ситуация удаления обслуживается точно так же. При удалении файла, напротив, его содержимое остается в неприкосновенности, пока не будет закрыт последний дескриптор этого файла.

В случае успеха функция `msgsnd` изменяет содержимое структуры `msqid_ds`, ассоциированной с заданной очередью: в поле `msg_lspid` заносится идентификатор вызывающего процесса, в поле `msg_stime` — время вызова, а значение поля `msg_qnum` (количество сообщений в очереди) увеличивается на единицу.

Выборка сообщений из очереди производится функцией `msgrcv`.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Возвращает блок данных из сообщения в случае успеха,
-1 — в случае ошибки

Как и в функции `msgsnd`, аргумент `ptr` содержит адрес, по которому будет сохранено длинное целое число — тип сообщения, за которым сразу же следует буфер для размещения данных сообщения. Если размер полученного сообщения превышает значение `nbytes` и при этом в аргументе `flag` установлен бит `MSG_NOERROR`, сообщение будет усечено до размера `nbytes`. (В этом случае приложение никогда не узнает, что сообщение было усечено.) Если размер полученного сообщения превышает значение `nbytes` и в аргументе `flag` не установлен бит `MSG_NOERROR`, вместо сообщения будет возвращен признак ошибки с кодом `E2BIG` (сообщение при этом останется в очереди).

Аргумент `type` позволяет определить желаемый тип сообщения.

`type == 0`

Будет возвращено первое сообщение в очереди.

`type > 0`

Будет возвращено первое сообщение, имеющее заданный тип.

`type < 0`

Будет возвращено первое сообщение, значение типа которого меньше или равно абсолютному значению аргумента `type`.

Ненулевое значение аргумента `type` используется, когда необходимо извлекать сообщения из очереди не в порядке их помещения в очередь. Например, значение `type`

может указывать приоритет сообщения. Еще один вариант использования поля *type* — клиент может передавать в нем идентификатор своего процесса, если сервер использует единственную очередь для обмена данными со всеми клиентами (разумеется, если идентификатор процесса умещается в длинное целое со знаком).

В аргументе *flag* можно указать значение `IPC_NOWAIT`, чтобы выполнить операцию в неблокирующем режиме. При наличии этого флага, если в очереди отсутствуют сообщения заданного типа, функция `msgrecv` будет возвращать значение `-1` с кодом `ENOMSG` в переменной *errno*. Если флаг `IPC_NOWAIT` не указан, операция будет заблокирована, пока не станет доступно сообщение указанного типа, пока очередь не будет удалена (`msgrecv` вернет `-1` и код ошибки `EIDRM` в переменной *errno*) или пока не будет перехвачен сигнал и обработчик сигнала вернет управление (`msgrecv` вернет `-1` и код ошибки `EINTR` в переменной *errno*).

В случае успешного завершения функции `msgrecv` ядро обновит содержимое структуры `msqid_ds`, ассоциированной с данной очередью: в поле `msg_lrpid` будет записан идентификатор вызывающего процесса, в поле `msg_rtime` — время вызова, а значение поля `msg_qnum` уменьшится на единицу.

Пример — сравнение производительности очередей сообщений и дуплексных каналов

Для организации двустороннего обмена между клиентом и сервером можно использовать либо очереди сообщений, либо дуплексные каналы. (В табл. 15.1 мы уже упоминали, что дуплексные каналы могут быть реализованы на основе сокетов домена UNIX (раздел 17.2), хотя на некоторых платформах имеется поддержка механизма дуплексных каналов на основе функции `pipe`.)

В табл. 15.5 приводятся результаты сравнения производительности в Solaris трех механизмов: очередей сообщений, дуплексных каналов (STREAMS) и сокетов домена UNIX. В процессе измерений тестовая программа создавала канал IPC, вызывала функцию `fork`, а затем родительский процесс передавал порядка 200 Мбайт данных дочернему процессу. Всего было отправлено 100 000 сообщений по 2000 байт в каждом. Время приводится в секундах.

Таблица 15.5. Результаты сравнения производительности альтернативных форм IPC в Solaris

Механизм IPC	Пользовательское время	Системное время	Общее время
Очередь сообщений	0,58	4,16	5,09
Дуплексный канал	0,61	4,30	5,24
Сокет домена UNIX	0,59	5,58	7,49

Результаты показывают, что очереди сообщений, которые изначально задумывались как скоростной механизм обмена данными, не дают значительного выигрыша по сравнению с другими IPC. (Когда были реализованы очереди сообщений, единственной доступной альтернативной формой IPC были полудуплексные каналы.) С учетом проблем, связанных с очередями сообщений (раздел 15.6.4), можно сделать вывод, что их не следует использовать в новых приложениях.

15.8. Семафоры

Семафоры не похожи на формы межпроцессных взаимодействий, которые мы уже описали (именованные и неименованные каналы и очереди сообщений). Семафор — это счетчик, который применяется для доступа к данным, совместно используемым несколькими процессами.

Стандарт Single UNIX Specification включает определение альтернативного набора функций для работы с семафорами, первоначально входивших в расширения реального времени. Эти интерфейсы будут обсуждаться в разделе 15.10.

Чтобы получить доступ к ресурсу, находящемуся в совместном использовании, процесс должен:

1. Проверить состояние семафора, который регулирует доступ к этому ресурсу.
2. Если семафор имеет положительное значение, процесс может обратиться к ресурсу. В этом случае процесс уменьшает значение семафора на 1, указывая тем самым, что он использовал единицу ресурса.
3. Иначе, если семафор имеет значение 0, процесс приостанавливается, пока значение семафора не станет больше 0. После этого процесс возобновит работу и вернется к шагу 1.

По окончании работы с ресурсом, доступ к которому регулируется семафором, значение семафора будет увеличено на 1. Если в этот момент какой-либо другой процесс находится в ожидании освобождения семафора, он возобновит свою работу.

Для корректной работы семафоров необходимо, чтобы проверка состояния семафора и его уменьшение выполнялись атомарно. По этой причине семафоры обычно реализуются внутри ядра.

Чаще всего используется разновидность семафоров, которая получила название *двоичный семафор*. Семафоры этого типа регулируют доступ к единственному ресурсу и инициализируются значением 1. Но вообще семафоры могут инициализироваться любым положительным значением, которое определяет, сколько единиц ресурса может одновременно использоваться несколькими процессами.

К сожалению, на практике семафоры XSI имеют более сложную организацию. Эта сложность обусловлена следующими тремя особенностями.

1. Семафор — это не просто одиночное неотрицательное значение. Чтобы определить семафор, необходимо определить набор из одного или более семафоров. Количество семафоров в наборе задается при создании этого набора.
2. Создание набора семафоров (`semget`) происходит независимо от его инициализации (`semctl`). Это очень серьезный недостаток, поскольку невозможно атомарно создать новый набор семафоров и инициализировать их значения.
3. Поскольку семафоры, как и все формы XSI IPC, продолжают существовать даже после завершения процессов, их использующих, необходимо предусматривать в программах алгоритмы освобождения размещенных ранее наборов семафоров. В этом может помочь операция *undo*, которая будет описана немного позже.

Каждому набору семафоров ядро ставит в соответствие структуру `semid_ds`:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* раздел 15.6.2 */
    unsigned short sem_nsems; /* количество семафоров в наборе */
    time_t         sem_otime; /* время последнего вызова функции semop() */
    time_t         sem_ctime; /* время последнего изменения */
    ...
};
```

Указанные поля структуры определены стандартом Single UNIX Specification, но реализации могут дополнять структуру `semid_ds` собственными полями.

Каждый из семафоров представлен в наборе анонимной структурой, которая содержит как минимум следующие поля:

```
struct {
    unsigned short semval; /* значение семафора, всегда >= 0 */
    pid_t        sempid; /* идентификатор процесса, выполнившего */
    /* последнюю операцию */
    unsigned short semncnt; /* количество процессов, */
    /* ожидающих выполнения условия semval>curval */
    unsigned short semzcnt; /* количество процессов, */
    /* ожидающих выполнения условия semval==0 */
    ...
};
```

В табл. 15.6 перечислены системные пределы, которые имеют отношение к наборам семафоров.

Таблица 15.6. Системные пределы, которые имеют отношение к наборам семафоров

Описание	Типичные значения			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Максимальное значение любого семафора	32 767	32 767	32 767	65 535
Максимальное значение корректировки (adjust-on-exit) для любого семафора (это значение добавляется к семафору при завершении процесса)	16 384	32 767	16 384	32 767
Максимальное количество наборов семафоров в системе	10	128	87 381	128
Максимальное количество семафоров в системе	60	32 000	87 381	Производное
Максимальное количество семафоров в наборе	60	250	87 381	512
Максимальное количество структур undo в системе	30	32 000	87 381	Производное
Максимальное количество записей в структуре undo	10	Не ограничено	10	Производное
Максимальное количество операций, выполняемых одним вызовом <code>semop</code>	100	32	5	512

Обычно при работе с семафорами прежде всего вызывается функция `semget`, которая возвращает идентификатор набора семафоров.

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

Возвращает идентификатор набора семафоров в случае успеха,
-1 — в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый набор семафоров, а когда открывается существующий. При создании нового набора инициализируются следующие поля структуры `semid_ds`:

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе `flag`. Значения для каждого конкретного права доступа приведены в табл. 15.2.
- В поле `sem_otime` записывается 0.
- В поле `sem_ctime` записывается текущее время.
- В поле `sem_nsems` записывается значение аргумента `nsems`.

Количество семафоров в наборе определяется аргументом `nsems`. Если создается новый набор семафоров (обычно на стороне сервера), мы должны указать значение `nsems`. Если открывается существующий набор семафоров, допускается в аргументе `nsems` передавать 0.

Операции над набором семафоров выполняются с помощью функции `semctl`.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
```

Возвращаемые значения описаны ниже

Четвертый аргумент функции является необязательным и зависит от выполняемой команды; если он присутствует, то представляет собой объединение `semun` различных аргументов команд:

```
union semun {
    int          val;      /* для SETVAL */
    struct semid_ds *buf;   /* для IPC_STAT и IPC_SET */
    unsigned short *array; /* для GETALL и SETALL */
};
```

Обратите внимание, что четвертый необязательный аргумент является объединением, а не указателем на объединение.

Как правило, объединение `semun` должно определяться в самом приложении. Однако в FreeBSD 8.0 это определение находится в заголовочном файле `<sys/sem.h>`.

Аргумент *cmd* определяет одну из следующих десяти операций, которые могут выполняться над набором семафоров, представленным аргументом *semid*. Пять команд, которые используются для работы с отдельным семафором, получают номер семафора в наборе из аргумента *semnum*. Значение *semnum* должно находиться в пределах от 0 до *nsems* – 1 включительно.

IPC_STAT Получить структуру *semid_ds*, которая соответствует заданному набору семафоров, и сохранить ее по адресу *arg.buf*.

IPC_SET Установить значения полей *sem_perm.uid*, *sem_perm.gid* и *sem_perm.mode* в соответствии со значениями этих же полей в структуре, на которую указывает *arg.buf*. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением *sem_perm.cuid* или *sem_perm.uid* либо если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить набор семафоров. Удаление происходит немедленно. Все процессы, которые продолжают использовать набор семафоров, получат код ошибки *EIDRM* при первой же попытке обращения к нему. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением *sem_perm.cuid* или *sem_perm.uid* либо если процесс обладает привилегиями суперпользователя.

GETVAL Вернуть значение поля *semval* для семафора с номером *semnum*.

SETVAL Установить значение поля *semval* для семафора с номером *semnum*. Значение определяется в *arg.val*.

GETPID Вернуть значение поля *sempid* для семафора с номером *semnum*.

GETNCNT Вернуть значение поля *semncnt* для семафора с номером *semnum*.

GETZNCNT Вернуть значение поля *semzcnt* для семафора с номером *semnum*.

GETALL Вернуть значения всех семафоров в наборе. Значения сохраняются в массиве, на который указывает *arg.array*.

SETALL Установить значения всех семафоров в наборе. Значения берутся из массива, на который указывает *arg.array*.

В случае всех команд **GET**, за исключением **GETALL**, функция возвращает соответствующее значение. Для остальных команд возвращается 0. В случае ошибки **semctl** возвращает –1 и сохраняет в переменной *errno* код ошибки.

Функция **semop** выполняет сразу несколько операций над набором семафоров.

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Возвращает 0 в случае успеха, –1 – в случае ошибки

Аргумент *semoparray* – это массив указателей на операции с семафорами, каждая из которых представлена в виде структуры *sembuf*:

```
struct sembuf{
    unsigned short sem_num; /* количество семафоров в наборе */
    /* (0, 1, ..., nsems1) */
```

```

short      sem_op; /* операция (<0, 0 или >0) */
short      sem_flg; /* IPC_NOWAIT, SEM_UNDO */
};

```

Аргумент *nops* определяет количество операций (элементов) в массиве.

Операция, выполняемая над каждым семафором из набора, определяется значением *sem_op*. Это значение может быть отрицательным, положительным или равным нулю. (Ниже мы будем упоминать флаг «undo». Этот флаг соответствует биту *SEM_UNDO* в поле *sem_flg*.)

1. Самый простой случай — положительное значение поля *sem_op*. Он соответствует случаю, когда процесс освобождает занятые ресурсы. Значение *sem_op* добавляется к значению семафора. Если указан флаг *SEM_UNDO*, это значение также вычитается из значения корректировки (adjust-on-exit) процесса.
2. Если значение *sem_op* отрицательное, это означает, что процесс желает получить ресурс, доступ к которому регулируется семафором.

Если значение семафора больше или равно абсолютному значению *sem_op* (ресурс доступен), абсолютное значение *sem_op* вычитается из значения семафора. Это гарантирует, что значение семафора ни при каких обстоятельствах не будет меньше нуля. Если указан флаг *SEM_UNDO*, абсолютное значение *sem_op* также прибавляется к величине корректировки семафора для данного процесса.

Если значение семафора меньше абсолютного значения *sem_op* (ресурс недоступен), вступают в силу следующие условия:

- 1) если указан флаг *IPC_NOWAIT*, функция *semop* возвращает управление с кодом ошибки *EAGAIN*;
 - 2) если флаг *IPC_NOWAIT* не указан, для данного семафора увеличивается значение *semncnt*, а выполнение вызывающего процесса приостанавливается, пока не будет соблюдено одно из следующих условий:
 - а) значение семафора стало больше или равно абсолютному значению *sem_op* (то есть другой процесс освободил требуемый ресурс). Значение *semncnt* для этого семафора уменьшается (поскольку ожидание освобождения семафора можно считать законченным), и абсолютное значение *sem_op* вычитается из значения семафора. Если был указан флаг *SEM_UNDO*, абсолютное значение *sem_op* также добавляется к величине корректировки семафора;
 - б) семафор был удален из системы. В этом случае функция *semop* вернет признак ошибки с кодом *EIDRM*;
 - в) процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение *semncnt* уменьшается (поскольку вызывающий процесс больше не ждет освобождения ресурса), и функция *semop* вернет признак ошибки с кодом *EINTR*.
 3. Если значение *sem_op* равно нулю, это означает, что процесс желает дождаться момента, когда значение семафора достигнет нуля.
- Если значение семафора уже равно нулю, функция сразу же вернет управление.

Если значение семафора больше нуля, тогда вступают в силу следующие условия:

- 1) если указан флаг `IPC_NOWAIT`, функция `semop` вернет управление с кодом ошибки `EAGAIN`;
- 2) если флаг `IPC_NOWAIT` не указан, для данного семафора увеличивается значение `semzcnt` и выполнение вызывающего процесса приостанавливается, пока не будет соблюдено одно из следующих условий:
 - a) значение семафора стало равным нулю. В этом случае значение `semzcnt` уменьшается (поскольку ожидание освобождения семафора можно считать законченным);
 - b) семафор был удален из системы. В этом случае функция `semop` вернет признак ошибки с кодом `EIDRM`;
 - b) процессом был перехвачен сигнал, и обработчик сигнала вернул управление. В этом случае значение `semzcnt` уменьшается (поскольку вызывающий процесс прекращает ожидание) и функция `semop` вернет признак ошибки с кодом `EINTR`.

Функция `semop` выполняет все операции атомарно — либо будут выполнены все запрошенные действия, либо ни одно из них не будет выполнено.

Корректировка семафора по завершении

Как уже упоминалось выше, завершение процесса в то время, когда он захватил какие-либо ресурсы посредством семафора, может потребовать значительных усилий. Всякий раз, когда мы устанавливаем для операции над семафором флаг `SEM_UNDO` (значение `sem_op` меньше нуля), ядро запоминает, как много ресурсов было захвачено процессом с помощью конкретного семафора (абсолютное значение `sem_op`). Когда процесс завершается, добровольно или принудительно, ядро проверяет, имеет ли процесс какие-либо невыполненные корректировки семафоров, и если таковые имеются, корректирует значения соответствующих семафоров.

Когда начальное значение семафора устанавливается функцией `semctl` с помощью команды `SETVAL` или `SETALL`, значение корректировки этого семафора во всех процессах сбрасывается в 0.

Пример — сравнение производительности семафоров, блокировок записей в файлах и мьютексов

При совместном использовании одного ресурса несколькими процессами порядок доступа к ресурсу может регулироваться с помощью семафора или блокировок записей в файле. Было бы интересно сравнить производительность этих двух методов.

В случае семафоров мы создали набор с единственным семафором. Он инициализируется значением 1. Чтобы захватить ресурс, процесс должен вызвать `semop` со значением `sem_op`, равным -1. Чтобы освободить ресурс, процесс должен вызвать `semop` со значением `sem_op`, равным +1. Кроме того, для каждой операции

мы указывали флаг `SEM_UNDO` на случай завершения процесса, который не успел освободить ресурс.

В случае с блокировками мы создали пустой файл и использовали его первый байт (который не обязательно должен существовать) для установки блокировки. Чтобы захватить ресурс, процесс должен установить блокировку для записи на этот байт, чтобы освободить ресурс – снять блокировку с байта. Одно из свойств блокировок заключается в том, что по завершении процесса, который удерживает блокировку, она будет автоматически снята ядром.

В случае с мьютексом необходимо, чтобы оба процесса отобразили один и тот же файл в свои адресные пространства и инициализировали мьютекс для защиты доступа к данным в файле с одним и тем же смещением, указав флаг `PTHREAD_PROCESS_SHARED`. Чтобы захватить ресурс, процесс должен запереть мьютекс, чтобы освободить ресурс – отпереть мьютекс. Если процесс завершится, не отперев мьютекс, восстановить нормальное использование ресурса будет сложно, если не использовать надежные мьютексы (вспомните функцию `pthread_mutex_consistent`, обсуждавшуюся в разделе 12.4.1).

В табл. 15.7 показано время выполнения этих трех методов блокировок в Linux. В каждом случае три тестовых процесса захватывали и освобождали ресурс 1 000 000 раз. Цифры в табл. 15.7 представляют общее время для всех трех процессов в секундах.

Таблица 15.7. Производительность трех альтернативных механизмов блокировки в Linux

Механизм IPC	Пользовательское время	Системное время	Общее время
Семафоры с флагом <code>SEM_UNDO</code>	0,50	6,08	7,55
Рекомендательная блокировка записи в файле	0,51	9,06	4,38
Мьютекс в разделяемой памяти	0,21	0,40	0,25

В Linux блокировки записей действуют быстрее, чем семафоры, но производительность мьютексов в разделяемой памяти значительно превосходит и семафоры, и блокировки записей. Если требуется урегулировать доступ к единственному ресурсу и нет необходимости во всех замысловатых особенностях семафоров XSI, предпочтительнее использовать блокировки записей, так как они проще в использовании, быстрее (на данной платформе) и система сама заботится о блокировках, которые не были сняты по завершении процесса. Хотя механизм мьютексов в разделяемой памяти оказался самым быстрым на этой платформе, предпочтение все же следует отдавать блокировкам записей, если производительность не ставится во главу угла. На то есть две причины. Во-первых, при использовании мьютекса в разделяемой памяти намного сложнее восстановить нормальное использование ресурса после неожиданного завершения процесса, удерживавшего мьютекс. Во-вторых, атрибут `process-shared` мьютексов пока поддерживается не

всеми системами. В прежних версиях стандарта Single UNIX Specification он был необязательным. Хотя он все еще остается необязательным в SUSv4, все XSI-совместимые реализации обязаны поддерживать его.

Из четырех платформ, рассматриваемых в этой книге, только Linux 3.2.0 и Solaris 10 поддерживают атрибут process-shared мьютексов.

15.9. Разделяемая память

Механизм разделяемой памяти позволяет двум и более процессам совместно использовать одну и ту же область памяти. Это самый скоростной вид IPC, поскольку при его использовании данные не нужно лишний раз копировать между клиентом и сервером. Единственный сложный момент при работе с разделяемой памятью — синхронизация доступа к ней. Если сервер размещает некоторые данные в области разделяемой памяти, клиент не должен пытаться читать данные, пока сервер не выполнит всю работу. Часто для синхронизации используются семафоры. (Но, как мы видели в конце предыдущего раздела, также могут использоваться блокировки записей в файлах.)

Стандарт Single UNIX Specification включает определение альтернативного набора функций для доступа к разделяемой памяти, первоначально входивших в расширения реального времени. Но в этой книге мы не будем рассматривать их.

Мы уже видели одну из форм организации разделяемой памяти — отображение одного и того же файла в адресные пространства нескольких процессов. Разделяемая память XSI отличается от файлов, отображаемых в память, тем, что не связана ни с какими файлами. Сегменты разделяемой памяти XSI являются анонимными. Каждому сегменту разделяемой памяти ядро ставит в соответствие структуру, содержащую как минимум следующий набор полей:

```
struct shmid_ds {
    struct ipc_perm  shm_perm;   /* раздел 15.6.2 */
    size_t          shm_segsz;  /* размер сегмента в байтах */
    pid_t           shm_lpid;   /* идентификатор процесса, последним вызвавшего
                                shmop() */
    pid_t           shm_cpid;   /* идентификатор процесса-создателя */
    shmat_t         shm_nattch; /* текущее количество подключений */
    time_t          shm_atime;  /* время последнего подключения */
    time_t          shm_dtime;  /* время последнего отключения */
    time_t          shm_ctime;  /* время последнего изменения */
    ...
};
```

(Реализации при необходимости могут добавлять собственные поля в эту структуру.)

Тип `shmat_t` определен как беззнаковое целое, по меньшей мере — `unsigned short`. В табл. 15.8 перечислены системные пределы, которые имеют отношение к разделяемой памяти.

Таблица 15.8. Системные пределы, имеющие отношение к разделяемой памяти

Описание	Типичные значения			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Максимальный размер сегмента разделяемой памяти в байтах	33 554 432	32 768	4 194 304	Производное
Минимальный размер сегмента разделяемой памяти в байтах	1	1	1	1
Максимальное количество сегментов разделяемой памяти в системе	192	4096	32	128
Максимальное количество сегментов разделяемой памяти для процесса	128	4096	8	128

Обычно при работе с разделяемой памятью сначала вызывается функция `shmget`, которая возвращает идентификатор сегмента разделяемой памяти.

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

Возвращает идентификатор сегмента разделяемой памяти в случае успеха, `-1` — в случае ошибки

В разделе 15.6.1 мы рассмотрели правила преобразования ключа в идентификатор и обсудили вопрос, когда создается новый сегмент, а когда открывается существующий. При создании нового сегмента инициализируются следующие поля структуры `shmid_ds`.

- Структура `ipc_perm` инициализируется, как описано в разделе 15.6.2. Поле `mode` устанавливается в соответствии со значениями битов прав доступа в аргументе `flag`. Значения для каждого конкретного права доступа приводятся в табл. 15.2.
- В поля `shm_lpid`, `shm_nattach`, `shm_atime` и `shm_dtime` записывается 0.
- В поле `shm_ctime` записывается текущее время.
- В поле `shm_segsz` записывается значение аргумента `size`.

Аргумент `size` определяет размер сегмента разделяемой памяти в байтах. Обычно реализации округляют это число, чтобы оно было кратно размеру страницы памяти в системе, но если приложение определяет в аргументе `size` число, не кратное размеру страницы памяти, остаток последней страницы будет недоступен для использования. Если необходимо создать новый сегмент разделяемой памяти (обычно на стороне сервера), его размер следует определить в аргументе `size`. Если нужно лишь получить ссылку на существующий сегмент (в случае клиента), в аргументе `size` можно передать 0. Когда создается новый сегмент, его содержимое очищается.

Функция `shmctl` выполняет различные операции над сегментом разделяемой памяти.

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Аргумент *cmd* представляет код операции, которая должна быть выполнена над сегментом, определяемым аргументом *shmid*.

IPC_STAT Получить структуру *shmid_ds* для данного сегмента памяти и сохранить ее по адресу *buf*.

IPC_SET Скопировать значения полей *shm_perm.uid*, *shm_perm.gid* и *shm_perm.mode* из *buf* в структуру *shmid_ds*, связанную с сегментом разделяемой памяти. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением *shm_perm.cuid* или *shm_perm.uid* либо если процесс обладает привилегиями суперпользователя.

IPC_RMID Удалить сегмент разделяемой памяти. Поскольку для сегментов разделяемой памяти поддерживается счетчик ссылок (поле *shm_nattach* в структуре *shmid_ds*), сегмент не будет удален, пока последний использующий его процесс не завершится или не отсоединит этот сегмент. Независимо от того, находится ли сегмент в использовании, его идентификатор немедленно удаляется из системы, что предотвращает возможность новых подключений сегмента вызовом функции *shmat*. Процесс сможет выполнить эту команду, только если его эффективный идентификатор пользователя совпадает со значением *shm_perm.cuid* или *shm_perm.uid* либо если процесс обладает привилегиями суперпользователя.

Linux и Solaris предоставляют две дополнительные команды, которые не являются частью стандарта Single UNIX Specification.

SHM_LOCK Заблокировать сегмент разделяемой памяти. Процесс сможет выполнить эту команду, только если обладает привилегиями суперпользователя.

SHM_UNLOCK Разблокировать сегмент разделяемой памяти. Процесс сможет выполнить эту команду, только если обладает привилегиями суперпользователя.

После создания сегмента разделяемой памяти процесс может присоединить его к своему адресному пространству с помощью функции *shmat*.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

Возвращает указатель на сегмент разделяемой памяти в случае успеха, -1 — в случае ошибки

Адрес, начиная с которого будет присоединен сегмент разделяемой памяти, зависит от значения аргумента *addr* и наличия флага **SHM_RND** в аргументе *flag*.

О Если в аргументе *addr* передано значение 0, сегмент будет присоединен к первому доступному адресу, который выберет ядро. Это рекомендованная методика.

- Если в аргументе *addr* передано ненулевое значение и флаг **SHM_RND** не указан, сегмент присоединяется, начиная с адреса *addr*.
- Если в аргументе *addr* передано ненулевое значение и указан флаг **SHM_RND**, сегмент будет присоединен с адреса, который вычисляется по формуле: $(addr - (addr \bmod SHMLBA))$. Имя константы **SHM_RND** происходит от слова «round» (округлить), а имя константы **SHMLBA**, величина которой всегда представлена степенью числа 2, — от «low boundary address multiple» (множитель адреса нижней границы). Приведенная выше формула округляет адрес вниз до ближайшего кратного числу **SHMLBA**.

Если мы не планируем, что приложение будет работать на единственной аппаратной платформе (что в наши дни весьма маловероятно), мы не должны указывать адрес присоединения сегмента разделяемой памяти. Вместо этого следует передавать в аргументе *addr* значение 0, позволяя системе самой выбрать адрес.

Если в аргументе *flag* указан флаг **SHM_RDONLY**, присоединенный сегмент будет доступен только для чтения. Иначе присоединенный сегмент доступен для чтения и записи.

Значение, возвращаемое функцией **shmat**, представляет адрес, начиная с которого был присоединен сегмент разделяемой памяти. В случае ошибки возвращается значение **-1**. Если вызов **shmat** завершился успехом, ядро увеличит счетчик **shm_nattch** в структуре **shmid_ds**, связанной с данным сегментом.

По окончании работы с сегментом разделяемой памяти следует вызывать функцию **shmdt** для его отсоединения. Обратите внимание: эта функция не удаляет из системы идентификатор и структуры данных, ассоциированные с сегментом памяти. Идентификатор продолжает существовать, пока какой-либо процесс (зачастую сервер) специально не удалит его вызовом функции **shmctl** с командой **IPC_RMID**.

```
#include <sys/shm.h>
int shmdt(void *addr);
```

Возвращает 0 в случае успеха, **-1** — в случае ошибки

В аргументе *addr* передается значение, полученное от функции **shmat**. В случае успеха **shmdt** уменьшает значение счетчика **shm_nattch** в структуре **shmid_ds**.

Пример

Адрес, к которому будет подключен сегмент разделяемой памяти, когда в аргументе *addr* передается значение 0, в значительной степени зависит от операционной системы. Листинг 15.11 содержит текст программы, которая выводит сведения о том, где размещаются различного рода данные в конкретной системе.

Листинг 15.11. Вывод сведений о размещении различного рода данных

```
#include "apue.h"
#include <sys/shm.h>
```

```

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 /* чтение и запись для владельца */

char array[ARRAY_SIZE]; /* неинициализированные данные = bss */

int
main(void)
{
    int      shmid;
    char    *ptr, *shmptr;

    printf("array[] от %p до %p\n", (void *)&array[0], (void*)&array[ARRAY_SIZE]);
    printf("стек примерно %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("ошибка вызова функции malloc");
    printf("динамически выделенная область от %p до %p\n",
           (void *)ptr, (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("ошибка вызова функции shmget");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("ошибка вызова функции shmat");
    printf("сегмент разделяемой памяти присоединен в адресах от %p до %p\n",
           (void *)shmptr, (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("ошибка вызова функции shmctl");

    exit(0);
}

```

Запуск этой программы в Linux на 64-разрядной платформе Intel дал следующие результаты:

```

$ ./a.out
array[] от 0x6020c0 до 0x60bd00
стек примерно 0x7fff957b146c
динамически выделенная область от 0x9e3010 до 0x9fb6b0
сегмент разделяемой памяти присоединен в адресах от 0x7fba578ab000 до
0x7fba578c36a0

```

На рис. 15.13 показана раскладка памяти, соответствующая полученным результатам. Обратите внимание: сегмент разделяемой памяти присоединен в адресах, расположенных значительно ниже стека.

В разделе 14.8 мы говорили, что с помощью функции `mmap` можно отобразить определенный участок файла в адресное пространство процесса. Концептуально это очень похоже на присоединение сегмента разделяемой памяти с помощью функции `shmat`. Главное отличие в том, что сегмент памяти, отображенный с помощью функции `mmap`, связан с файлом, тогда как сегмент разделяемой памяти вообще никак не связан с файлами.

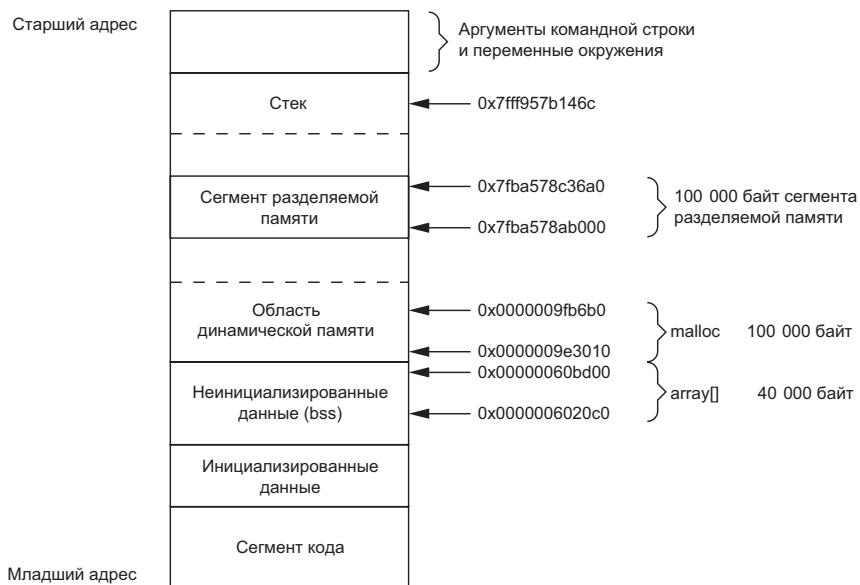


Рис. 15.13. Раскладка памяти в Linux на платформе Intel

Пример — отображение в память файла /dev/zero

Разделяемая память может использоваться для организации взаимодействия между процессами, которые не связаны родственными отношениями. Но если процессы взаимосвязаны, некоторые реализации предоставляют иную методику.

Следующий прием работает в FreeBSD 8.0, Linux 3.2.0 и Solaris 10. В Mac OS X 10.6.8 в настоящее время отображение символьных устройств в память процесса не поддерживается.

Устройство `/dev/zero` при чтении из него служит неиссякаемым источником нулевых байтов. Оно также может принимать любые объемы данных, совершенно игнорируя их. Это устройство представляет для нас интерес из-за особых свойств, которые оно проявляет при отображении в память.

- Создается неименованная область памяти, размер которой передается функции `mmap` во втором аргументе. Это число округляется до ближайшего целого, кратного размеру страницы.
- Область памяти инициализируется нулями.
- Эта область может совместно использоваться несколькими процессами, если их общий предок передал функции `mmap` флаг `MAP_SHARED`.

Пример работы с этим устройством приводится в листинге 15.12.

Листинг 15.12. Взаимодействие между родительским и дочерним процессами с использованием операций ввода/вывода над устройством /dev/zero, отображенными в память

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long) /* размер сегмента разделяемой памяти */

static int
update(long *ptr)
{
    return((*ptr)++); /* вернуть значение до увеличения */
}

int
main(void)
{
    int      fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("ошибка вызова функции open");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
                    fd, 0)) == MAP_FAILED)
        err_sys("ошибка вызова функции mmap");
    close(fd); /* после отображения, /dev/zero можно закрыть */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid > 0) { /* родительский процесс */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("предок: ожидалось %d, получено %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else { /* дочерний процесс */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("потомок: ожидалось %d, получено %d", i, counter);

            TELL_PARENT(getppid());
        }
    }
    exit(0);
}
```

Эта программа открывает устройство /dev/zero и вызывает функцию `mmap`, указывая ей размер отображаемой области. Обратите внимание: отобразив участок этого специального файла, мы можем закрыть его. После этого создается дочерний

процесс. Поскольку при отображении указан флаг `MAP_SHARED`, данные, которые запишет в эту область один процесс, сможет прочитать другой. (Если при отображении указать флаг `MAP_PRIVATE`, этот пример работать не будет.)

Затем родительский и дочерний процессы поочередно начинают увеличивать число, находящееся в разделяемой области отображеной памяти, используя для синхронизации функции из раздела 8.9. Число, находящееся в разделяемой памяти, инициализируется значением 0. Родительский процесс увеличивает его до значения 1, затем дочерний процесс увеличивает его до 2, потом родительский процесс увеличивает его до 3 и т. д. Обратите внимание, что в функции `update` используются круглые скобки, потому что нам нужно увеличить число в памяти, а не сам указатель.

Основное преимущество такого подхода заключается в отсутствии необходимости существования файла перед созданием отображенной области вызовом `mmap`. Отображение устройства `/dev/zero` автоматически создает область отображенной памяти заданного размера. Недостаток же состоит в том, что такой прием работает только с процессами, которые связаны родственными отношениями. Однако для родственных процессов, вероятно, более простым и эффективным решением было бы использование потоков (главы 11 и 12). Обратите внимание: независимо от выбранной методики, все равно необходимо синхронизировать доступ к разделяемым данным.

Пример — анонимные области отображаемой памяти

Многие реализации позволяют создавать анонимные области отображаемой памяти — примерно так же, как это делается при отображении устройства `/dev/zero`. Чтобы воспользоваться этой возможностью, нужно передать функции `mmap` флаг `MAP_ANON` и число `-1` вместо дескриптора файла. В результате мы получим анонимную (поскольку она не связана с именем какого-либо файла) область памяти, которая может совместно использоваться родственными процессами.

Возможность создания анонимных областей отображеной памяти имеется на всех четырех платформах, обсуждаемых в этой книге. Обратите внимание, что Linux определяет флаг `MAP_ANONYMOUS`, поддерживающий эту возможность, но при этом также определяет и флаг `MAP_ANON` с тем же значением для сохранения совместимости.

Чтобы программа из листинга 15.12 использовала эту возможность, в нее нужно внести три изменения: (а) убрать операцию открытия устройства `/dev/zero`, (б) убрать операцию закрытия дескриптора и (в) изменить обращение к функции `mmap` следующим образом:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, 1, 0)) == MAP_FAILED)
```

В этом вызове мы указали флаг `MAP_ANON` и передали значение `-1` вместо дескриптора файла. Остальная часть программы из листинга 15.12 остается без изменений. Последние два примера демонстрируют совместное использование области памяти двумя родственными процессами. Если необходимо использовать разделяемую память для организации взаимодействия между процессами, которые не связаны родственными отношениями, мы можем выбрать один из двух вариантов. При-

ложения могут использовать функции XSI, предназначенные для работы с разделяемой памятью, или функцию `mmap` с флагом `MAP_SHARED` для отображения одного и того же файла в собственные адресные пространства.

15.10. Семафоры POSIX

Механизм семафоров POSIX является одним из трех механизмов IPC, начинавшихся как расширения реального времени в POSIX.1. В стандарте Single UNIX Specification все три механизма (очереди сообщений, семафоры и разделяемая память) отнесены к разряду необязательных. До версии SUSv4 поддержка семафоров POSIX была включена как расширение семафоров. В SUSv4 она была перемещена в категорию базовых спецификаций, но интерфейсы для работы с очередями сообщений и разделяемой памятью остались необязательными для реализации.

Интерфейс семафоров POSIX призван восполнить нехватку некоторых особенностей в интерфейсе семафоров XSI:

- Семафоры POSIX обеспечивают более высокую скорость работы в сравнении с семафорами XSI.
- Семафоры POSIX проще в обращении: для работы с ними не требуется создавать множества семафоров, а некоторые операции с ними следуют знакомому шаблону операций с файловой системой. При этом не требуется, чтобы они были реализованы в файловой системе, хотя некоторые реализации пошли именно по такому пути.
- Семафоры POSIX создают меньше проблем при удалении. Вспомните, что при удалении семафора XSI операции, использующие тот же идентификатор семафора, терпят неудачу с кодом ошибки `EIDRM` в переменной `errno`. При использовании семафоров POSIX операции продолжают работать как обычно, пока не будет удалена последняя ссылка на семафор.

Существует две разновидности семафоров POSIX: именованные и неименованные. Они отличаются порядком создания и удаления, но в остальном действуют одинаково. Неименованные семафоры существуют только в памяти и требуют, чтобы процессы, использующие их, обладали правом доступа к этой памяти. То есть неименованные семафоры могут использоваться только потоками выполнения внутри одного и того же процесса или потоками выполнения в разных процессах, отображающих одну и ту же область памяти в свои адресные пространства. Именованные семафоры, напротив, доступны по именам и могут использоваться потоками выполнения в любых процессах, где известны их имена.

Чтобы создать новый именованный семафор или задействовать существующий, следует вызвать функцию `sem_open`.

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode,
                                                 unsigned int value */ );
```

Возвращает указатель на семафор в случае успеха,
значение `SEM_FAILED` — в случае неудачи

При использовании существующего именованного семафора необходимо передать два аргумента: имя семафора в аргументе *name* и ноль — в аргументе *oflag*. Если в аргументе *oflag* передать флаг *O_CREAT*, будет создан новый именованный семафор, если он еще не существует; иначе будет открыт уже имеющийся семафор, но дополнительные шаги по его инициализации будут опущены.

Когда указывается флаг *O_CREAT*, необходимо также передать два дополнительных аргумента. Аргумент *mode* описывает, кто имеет право доступа к семафору. Он может принимать те же биты прав доступа, что передаются при открытии файла: *user-read*, *user-write*, *user-execute*, *group-read*, *group-write*, *group-execute*, *other-read*, *other-write* и *other-execute*. Окончательные права доступа к семафору определяются значением аргумента *mode* и маской процесса для прав доступа при создании файлов (разделы 4.5 и 4.8). Но имейте в виду, что учитываются только права на чтение и на запись. При открытии существующего семафора интерфейсы не позволяют указывать права доступа. Обычно реализации открывают семафоры одновременно для чтения и для записи.

Аргумент *value* определяет начальное значение создаваемого семафора. Он может принимать любое значение в диапазоне от 0 до *SEM_VALUE_MAX* (табл. 2.9).

Если требуется гарантировать создание нового семафора, в аргументе *oflag* следует передать значение *O_CREAT|O_EXCL*. В этом случае вызов *sem_open* потерпит неудачу, если семафор уже существует.

Для переносимости необходимо следовать определенным соглашениям при выборе имени семафора.

- Первым символом в имени должен быть слеш (/). Хотя не требуется, чтобы семафоры POSIX были реализованы на основе файловой системы, тем не менее, если файловая система все-таки используется, необходимо устраниТЬ любую неоднозначность, которая может возникнуть при интерпретации имен семафоров.
- Имя не должно содержать других символов слеша, чтобы избежать любой зависимости от конкретных особенностей реализации. Например, если в используемой файловой системе имена /*mysem* и //*mysem* считаются одинаковыми, а данная конкретная реализация не использует файловую систему для поддержки семафоров, эти два имени могут интерпретироваться как разные (представьте, что произойдет, если реализация хеширует имена семафоров в целочисленные значения).
- Максимальная длина имени семафора определяется реализацией. Имя не должно быть длиннее, чем *_POSIX_NAME_MAX* символов (табл. 2.8), потому что это минимально допустимый предел максимальной длины имени для реализаций, где поддержка семафоров основана на файловой системе.

Функция *sem_open* возвращает указатель на семафор, который можно передавать другим функциям управления семафорами. По завершении работы с семафором можно вызвать функцию *sem_close*, чтобы освободить все ресурсы, ассоциированные с ним.

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

Если процесс завершится без вызова `sem_close`, ядро автоматически закроет все открытые им семафоры. Имейте в виду, что это не оказывает влияния на состояние значения семафора: даже если процесс увеличил значение семафора, оно не изменится при выходе. Аналогично, вызов `sem_close` не оказывает влияния на значение семафора. Семафоры POSIX не предусматривают механизма, эквивалентного флагу `SEM_UNDO`, имеющемуся в семафорах XSI.

Чтобы удалить именованный семафор, можно вызывать функцию `sem_unlink`.

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

Функция `sem_unlink` удаляет имя семафора. Если в системе не остается открытых ссылок на семафор, он также удаляется. Иначе удаление семафора откладывается, пока не будет закрыта последняя ссылка на него.

В отличие от семафоров XSI, одним вызовом функции можно увеличить или уменьшить значение семафора POSIX только на единицу. Уменьшение счетчика действует аналогично запиранию двоичного семафора или захвату единицы ресурса, связанного с семафором-счетчиком.

Обратите внимание, что семафоры POSIX не различаются по типам. Будет ли семафор действовать как счетчик или как двоичный семафор, зависит от того, как он инициализируется и используется. Если семафор может принимать только два значения, 0 или 1, — это двоичный семафор. Когда двоичный семафор получает значение 1, мы говорим, что семафор «открыт», когда он получает значение 0, мы говорим «заперт».

Уменьшить значение семафора можно вызовом функции `sem_wait` или `sem_trywait`.

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Обе возвращают 0 в случае успеха, -1 — в случае неудачи

Процесс блокируется в вызове функции `sem_wait`, если семафор имеет значение 0. Она не вернет управление, пока не сможет успешно уменьшить значение семафора или не будет прервана сигналом. Чтобы избежать блокировки, можно использовать функцию `sem_trywait`. Если семафор имеет значение, равное 0, функция `sem_trywait` вернет -1 с кодом ошибки `EAGAIN` в переменной `errno`.

Третья альтернатива — блокирование процесса на ограниченный промежуток времени. Для этой цели можно использовать функцию `sem_timedwait`.

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsprtr);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

Аргумент *tsprtr* определяет абсолютное время, когда следует прервать ожидание на семафоре. Ожидание реализовано на основе часов **CLOCK_REALTIME** (табл. 6.7). Если значение семафора можно уменьшить немедленно, аргумент *tsprtr* игнорируется: даже если он будет содержать отметку времени в прошлом, попытка уменьшить значение семафора все равно будет выполнена. По истечении тайм-аута, если уменьшить значение семафора так и не удалось, **sem_timedwait** вернет -1 с кодом ошибки **ETIMEDOUT** в переменной *errno*.

Увеличить значение семафора можно вызовом функции **sem_post**. Эта операция является аналогом отпирания двоичного семафора или освобождения единицы ресурса, связанного с семафором-счетчиком.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

Если к моменту вызова **sem_post** имелся процесс, заблокированный в вызове **sem_wait** (или **sem_timedwait**), он будет разблокирован и значение семафора, только что увеличенное функцией **sem_post**, будет уменьшено функцией **sem_wait** (или **sem_timedwait**).

Когда семафоры POSIX применяются только в рамках одного процесса, проще использовать неименованные семафоры. От именованных семафоров они отличаются только способом создания и удаления. Создать неименованный семафор можно вызовом функции **sem_init**.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

Аргумент *pshared* указывает, планируется ли использовать новый семафор одновременно в нескольких процессах. Если да, в этом аргументе следует передать не нулевое значение. Аргумент *value* определяет начальное значение семафора.

В отличие от **sem_open**, функция **sem_init** возвращает признак ошибки, а не указатель на семафор, поэтому мы должны объявить переменную типа **sem_t** и передать ее адрес функции **sem_init** для инициализации. Если предполагается использовать семафор для синхронизации процессов, аргумент *sem* должен ссылаться на область памяти, доступную этим процессам.

Завершив работу с неименованным семафором, его можно удалить вызовом функции `sem_destroy`.

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

После вызова `sem_destroy` семафор нельзя больше использовать для вызова функций управления семафором — только после повторной его инициализации вызовом `sem_init`.

Существует еще одна функция, позволяющая извлекать текущее значение семафора, — `sem_getvalue`.

```
#include <semaphore.h>
int sem_getvalue(sem_t *restrict sem, int *restrict valp);
```

Возвращает 0 в случае успеха, -1 — в случае неудачи

В случае успеха по адресу в аргументе `valp` функция вернет целочисленное значение семафора. Но будьте осторожны, фактическое значение семафора может измениться к моменту, когда вы будете пытаться использовать только что прочитанное значение. Если не использовать дополнительные механизмы синхронизации, чтобы предотвратить состояние гонки, функцию `sem_getvalue` имеет смысл использовать только для отладки.

Функция `sem_getvalue` не поддерживается системой Mac OS X 10.6.8.

Пример

Одной из побудительных причин введения семафоров POSIX было стремление получить более высокую производительность в сравнении с семафорами XSI. Было бы весьма поучительно посмотреть, достигнута ли эта цель в существующих системах, даже при том, что эти системы не предназначены для поддержки приложений реального времени.

В табл. 15.9 приводятся результаты хронометража производительности семафоров XSI (без флага `SEM_UNDO`) и POSIX в ситуации, когда три процесса конкурируют между собой, пытаясь захватить и освободить семафор 1 000 000 раз.

Хронометраж выполнялся на двух платформах: Linux 3.2.0 и Solaris 10.

В табл. 15.9 видно, что в Solaris семафоры POSIX дают лишь 12%-ный прирост производительности в сравнении с семафорами XSI, но в Linux этот прирост составляет 94% (почти в 18 раз быстрее)! Если заняться трассировкой программ, можно обнаружить, что реализация семафоров POSIX в Linux отображает файл в адресное пространство процесса и затем выполняет отдельные операции с семафорами без использования системных вызовов.

Таблица 15.9. Результаты хронометража производительности разных реализаций семафоров

Семафоры	Solaris 10			Linux 3.2.0		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
XSI	11,85	15,85	27,91	0,33	5,93	7,33
POSIX	13,72	10,52	24,44	0,26	0,75	0,41

Пример

Как показано в табл. 12.4, стандарт Single UNIX Specification не определяет, что произойдет, если один поток выполнения запрет обычный мьютекс, а другой попытается отпереть его. Но для мьютексов с проверкой ошибок и рекурсивных мьютексов в этом случае возвращается признак ошибки. Так как двоичные семафоры могут использоваться в роли мьютексов, мы можем использовать их для создания собственных примитивов синхронизации.

Допустим, что нам требуется создать блокировку, которая может быть заперта одним потоком выполнения и отпerta другим. Определение структуры такой блокировки могло бы выглядеть так:

```
struct slock {
    sem_t *semp;
    char name[_POSIX_NAME_MAX];
};
```

Программа в листинге 15.13 демонстрирует реализацию примитива синхронизации на основе семафора POSIX.

Листинг 15.13. Реализация мьютекса с использованием семафора POSIX

```
#include "slock.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

struct slock *

s_alloc()
{
    struct slock *sp;
    static int cnt;

    if ((sp = malloc(sizeof(struct slock))) == NULL)
        return(NULL);
    do {
        sprintf(sp->name, sizeof(sp->name), "%ld.%d", (long)getpid(), cnt++);
        sp->smp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1);
    } while ((sp->smp == SEM_FAILED) && (errno == EEXIST));
    if (sp->smp == SEM_FAILED) {
        free(sp);
        return(NULL);
    }
    return(sp);
}
```

```

        return(NULL);
    }
    sem_unlink(sp->name);
    return(sp);
}

void
s_free(struct slock *sp)
{
    sem_close(sp->semp);
    free(sp);
}

int
s_lock(struct slock *sp)
{
    return(sem_wait(sp->semp));
}

int
s_trylock(struct slock *sp)
{
    return(sem_trywait(sp->semp));
}

int
s_unlock(struct slock *sp)
{
    return(sem_post(sp->semp));
}

```

Наша реализация конструирует имя семафора на основе идентификатора процесса и счетчика. Мы не стали защищать счетчик с помощью мьютекса, потому что если два потока выполнения одновременно вызовут `s_alloc` и получат одно и то же имя, наличие флага `O_EXCL` в вызове `sem_open` обеспечит успешное создание семафора только в одном потоке. Второй получит код ошибки `EEXIST` в переменной `errno` и сможет просто повторить попытку. Обратите внимание, что сразу после создания семафора мы тут же вызываем функцию `sem_unlink`. Она удалит имя семафора, благодаря чему никакой другой процесс не сможет получить доступ к нему и упрощается процедура освобождения ресурсов по завершении процесса.

15.11. Свойства взаимодействий типа клиент-сервер

Рассмотрим подробнее некоторые свойства клиентов и серверов, которые имеют отношение к различным механизмам IPC, используемым для взаимодействия между ними. Самый простой тип взаимоотношений — когда клиент с помощью функций `fork` и `exec` запускает требуемый сервер. В этом случае перед вызовом функции `fork` могут быть созданы два полудуплексных канала, чтобы организовать движение данных в обе стороны. На рис. 15.8 показан пример такой организации взаимодействий. Запускаемый сервер может быть программой с установленным битом set-user-ID, что дает ему специальные привилегии. Кроме того, сервер

может идентифицировать клиента, получив собственный реальный идентификатор пользователя. (В разделе 8.10 мы говорили, что реальные идентификаторы пользователя и группы не изменяются при запуске новой программы с помощью функции `exec`.)

На основе этой схемы мы можем разработать сервер открытия файлов. (Реализация его будет показана в разделе 17.5.) Он будет открывать файлы по запросу клиента. То есть мы можем добавить проверку дополнительных прав доступа, кроме обычных для UNIX прав пользователь/группа/остальные. Этот подход предполагает, что программа-сервер должна иметь установленный бит set-user-ID, чтобы получить дополнительные привилегии (возможно, привилегии суперпользователя). Сервер на основе реального идентификатора пользователя клиента определяет, разрешен ли ему доступ к запрошенному файлу. Благодаря этому мы можем создать сервер, который предоставляет определенным пользователям дополнительные привилегии, которых они обычно не имеют.

В этом примере, поскольку сервер является дочерним процессом по отношению к клиенту, он может передать родительскому процессу только содержимое файла. Хотя такой подход вполне применим к обычным файлам, он не может быть использован, например, для специальных файлов устройств. Было бы лучше, если бы сервер открывал требуемый файл и передавал клиенту дескриптор файла. Родительский процесс может передать дескриптор потомку, но передать дескриптор в обратном направлении, от дочернего процесса родительскому, невозможно (если не использовать специальные приемы, о которых мы расскажем в главе 17).

Следующий тип сервера был показан на рис. 15.12. Это процесс-демон, который взаимодействует со всеми клиентами посредством некоторого механизма IPC. Для такого рода взаимодействий между клиентами и сервером нельзя использовать неименованные каналы. Здесь требуется именованная форма IPC, например именованные каналы (FIFO) или очереди сообщений. В случае именованных каналов, как мы уже видели, необходимо создавать отдельный именованный канал для связи с каждым из клиентов, если предполагается передача данных клиенту от сервера. Если же данные передаются только от клиента, достаточно будет создать единственный именованный канал с предопределенным именем. (Такую форму взаимодействий использует демон печати в System V. В этом случае в роли клиента выступает команда `lp(1)`, а сервер представлен демоном `lpsched`. Данные в этой схеме передаются только от клиента к серверу, обратная связь полностью отсутствует.)

При использовании очередей сообщений мы получаем дополнительные возможности.

1. Для взаимодействия сервера со всеми клиентами достаточно одной очереди. Поле `type` в сообщении может служить для идентификации получателя сообщения. Например, клиенты могут отправлять запросы, указывая в поле `type` число 1. При этом каждый клиент должен включать в сообщение идентификатор своего процесса. В результате сервер может принимать только сообщения, в которых поле `type` имеет значение 1 (четвертый аргумент функции `msgrcv`), а клиенты — принимать только сообщения, в которых значение поля `type` совпадает с идентификаторами их процессов.

2. Для каждого клиента также может быть создана отдельная очередь сообщений. Перед отправкой первого запроса клиент создает собственную очередь сообщений с ключом `IPC_PRIVATE`. Сервер также должен создать очередь с ключом или идентификатором, которые известны клиентам. Первый запрос клиент передает через предопределенную очередь сообщений, отсылая серверу идентификатор своей очереди, а весь последующий обмен данными уже происходит через отдельную очередь, созданную клиентом. Свой первый и все последующие отклики сервер передает через очередь сообщений клиента.

Один из недостатков такого подхода заключается в том, что каждая клиентская очередь может содержать всего одно сообщение — либо запрос клиента, либо ответ сервера. Это выглядит слишком расточительно из-за ограничений на количество очередей в системе, поэтому вместо отдельных очередей лучше использовать именованные каналы. Другая проблема состоит в том, что сервер вынужден получать сообщения из нескольких очередей сразу, но ни `select`, ни `poll` не могут работать с очередями сообщений.

Любая из этих двух методик, основанных на очередях сообщений, может быть реализована на базе разделяемой памяти с применением методов синхронизации (семафоры или блокировка записей в файле).

Проблема с таким видом взаимодействий клиента и сервера (когда они не связаны родственными отношениями) состоит в том, что сервер должен точно идентифицировать клиента. Если сервер выполняет привилегированные операции, он должен точно знать, кто является клиентом. Это совершенно необходимо, если сервер, например, является программой с установленным битом `set-user-ID`. Хотя все эти формы IPC проходят через ядро, оно не предоставляет никаких средств идентификации отправителя.

В случае очередей сообщений, когда для передачи данных между сервером и клиентом используется единственная очередь, в которой может одновременно находиться только одно сообщение, поле `msg_lspid` будет содержать идентификатор процесса отправителя. Но это не совсем то, что нам нужно, — желательно было бы иметь эффективный идентификатор пользователя заданного процесса. К сожалению, переносимого способа получения эффективного идентификатора пользователя по идентификатору процесса не существует. (Естественно, ядро хранит оба этих значения в таблице процессов, но, обладая одним, мы не можем получить другой без прямого поиска в памяти ядра.)

В разделе 17.2 мы будем применять следующую методику идентификации клиента на стороне сервера. Этот прием также может использоваться при работе с именованными каналами, очередями сообщений, семафорами или разделяемой памятью. Предположим, что для организации взаимодействий, схема которых представлена на рис. 15.12, используются именованные каналы. Клиент должен создать собственный канал FIFO и установить права доступа к нему так, чтобы он был доступен для чтения и записи только владельцу. Здесь мы исходим из предположения, что сервер обладает привилегиями суперпользователя (иначе нет большого смысла беспокоиться по поводу идентификации клиента), то есть сервер может выполнять операции чтения и записи с данным каналом. Когда по предопределенному каналу FIFO поступает первый запрос от клиента (который

должен содержать идентификатор канала клиента), сервер вызывает функцию `stat` или `fstat` для канала клиента. Предполагается, что эффективный идентификатор пользователя клиента — это идентификатор владельца FIFO (поле `st_uid` структуры `stat`). Сервер должен убедиться, что доступ к каналу разрешен только для его владельца. Дополнительно сервер должен проверить, имеют ли три поля времени, связанные с FIFO (поля `st_atime`, `st_mtime` и `st_ctime` структуры `stat`), допустимые значения (например, не более 15 или 30 секунд). Если злоумышленник сможет создать канал FIFO с другим эффективным идентификатором и установить право только на чтение и на запись для владельца, значит, система имеет серьезные проблемы с безопасностью.

Чтобы реализовать эту методику для XSI IPC, вспомните, что с каждой очередью сообщений, семафором и сегментом разделяемой памяти ассоциируется структура `ipc_perm`, которая идентифицирует создателя объекта IPC (поля `cuid` и `cgid`). Как и в случае FIFO, сервер должен требовать от клиента, чтобы создаваемая им структура IPC имела права доступа только для владельца. Кроме того, сервер должен убедиться, что все характеристики времени имеют надлежащие значения (поскольку эти структуры IPC могут существовать в системе, пока явно не будут удалены).

В разделе 17.3 мы увидим, что существует более надежный способ идентификации, когда эффективные идентификаторы пользователя и группы клиента предстаются ядром. Сделать это можно с помощью подсистемы сокетов, передавая дескрипторы файлов между процессами.

15.12. Подведение итогов

Мы рассмотрели разнообразные формы взаимодействий между процессами: именованные и неименованные каналы, три формы IPC, которые обычно называют XSI IPC (очереди сообщений, семафоры и разделяемую память), и альтернативный механизм семафоров POSIX. Семафоры в действительности представляют собой механизм синхронизации, а не обмена данными и часто используются для синхронизации доступа к разделяемым ресурсам, таким как сегменты разделяемой памяти. При обсуждении неименованных каналов мы рассмотрели реализацию функции `ropen`, понятие сопроцессов и возможные ловушки, связанные с режимом буферизации в стандартной библиотеке ввода/вывода.

После сравнения производительности очередей сообщений с дуплексными каналами и семафорами с механизмом блокировки записей в файлах мы можем дать следующие рекомендации. Изучайте именованные и неименованные каналы, поскольку эти два механизма по-прежнему остаются эффективным средством организации обмена данными для большинства приложений. Избегайте использования очередей сообщений и семафоров в новых приложениях. Вместо них следует применять дуплексные каналы и блокировки записей в файлах, так как они намного проще. Разделяемая память может найти применение, хотя те же возможности предоставляются функцией `mmap` (раздел 14.8).

В следующей главе мы рассмотрим механизмы сетевых взаимодействий, которые помогают организовать обмен информацией между разными машинами.

Упражнения

- 15.1 В программе из листинга 15.2 удалите обращение к функции `close` перед вызовом `waitpid` в конце кода родителя. Объясните, что произойдет.
- 15.2 В программе из листинга 15.2 удалите обращение к функции `waitpid` в конце кода родителя. Объясните, что произойдет.
- 15.3 Что случится, если функции `popen` передать имя несуществующей команды? Напишите небольшую программу, чтобы проверить эту ситуацию.
- 15.4 В программе из листинга 15.9 удалите обработчик сигнала, запустите программу и завершите дочерний процесс. Как можно убедиться, что родительский процесс завершился при получении сигнала `SIGPIPE` после ввода строки?
- 15.5 Попробуйте в программе из листинга 15.9 использовать для работы с неименованными каналами вместо функций `read` и `write` функции чтения и записи из стандартной библиотеки ввода/вывода.
- 15.6 В пояснениях к стандарту POSIX.1 в качестве одной из причин появления функции `waitpid` приводится описание ситуации, которая не может быть обработана без этой функции:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

Что получится в результате выполнения этого кода, если вместо функции `waitpid` использовать функцию `wait`?

- 15.7 Объясните, как функции `select` и `poll` обрабатывают ситуацию закрытия неименованного канала пишущим процессом. Чтобы ответить на этот вопрос, напишите две небольшие программы: одну — с использованием функции `select`, другую — с использованием функции `poll`.
Повторите это упражнение для проверки ситуации, когда канал закрывается читающим процессом.
- 15.8 Что случится, если команда `cmdstring`, запущенная функцией `popen` со значением "r" в аргументе `type`, попытается вывести что-нибудь в стандартный вывод сообщений об ошибках?
- 15.9 Для выполнения команды из аргумента `cmdstring` функция `popen` вызывает командный интерпретатор. Что происходит по завершении `cmdstring`? (Подсказка: нарисуйте схему происходящего.)
- 15.10 Стандарт POSIX.1 особо отмечает, что возможность открытия канала FIFO с помощью функции `open` одновременно для чтения и записи не предусмотрена, хотя большинство версий UNIX допускают это. Продемонстрируйте другой метод открытия FIFO для чтения и записи без использования блокировок.

- 15.11** Если файл не содержит секретной информации, его доступность для чтения для всех пользователей не несет никакого вреда. (Хотя обычно попытки совать нос в чужие файлы не одобряются.) Но что может произойти, если злонамеренный процесс получит доступ для чтения к очереди сообщений, которая используется для взаимодействия сервера и нескольких клиентов? Какой информацией должен обладать злонамеренный процесс, чтобы прочитать содержимое очереди сообщений?
- 15.12** Напишите программу, которая выполняет следующие действия: пять раз в цикле создает очередь сообщений, выводит идентификатор очереди, удаляет очередь сообщений; затем в другом цикле пять раз создает очередь сообщений с ключом `IPC_PRIVATE` и размещает в очереди одно сообщение. После завершения программы просмотрите очереди сообщений с помощью команды `ipcs(1)`. Объясните, что происходит с идентификаторами очередей.
- 15.13** Опишите, как создать связанный список объектов данных в сегменте разделяемой памяти. Что следует хранить в качестве указателей в списке?
- 15.14** Нарисуйте временную диаграмму работы программы из листинга 15.12, показывающую значение переменной `i` в родительском и дочернем процессах, значения числа в разделяемой памяти и возвращаемые значения функции `update`. Исходите из предположения, что после вызова функции `fork` первым получает управление дочерний процесс.
- 15.15** Перепишите программу из листинга 15.12 так, чтобы она вместо отображаемой памяти использовала функции для работы с разделяемой памятью XSI из раздела 15.9.
- 15.16** Перепишите программу из листинга 15.12 так, чтобы она использовала семафоры XSI для синхронизации родительского и дочернего процессов.
- 15.17** Перепишите программу из листинга 15.12 так, чтобы она использовала механизм блокировки записей в файле для синхронизации родительского и дочернего процессов.
- 15.18** Перепишите программу из листинга 15.12 так, чтобы она использовала семафоры POSIX из раздела 15.10 для синхронизации родительского и дочернего процессов.

16

Межпроцессные взаимодействия в сети: сокеты

16.1. Введение

В предыдущей главе мы рассмотрели именованные и неименованные каналы, очереди сообщений, семафоры и разделяемую память — классические механизмы межпроцессных взаимодействий, предоставляемые различными версиями UNIX. Эти механизмы позволяют организовать взаимодействие между процессами, работающими на одной машине. В этой главе мы рассмотрим сетевые механизмы IPC, которые позволяют взаимодействовать процессам, выполняющимся на разных машинах (объединенных в общую сеть).

В этой главе будет описан интерфейс сетевых сокетов, который можно использовать для взаимодействий между процессами независимо от того, где они работают — на одной машине или на разных. Это было одной из основных целей при разработке интерфейса сокетов: один и тот же набор функций должен был использоваться как для внутримашинного, так и для межмашинного обмена данными. Несмотря на то что интерфейс сокетов может использоваться для работы по многим сетевым протоколам, в этой главе мы ограничимся обсуждением только протоколов TCP/IP, поскольку де-факто они стали стандартом для взаимодействий через Интернет.

Как указывает стандарт POSIX.1, интерфейс сокетов основан на интерфейсе сокетов 4.4BSD. Хотя за прошедшие годы и были внесены некоторые изменения, тем не менее современный интерфейс весьма напоминает тот, что впервые появился в начале 80-х годов в 4.2BSD.

Эта глава — лишь краткий обзор прикладного программного интерфейса сокетов. Детальное обсуждение сокетов вы найдете в книге, посвященной сетевому программированию в UNIX [Stevens, Fenner, and Rudoff, 2004].

16.2. Дескрипторы сокетов

Сокет — это абстракция конечной точки взаимодействия. Подобно тому как для работы с файлами приложения используют дескрипторы файлов, для работы с сокетами они используют дескрипторы сокетов. В UNIX дескрипторы сокетов реализованы так же, как дескрипторы файлов. В действительности большинство

функций, работающих с дескрипторами файлов, таких как `read` или `write`, будут работать и с дескрипторами сокетов.

Создается дескриптор сокета с помощью функции `socket`.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Возвращает дескриптор файла (сокета) в случае успеха,
-1 – в случае ошибки

Аргумент *domain* определяет природу взаимодействия, включая формат адреса (более подробно он будет описан в следующем разделе). В табл. 16.1 приводится список доменов, которые определены стандартом POSIX.1. Имена констант начинаются с префикса `AF_` (от address family – семейство адресов), потому что каждый домен обладает своим собственным форматом представления адресов.

Таблица 16.1. Домены сокетов

Домен	Описание
<code>AF_INET</code>	Домен Интернета IPv4
<code>AF_INET6</code>	Домен Интернета IPv6 (необязательный в POSIX.1)
<code>AF_UNIX</code>	Домен UNIX
<code>AF_UNSPEC</code>	Неопределенный домен

Домен UNIX будет обсуждаться в разделе 17.2. Большинство систем определяют дополнительный домен `AF_LOCAL`, который является псевдонимом домена `AF_UNIX`. Константа `AF_UNSPEC` обозначает неопределенный домен, который может представлять любой домен. Некоторые платформы традиционно реализуют поддержку дополнительных сетевых протоколов, таких как `AF_IPX` для семейства протоколов NetWare, но стандарт POSIX.1 не определяет константы доменов для этих протоколов.

В аргументе *type* указывается тип сокета, который, в свою очередь, определяет характеристики взаимодействия. Типы сокетов, определенные стандартом POSIX.1, перечислены в табл. 16.2, но реализации могут добавлять поддержку дополнительных типов.

В аргументе *protocol* обычно передается значение 0, чтобы выбрать протокол по умолчанию для данного домена и типа сокета. Если для одного и того же домена и типа сокета поддерживается несколько протоколов, можно использовать этот аргумент для выбора конкретного протокола. Протокол по умолчанию для сокетов типа `SOCK_STREAM` из домена `AF_INET` – TCP (Transmission Control Protocol – протокол управления передачей данных). Протокол по умолчанию для сокетов типа `SOCK_DGRAM` из домена `AF_INET` – UDP (User Datagram Protocol – протокол пользовательских дейтаграмм). В табл. 16.3 перечислены протоколы, определенные для сокетов из домена Интернета.

Таблица 16.2. Типы сокетов

Тип	Описание
SOCK_DGRAM	Не ориентированы на создание логического соединения, сообщения фиксированной длины, доставка сообщений не гарантируется
SOCK_RAW	Интерфейс дейтаграмм к протоколу IP (необязателен в POSIX.1)
SOCK_SEQPACKET	Ориентированы на создание логического соединения, упорядоченность передачи данных, сообщения фиксированной длины, гарантируется доставка сообщений
SOCK_STREAM	Ориентированы на создание логического соединения, упорядоченность передачи данных, гарантируется доставка сообщений, двунаправленный поток байтов

Таблица 16.3. Протоколы для сокетов из домена Интернета

Протокол	Описание
IPPROTO_IP	Протокол Интернета IPv4
IPPROTO_IPV6	Протокол Интернета IPv6 (необязателен в POSIX.1)
IPPROTO_ICMP	Протокол управляющих сообщений Интернета
IPPROTO_RAW	Протокол простых пакетов IP (необязателен в POSIX.1)
IPPROTO_TCP	Протокол управления передачей данных
IPPROTO_UDP	Протокол пользовательских дейтаграмм

При использовании интерфейса дейтаграмм (**SOCK_DGRAM**) не требуется устанавливать логическое соединение, чтобы обмениваться данными между конечными точками взаимодействия. Все, что нужно сделать, — передать сообщение по адресу сокета, который используется процессом на другом конце.

Поэтому дейтаграммы представляют службу, не ориентированную на установление логического соединения. Потоки байтов (**SOCK_STREAM**), с другой стороны, требуют, чтобы перед началом обмена данными между нашим сокетом и сокетом, принадлежащим сетевому узлу, с которым предполагается взаимодействовать, было установлено логическое соединение.

Дейтаграмма представляет собой самостоятельное сообщение. Передача дейтаграммы напоминает отправку письма по почте. Можно отправить множество писем, но нельзя гарантировать, что они будут доставлены в определенном порядке и что некоторые из них не потеряются по дороге. Каждое письмо содержит адрес получателя, благодаря чему оно не зависит от других писем. Письма даже могут быть отправлены разным получателям.

Напротив, протоколы, ориентированные на создание логического соединения, организованы как телефонный звонок. Прежде всего необходимо установить соединение, набрав номер телефона, и после того, как соединение будет установлено, оно обеспечивает двунаправленную связь с удаленным абонентом. Соединение обеспечивается одноранговым коммуникационным каналом. Такого рода соединение, через которое вы имеете возможность общаться, является соединением типа «точка-

точка». Ваши слова не содержат адресной информации, так как подключение этого типа логически связывает оба конца коммуникационного канала и само по себе подразумевает однозначную идентификацию отправителя и получателя.

При использовании сокетов типа `SOCK_STREAM` приложения не распознают границ отдельных сообщений, поскольку сокеты такого типа реализуют услугу передачи потока байтов. Это означает, что операция чтения данных из сокета может вернуть не то количество байтов, которое было записано передающим процессом. В конечном счете будет получено все, что было отправлено, но для этого может потребоваться несколько вызовов функций.

Сокеты типа `SOCK_SEQPACKET` очень похожи на сокеты типа `SOCK_STREAM`, за исключением того, что вместо услуги приема/передачи данных в виде потока байтов они реализуют услугу передачи отдельных сообщений. Это означает, что объем данных, полученных из сокета типа `SOCK_SEQPACKET`, всякий раз в точности совпадает с объемом отправленных данных. Служба передачи последовательности пакетов в домене Интернета реализуется на базе протокола SCTP (Stream Control Transmission Protocol — протокол передачи с управлением потоком).

Сокеты типа `SOCK_RAW` представляют интерфейс дейтаграмм на сетевом уровне (то есть интерфейс к протоколу IP в домене Интернета). При использовании этого интерфейса вся ответственность за построение заголовков пакетов возлагается на приложения, поскольку сокеты этого типа не используют протоколы транспортного уровня (такие, как TCP или UDP). Чтобы предотвратить использование сокетов типа `SOCK_RAW` в неблаговидных целях, для их создания приложение должно обладать привилегиями суперпользователя.

Вызов функции `socket` напоминает вызов функции `open`. В обоих случаях мы получаем дескриптор файла, который затем используется в операциях ввода/вывода. По окончании работы с сокетом вызывается функция `close`, которая закрывает соединение и освобождает номер дескриптора для повторного использования.

Хотя дескриптор сокета является файловым дескриптором, его можно использовать не во всех функциях, которые принимают дескриптор файла. В табл. 16.4 приводится перечень большинства описанных нами функций, которые работают с файловыми дескрипторами, и дается описание их поведения при обслуживании дескрипторов сокетов. Если в ячейке таблицы указано «не определено» или «зависит от реализации», это означает, что, как правило, данная функция не может работать с дескрипторами сокетов. Например, функция `lseek` не может работать с сокетами, поскольку сокеты не поддерживают понятие текущей позиции в файле.

Таблица 16.4. Поведение некоторых функций при работе с сокетами

Функция	Поведение при работе с сокетами
<code>close</code> (раздел 3.5)	Освобождает сокет
<code>dup</code> , <code>dup2</code> (раздел 3.12)	Создают дубликат дескриптора
<code>fchdir</code> (раздел 4.23)	Завершается с кодом ошибки <code>ENOTDIR</code> в переменной <code>errno</code>
<code>fchmod</code> (раздел 4.9)	Не определено
<code>fchown</code> (раздел 4.11)	Зависит от реализации

Таблица 16.4 (окончание)

Функция	Поведение при работе с сокетами
<code>fcntl</code> (раздел 3.14)	Поддерживает некоторые команды, включая <code>F_DUPFD</code> , <code>F_GETFD</code> , <code>F_GETFL</code> , <code>F_GETOWN</code> , <code>F_SETFD</code> , <code>F_SETFL</code> и <code>F_SETOWN</code>
<code>fdatasync</code> , <code>fsync</code> (раздел 3.13)	Зависит от реализации
<code>fstat</code> (раздел 4.2)	Поддерживает некоторые поля структуры <code>stat</code> , но правила поддержки определяются реализацией
<code>ftruncate</code> (раздел 4.13)	Не определено
<code>ioctl</code> (раздел 3.15)	Выполняет ограниченный набор команд, который зависит от реализации драйвера устройства
<code>lseek</code> (раздел 3.6)	Зависит от реализации (обычно завершается с кодом ошибки <code>ESPIPE</code>)
<code>mmap</code> (раздел 14.8)	Не определено
<code>poll</code> (раздел 14.4.2)	Работает так, как и следует ожидать
<code>pread</code> , <code>pwrite</code> (раздел 3.11)	Завершается с кодом ошибки <code>ESPIPE</code> в переменной <code>errno</code>
<code>read</code> (раздел 3.7) и <code>readv</code> (раздел 14.6)	Эквивалентны вызову функции <code>recv</code> (раздел 16.5) без каких-либо флагов
<code>select</code> (раздел 14.4.1)	Работает так, как и следует ожидать
<code>write</code> (раздел 3.8) и <code>writen</code> (раздел 14.6)	Эквивалентны вызову функции <code>send</code> (раздел 16.5) без каких-либо флагов

Обмен данными через сокеты является двунаправленным. Выполнение отдельных операций над сокетами можно запретить с помощью функции `shutdown`.

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Если в аргументе `how` передать значение `SHUT_RD`, операция чтения из сокета будет запрещена. Если передать значение `SHUT_WR`, будет запрещена операция записи в сокет. Если передать значение `SHUT_RDWR`, будет запрещена возможность передачи данных в обоих направлениях.

Зачем же нужна функция `shutdown`, если `close` умеет работать с сокетами? На то есть несколько причин. Во-первых, функция `close` закрывает соединение и освобождает дескриптор, только когда будет закрыта последняя активная ссылка на сокет. Это означает, что если мы создали дубликат дескриптора сокета (например, с помощью функции `dup`), функция `close` не сможет закрыть сокет, пока не будет закрыт последний файловый дескриптор, ссылающийся на него. Функция `shutdown` позволяет деактивировать сокет, независимо от количества ссылающихся на него активных дескрипторов. Во-вторых, иногда возникает потребность запретить передачу данных в одном из направлений. Например, можно запретить

операцию записи, чтобы дать возможность процессу, с которым мы взаимодействуем, определить момент окончания передачи данных, но при этом мы хотели бы сохранить возможность приема данных, которые еще могут быть посланы удаленным процессом.

16.3. Адресация

В предыдущем разделе мы рассмотрели порядок создания и удаления сокетов. Прежде чем двинуться дальше, нужно узнать, как производится идентификация процесса, с которым мы собираемся взаимодействовать. Идентификационная информация состоит из двух частей. Сетевой адрес компьютера позволяет идентифицировать сетевой узел, с которым мы предполагаем вступить в контакт, а номер службы (или номер порта) помогает идентифицировать конкретный процесс на этом компьютере.

16.3.1. Порядок байтов

При организации взаимодействий между процессами, работающими на одной машине, мы обычно не задумываемся о порядке следования байтов. Порядок байтов — это характеристика аппаратной архитектуры, определяющая, в каком порядке следуют байты в данных длинных типов, таких как целые числа. На рис. 16.1 показан порядок байтов в 32-разрядном целом числе.

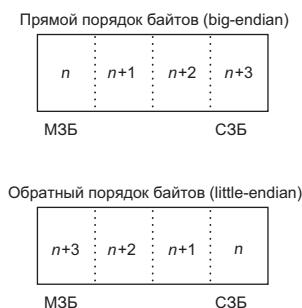


Рис. 16.1. Порядок следования байтов в 32-разрядном целом числе

Если архитектура поддерживает *прямой* (big-endian) порядок байтов, в старшем адресе будет располагаться младший значащий байт (МЗБ). В случае *обратного* (little-endian) порядка байтов младший значащий байт будет храниться в старшем адресе. Обратите внимание: независимо от порядка байтов старший значащий байт (СЗБ) всегда располагается слева, а младший — справа. То есть если присвоить переменной 32-разрядное целое значение `0x04030201`, старший значащий байт будет иметь значение 4, а младший значащий байт — значение 1, независимо от порядка байтов. Если теперь привести адрес переменной к типу `char*(cp)`, мы сможем наблюдать различия в порядке байтов на разных аппа-

ратных архитектурах. Если архитектура поддерживает обратный (little-endian) порядок байтов, `cp[0]` будет ссылаться на младший значащий байт, который содержит значение 1, а `cp[3]` — на старший значащий байт, имеющий значение 4. Если архитектура поддерживает прямой (big-endian) порядок байтов, `cp[0]` будет ссылаться на старший значащий байт со значением 4, а `cp[3]` — на младший значащий байт со значением 1. В табл. 16.5 показано, какие платформы какой порядок байтов поддерживают.

Таблица 16.5. Порядок байтов на тестовых платформах

Операционная система	Аппаратная архитектура	Порядок следования байтов
FreeBSD 8.0	Intel Pentium	Обратный (little-endian)
Linux 3.2.0	Intel Core i5	Обратный
Mac OS X 10.6.8	Intel Core 2 Duo	Обратный
Solaris 10	Sun SPARC	Прямой (big-endian)

Некоторые типы процессоров допускают возможность изменения порядка байтов, что вносит еще большую путаницу.

Чтобы не возникало путаницы с порядком байтов при обмене данными между разнородными компьютерными системами, сетевые протоколы жестко задают порядок байтов. Набор протоколов TCP/IP использует сетевой (прямой, big-endian) порядок байтов. Порядок байтов приобретает важность, когда приложения начинают обмениваться форматированными данными. При использовании протоколов TCP/IP адреса имеют сетевой порядок байтов, поэтому в приложениях иногда возникает необходимость преобразовать порядок байтов, поддерживаемый аппаратной архитектурой, в сетевой порядок байтов. Такое преобразование обычно производится, например, при выводе адреса в удобочитаемой форме.

Преобразования между сетевым и аппаратным порядком байтов производятся с помощью следующих четырех функций.

```
#include <arpa/inet.h>
```

```
uint32_t htonl(uint32_t hostint32);
```

Возвращает 32-разрядное целое с сетевым порядком байтов

```
uint16_t htons(uint16_t hostint16);
```

Возвращает 16-разрядное целое с сетевым порядком байтов

```
uint32_t ntohl(uint32_t netint32);
```

Возвращает 32-разрядное целое с аппаратным порядком байтов

```
uint16_t ntohs(uint16_t netint16);
```

Возвращает 16-разрядное целое с аппаратным порядком байтов

В именах функций буква `n` означает «network» (сетевой порядок байтов), а `h` — «host» (аппаратный). Буква `l` означает «long» (длинное, то есть 4-байтное целое), а `s` — «short» (короткое, то есть 2-байтное целое). Хотя для доступа к этим функциям мы обычно подключаем заголовочный файл `<arpa/inet.h>`, реализации систем часто определяют их в других заголовочных файлах, подключаемых файлом `<arpa/inet.h>`. Также системы часто реализуют эти функции как макросы.

16.3.2. Форматы адресов

Адреса используются для идентификации сокетов в конкретном домене. Для каждого домена определен свой формат представления адреса. Чтобы адреса различных форматов могли передаваться функциям, работающим с сокетами, выполняется приведение адресов к обобщенной структуре адреса `sockaddr`:

```
struct sockaddr {
    sa_family_t sa_family; /* семейство адресов */
    char        sa_data[]; /* адрес переменной длины */
    ...
};
```

Реализации могут дополнять эту структуру своими полями и определять размер поля `sa_data`. Например, в Linux эта структура определена так:

```
struct sockaddr {
    sa_family_t sa_family; /* семейство адресов */
    char        sa_data[14]; /* адрес переменной длины */
};
```

а в FreeBSD так:

```
struct sockaddr {
    unsigned char sa_len;      /* общая длина */
    sa_family_t   sa_family;    /* семейство адресов */
    char         sa_data[14];   /* адрес переменной длины */
};
```

Формат представления адресов Интернета определен в заголовочном файле `<netinet/in.h>`. Адреса сокетов из домена IPv4 (`AF_INET`) представлены структурой `sockaddr_in`:

```
struct in_addr {
    in_addr_t s_addr; /* адрес IPv4 */
};

struct sockaddr_in {
    sa_family_t   sin_family; /* семейство адресов */
    in_port_t     sin_port;   /* номер порта */
    struct in_addr sin_addr; /* адрес IPv4 */
};
```

Тип данных `in_port_t` определен как `uint16_t`, а тип `in_addr_t` — как `uint32_t`. Эти целочисленные типы задают количество используемых разрядов и определены в заголовочном файле `<stdint.h>`.

В отличие от домена AF_INET, адреса сокетов домена Интернета IPv6 (AF_INET6) представлены структурой sockaddr_in6:

```
struct in6_addr {
    uint8_t s6_addr[16]; /* адрес IPv6 */
};

struct sockaddr_in6 {
    sa_family_t     sin6_family;    /* семейство адресов */
    in_port_t       sin6_port;      /* номер порта */
    uint32_t        sin6_flowinfo;  /* класс трафика и сведения о потоке */
    struct in6_addr sin6_addr;     /* адрес IPv6 */
    uint32_t        sin6_scope_id;  /* идентификатор области видимости */
};
```

Это определения, которые требует стандарт Single UNIX Specification. Реализации могут добавлять в эти структуры дополнительные поля. Например, в Linux структура sockaddr_in определена так:

```
struct sockaddr_in {
    sa_family_t     sin_family;    /* семейство адресов */
    in_port_t       sin_port;      /* номер порта */
    struct in_addr  sin_addr;     /* адрес IPv4 */
    unsigned char   sin_zero[8];   /* заполнитель */
};
```

где поле `sin_zero` является заполнителем и должно содержать только нулевые значения.

Обратите внимание, что хотя структуры `sockaddr_in` и `sockaddr_in6` совершенно различны, обе они приводятся к типу `sockaddr` при передаче функциям, работающим с сокетами. В разделе 17.2 мы увидим, что структура представления адресов сокетов домена UNIX отличается от обеих структур представления адресов домена Интернета.

Иногда бывает необходимо вывести адреса в виде, удобном для человека. Сетевое программное обеспечение BSD включало функции `inet_ntoa` и `inet_addr`, преобразующие адреса между двоичным представлением и представлением в виде строки в десятично-точечной нотации (a.b.c.d). Однако эти функции могут работать только с адресами IPv4. Позднее появились две новые функции — `inet_ntop` и `inet_pton`, которые имели аналогичную функциональность, но могли работать также с адресами IPv6.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

Возвращает указатель на строку с адресом в случае успеха,
NULL — в случае ошибки

```
int inet_pton(int domain, const char *restrict str, void *restrict addr);
```

Возвращает 1 в случае успеха, 0 — при неверном формате,
-1 — в случае ошибки

Функция `inet_ntop` преобразует адрес из двоичного представления с сетевым порядком байтов в текстовую строку. Функция `inet_pton` преобразует текстовую строку в двоичное представление с сетевым порядком байтов. Эти функции поддерживают только два значения аргумента `domain`: `AF_INET` и `AF_INET6`.

Аргумент `size` функции `inet_ntop` задает размер буфера (`str`), в котором будет размещена строка. Для удобства существуют две константы: `INET_ADDRSTRLEN`, которая определяет размер буфера, достаточный для хранения строки с адресом IPv4, и `INET6_ADDRSTRLEN`, которая определяет размер буфера, достаточный для хранения строки с адресом IPv6. Аргумент `addr` функции `inet_pton` должен содержать адрес буфера достаточного размера для хранения 32-разрядного адреса, если в аргументе `domain` передается значение `AF_INET`, и 128-разрядного адреса, если в аргументе `domain` передается значение `AF_INET6`.

16.3.3. Определение адреса

В идеале приложения ничего не должны знать о внутренней структуре адреса со-кета. Если приложение просто передает адреса в виде структуры `sockaddr` и не использует какие-либо специфические для протокола особенности, оно сможет работать с самыми разными протоколами, которые предоставляют один и тот же вид услуги.

Сетевая подсистема BSD традиционно предоставляла интерфейсы для доступа к различной информации о конфигурации сети. В разделе 6.7 мы уже вкратце рассмотрели некоторые файлы с сетевой информацией и функции для работы с эти-ми файлами. В этом разделе мы обсудим их подробнее и рассмотрим новые функции, применяемые для поиска адресной информации.

Информация о конфигурации сети может храниться в статических файлах (`/etc/hosts`, `/etc/services` и др.) или предоставляться различными сетевыми службами, такими как DNS (Domain Name System — система доменных имен) и NIS (Network Information Service — сетевая информационная служба). Независимо от того, где хранится информация, для доступа к ней используются одни и те же функции.

Адреса хостов, известных заданной системе, можно получить с помощью функции `gethostent`.

```
#include <netdb.h>
struct hostent *gethostent(void);
```

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
void sethostent(int stayopen);
void endhostent(void);
```

Функция `gethostent` возвращает очередную запись из файла с данными об адресах. Если файл еще не открыт, функция `gethostent` откроет его. Функция

`sethostent` открывает файл или переходит в его начало, если он уже открыт. Когда в аргументе `stayopen` передается ненулевое значение, файл останется открытым после вызова `gethostent`. Функция `endhostent` закрывает файл.

Когда функция `gethostent` возвращает управление, мы получаем указатель на структуру `hostent`, которая может размещаться в области статической памяти, которая будет затерта при следующем обращении к этой функции. Структура `hostent` содержит как минимум следующие поля:

```
struct hostent {
    char      *h_name;      /* имя хоста */
    char    **h_aliases;   /* указатель на массив псевдонимов */
    int       h_addrtype;  /* тип адреса */
    int       h_length;    /* длина адреса в байтах */
    char    **h_addr_list; /* указатель на массив сетевых адресов */
    ...
};
```

Возвращаемые адреса имеют сетевой порядок байтов.

Существуют еще две функции, `gethostbyname` и `gethostbyaddr`, которые также работают со структурами `hostent`, но в настоящее время считаются устаревшими. Они были удалены из версии 4 стандарта Single UNIX Specification. Вскоре мы рассмотрим функции, которые пришли им на смену.

С помощью следующего набора функций можно получить имена сетей и их номера.

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);

All functions return a pointer to a hostent structure on success, or NULL on error.

void setnetent(int stayopen);
void endnetent(void);
```

Структура `netent` содержит как минимум следующие поля:

```
struct netent {
    char      *n_name;      /* имя сети */
    char    **n_aliases;   /* указатель на массив псевдонимов сети */
    int       n_addrtype;  /* тип адреса */
    uint32_t  n_net;        /* номер сети */
    ...
};
```

Возвращаемый номер сети имеет сетевой порядок байтов. Тип адреса — одна из констант, определяющих семейство адресов (например, `AF_INET`).

Преобразования между именами протоколов и их номерами производятся с помощью следующих функций.

```
#include <netdb.h>

struct protoent *getprotobynumber(const char *name);

struct protoent *getprotoent(void);
```

Все возвращают указатель в случае успеха, NULL – в случае ошибки

```
void setprotoent(int stayopen);

void endprotoent(void);
```

Структура **protoent** определена стандартом POSIX.1 и должна содержать как минимум следующие поля:

```
struct protoent {
    char    *p_name;      /* имя протокола */
    char   **p_aliases; /* указатель на массив псевдонимов протокола */
    int     p_proto;     /* номер протокола */
    ...
};
```

Службы определяются номером порта, который является частью адреса. Каждой сетевой службе присвоен свой уникальный номер порта. Получить номер порта по имени службы можно с помощью функции **getservbyname**, а имя службы по номеру порта – с помощью функции **getservbyport**. С помощью функции **getservent** можно последовательно просмотреть все записи в базе данных служб.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);
```

Все возвращают указатель в случае успеха, NULL – в случае ошибки

```
void setservent(int stayopen);

void endservent(void);
```

Структура **servent** содержит как минимум следующие поля:

```
struct servent {
    char    *s_name;      /* имя службы */
    char   **s_aliases; /* указатель на массив псевдонимов службы */
    int     s_port;       /* номер порта */
    char    *s_proto;     /* имя протокола */
    ...
};
```

Стандарт POSIX.1 определяет ряд новых функций, которые позволяют получать сетевой адрес по имени хоста и имени службы, и наоборот. Эти функции заменили устаревшие `gethostbyname` и `gethostbyaddr`.

Функция `getaddrinfo` позволяет получить адрес по имени хоста и сетевой службы.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host, const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);

void freeaddrinfo(struct addrinfo *ai);
```

Возвращает 0 в случае успеха, неотрицательный
код ошибки — в случае неудачи

Мы должны передать функции имя хоста, имя службы или и то и другое. Если передается только одно имя, второе должно быть пустым указателем. Имя хоста может быть именем сетевого узла или адресом в десятично-точечной нотации.

Функция `getaddrinfo` возвращает связанный список структур `addrinfo`. Функция `freeaddrinfo` используется для освобождения памяти, занимаемой списком этих структур, связанных между собой через поле `ai_next`.

Структура `addrinfo` содержит как минимум следующие поля:

```
struct addrinfo {
    int          ai_flags;      /* флаги */
    int          ai_family;     /* семейство адресов */
    int          ai_socktype;   /* тип сокета */
    int          ai_protocol;   /* протокол */
    socklen_t    ai_addrlen;   /* длина адреса в байтах */
    struct sockaddr *ai_addr;   /* адрес */
    char        *ai_canonname; /* каноническое имя хоста */
    struct addrinfo *ai_next;   /* следующий элемент списка */
    ...
};
```

Таблица 16.6. Флаги для структуры `addrinfo`

Флаг	Описание
<code>AI_ADDRCONFIG</code>	Запрос типа адреса (IPv4 или IPv6)
<code>AI_ALL</code>	Поиск обоих типов адресов — IPv4 и IPv6 (используется только совместно с флагом <code>AI_V4MAPPED</code>)
<code>AI_CANONNAME</code>	Запрос канонического имени (в противоположность псевдониму)
<code>AI_NUMERICHOST</code>	Вернуть адрес в числовом формате
<code>AI_NUMERICSERV</code>	Вернуть службу в виде номера порта
<code>AI_PASSIVE</code>	Сокет предназначен для работы в режиме прослушивания
<code>AI_V4MAPPED</code>	Если адреса IPv6 не найдены, возвращать адреса IPv4 в формате IPv6

Аргумент *hint* можно использовать для задания дополнительных критериев выбора адресов. Этот аргумент представляет шаблон для фильтрации адресов, в котором используются только поля *ai_family*, *ai_flags*, *ai_protocol* и *ai_socktype*. Остальные поля целочисленного типа должны содержать значения 0, а указатели — **NULL**. В табл. 16.6 перечисляются флаги, которые можно использовать в поле *ai_flags*, и их назначение.

Если вызов `getaddrinfo` завершится ошибкой, мы не сможем воспользоваться функцией `perror` или `strerror`, чтобы получить текст сообщения об ошибке.

Вместо них для преобразования кода ошибки в текстовое представление нужно пользоваться функцией `gai_strerror`.

```
#include <netdb.h>
const char *gai_strerror(int error);
```

Возвращает указатель на строку с описанием ошибки

Функция `getnameinfo` преобразует адрес в имя хоста и имя сетевой службы.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                char *restrict host, socklen_t hostlen,
                char *restrict service, socklen_t servlen, int flags);
```

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

Адрес сокета (*addr*) преобразуется в имя хоста и имя сетевой службы. Если в аргументе *host* передается непустой указатель, он должен указывать на буфер, размер которого указывается в аргументе *hostlen*. Имя хоста будет возвращено в этом буфере. Аналогично, если аргумент *service* не является пустым указателем, он указывает на буфер размером *servlen* байт, в котором будет возвращено имя сетевой службы. С помощью аргумента *flags* можно влиять на порядок преобразования. В табл. 16.7 перечислены поддерживаемые значения этого аргумента.

Таблица 16.7. Флаги для функции `getnameinfo`

Флаг	Описание
<code>NI_DGRAM</code>	Служба основана на интерфейсе дейтаграмм, а не потоков
<code>NI_NAMEREQD</code>	Если имя хоста не найдено, считать это ошибкой
<code>NI_NOFQDN</code>	Для локальных хостов вместо полного доменного имени возвращать только имя узла
<code>NI_NUMERICHOST</code>	Вместо имени хоста возвращать его адрес в числовой форме
<code>NI_NUMERICSCOPE</code>	Для IPv6. Вместо имени хоста вернуть его идентификатор области действия адреса (Scope ID) в числовой форме
<code>NI_NUMERICSERV</code>	Возвращать имя службы в числовом представлении (то есть номер порта)

Пример

В листинге 16.1 показан пример использования функции `getaddrinfo`.

Листинг 16.1. Вывод сведений о хостах и сетевых службах

```
#include "apue.h"
#if defined(SOLARIS)
#include <netinet/in.h>
#endif
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" семейство ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
        printf("не определено");
        break;
    default:
        printf("неизвестно");
    }
}

void
print_type(struct addrinfo *aip)
{
    printf(" тип ");
    switch (aip->ai_socktype) {
    case SOCK_STREAM:
        printf("stream");
        break;
    case SOCK_DGRAM:
        printf("datagram");
        break;
    case SOCK_SEQPACKET:
        printf("seqpacket");
        break;
    case SOCK_RAW:
        printf("raw");
        break;
    default:
        printf("неизвестный (%d)", aip->ai_socktype);
    }
}

void
```

```
print_protocol(struct addrinfo *aip)
{
    printf(" протокол ");
    switch (aip->ai_protocol) {
        case 0:
            printf("по умолчанию");
            break;
        case IPPROTO_TCP:
            printf("TCP");
            break;
        case IPPROTO_UDP:
            printf("UDP");
            break;
        case IPPROTO_RAW:
            printf("raw");
            break;
        default:
            printf("неизвестный (%d)", aip->ai_protocol);
    }
}

void
print_flags(struct addrinfo *aip)
{
    printf("флаги");
    if (aip->ai_flags == 0) {
        printf(" 0");
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
        if (aip->ai_flags & AI_ALL)
            printf(" all");
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo    *ailist, *aip;
    struct addrinfo    hint;
    struct sockaddr_in *sinp;
    const char         *addr;
    int                err;
    char               abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("Использование: %s имя_узла служба", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
```

```

hint.ai_addr = NULL;
hint.ai_next = NULL;
if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
    err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));
for (aip = ailist; aip != NULL; aip = aip->ai_next) {
    print_flags(aip);
    print_family(aip);
    print_type(aip);
    print_protocol(aip);
    printf("\n\tхост %s", aip->ai_canonname?aip->ai_canonname:"-");
    if (aip->ai_family == AF_INET) {
        sinp = (struct sockaddr_in *)aip->ai_addr;
        addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                         INET_ADDRSTRLEN);
        printf(" адрес %s", addr?addr:"не известен");
        printf(" порт %d", ntohs(sinp->sin_port));
    }
    printf("\n");
}
exit(0);
}

```

Эта программа иллюстрирует работу с функцией `getaddrinfo`. Если заданный хост предоставляет заданную службу по нескольким протоколам, программа выведет несколько записей. В нашем примере выводится адресная информация только для протоколов IPv4 (`ai_family` имеет значение `AF_INET`). Если необходимо ограничиться только семейством протоколов `AF_INET`, следует записать это значение в поле `ai_family` аргумента `hint`.

После запуска программы на одной из наших тестовых систем мы получили:

```

$ ./a.out harry nfs
флаги canon семейство inet тип stream протокол TCP
    хост harry адрес 192.168.1.99 порт 2049
флаги canon семейство inet тип datagram протокол UDP
    хост harry адрес 192.168.1.99 порт 2049

```

16.3.4. Присваивание адресов сокетам

Адрес, присваиваемый клиентскому сокету, не представляет для нас особого интереса, потому мы можем позволить системе выбирать адрес по умолчанию. Однако для сервера важно присвоить сокету предопределенный адрес, на который клиенты будут присыпать запросы. Клиентам необходимо заранее знать требуемый адрес, чтобы войти в контакт с сервером, и самое простое решение заключается в том, чтобы зарезервировать адрес сервера в файле `/etc/services` или в службе имен.

Присвоить адрес сокету можно с помощью функции `bind`.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Существует несколько ограничений, касающихся адресов:

- Указываемый адрес должен быть действительным адресом для машины, на которой выполняется процесс, — нельзя задать адрес, который принадлежит другой машине.
- Формат адреса должен совпадать с форматом, который поддерживается семейством адресов, указанным при создании сокета.
- Номер порта не может быть меньше 1024, если процесс не имеет соответствующих привилегий (например, привилегий суперпользователя).
- Обычно каждый конкретный адрес может быть связан только с одним сокетом, хотя некоторые протоколы допускают присвоение одного и того же адреса нескольким сокетам.

В домене Интернета имеется специальный IP-адрес `INADDR_ANY`, который соответствует адресам всех сетевых интерфейсов в системе. Это означает, что существует возможность принимать пакеты с любого сетевого интерфейса, установленного в системе. В следующем разделе мы увидим, что система сама может присвоить адрес сокету при обращении к функциям `connect` и `listen`.

Получить адрес, присвоенный сокету, можно с помощью функции `getsockname`.

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

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Перед вызовом `getsockname` необходимо занести в аргумент `alenp` адрес целого числа, которое определяет размер буфера `addr`. По возвращении из функции это число будет содержать фактический размер полученного адреса. Если адрес не умещается в предоставленный буфер, он будет усечен. Если сокету не присвоен адрес, результат функции неопределен.

Если сокет соединен с удаленным узлом, мы можем получить адрес удаленного узла, обратившись к функции `getpeername`.

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

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Кроме того, что эта функция возвращает адрес удаленной стороны, она полностью идентична функции `getsockname`.

16.4. Установка соединения

Если мы имеем дело с сетевой службой, которая ориентирована на установление логического соединения (`SOCK_STREAM` или `SOCK_SEQPACKET`), прежде чем начать обмениваться данными, необходимо установить соединение между сокетом процесса, посылающего запрос (клиентом), и процессом, предоставляющим услугу (сервером). Для создания соединения используется функция `connect`.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Адрес, который передается функции `connect`, — это адрес сервера, с которым предполагается установить связь. Если сокету `sockfd` не присвоен какой-либо адрес, функция присвоит ему адрес по умолчанию.

Попытка соединения с сервером может потерпеть неудачу по нескольким причинам. Машина, с которой устанавливается соединение, должна быть включена и связана с сетью. Серверу должен быть присвоен адрес, с которым мы пытаемся соединиться, и в очереди запросов на соединение на стороне сервера должно быть достаточно места, чтобы поставить в очередь наш запрос (вскоре мы поговорим об этом более подробно). То есть приложение должно уметь обрабатывать возможные ошибки соединения.

Пример

В листинге 16.2 показан пример обработки ошибочных ситуаций, возникающих при попытке установить соединение. Такие ошибки наиболее вероятны при попытке связаться с сервером, испытывающим сильные нагрузки.

Листинг 16.2. Попытка соединения с повторением

```
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int      numsec;

    /*
     * Попытаться установить соединение с экспоненциальной задержкой.
     */
    for (numsec = 1; nsec <= MAXSLEEP; nsec <= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Соединение установлено.
             */
            return(0);
        }
    }
}
```

```

    }

    /*
     * Задержка перед следующей попыткой.
     */
    if (numsec <= MAXSLEEP/2)
        sleep(nsec);
}
return(-1);
}

```

Эта функция демонстрирует известный алгоритм с экспоненциальной задержкой. Если функция `connect` терпит неудачу, процесс приостанавливается на короткое время и затем повторяет попытку, всякий раз увеличивая время задержки, пока оно не достигнет максимума — около 2 минут.

Решение, представленное в листинге 16.2, обладает одним недостатком: оно не переносимо. Данный прием можно использовать только в Linux и Solaris, но он не даст положительного результата в FreeBSD и Mac OS X. Если первая попытка установить соединение потерпит неудачу, реализация сокетов на основе BSD будет продолжать объявлять неудачными успешные попытки соединения, если использовать тот же самый дескриптор сокета с протоколом TCP. Это результат просачивания в приложение зависимости поведения реализации от типа протокола через (независимый от протокола) интерфейс сокетов. Причины такого поведения корнями уходят в прошлое, и именно поэтому стандарт Single UNIX Specification предупреждает, что состояние сокета не определено в случае неудачной попытки установить соединение.

Поэтому переносимые приложения должны закрывать сокеты после неудачной попытки установить соединение. Если требуется повторить попытку, следует открыть новый сокет. В листинге 16.3 представлен более переносимый прием.

Листинг 16.3. Переносимый способ соединения с повторением

```

#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int domain, int type, int protocol,
              const struct sockaddr *addr, socklen_t alen)
{
    int      numsec, fd;

    /*
     * Попытаться установить соединение с экспоненциальной задержкой.
     */
    for (numsec = 1; numsec <= MAXSLEEP; numsec <<= 1) {
        if ((fd = socket(domain, type, protocol)) < 0)
            return(-1);
        if (connect(fd, addr, alen) == 0) {
            /*
             * Соединение установлено.
             */
            return(fd);
        }
    }
}

```

```

    }
    close(fd);

    /*
     * Задержка перед следующей попыткой.
     */
    if (numsec <= MAXSLEEP/2)
        sleep(numsec);
}
return(-1);
}

```

Обратите внимание: из-за того что может потребоваться создать новый сокет, нет смысла передавать дескриптор сокета функции `connect_retry`. Новая версия возвращает теперь не признак успеха, а дескриптор сокета, подключенного к удаленной стороне.

Если сокет находится в неблокирующем режиме, который мы обсудим в разделе 16.8, и соединение не может быть установлено немедленно, функция `connect` вернет значение `-1` и код ошибки `EINPROGRESS` в переменной `errno`. Приложение может определить, когда дескриптор станет доступен для записи, с помощью функции `poll` или `select`. В этот момент установление соединения будет завершено.

Функция `connect` также может использоваться для работы со службами, которые не требуют установления соединения (`SOCK_DGRAM`). Казалось бы, здесь кроется какое-то противоречие, но это не так, поскольку это своего рода оптимизация. Если вызвать функцию `connect` для сокета `SOCK_DGRAM`, для всех исходящих пакетов будет установлен адрес, который мы передадим функции `connect`, что освобождает нас от необходимости указывать адрес при передаче каждой дейтаграммы. Кроме того, мы будем получать дейтаграммы только с указанного адреса.

С помощью функции `listen` сервер заявляет о своем желании принимать запросы на установление соединения.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Возвращает 0 в случае успеха, `-1` — в случае ошибки

Аргумент `backlog` определяет желаемое количество ожидающих обработки запросов, которые должны быть поставлены в очередь от имени процесса. Фактическое значение определяется самой системой, но верхний предел определен под именем `SOMAXCONN` в заголовочном файле `<sys/socket.h>`.

B Solaris предел `SOMAXCONN` из `<sys/socket.h>` игнорируется системой. Значение этого предела зависит от реализации каждого конкретного протокола. Так, для TCP этот предел равен 128.

После заполнения очереди система будет отвергать дополнительные запросы на соединение, поэтому значение `backlog` должно выбираться с учетом возможной нагрузки на сервер и объема ресурсов, необходимого для принятия запроса и запуска службы.

После вызова функции `listen` указанный сокет будет использоваться для приема запросов на соединение. Функция `accept` принимает запрос и преобразует его в соединение.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Возвращает дескриптор файла (сокета) в случае успеха,
-1 — в случае ошибки

Функция `accept` возвращает дескриптор сокета, соединенного с клиентом, вызвавшим функцию `connect`. Этот новый сокет имеет тот же тип и семейство адресов, что и сокет `sockfd`. Первоначальный сокет, который передается функции `accept`, не связан с установленным соединением, он остается свободным для приема последующих запросов на соединение.

Если нас не беспокоит проблема идентификации клиента, мы можем передать в аргументах `addr` и `len` значение `NULL`. Иначе необходимо передать в `addr` адрес буфера достаточного размера для хранения адреса, а в аргументе `len` — адрес целого числа, определяющего размер буфера. По возвращении из функции `accept` в буфере будет находиться адрес клиента, а по адресу `len` — фактический размер адреса.

Если запросы, ожидающие обработки, отсутствуют, функция `accept` будет заблокирована, пока не поступит хотя бы один запрос. Если `sockfd` находится в неблокирующем режиме, функция `accept` вернет значение `-1` и код ошибки `EAGAIN` или `EWOULDBLOCK` в переменной `errno`.

На всех четырех платформах, обсуждаемых в этой книге, константа `EAGAIN` определена с тем же значением, что и `EWOULDBLOCK`.

Если сервер вызовет функцию `accept` при отсутствии запросов на соединение, он окажется заблокированным, пока не придет хотя бы один запрос. Как вариант сервер может использовать функцию `poll` или `select` для ожидания прибытия запросов. В этом случае сокет, содержащий запросы на соединение, будет выглядеть как доступный для чтения.

Пример

В листинге 16.4 приводится функция, которая размещает и инициализирует сокет для серверного процесса.

Листинг 16.4. Инициализация сокета для сервера

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen, int qlen)
{
    int      fd;
    int      err = 0;
```

```

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0)
        goto errout;
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET){
        if (listen(fd, qlen) < 0)
            goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}

```

Позднее мы увидим, что несколько странные правила протокола TCP, касающиеся многократного использования адреса, делают этот пример неадекватным. В листинге 16.10 приводится другая версия функции, которая обходит эти правила и исправляет главный недостаток данной версии.

16.5. Передача данных

Поскольку сокет представлен файловым дескриптором, мы можем использовать функции `read` и `write`, когда он соединен с удаленной стороной. Мы уже говорили, что сокеты типа `SOCK_DGRAM` также могут находиться в состоянии «установленного соединения», если с помощью функции `connect` был задан адрес удаленного узла по умолчанию. Возможность использовать функции `read` и `write` для работы с дескрипторами сокетов является большим плюсом, так как это означает, что мы можем передавать дескрипторы сокетов функциям, которые изначально проектировались для работы с локальными файлами. Кроме того, дескрипторы сокетов можно передавать дочерним процессам, которые запускают программы, ничего не знающие о сокетах. Мы можем использовать функции `read` и `write` для обмена данными через сокеты, но это практически все, что возможно. Если нам понадобится определить какие-либо дополнительные возможности, принимать пакеты от нескольких клиентов или передавать экстренные данные, придется использовать одну из шести функций, специально разработанных для передачи данных через сокеты.

Три функции из этих шести предназначены для передачи данных, а три — для приема. В первую очередь рассмотрим функции передачи данных.

Самая простая из них — функция `send`. Она похожа на функцию `write`, но позволяет указать дополнительные флаги, влияющие на процесс передачи.

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

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

Возвращает количество отправленных байтов в случае успеха,
–1 — в случае ошибки

Как и в случае с функцией `write`, к моменту вызова функции `send` сокет должен быть соединен с удаленной стороной. Аргументы `buf` и `nbytes` имеют тот же смысл, что и для функции `write`.

Однако, в отличие от `write`, функция `send` поддерживает дополнительный аргумент `flags`. Стандарт Single UNIX Specification определяет три флага, но большинство реализаций поддерживают дополнительные флаги. Перечень флагов приводится в табл. 16.8.

Успешное завершение функции `send` еще не означает, что процесс на другом конце соединения получил отправленные данные. Все, что гарантирует функция `send` в случае успеха, — это отсутствие ошибок при передаче данных сетевым драйверам.

Если при использовании протокола, который ограничивает размер сообщения, попытаться послать сообщение размером больше максимально допустимого, функция `send` вернет признак ошибки с кодом `EMSGSIZE` в переменной `errno`. При использовании протоколов, которые поддерживают обмен данными в виде потоков байтов, функция `send` будет заблокирована, пока не передаст весь объем данных. Функция `sendto` напоминает функцию `send` и отличается лишь тем, что позволяет указать адрес получателя при работе с сокетами типа `SOCK_DGRAM`.

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
```

Возвращает количество отправленных байтов в случае успеха,
–1 — в случае ошибки

При использовании сокетов, ориентированных на установление соединения, адрес получателя игнорируется, так как он определяется самим соединением. Для обслуживания сокетов, которые не создают соединение, нельзя использовать функцию `send`, если предварительно не была вызвана функция `connect`, поэтому `sendto` предоставляет альтернативный способ передачи данных.

Таблица 16.8. Флаги, используемые функцией `send`

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>MSG_CONFIRM</code>	Сообщить уровню соединения о необходимости сохранить отображение адреса действительным			✓		
<code>MSG_DONTROUTE</code>	Не отправлять пакет за пределы локальной сети		✓	✓	✓	✓
<code>MSG_DONTWAIT</code>	Разрешить неблокирующий режим выполнения операции (эквивалент флага <code>O_NONBLOCK</code>)		✓	✓	✓	✓

Таблица 16.8 (окончание)

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
MSG_EOF	Разорвать соединение на отправляющей стороне после отправки данных		✓		✓	
MSG_EOR	Обозначает конец записи, если поддерживается протоколом	✓	✓	✓	✓	✓
MSG_MORE	Отложить отправку пакета, чтобы дать возможность записать дополнительные данные			✓		
MSG_NOSIGNAL	Не генерировать сигнал SIGPIPE при попытке записи в неподключенный сокет	✓	✓	✓		
MSG_OOB	Обозначает передачу внеочередных данных, если поддерживается протоколом (раздел 16.7)	✓	✓	✓	✓	✓

В нашем распоряжении имеется еще одна функция, предназначенная для передачи данных через сокет. Функция `sendmsg` принимает структуру `msghdr`, которая определяет сразу несколько буферов с данными для передачи, что делает ее похожей на `writev` (раздел 14.6).

```
#include <sys/socket.h>
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Возвращает количество отправленных байтов в случае успеха,
–1 – в случае ошибки

Согласно стандарту POSIX.1, структура `msghdr` должна содержать как минимум следующие поля:

```
struct msghdr {
    void        *msg_name;      /* необязательный адрес */
    socklen_t   msg_namelen;    /* размер адреса в байтах */
    struct iovec *msg iov;     /* массив буферов ввода/вывода */
    int         msg iovlen;    /* количество элементов в массиве */
    void        *msg_control;   /* вспомогательные данные */
    socklen_t   msg_controllen; /* объем вспомогательных данных в байтах */
    int         msg_flags;     /* флаги принятого сообщения */
    ...
};
```

Мы уже рассматривали структуру `iovec` в разделе 14.6. Назначение вспомогательных данных будет рассмотрено в разделе 17.4.

Функция `recv` похожа на функцию `read`, но позволяет указать дополнительные флаги, влияющие на процесс приема данных.

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

Возвращает длину сообщения в байтах, 0 — при отсутствии доступных сообщений и если на удаленном конце соединения запрещена операция записи в сокет, -1 — в случае ошибки

Перечень флагов, которые можно передать функции `recv`, приводится в табл. 16.9. Только три из них определены стандартом Single UNIX Specification.

Таблица 16.9. Флаги, используемые функцией `recv`

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>MSG_CMSG_CLOEXEC</code>	Устанавливать флаг закрытия при вызове <code>exec</code> (<code>close-on-exec</code>) для файловых дескрипторов, принимаемых через сокеты (раздел 17.4)			✓		
<code>MSG_DONTWAIT</code>	Разрешить неблокирующий режим выполнения операции (эквивалент флага <code>O_NONBLOCK</code>)		✓	✓		✓
<code>MSG_ERRQUEUE</code>	Принимать информацию об ошибках в виде дополнительных данных			✓		
<code>MSG_OOB</code>	Принять внеочередные данные, если поддерживается протоколом (раздел 16.7)	✓	✓	✓	✓	✓
<code>MSG_PEEK</code>	Вернуть содержимое пакета, но не удалять его из приемной очереди	✓	✓	✓	✓	✓
<code>MSG_TRUNC</code>	Запросить фактический размер пакета, даже если он был обрезан			✓		
<code>MSG_WAITALL</code>	Ждать получения всех данных (только для <code>SOCK_STREAM</code>)	✓	✓	✓	✓	✓

Если указан флаг `MSG_PEEK`, можно «подсмотреть» содержимое следующего сообщения, не удаляя его из приемной очереди. Эти данные будут повторно получены при следующем обращении к функции `read` или к одной из функций `recv`.

При использовании сокетов типа `SOCK_STREAM` можно получить меньший объем данных, чем было запрошено. Флаг `MSG_WAITALL` запрещает такое поведение функции `recv`, заставляя ее дождаться получения всего запрошенного объема данных. Для сокетов типа `SOCK_DGRAM` и `SOCK_SEQPACKET` флаг `MSG_WAITALL` не оказывает влияния на поведение функции `recv`, поскольку для них за одну операцию чтения всегда возвращается сообщение целиком.

Если отправитель вызвал функцию `shutdown` (раздел 16.2), чтобы завершить передачу данных, или если протокол поддерживает завершение передачи по умолчанию и отправитель закрыл свой сокет, функция `recv` вернет 0 после получения всех данных.

Чтобы получить сведения об отправителе, можно воспользоваться функцией `recvfrom`, возвращающей адрес, с которого произведена передача данных.

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

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr, socklen_t *restrict addrlen);
```

Возвращает длину сообщения в байтах, 0 — при отсутствии доступных сообщений и если на удаленном конце соединения запрещена операция записи в сокет, -1 — в случае ошибки

Если аргумент `addr` содержит непустой указатель, по указанному адресу будет записан адрес сокета, с которого отправлены данные. При вызове `recvfrom` необходимо передать в аргументе `addrlen` указатель на целое число с размером буфера `addr`. По возвращении из функции это число будет содержать фактический размер адреса в байтах.

Так как функция позволяет получить адрес отправителя, она обычно используется для работы с сокетами типа `SOCK_DGRAM`. В остальном функция `recvfrom` ничем не отличается от `recv`.

Чтобы принять данные сразу в несколько буферов, как это делает функция `readv` (раздел 14.6), или получить вспомогательные данные (раздел 17.4), можно использовать функцию `recvmsg`.

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Возвращает длину сообщения в байтах, 0 — при отсутствии доступных сообщений и если на удаленном конце соединения запрещена операция записи в сокет, -1 — в случае ошибки

Перечень приемных буферов определяется структурой `msghdr` (которая также используется с функцией `sendmsg`). Чтобы изменить поведение по умолчанию функции `recvmsg`, можно установить дополнительные флаги в аргументе `flags`. По возвращении из функции поле `msg_flags` структуры `msghdr` будет хранить раз-

личные характеристики полученных данных. (На входе в `recvmsg` поле `msg_flags` игнорируется.) Перечень флагов, которые может вернуть `recvmsg`, приводится в табл. 16.10. Пример использования этой функции мы увидим в главе 17.

Таблица 16.10. Флаги, возвращаемые функцией `recvmsg` в поле `msg_flags`

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>MSG_CTRUNC</code>	Управляющая информация обрезана	✓	✓	✓	✓	✓
<code>MSG_EOR</code>	Получен признак конца записи	✓	✓	✓	✓	✓
<code>MSG_ERRQUEUE</code>	Принята информация об ошибках в виде дополнительных данных			✓		
<code>MSG_OOB</code>	Приняты экстренные данные	✓	✓	✓	✓	✓
<code>MSG_TRUNC</code>	Данные обрезаны	✓	✓	✓	✓	✓

Пример — клиент, ориентированный на создание соединения

В листинге 16.5 приводится исходный код клиентской программы, которая запрашивает у сервера результат выполнения команды `uptime`. Мы назвали эту службу «remote uptime» (удаленный `uptime`), или для краткости «`ruptime`».

Листинг 16.5. Клиент, получающий результат выполнения команды `uptime` на сервере

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUflen 128

extern int connect_retry(int, int, int const struct sockaddr *, socklen_t);

void
print_uptime(int sockfd)
{
    int      n;
    char    buf[BUflen];

    while ((n = recv(sockfd, buf, BUflen, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("ошибка вызова функции recv");
}

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

```

struct addrinfo *ailist, *aip;
struct addrinfo hint;
int sockfd, err;

if (argc != 2)
    err_quit("Использование: ruptime hostname");
memset(&hint, 0, sizeof(hint));
hint.ai_socktype = SOCK_STREAM;
hint.ai_canonname = NULL;
hint.ai_addr = NULL;
hint.ai_next = NULL;
if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
    err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));
for (aip = ailist; aip != NULL; aip = aip->ai_next) {
    if ((sockfd = connect_retry(aip->ai_family, SOCK_STREAM, 0,
                                aip->ai_addr, aip->ai_addrlen)) < 0) {
        err = errno;
    } else {
        print_uptime(sockfd);
        exit(0);
    }
}
err_exit(err, "невозможно соединиться с %s", argv[1]);
}

```

Эта программа соединяется с сервером, читает переданную им строку и выводит ее в стандартный вывод. Так как в программе используется сокет типа `SOCK_STREAM`, нет уверенности, что строка будет прочитана целиком за одно обращение к функции `recv`, поэтому попытки получения данных повторяются в цикле, пока функция не вернет 0.

Функция `getaddrinfo` может вернуть несколько адресов, если сервер поддерживает несколько сетевых интерфейсов или несколько сетевых протоколов. Мы пробуем соединиться поочередно с каждым из них, пока не будет найден адрес требуемой службы. Для установки соединения с сервером используется функция `connect_retry` из листинга 16.3.

Пример — сервер, ориентированный на создание соединения

В листинге 16.6 приводится исходный код сервера, который возвращает результат команды `uptime` по запросу клиента из листинга 16.5.

Листинг 16.6. Сервер, возвращающий результат команды `uptime` по запросу клиента

```

#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUflen 128
#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

```

```
extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int      clfd;
    FILE    *fp;
    char    buf[BUFSIZE];

    set_cloexec(sockfd);
    for (;;) {
        if ((clfd = accept(sockfd, NULL, NULL)) < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции accept: %s",
                   strerror(errno));
            exit(1);
        }
        set_cloexec(clfd);
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "ошибка: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFSIZE, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("Использование: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX; /* лучшее, что можно сделать */
    if ((host = malloc(n)) == NULL)
        err_sys("ошибка вызова функции malloc");
    if (gethostname(host, n) < 0)
        err_sys("ошибка вызова функции gethostname");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
               gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
```

```

        aip->ai_addrlen, QLEN)) >= 0) {
    serve(sockfd);
    exit(0);
}
exit(1);
}

```

Чтобы определить собственный адрес, сервер должен получить сетевое имя компьютера, на котором он запущен. Если максимальная длина имени хоста не определена, мы используем константу `HOST_NAME_MAX`. Если система не определяет константу `HOST_NAME_MAX`, мы задаем ее самостоятельно. Стандарт POSIX.1 указывает, что минимальное значение длины сетевого имени хоста должно быть равно 255 байт без учета завершающего нулевого символа, поэтому мы определяем константу `HOST_NAME_MAX` со значением 256 — с учетом завершающего нулевого символа.

Сервер получает сетевое имя хоста с помощью функции `gethostname` и отыскивает адрес службы `uptime`. Функция `getaddrinfo` может вернуть несколько адресов, но для простоты мы выбираем первый из них, на котором будет возможно установить пассивный сокет. Обработку нескольких адресов мы оставляем вам в качестве упражнения.

Для инициализации сокета мы использовали функцию `initserver` из листинга 16.4. Этот сокет будет ожидать поступления запросов на соединение. (Фактически мы использовали версию функции из листинга 16.10, почему — вы узнаете, когда мы перейдем к обсуждению параметров сокетов в разделе 16.6.)

Пример — альтернативный сервер, ориентированный на создание соединения

Ранее мы уже говорили, что возможность использования файловых дескрипторов для организации доступа к сокетам играет важную роль, так как позволяет использовать для работы в сети программы, которые ничего не знают о сетях. Программа в листинге 16.7 как раз является таким сервером. Вместо того чтобы читать данные со стандартного вывода команды `uptime` и передавать их клиенту, сервер связывает стандартный вывод и стандартный вывод сообщений об ошибках команды `uptime` с сокетом, который соединен с клиентом.

Листинг 16.7. Сервер, демонстрирующий запись вывода команды прямо в сокет

```

#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

```

```
extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int      clfd, status;
    pid_t   pid;

    set_cloexec(sockfd);
    for (;;) {
        if((clfd = accept(sockfd, NULL, NULL)) < 0){
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции accept: %s",
                   strerror(errno));
            exit(1);
        }
        if ((pid = fork()) < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции fork: %s",
                   strerror(errno));
            exit(1);
        } else if (pid == 0) { /* дочерний процесс */
            /*
             * Родительский процесс вызвал функцию daemonize (листинг 13.1),
             * поэтому STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO уже открыты
             * на устройстве /dev/null. В результате нет необходимости защищать
             * вызов close проверкой на равенство clfd одному из этих значений.
             */
            if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
                dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                syslog(LOG_ERR, "ruptimed: неожиданная ошибка");
                exit(1);
            }
            close(clfd);
            execl("/usr/bin/uptime", "uptime", (char *)0);
            syslog(LOG_ERR, "ruptimed: неожиданный возврат из exec: %s",
                   strerror(errno));
        } else { /* родительский процесс */
            close(clfd);
            waitpid(pid, &status, 0);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int          sockfd, err, n;
    char         *host;

    if (argc != 1)
        err_quit("Использование: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX; /* лучшее, что можно сделать */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
```

```

memset(&hint, 0, sizeof(hint));
hint.ai_flags = AI_CANONNAME;
hint.ai_socktype = SOCK_STREAM;
hint.ai_canonname = NULL;
hint.ai_addr = NULL;
hint.ai_next = NULL;
if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
    syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
           gai_strerror(err));
    exit(1);
}
for (aip = ailist; aip != NULL; aip = aip->ai_next) {
    if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
                             aip->ai_addrlen, QLEN)) >= 0) {
        serve(sockfd);
        exit(0);
    }
}
exit(1);
}

```

Чтобы запустить команду `uptime` и получить ее вывод, мы вместо `ropen` вызвали функцию `fork` и запустили дочерний процесс, который затем с помощью `dup2` связал дескрипторы `STDOUT_FILENO` и `STDERR_FILENO` с сокетом. Команда `uptime` выводит результаты в стандартный вывод, который связан с сокетом, и данные отправляются клиенту.

Родительский процесс может закрыть свой дескриптор сокета, соединенный с клиентом, поскольку дочерний процесс удержит его открытым. Родительский процесс ожидает завершения дочернего процесса, что предотвращает появление зомби. Так как время работы команды `uptime` невелико, родительский процесс может позволить себе дождаться завершения потомка, прежде чем перейти к приему следующего запроса на соединение. Такая стратегия может оказаться неприемлемой, если дочерний процесс выполняется достаточно продолжительное время.

В предыдущем примере использовался сокет, ориентированный на создание логического соединения. Но как правильно выбрать тип сокета? В каких случаях следует использовать сокеты, ориентированные либо не ориентированные на создание соединения? Ответ зависит от того, какой объем работы предполагается выполнить и насколько приложение чувствительно к ошибкам.

При использовании сокетов, не ориентированных на создание соединений, пакеты могут прибывать не в том порядке, в каком они были отправлены. Поэтому если все данные не смогут уместиться в один пакет, придется побеспокоиться о порядке доставки пакетов. Максимальный размер пакета является характеристикой используемого протокола. Кроме того, следует учитывать, что при использовании сокетов, не ориентированных на создание соединений, пакеты могут теряться. Если логика приложения не допускает таких потерь, необходимо использовать сокеты, ориентированные на создание соединений.

Есть два способа сделать приложение нечувствительным к потере пакетов. Если необходимо обеспечить надежную связь с удаленной стороной, можно пронумеровать пакеты и запрашивать повторную передачу отсутствующего пакета при обнаружении потери. Кроме того, потребуется предусмотреть обработку дубликатов

пакетов, поскольку пакет может задержаться, а приложение может посчитать, что он потерян, и запросить повторную передачу, после чего могут быть доставлены оба пакета.

Второй вариант — разрешить пользователю повторить команду при появлении ошибки. Для простых приложений такой вариант может оказаться вполне приемлемым, но для сложных программ он не подходит — лучше использовать сокеты, ориентированные на создание логических соединений.

Один из недостатков сокетов, ориентированных на создание логических соединений, состоит в том, что для установки соединения необходим больший объем работы и требуется больше времени, а кроме того, каждое соединение потребляет больше ресурсов операционной системы.

Пример — клиент, не ориентированный на создание соединения

Программа в листинге 16.8 является версией клиента из листинга 16.5, использующей интерфейс дейтаграмм.

Листинг 16.8. Клиент, использующий интерфейс дейтаграмм

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUflen 128
#define TIMEOUT 20

void
sigalrm(int signo)
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int      n;
    char    buf[BUflen];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("ошибка вызова функции sendto");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUflen, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("ошибка вызова функции recv");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}

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

```

struct addrinfo    *ailist, *aip;
struct addrinfo    hint;
int                sockfd, err;
struct sigaction   sa;

if (argc != 2)
    err_quit("Использование: ruptime hostname");
sa.sa_handler = sigalarm;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGALRM, &sa, NULL) < 0)
    err_sys("ошибка вызова функции sigaction");
memset(&hint, 0, sizeof(hint));
hint.ai_socktype = SOCK_DGRAM;
hint.ai_canonname = NULL;
hint.ai_addr = NULL;
hint.ai_next = NULL;
if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
    err_quit("ошибка вызова функции getaddrinfo: %s", gai_strerror(err));

for (aip = ailist; aip != NULL; aip = aip->ai_next) {
    if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
        err = errno;
    } else {
        print_uptime(sockfd, aip);
        exit(0);
    }
}
fprintf(stderr, "невозможно соединиться с %s: %s\n", argv[1],
        strerror(err));
exit(1);
}

```

Функция `main` в этой версии клиента практически не изменилась, добавилась только установка обработчика сигнала `SIGALRM`. Функция `alarm` используется, чтобы предотвратить блокировку вызова функции `recvfrom` на длительное время.

При использовании протокола, ориентированного на создание соединения, мы должны были подключиться к серверу до начала обмена данными. Для сервера достаточно получить запрос на соединение, чтобы понять, что он должен обслужить клиента. Но при использовании протокола передачи дейтаграмм необходимо оповестить сервер, что мы хотим получить услугу. В этом примере мы просто посылаем серверу 1-байтное сообщение. Сервер принимает его, извлекает из пакета наш адрес и по этому адресу отправляет ответ. Если бы сервер предоставлял несколько услуг, мы могли бы в запросе посыпать идентификатор требуемой услуги, но поскольку наш сервер выполняет всего одну команду, содержимое 1-байтного сообщения не играет никакой роли.

Если сервер не запущен, клиент может оказаться заблокированным на неопределенное время в функции `recvfrom`. В предыдущем примере, ориентированном на создание соединения, функция `connect` возвращает управление с признаком ошибки, если сервер не отвечает. Во избежание блокировки на неопределенное время мы устанавливаем таймер перед вызовом функции `recvfrom`.

Пример — сервер, не ориентированный на создание соединения

В листинге 16.9 приводится исходный код версии сервера `uptime`, которая реализует обмен дейтаграммами.

Листинг 16.9. Сервер, который реализует службу `uptime` на основе обмена дейтаграммами

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUflen 128
#define MAXADDRLEN 256

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int             n;
    socklen_t       alen;
    FILE           *fp;
    char            buf[BUflen];
    char            abuf[MAXADDRLEN];
    struct sockaddr *addr = (struct sockaddr *)abuf;

    set_cloexec(sockfd);
    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUflen, 0, addr, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: ошибка вызова функции recvfrom: %s",
                   strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "ошибка: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0, addr, alen);
        } else {
            if (fgets(buf, BUflen, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0, addr, alen);
            pclose(fp);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
```

```

int          sockfd, err, n;
char        *host;

if (argc != 1)
    err_quit("Использование: ruptimed");
if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
    n = HOST_NAME_MAX; /* лучшее, что можно сделать */
if ((host = malloc(n)) == NULL)
    err_sys("ошибка вызова функции malloc");
if (gethostname(host, n) < 0)
    err_sys("ошибка вызова функции gethostname");
daemonize("ruptimed");
memset(&hint, 0, sizeof(hint));
hint.ai_flags = AI_CANONNAME;
hint.ai_socktype = SOCK_DGRAM;
hint.ai_canonname = NULL;
hint.ai_addr = NULL;
hint.ai_next = NULL;
if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
    syslog(LOG_ERR, "ruptimed: ошибка вызова функции getaddrinfo: %s",
           gai_strerror(err));
    exit(1);
}
for (aip = ailist; aip != NULL; aip = aip->ai_next) {
    if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
                           aip->ai_addrlen, 0)) >= 0) {
        serve(sockfd);
        exit(0);
    }
}
exit(1);
}

```

Сервер блокируется в функции `recvfrom` в ожидании прибытия запроса. Когда приходит запрос, сервер извлекает адрес отправителя и с помощью функции `popen` запускает команду `uptime`. Результат работы команды с помощью функции `sendto` отправляется клиенту по адресу, откуда пришел запрос.

16.6. Параметры сокетов

Механизм сокетов предоставляет две функции доступа к параметрам сокетов, которые управляют их поведением. Одна функция используется для изменения, а другая возвращает текущие значения параметров. Мы можем получить и изменить три типа параметров.

1. Универсальные параметры, присущие всем типам сокетов.
2. Параметры, которые поддерживаются на уровне сокета, но зависят от используемого протокола.
3. Параметры, уникальные для каждого отдельно взятого протокола.

Стандарт Single UNIX Specification определяет только параметры, которые поддерживаются на уровне сокетов (первые два типа параметров из предыдущего списка).

Изменить параметры сокета можно с помощью функции `setsockopt`.

```
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);
```

Возвращает 0 в случае успеха, -1 – в случае ошибки

Аргумент *level* определяет протокол, на который будет воздействовать параметр. Если параметр относится к разряду универсальных, в аргументе *level* передается значение `SOL_SOCKET`. Иначе в аргументе *level* должен быть записан номер протокола, например `IPPROTO_TCP` для протокола TCP и `IPPROTO_IP` для протокола IP. В табл. 16.11 приводится перечень универсальных параметров, которые определены стандартом Single UNIX Specification.

Таблица 16.11. Параметры сокетов

Параметр	Тип аргумента <i>val</i>	Описание
<code>SO_ACCEPTCONN</code>	<code>int</code>	Определяет, находится ли сокет в режиме приема запросов на соединение (только для <code>getsockopt</code>)
<code>SO_BROADCAST</code>	<code>int</code>	Допускается передача широковещательных дейтаграмм, если значение <i>*val</i> не равно нулю
<code>SO_DEBUG</code>	<code>int</code>	Сетевому драйверу разрешена запись отладочной информации, если значение <i>*val</i> не равно нулю
<code>SO_DONTROUTE</code>	<code>int</code>	Передавать сообщения в обход процедуры маршрутизации, если значение <i>*val</i> не равно нулю
<code>SO_ERROR</code>	<code>int</code>	Получить ибросить состояние ошибки, ожидающей обработки (только для <code>getsockopt</code>)
<code>SO_KEEPALIVE</code>	<code>int</code>	Разрешена периодическая передача служебных сообщений для поддержания соединения в активном состоянии, если значение <i>*val</i> не равно нулю
<code>SO_LINGER</code>	<code>struct linger</code>	Время задержки закрытия сокета, если в нем имеются неотправленные сообщения
<code>SO_OOBINLINE</code>	<code>int</code>	Экстренные сообщения помещаются во входной поток, если значение <i>*val</i> не равно нулю
<code>SO_RCVBUF</code>	<code>int</code>	Размер приемного буфера в байтах
<code>SO_RCVLOWAT</code>	<code>int</code>	Минимальный объем данных, который должен возвращаться функциями приема
<code>SO_RCVTIMEO</code>	<code>struct timeval</code>	Максимальное время ожидания для операций чтения из сокета
<code>SO_REUSEADDR</code>	<code>int</code>	Разрешает повторное использование локальных адресов функцией <code>bind</code> , если значение <i>*val</i> не равно нулю
<code>SO_SNDBUF</code>	<code>int</code>	Размер буфера передачи в байтах

Таблица 16.11 (окончание)

SO_SNDLOWAT	int	Минимальный объем данных в байтах, который должен передаваться функциями отправки
SO_SNDFTIMEO	struct timeval	Максимальное время ожидания для операций записи в сокет
SO_TYPE	int	Тип сокета (только для getsockopt)

Аргумент *val* может содержать указатель на целое число или на структуру, в зависимости от параметра. Некоторые параметры являются флагами, которые могут иметь только два значения — включено и выключено. Если целое число не равно нулю, параметр включен. Если целое число равно нулю, параметр выключен. Аргумент *len* определяет размер объекта, на который указывает *val*.

Текущие значения параметров можно получить с помощью функции `getsockopt`.

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

int getsockopt(int sockfd, int Level, int option, void *restrict val,
               socklen_t *restrict lenp);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Аргумент *lenp* — это указатель на целое число. Перед вызовом функции `getsockopt` нужно установить это число равным размеру буфера, куда будет скопировано текущее значение параметра. Если фактический размер параметра больше размера буфера, параметр будет обрезан. Если фактический размер параметра меньше размера буфера, по адресу *lenp* будет записано фактическое значение размера параметра.

Пример

Функция в листинге 16.4 не отрабатывает ситуацию, когда сервер завершается и мы пытаемся перезапустить его. Как правило, реализация протокола TCP не позволяет присвоить тот же адрес другому сокету, пока не пройдет определенный промежуток времени, который обычно составляет несколько минут. К счастью, это ограничение легко обойти с помощью параметра `SO_REUSEADDR`, как показано в листинге 16.10.

Листинг 16.10. Инициализация сокета для сервера с возможностью повторного использования адреса

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen, int qlen)
{
    int      fd, err;
    int      reuse = 1;
```

```

if ((fd = socket(addr->sa_family, type, 0)) < 0)
    return(-1);
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
               sizeof(int)) < 0)
    goto errout;
if (bind(fd, addr, alen) < 0)
    goto errout;
if (type == SOCK_STREAM || type == SOCK_SEQPACKET)
    if (listen(fd, qlen) < 0)
        goto errout;
return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}

```

Чтобы разрешить повторное использование адреса, необходимо установить параметр `SO_REUSEADDR`, для этого мы записываем в переменную ненулевое значение и передаем ее адрес функции `setsockopt` в аргументе `val`. В аргументе `len` передается размер целого числа, определяющего размер объекта, на который указывает `val`.

16.7. Экстренные данные

Передача экстренных данных — это необязательная функциональная возможность, поддерживаемая некоторыми протоколами, которая позволяет производить доставку высокоприоритетных данных. Экстренные данные отправляются в первую очередь. Протокол TCP поддерживает такую возможность, а UDP — нет. Интерфейс доступа к экстренным данным в сокетах очень тесно связан с реализацией экстренных данных в протоколе TCP.

Протокол TCP называет экстренные данные «срочными» (`urgent`). Он поддерживает только однобайтные срочные данные, но позволяет доставлять их в первую очередь. Чтобы послать срочные данные, нужно передать флаг `MSG_OOB` любой из трех функций `send`. Если с флагом `MSG_OOB` передается более одного байта срочных данных, только последний байт в сообщении будет воспринят как срочные данные.

Можно предусмотреть посылку сигнала `SIGURG` при получении срочных данных. В разделах 3.14 и 14.5.2 мы уже видели, что для этого можно использовать команду `F_SETOWN` функции `fcntl`, которая назначает владельца дескриптора. Третий аргумент функции `fcntl` задает идентификатор процесса, если он является положительным числом, и группу процессов — если отрицательным числом, отличным от `-1`. То есть мы можем указать, что процесс должен получать сигнал от сокета, вызвав

```
fcntl(sockfd, F_SETOWN, pid);
```

Чтобы узнать, какой процесс владеет сокетом, можно использовать команду `F_GETOWN`. Как и в случае команды `F_SETOWN`, отрицательное значение представляется

идентификатор группы процессов, а положительное — идентификатор процесса. Таким образом, вызов

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

запишет в переменную `owner` идентификатор процесса, который будет получать сигналы от сокета, если возвращаемое значение положительное, а если оно отрицательное, абсолютное значение переменной `owner` будет соответствовать идентификатору группы процессов, которая будет получать сигналы от сокета.

Протокол TCP поддерживает понятие *маркера срочности*: это позиция в потоке обычных данных, куда помещаются срочные данные. Можно задать такой режим приема срочных данных, когда они размещаются в потоке обычных данных, для чего необходимо включить параметр `SO_OOBINLINE`. Проверить достижение маркера срочных данных можно с помощью функции `sockatmark`.

```
#include <sys/socket.h>
int sockatmark(int sockfd);
```

Возвращает 1, если маркер достигнут, 0 — если нет,
–1 — в случае ошибки

Если следующий байт, доступный для чтения, находится в позиции маркера срочности, функция `sockatmark` вернет значение 1.

При наличии в сокете срочных данных функция `select` (раздел 14.4.1) вернет дескриптор файла как имеющий исключительную ситуацию, ожидающую обработки. У нас есть возможность выбора: получать срочные данные в потоке обычных данных или же с помощью одной из функций `recv`, используя флаг `MSG_OOB` для выборки срочных данных в первую очередь. Протокол TCP помещает в очередь только один байт срочных данных. Если другой срочный байт прибудет до того, как мы получим текущий, существующий байт будет утерян.

16.8. Неблокирующий и асинхронный ввод/вывод

Если в сокете нет данных, доступных для чтения, выполнение функции `recv` обычно блокируется. Аналогично блокируется и функция `send`, если в выходной очереди сокета не хватает места для отправляемого сообщения.

Если сокет находится в неблокирующем режиме, поведение функций изменяется. В этом случае данные функции не блокируются и возвращают признак ошибки с кодом `EWOULDBLOCK` или `EAGAIN` в переменной `errno`.

В такой ситуации для определения момента, когда можно принять или послать данные, можно использовать функцию `poll` или `select`.

Стандарт Single UNIX Specification включает поддержку обобщенного механизма асинхронного ввода/вывода. Механизм сокетов обрабатывает асинхрон-

ный ввод/вывод собственным способом, но он не стандартизован в Single UNIX Specification. В некоторых книгах классический механизм асинхронного ввода/вывода для сокетов называется «ввод/вывод, основанный на сигналах», чтобы отличать его от механизма асинхронного ввода/вывода в стандарте Single UNIX Specification.

При использовании механизма асинхронного ввода/вывода для сокетов можно организовать посылку сигнала `SIGIO`, когда в сокете появятся доступные для чтения данные или освободится место в выходной очереди. Процесс включения механизма асинхронного ввода/вывода состоит из двух действий.

1. Назначить владельца сокета так, чтобы сигнал доставлялся соответствующему процессу.
2. Информировать сокет, чтобы он посыпал сигнал, когда выполнение операций ввода/вывода не будет блокировать процесс.

Первое действие можно выполнить тремя способами.

1. Воспользоваться командой `F_SETOWN` функции `fcntl`.
2. Воспользоваться командой `FIOSETOWN` функции `ioctl`.
3. Воспользоваться командой `SIOCSPGRP` функции `ioctl`.

Второе действие можно выполнить двумя способами:

1. С помощью команды `F_SETFL` функции `fcntl` установить флаг `O_ASYNC`.
2. Воспользоваться командой `FIOASYNC` функции `ioctl`.

У нас на выбор несколько вариантов, но они не универсальны. В табл. 16.12 показано, какие из вариантов на каких платформах поддерживаются.

Таблица 16.12. Команды управления режимом асинхронного ввода/вывода для сокетов

Механизм	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>fcntl(fd, F_SETOWN, pid)</code>	✓	✓	✓	✓	✓
<code>ioctl(fd, FIOSETOWN, pid)</code>		✓	✓	✓	✓
<code>ioctl(fd, SIOCSPGRP, pid)</code>		✓	✓	✓	✓
<code>fcntl(fd, F_SETFL, flags O_ASYNC)</code>		✓	✓	✓	
<code>ioctl(fd, FIOASYNC, &n)</code>		✓	✓	✓	✓

16.9. Подведение итогов

В этой главе мы рассмотрели механизмы IPC, которые позволяют процессам обмениваться данными с другими процессами, выполняющимися как на других машинах, так и на той же самой машине. Мы узнали, как назначить сокету адрес и как получить адреса, используемые для соединения с серверами.

Мы привели примеры клиентов и серверов, которые используют сокеты, ориентированные и не ориентированные (интерфейсдейтаграмм) на создание логического соединения. Коротко обсудили неблокирующий и асинхронный ввод/вывод для сокетов и функции, которые используются для работы с параметрами сокетов. В следующей главе мы рассмотрим ряд более сложных тем, связанных с IPC, включая передачу дескрипторов файлов между процессами, выполняющимися на одной машине.

Упражнения

- 16.1** Напишите программу, которая определяла бы порядок байтов, используемый системой.
- 16.2** Напишите программу для вывода полей структуры `stat`, которые поддерживаются для сокетов. Запустите ее хотя бы на двух различных платформах и опишите, какие различия вы обнаружили.
- 16.3** Программа в листинге 16.6 предоставляет услугу только через один сокет. Измените ее так, чтобы она поддерживала обслуживание через несколько сокетов одновременно (каждый из которых имеет свой адрес).
- 16.4** Напишите программы клиента и сервера, с помощью которых можно получать количество процессов, работающих на заданной машине.
- 16.5** В программе из листинга 16.7 сервер ожидает, пока дочерний процесс запустит команду `uptime` и завершится, прежде чем принять очередной запрос на соединение. Перепишите сервер так, чтобы он мог принимать входящие запросы на соединение без задержки.
- 16.6** Напишите две библиотечные процедуры, первая из которых должна включать асинхронный режим ввода/вывода для сокета, а вторая — выключать его. Воспользуйтесь табл. 16.12, чтобы обеспечить работоспособность этих функций на всех платформах и возможность обрабатывать как можно больше типов сокетов.

17

Расширенные возможности IPC

17.1. Введение

В предыдущих двух главах мы обсудили различные формы IPC, включая каналы и сокеты. В этой главе рассматривается дополнительная форма IPC — сокеты домена UNIX. С помощью этой формы IPC процессы могут передавать друг другу открытые файловые дескрипторы, серверы могут присваивать имена своим файловым дескрипторам, а клиенты — использовать эти имена для взаимодействия с серверами. Кроме того, мы увидим, как операционная система может обеспечить уникальность канала связи с каждым клиентом.

17.2. Сокеты домена UNIX

Сокеты домена UNIX используются для взаимодействий процессов, работающих на одной и той же машине. Сокеты домена Интернета также могут служить для этих целей, но сокеты домена UNIX выполняют эту работу более эффективно. Сокеты домена UNIX просто копируют данные — они никак не обрабатывают сетевые протоколы, не удаляют и не добавляют никаких заголовков пакетов, не вычисляют контрольные суммы, не генерируют последовательные номера и не высылают подтверждения о приеме.

Сокеты домена UNIX реализуют два интерфейса — интерфейс дейтаграмм и интерфейс потоков. При этом интерфейс дейтаграмм домена UNIX гарантирует доставку пакета получателю. Сообщения никогда не теряются или всегда доставляются в порядке отправки. Сокеты домена UNIX сочетают в себе особенности сокетов и неименованных каналов. Для взаимодействия с сокетом домена UNIX можно использовать интерфейс сетевого сокета или создать пару неименованных, связанных между собой сокетов домена UNIX с помощью функции `socketpair`.

```
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Возвращает 0 в случае успеха, -1 — в случае ошибки

Несмотря на то что интерфейс выглядит достаточно общим, чтобы использовать функцию `socketpair` для создания сокетов произвольного домена, в большинстве операционных систем эта функция поддерживает только домен UNIX.

Пара соединенных друг с другом сокетов домена UNIX действует подобно дуплексному каналу: оба конца соединения открыты для чтения и записи (рис. 17.1). Мы будем называть их «fd-каналы», чтобы отличать от обычных неименованных каналов.

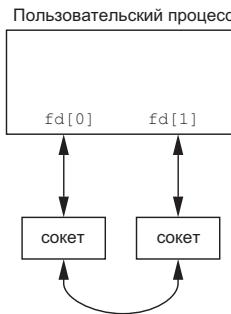


Рис. 17.1. Пара сокетов

Пример — функция `fd_pipe`

В листинге 17.1 приводится функция `fd_pipe`, реализованная на основе функции `socketpair` и создающая пару соединенных между собой сокетов домена UNIX.

Листинг 17.1. Функция создания дуплексного канала

```

#include "apue.h"
#include <sys/socket.h>

/*
 * Возвращает дуплексный канал (сокет домена UNIX)
 * с двумя файловыми дескрипторами fd[0] и fd[1].
 */
int
fd_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}

```

Некоторые системы, производные от BSD, используют сокеты домена UNIX для реализации неименованных каналов. Но при вызове функции `pipe` открытый для записи конец первого дескриптора и открытый для чтения конец второго дескриптора закрываются. Чтобы получить дуплексный канал, необходимо напрямую вызвать функцию `socketpair`.

Пример — опрос очередей сообщений XSI с помощью сокетов домена UNIX

В разделе 15.6.4 рассказывалось об одной проблеме, связанной с использованием очередей сообщений XSI, которая заключается в невозможности использовать функцию `poll` или `select` для их опроса, потому что они не связаны с файловыми дескрипторами. Сокеты, напротив, связаны с файловыми дескрипторами, поэтому их можно использовать для определения моментов прибытия новых сообщений.

В этом примере мы используем отдельный поток для обслуживания каждой очереди. Каждый поток будет блокироваться в вызове функции `msgrecv`. Когда в очередь поступит новое сообщение, поток запишет его в один конец канала на основе сокетов домена UNIX. Наше приложение будет извлекать сообщения из другого конца, когда функция `poll` сообщит, что данные могут быть прочитаны из сокета. Этот прием иллюстрирует программа в листинге 17.2. Функция `main` создает очереди сообщений и сокеты домена UNIX и запускает по одному потоку выполнения для обслуживания каждой очереди. Затем она входит в бесконечный цикл опроса одного из концов канала. Когда сокет будет готов для выполнения операции чтения, функция прочитает данные из него и выведет сообщение на стандартный вывод.

Листинг 17.2. Опрос очередей сообщений XSI с использованием сокетов домена UNIX

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ 3          /* количество очередей*/
#define MAXMSZ 512 /* максимальный размер сообщения */
#define KEY 0x123 /* ключ для первой очереди сообщений */

struct threadinfo {
    int qid;
    int fd;
};

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

void *
helper(void *arg)
{
    int n;
    struct mymesg m;
    struct threadinfo *tip = arg;

    for(;;) {
        memset(&m, 0, sizeof(m));
        if ((n = msgrecv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) < 0)
            err_sys("ошибка вызова функции msgrecv");
        if (write(tip->fd, m.mtext, n) < 0)
            err_sys("ошибка вызова функции write");
    }
}

int
main()
{
    int i, n, err;
    fd[2];
    qid[NQ];
```

```

struct pollfd      pfd[NQ];
struct threadinfo ti[NQ];
pthread_t          tid[NQ];
char               buf[MAXMSZ];

for (i = 0; i < NQ; i++) {
    if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
        err_sys("ошибка вызова функции msgget");

    printf("очередь %d получила идентификатор %d\n", i, qid[i]);

    if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
        err_sys("ошибка вызова функции socketpair");
    pfd[i].fd = fd[0];
    pfd[i].events = POLLIN;
    ti[i].qid = qid[i];
    ti[i].fd = fd[1];
    if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
        err_exit(err, "ошибка вызова функции pthread_create");
}

for (;;) {
    if (poll(pfd, NQ, -1) < 0)
        err_sys("ошибка вызова функции poll");
    for (i = 0; i < NQ; i++) {
        if (pfd[i].revents & POLLIN) {
            if ((n = read(pfd[i].fd, buf, sizeof(buf))) < 0)
                err_sys("ошибка вызова функции read");
            buf[n] = 0;
            printf("очередь: %d, сообщение: %s\n", qid[i], buf);
        }
    }
}
exit(0);
}

```

Обратите внимание, что здесь используются сокеты типа SOCK_DGRAM, а не потоковые сокеты. Это позволило избавиться от проверки границ сообщений, так как одна операция чтения из сокета может прочитать только одно сообщение.

Данный прием дает возможность использовать функцию `poll` или `select` (косвенно) для опроса очередей сообщений. Пока накладные расходы на поддержку одного потока выполнения для каждой очереди и копирование каждого сообщения два лишних раза (один раз при записи в сокет и один раз при чтении из сокета) остаются приемлемыми, такой прием использования очередей сообщений XSI оказывается проще.

Теперь воспользуемся программой в листинге 17.3, чтобы послать сообщения тестовой программе из листинга 17.2.

Листинг 17.3. Отправка сообщений в очередь сообщений XSI

```

#include "apue.h"
#include <sys/msg.h>

#define MAXMSZ 512

struct mymesg {

```

```

long mtype;
char mtext[MAXMSZ];
};

int
main(int argc, char *argv[])
{
    key_t      key;
    long       qid;
    size_t     nbytes;
    struct mymesg m;

    if (argc != 3) {
        fprintf(stderr, "Использование: sendmsg KEY message\n");
        exit(1);
    }
    key = strtol(argv[1], NULL, 0);
    if ((qid = msgget(key, 0)) < 0)
        err_sys("невозможно открыть очередь с ключом %s", argv[1]);
    memset(&m, 0, sizeof(m));
    strncpy(m.mtext, argv[2], MAXMSZ-1);
    nbytes = strlen(m.mtext);
    m.mtype = 1;
    if (msgsnd(qid, &m, nbytes, 0) < 0)
        err_sys("невозможно отправить сообщение");
    exit(0);
}

```

Эта программа принимает два аргумента: ключ очереди и строку для отправки в теле сообщения. Получая сообщения, сервер выводит их, как показано ниже.

```

$ ./pollmsg &                               запустить сервер в фоновом режиме
[1] 12814
$ queue ID 0 is 196608
очередь 1 получила идентификатор 196609
очередь 2 получила идентификатор 196610
$ ./sendmsg 0x123 "привет, мир"           отправить сообщение в первую очередь
очередь: 196608, сообщение: привет, мир
$ ./sendmsg 0x124 "просто тест"           отправить сообщение во вторую очередь
очередь: 196609, сообщение: просто тест
$ ./sendmsg 0x125 "пока"                  отправить сообщение в третью очередь
очередь: 196610, сообщение: пока

```

17.2.1. Именованные сокеты домена UNIX

Хотя функция `socketpair` и создает соединенные друг с другом сокеты, но у них нет имен. Это означает, что их нельзя использовать для взаимодействий независимых процессов.

В разделе 16.3.4 мы узнали, как присваиваются адреса сокетам домена Интернета. Точно так же можно присваивать адреса сокетам домена UNIX и использовать их для получения запросов на соединение. Однако формат адреса сокетов домена UNIX отличается от формата адреса сокетов домена Интернета.

В разделе 16.3 мы уже говорили, что форматы адресов сокетов отличаются в разных реализациях. Адрес сокетов домена UNIX представлен структурой `sockaddr_un`.

В Linux 3.2.0 и Solaris 10 структура `sockaddr_un` объявлена в заголовочном файле `<sys/un.h>` так:

```
struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    char        sun_path[108]; /* полное имя */
};
```

Но в FreeBSD 8.0 и Mac OS X 10.6.8 структура `sockaddr_un` объявлена несколько иначе:

```
struct sockaddr_un {
    unsigned char sun_len;      /* длина адреса сокета */
    sa_family_t   sun_family;   /* AF_UNIX */
    char         sun_path[104]; /* полное имя */
};
```

Поле `sun_path` содержит полное имя файла. Присваивая имя сокету домена UNIX, система создает файл типа `S_IFSOCK` с этим именем.

Этот файл существует, только чтобы сообщить имя сокета клиентам. Сам файл не может быть открыт или как-то иначе использован для взаимодействия приложений.

Если во время попытки присвоить имя сокету файл уже существует, вызов функции `bind` завершится с признаком ошибки. При закрытии сокета этот файл не удаляется автоматически, поэтому мы должны сами побеспокоиться о его удалении перед завершением приложения.

Пример

Программа в листинге 17.4 демонстрирует порядок присваивания адреса сокету домена UNIX.

Листинг 17.4. Присваивание адреса сокету домена UNIX

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int                  fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("ошибка вызова функции socket");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("ошибка вызова функции bind");
    printf("имя сокету домена UNIX присвоено\n");
    exit(0);
}
```

Когда эта программа запустится в первый раз, вызов функции `bind` завершится успехом, но если запустить программу повторно, мы получим сообщение об ошибке, потому что файл с заданным именем уже существует. Программа будет терпеть неудачу, пока файл не будет удален.

```
$ ./a.out                                         запустить программу
имя сокету домена UNIX присвоено
$ ls -l foo.socket                             проверить наличие файла сокета
srwxr-xr-x 1 sar      0 May 18 00:44 foo.socket
$ ./a.out                                         попытаться запустить программу
ошибка вызова функции bind: Address already in use
$ rm foo.socket                                удалить файл сокета
$ ./a.out                                         запустить программу в третий раз
имя сокету домена UNIX присвоено              теперь опять все в порядке
```

Чтобы получить размер адреса, мы определяем смещение поля `sun_path` в структуре `sockaddr_un` и добавляем к нему длину имени без учета завершающего нулевого символа. Так как в различных реализациях полю `sun_path` в структуре `sockaddr_un` может предшествовать разное количество полей, для определения его смещения от начала структуры мы воспользовались макроопределением `offsetof`, которое объявлено в заголовочном файле `<stddef.h>` (подключается в заголовочном файле `arie.h`). В файле `<stddef.h>` вы найдете примерно такое определение:

```
#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)
```

Это выражение вычисляет адрес начала заданного поля в предположении, что структура начинается с адреса 0.

17.3. Уникальные соединения

Сервер может создавать через сокеты домена UNIX уникальные соединения с клиентами, используя стандартные функции `bind`, `listen` и `accept`. Для соединения с сервером клиент может использовать функцию `connect`; после того как сервер примет запрос на соединение, между клиентом и сервером будет установлено уникальное соединение. Этот способ взаимодействия аналогичен тому, что использовался в листингах 16.5 и 16.6.

На рис. 17.2 изображены клиентский и серверный процессы перед установкой соединения между ними. Сервер присвоил своему сокету адрес `sockaddr_un` и ожидает получения запросов на соединение. На рис. 17.3 изображено уникальное соединение между клиентом и сервером после того, как сервер примет запрос на соединение от клиента.

Теперь определим три функции для создания уникальных соединений между независимыми процессами, выполняющимися на одной машине. Эти функции имитируют поведение функций создания логических соединений на основе сокетов, обсуждавшихся в разделе 16.4. Но здесь в качестве механизма взаимодействий мы будем использовать сокеты домена UNIX.

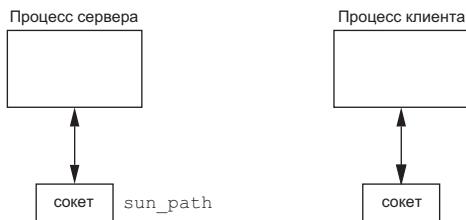


Рис. 17.2. Сокеты клиента и сервера перед установлением соединения

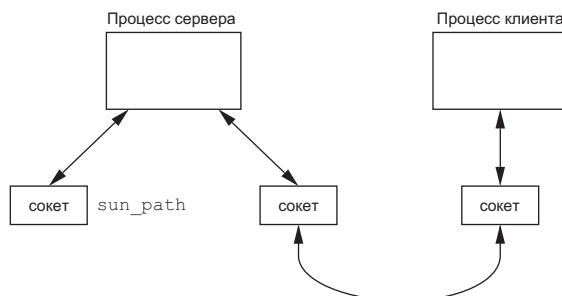


Рис. 17.3. Сокеты клиента и сервера после установления соединения

```
#include "apue.h"

int serv_listen(const char *name);
```

Возвращает файловый дескриптор сокета, ожидающего появления запросов, в случае успеха; отрицательное значение — в случае неудачи

```
int serv_accept(int Listenfd, uid_t *uidptr);
```

Возвращает новый файловый дескриптор в случае успеха; отрицательное значение — в случае неудачи

```
int cli_conn(const char *name);
```

Возвращает новый файловый дескриптор в случае успеха; отрицательное значение — в случае неудачи

Функцию `serv_listen` (листинг 17.5) можно использовать на стороне сервера для приема запросов на соединение по предопределенному имени (некоторый путь в файловой системе). Когда клиенту потребуется соединиться с сервером, он будет использовать это имя. Функция возвращает серверный сокет домена UNIX, готовый к приему запросов от клиентов.

Функция `serv_accept` (листинг 17.6) используется сервером для ожидания запросов на соединение от клиентов. Когда прибывает такой запрос, система автомати-

чески создает новый сокет домена UNIX, соединяет его с сокетом клиента и возвращает новый сокет серверу. Кроме того, эффективный идентификатор клиента сохраняется в памяти по адресу *uidptr*.

Функция *cli_conn* (листинг 17.7) вызывается клиентом, чтобы установить соединение с сервером. Значение аргумента *name*, определяемое клиентом, должно совпадать с именем, назначенным сервером при вызове функции *serv_listen*. Функция возвращает файловый дескриптор сокета, подключенного к серверу.

Листинг 17.5. Функция serv_listen

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define QLEN 10

/*
 * Создает точку соединения на стороне сервера.
 * Возвращает fd в случае успеха, <0 - в случае ошибки.
 */
int
serv_listen(const char *name)
{
    int             fd, len, err, rval;
    struct sockaddr_un un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* создать сокет домена UNIX типа SOCK_STREAM */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-2);

    unlink(name); /* если name уже существует */

    /* заполнить структуру с адресом */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* присвоить имя дескриптору */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -3;
        goto errout;
    }

    /* сообщить ядру, что процесс является сервером */
    if (listen(fd, QLEN) < 0) {
        rval = -4;
        goto errout;
    }
    return(fd);
```

```

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}

```

Сначала с помощью функции `socket` создается сокет домена UNIX. Затем в структуру `sockaddr_un` заносится предопределеное имя, связанное с сокетом. Эта структура будет служить аргументом функции `bind`. Обратите внимание: не нужно записывать значение в поле `sun_len`, которое присутствует на некоторых платформах, потому что операционная система сама сделает это, используя аргумент длины адреса, передаваемый функции `bind`.

В заключение вызывается функция `listen` (раздел 16.4), которая сообщает ядру, что процесс выступает в роли сервера, ожидая запросов на соединение от клиентов. Чтобы принять запрос на соединение, сервер должен вызвать функцию `serv_accept` (листинг 17.6).

Листинг 17.6. Функция serv_accept

```

#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE 30 /* имя, используемое клиентом, не должно быть старше */
               /* этого значения (сек) */

/*
 * Ждет запроса на соединение от клиента и принимает его.
 * Также получает идентификатор пользователя клиента из характеристик файла,
 * который должен быть связан с сокетом перед вызовом сервера.
 * Возвращает новый fd в случае успеха, <0 - в случае ошибки.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, err, rval;
    socklen_t           len;
    time_t              staletime;
    struct sockaddr_un un;
    struct stat          statbuf;
    char                *name;

    /*
     * Выделить объем памяти, достаточный для самого длинного имени
     * с учетом завершающего нулевого символа
     */
    if ((name = malloc(sizeof(un.sun_path) + 1)) == NULL)
        return(-1);
    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0) {
        free(name);
        return(-2); /* чаще всего errno=EINTR, если перехвачен сигнал */
    }
}

```

```

/* получить идентификатор пользователя клиента из адреса вызова */
len -= offsetof(struct sockaddr_un, sun_path); /* длина имени */
memcpy(name, un.sun_path, len);
name[len] = 0; /* завершающий нулевой символ */
if (stat(name, &statbuf) < 0) {
    rval = -3;
    goto errout;
}

#endif
if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
    (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
    rval = -4; /* это не сокет */
    goto errout;
}
if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
    (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
    rval = -5; /* не rwx----- */
    goto errout;
}
staletime = time(NULL) - STALE;
if (statbuf.st_atime < staletime ||
    statbuf.st_ctime < staletime ||
    statbuf.st_mtime < staletime) {
    rval = -6; /* индексный узел слишком стар */
    goto errout;
}

if (uidptr != NULL)
    *uidptr = statbuf.st_uid; /* вернуть UID клиента */
unlink(un.sun_path); /* работа с файлом закончена */
free(name);
return(clifd);

errout:
err = errno;
close(clifd);
free(name);
errno = err;
return(rval);
}

```

Сервер блокируется в функции `accept`, ожидая, пока клиент вызовет функцию `cli_conn`. Когда `accept` вернет управление, мы получим от нее дескриптор, соединенный с клиентом. Кроме того, имя файла, которое клиент присвоит своему сокету (содержащее идентификатор клиентского процесса), также будет возвращено функцией `accept` во втором аргументе (указатель на структуру `sockaddr_un`). Далее мы добавляем завершающий нулевой символ в конец имени файла (если имя займет все пространство в поле `sun_path` структуры `sockaddr_un`, там не останется места для завершающего нулевого байта). Затем вызываем функцию `stat`, чтобы убедиться, что файл действительно является сокетом и права доступа к нему разрешают чтение, запись и выполнение только для владельца файла. Кроме того, проверяем три значения времени, которые не должны превосходить текущее время более чем на 30 секунд. (В разделе 6.10 уже упоминалось, что функция `time`

возвращает текущее время и дату в виде количества секунд, прошедших с начала Эпохи.)

Если все эти проверки увенчались успехом, мы предполагаем, что клиент действительно является владельцем сокета. Эти проверки далеки от совершенства, но это максимум, что можно сделать в современных системах. (Было бы лучше, если бы эффективный идентификатор пользователя возвращался ядром через функцию `accept()`.)

Клиент инициирует соединение с сервером, вызывая функцию `cli_conn` (листинг 17.7).

Листинг 17.7. Функция `cli_conn`

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH "/var/tmp/"
#define CLI_PERM S_IRWXU /* rwx только для владельца */

/*
 * Создает точку соединения на стороне клиента и соединяет ее с сервером.
 * Возвращает fd в случае успеха, <0 - в случае ошибки.
 */
int
cli_conn(const char *name)
{
    int             fd, len, err, rval;
    struct sockaddr_un un, sun;
    int             do_unlink = 0;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* создать сокет домена UNIX типа SOCK_STREAM */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-2);

    /* заполнить структуру с адресом */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05ld", CLI_PATH, (long)getpid());
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path); /* если файл уже существует */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -3;
        goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -4;
        do_unlink = 1;
        goto errout;
    }
}
```

```
/* записать в структуру адрес сервера */
memset(&sun, 0, sizeof(sun));
sun.sun_family = AF_UNIX;
strcpy(sun.sun_path, name);
len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
if (connect(fd, (struct sockaddr *)&sun, len) < 0) {
    rval = -5;
    do_unlink = 1;
    goto errout;
}
return(fd);

errout:
err = errno;
close(fd);
if (do_unlink)
    unlink(un.sun_path);
errno = err;
return(rval);
}
```

Для создания сокета на стороне клиента вызывается функция `socket`. После этого в структуру `sockaddr_un` заносится имя файла, созданного клиентом.

Мы не можем позволить системе выбирать адрес по умолчанию, поскольку в этом случае сервер не сможет отличить одного клиента от другого (если явно не присвоить имя сокету домена UNIX, ядро автоматически присвоит ему адрес по своему усмотрению и в файловой системе не будет создан файл, представляющий сокет). Вместо этого мы присваиваем сокету указанный нами адрес, что обычно не делается при разработке клиентских приложений с использованием сокетов.

Последние пять символов в имени файла отводятся под идентификатор процесса клиента. На всякий случай мы вызываем функцию `unlink`, чтобы гарантировать отсутствие сгенерированного имени в файловой системе. Затем вызывается функция `bind`, которая связывает сокет с заданным именем. При этом она создает файл сокета с тем же именем. После этого функция `chmod` устанавливает права доступа к файлу так, чтобы право на чтение, запись и выполнение имел только владелец файла. Функция `serv_accept`, запускаемая сервером, проверяет эти права и идентифицирует клиента по идентификатору пользователя сокета.

Затем мы заполняем вторую структуру `sockaddr_un`, записывая в нее предопределенное имя сокета сервера. В завершение вызывается функция `connect`, которая инициирует соединение с сервером.

17.4. Передача дескрипторов файлов

Способность передавать дескрипторы файлов между процессами дает неоцененную возможность по-иному подойти к разработке архитектуры клиент-серверных приложений. Она позволяет одному процессу (обычно серверу) выполнять все действия, связанные с открытием файла (включая преобразование сетевых имен в адреса, соединение через модем, установку блокировок на файл и т. п.), и возвращать вызывающему процессу дескриптор, который можно использовать

в операциях ввода/вывода. Такой подход помогает скрыть от клиента все сложные механизмы, связанные с открытием файла.

Нам необходимо уточнить смысл понятия «передача открытого дескриптора файла от одного процесса другому». На рис. 3.2 показан случай, когда два процесса открыли один и тот же файл. Хотя они и используют один и тот же виртуальный узел (*v-node*), каждый из процессов обращается к нему через свою запись в таблице файлов.

Когда открытый дескриптор файла передается от одного процесса другому, необходимо, чтобы передающий и принимающий процессы совместно использовали одну и ту же запись в таблице файлов. На рис. 17.4 показано, что мы хотим получить.

Таблица дескрипторов процесса

	Флаги дескриптора	Указатель на запись в таблице файлов
fd 0:		
fd 1:		
fd 2:		
fd 3:		
...		

Запись в таблице файлов

Флаги состояния файла
Текущая позиция в файле
Указатель на виртуальный узел

Запись в таблице виртуальных узлов

Информация виртуального узла
v_data
Информация индексного узла
Текущий размер файла
i_vnode

Таблица дескрипторов процесса

	Флаги дескриптора	Указатель на запись в таблице файлов
fd 0:		
fd 1:		
fd 2:		
fd 3:		
fd 4:		
...		

Рис. 17.4. Передача открытого дескриптора файла от верхнего процесса нижнему

Технически нам нужно передать указатель на запись в таблице открытых файлов от одного процесса другому. Этот указатель должен быть связан с первым доступным дескриптором в принимающем процессе. (Выражение «передача открытого дескриптора файла» создает ошибочное впечатление, что номер дескриптора в принимающем процессе должен совпадать с номером дескриптора в передающем процессе, — такое возможно, но это не всегда верно.) Ситуация, когда два процесса совместно используют одну и ту же запись в таблице файлов, в точности совпадает с ситуацией, возникающей после вызова функции `fork` (рис. 8.1).

Как правило, после передачи дескриптора от одного процесса другому передающий процесс закрывает свой дескриптор. Закрытие дескриптора в этом случае не означает закрытия файла или устройства, так как считается, что файл остается открытим в принимающем процессе (даже если принимающий процесс еще не успел получить дескриптор).

Сейчас мы определим три функции, которые будут использоваться в этой главе для передачи и приема дескрипторов файлов. Далее в этом разделе мы продемонстрируем код реализации этих трех функций.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errormsg);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Возвращает дескриптор файла в случае успеха, отрицательное значение — в случае ошибки

Процесс (обычно сервер), желающий передать дескриптор другому процессу, вызывает `send_fd` или `send_err`. Процесс, ожидающий получения дескриптора (обычно клиент), вызывает `recv_fd`.

Функция `send_fd` передает дескриптор `fd_to_send` посредством сокета домена UNIX, представленного дескриптором `fd`. Функция `send_err` передает сообщение об ошибке `errormsg`, сопровождаемое кодом ошибки `status`. Значение аргумента `status` должно находиться в диапазоне от -1 до -255.

Для приема дескриптора клиент вызывает функцию `recv_fd`. Если все в порядке (передающий процесс обратился к функции `send_fd`), функция вернет клиенту неотрицательный дескриптор. Иначе возвращаемое значение будет представлять код ошибки `status`, отправленный функцией `send_err` (отрицательное число в диапазоне от -1 до -255). Кроме того, если сервер передал сообщение об ошибке, для его обработки будет вызвана клиентская функция `userfunc`. В первом аргументе этой функции передается константа `STDERR_FILENO`, во втором — указатель на строку сообщения, а в третьем — длина сообщения. Возвращаемое значение `userfunc` — количество записанных байтов или отрицательный код ошибки. Часто в качестве `userfunc` клиентские приложения используют стандартную функцию `write`.

Мы реализуем собственный протокол для этих трех функций. Чтобы передать дескриптор, функция `send_fd` посыпает два нулевых байта, за которыми следует фактический дескриптор. Чтобы передать сообщение об ошибке, функция `send_err` посыпает строку `errormsg`, за которой следуют нулевой байт и абсолютное значение байта с кодом ошибки (1-255). Функция `recv_fd` читает все, что поступает через сокет, пока не встретит нулевой байт. Все символы, прочитанные до этого момента, передаются функции `userfunc`. Следующий байт, который будет прочи-

тан функцией `recv_fd`, — это код ошибки. Если он равен 0, значит, передан дескриптор файла, иначе дескриптор не был получен.

После записи сообщения об ошибке в сокет функция `send_err` вызывает `send_fd`. Реализация `send_err` приводится в листинге 17.8.

Листинг 17.8. Функция `send_err`

```
#include "apue.h"

/*
 * Эта функция используется для передачи сообщения об ошибке с помощью
 * протокола send_fd()/recv_fd(), если при передаче дескриптора возникла ошибка.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int      n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n) /* передать сообщение об ошибке */
            return(-1);

    if (errcode >= 0)
        errcode = -1; /* код ошибки должен быть отрицательным числом */

    if (send_fd(fd, errcode) < 0)
        return(-1);

    return(0);
}
```

Для передачи дескрипторов файлов между процессами с помощью сокетов домена UNIX можно использовать функции `recvmsg(2)` и `sendmsg(2)` (раздел 16.5). Обе принимают указатель на структуру `msghdr`, которая содержит всю необходимую информацию о том, что передается и принимается. Эта структура может выглядеть примерно так:

```
struct msghdr {
    void          *msg_name;      /* необязательный адрес */
    socklen_t     msg_namelen;   /* размер адреса в байтах */
    struct iovec  *msg_iov;       /* массив буферов ввода/вывода */
    int           msg iovlen;    /* количество элементов в массиве */
    void          *msg_control;   /* вспомогательные данные */
    socklen_t     msg_controllen; /* объем вспомогательных данных в байтах */
    int           msg_flags;     /* флаги принятого сообщения */
};
```

Первые два поля обычно используются при передачедейтограмм через сетевое соединение, когда адрес назначения можно указать для каждой дейтограммы. Следующие два поля позволяют определить массив буферов ввода/вывода как для функций `readv` и `writev` (чтение вразброс и запись со слиянием, раздел 14.6). Поле `msg_flags` содержит флаги, описывающие принятое сообщение (перечень флагов приводился в табл. 16.10).

Два поля имеют отношение к передаче и приему управляющей информации. Поле `msg_control` содержит указатель на структуру `cmsgdr` (заголовок блока управля-

иющей информации), а `msg_controllen` — количество байтов управляющей информации.

```
struct cmsghdr {
    socklen_t    cmsg_len;    /* количество байтов данных, включая заголовок */
    int          cmsg_level; /* определяет протокол */
    int          cmsg_type;   /* тип управляющей информации */
    /* далее следует управляющая информация */
};
```

Чтобы передать дескриптор, необходимо записать в поле `cmsg_len` размер структуры `cmsghdr` плюс размер целого числа (дескриптора). В поле `cmsg_level` записывается значение `SOL_SOCKET`, а в поле `cmsg_type` — значение `SCM_RIGHTS`, которое указывает, что передаются права доступа. (Аббревиатура `SCM` означает socketlevel control message — управляющее сообщение уровня сокетов.) Права доступа могут передаваться только через сокеты домена UNIX. Дескриптор следует сразу же за полем `cmsg_type`, а чтобы получить указатель на него, можно воспользоваться макросом `CMSG_DATA`.

Для доступа к управляющей информации используются три макроса, и еще один служит для вычисления значения, которое заносится в поле `cmsg_len`.

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

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Возвращает указатель на данные, связанные со структурой `cmsghdr`

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Возвращает указатель на первую структуру `cmsghdr`, связанную со структурой `msghdr`, или `NULL`, если таких не существует

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
```

Возвращает указатель на следующую структуру `cmsghdr`, связанную со структурой `msghdr`, относительно заданной структуры `cmsghdr` или `NULL`, если такой не существует

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Возвращает объем памяти, который необходимо выделить для хранения объекта размером `nbytes`

В Single UNIX Specification определены первые три макроса, но отсутствует `CMSG_LEN`.

Макрос `CMSG_LEN` возвращает количество байтов, необходимое для хранения данных объемом `nbytes` после добавления размера структуры `cmsghdr` с учетом всех ограничений по выравниванию полей, накладываемых аппаратной архитектурой процессора.

В листинге 17.9 приводится исходный код функции `send_fd`, передающей файловый дескриптор через сокет домена UNIX. Функции `sendmsg` передаются данные протокола (нулевой байт и код ошибки) и дескриптор.

Листинг 17.9. Передача файлового дескриптора через сокет домена UNIX

```
#include "apue.h"
#include <sys/socket.h>

/*
 * размер буфера с управляющей информацией
 * для приема/передачи одного дескриптора
 */
#define CONTROLLEN CMSG_LEN(sizeof(int))

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Передает дескриптор файла другому процессу.
 * Если fd<0, в качестве кода ошибки, отправляется -fd.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* 2-байтный протокол send_fd()/recv_fd() */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control     = NULL;
        msg.msg_controllen  = 0;
        buf[1] = -fd_to_send; /* ненулевое значение означает ошибку */
        if (buf[1] == 0)
            buf[1] = 1; /* протокол преобразует в -256 */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control     = cmptr;
        msg.msg_controllen  = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send; /* записать дескриптор */
        buf[1] = 0; /* нулевое значение означает отсутствие ошибки */
    }

    buf[0] = 0; /* нулевой байт - флаг для recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Чтобы принять дескриптор (листинг 17.10), мы выделяем память для размещения структуры `cmsghdr` и дескриптора, затем помещаем в поле `msg_control` указатель на выделенную память и вызываем функцию `recvmsg`. Для расчета объема выделяемого пространства мы воспользовались макросом `CMSG_LEN`.

Чтение данных из сокета производится до получения нулевого байта, предшествующего заключительному байту с кодом ошибки. Все, что было получено до этого байта, рассматривается как сообщение об ошибке от отправителя.

Листинг 17.10. Прием файлового дескриптора через сокет домена UNIX

```
#include "apue.h"
#include <sys/socket.h> /* struct msghdr */

/*
 * размер буфера с управляющей информацией
 * для приема/передачи одного дескриптора
 */
#define CONTROLLEN CMSG_LEN(sizeof(int))

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Принимает дескриптор файла от серверного процесса. Кроме того, любые
 * принятые данные передаются функции (*userfunc)(STDERR_FILENO, buf, nbytes).
 * Чтобы принять дескриптор, мы должны соблюдать 2-байтный протокол.
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int          newfd, nr, status;
    char         *ptr;
    char         buf[MAXLINE];
    struct iovec iov[1];
    struct msghdr msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control  = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("ошибка вызова функции recvmsg");
            return(-1);
        } else if (nr == 0) {
            err_ret("соединение закрыто сервером");
            return(-1);
        }
        /*
         * Проверить, являются ли два последних байта нулевым байтом
         * и кодом ошибки. Нулевой байт должен быть предпоследним,
         */
    }
}
```



```

    gid_t cmcred_gid; /* реальный идентификатор группы передающего процесса */
    short cmcred_ngroups; /* количество групп */
    gid_t cmcred_groups[CMGROUP_MAX]; /* список групп */
};

 Для передачи идентификационной информации нужно только зарезервировать место в памяти под структуру cmmsgcred. Ядро само заполняет ее, чтобы предотвратить подделку этой информации.

```

В Linux идентификационные сведения передаются в виде структуры `ucred`:

```

struct ucred {
    pid_t pid; /* идентификатор передающего процесса */
    uid_t uid; /* идентификатор пользователя передающего процесса */
    gid_t gid; /* идентификатор группы передающего процесса */
};

 В отличие от FreeBSD, Linux требует, чтобы приложение само инициализировало структуру перед отправкой. Ядро лишь гарантирует, что предоставленные данные соответствуют вызывающему процессу либо он обладает соответствующими привилегиями, чтобы идентифицировать себя таким способом.

```

В листинге 17.11 демонстрируется функция `send_fd`, в которую добавлена возможность передачи идентификационной информации о передающем процессе.

Листинг 17.11. Передача идентификационной информации через сокеты домена UNIX

```

#include "apue.h"
#include <sys/socket.h>

#if defined(SCM_CREDS)           /* интерфейс BSD */
#define CREDSTRUCT cmmsgcred
#define SCM_CREDTYPE SCM_CREDS
#elif defined(SCM_CREDENTIALS) /* интерфейс Linux */
#define CREDSTRUCT ucred
#define SCM_CREDTYPE SCM_CREDENTIALS
#else
#error передача идентификационной информации не поддерживается!
#endif

/*
 * размер буфера с управляющей информацией
 * для приема/передачи одного дескриптора
 */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSSLN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLEN   (RIGHTSLEN + CREDSSLN)

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Передает дескриптор файла другому процессу.
 * Если fd<0, то в качестве кода ошибки отправляется -fd.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT *credp;

```

```

struct cmsghdr      *cmp;
struct iovec         iov[1];
struct msghdr        msg;
char                 buf[2]; /* 2-байтный протокол send_fd/recv_fd */

iov[0].iov_base = buf;
iov[0].iov_len   = 2;
msg.msg_iov       = iov;
msg.msg_iovlen    = 1;
msg.msg_name      = NULL;
msg.msg_namelen   = 0;
msg.msg_flags     = 0;
if (fd_to_send < 0) {
    msg.msg_control    = NULL;
    msg.msg_controllen = 0;
    buf[1] = -fd_to_send; /* ненулевое значение означает наличие ошибки */
    if (buf[1] == 0)
        buf[1] = 1; /* протокол преобразует в -256 */
} else {
    if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
        return(-1);
    msg.msg_control    = cmptr;
    msg.msg_controllen = CONTROLLEN;
    cmp = cmptr;
    cmp->cmsg_level = SOL_SOCKET;
    cmp->cmsg_type  = SCM_RIGHTS;
    cmp->cmsg_len    = RIGHTSLEN;
    *(int *)CMSG_DATA(cmp) = fd_to_send; /* дескриптор для передачи */
    cmp = CMSG_NXTHDR(&msg, cmp);
    cmp->cmsg_level = SOL_SOCKET;
    cmp->cmsg_type  = SCM_CREDTYPE;
    cmp->cmsg_len    = CREDSLLEN;
    credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);

#if defined(SCM_CREDENTIALS)
    credp->uid = geteuid();
    credp->gid = getegid();
    credp->pid = getpid();
#endif
    buf[1] = 0; /* нулевое значение означает отсутствие ошибки */
}
buf[0] = 0; /* нулевой байт - флаг для recv_ufd() */
if (sendmsg(fd, &msg, 0) != 2)
    return(-1);
return(0);
}

```

Обратите внимание, что инициализация структуры с идентификационной информацией должна производиться только в Linux.

В листинге 17.12 приводится модифицированная версия функции `recv_fd` с именем `recv_ufd`. Она возвращает идентификатор пользователя серверного процесса через аргумент, передаваемый по ссылке.

Листинг 17.12. Прием идентификационной информации через сокеты домена UNIX

```

#include "apue.h"
#include <sys/socket.h> /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)           /* интерфейс BSD */

```

```

#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS) /* интерфейс Linux */
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error передача идентификационной информации не поддерживается!
#endif

/*
 * размер буфера с управляющей информацией
 * для приема/передачи одного дескриптора
 */
#define RIGHTSLEN     CMSG_LEN(sizeof(int))
#define CREDSSLLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN   (RIGHTSLEN + CREDSSLLEN)

static struct cmsghdr *cmptr = NULL; /* размещается при первом вызове */

/*
 * Принимает дескриптор файла от серверного процесса. Кроме того, любые
 * принятые данные передаются функции (*userfunc)(STDERR_FILENO, buf, nbytes).
 * Прием дескриптора выполняется с соблюдением 2-байтного протокола.
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    char                 *ptr;
    char                 buf[MAXLINE];
    struct iovec         iov[1];
    struct msghdr        msg;
    int                  nr;
    int                  newfd = -1;
    int                  status = -1;
#if defined(CREDOPT)
    const int            on = 1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("ошибка вызова функции setsockopt");
        return(-1);
    }
#endif
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg iov     = iov;
        msg.msg iovlen  = 1;
        msg.msg name    = NULL;
        msg.msg namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg control   = cmptr;
        msg.msg controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("ошибка вызова функции recvmsg");

```

```

        return(-1);
    } else if (nr == 0) {
        err_ret("соединение закрыто сервером");
        return(-1);
    }

/*
 * Проверить, являются ли два последних байта нулевым байтом
 * и кодом ошибки. Нулевой байт должен быть предпоследним,
 * а код ошибки - последним байтом в буфере.
 * Нулевой код ошибки означает, что мы должны принять дескриптор.
 */
for (ptr = buf; ptr < &buf[nr]; ) {
    if (*ptr++ == 0) {
        if (ptr != &buf[nr-1])
            err_dump("нарушение формата сообщения");
        status = *ptr & 0xFF; /* предотвратить расширение */
        /* знакового разряда */

        if (status == 0) {
            if (msg.msg_controllen < CONTROLLEN)
                err_dump("получен код 0, но отсутствует fd");

            /* обработка управляющей информации */
            for (cmp = CMSG_FIRSTHDR(&msg);
                 cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                if (cmp->cmsg_level != SOL_SOCKET)
                    continue;
                switch (cmp->cmsg_type) {
                case SCM_RIGHTS:
                    newfd = *(int *)CMSG_DATA(cmp);
                    break;
                case SCM_CREDTYPE:
                    credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                    *uidptr = credp->CR_UID;
                }
            }
            if (status >= 0) /* доставлены заключительные данные */
                return(newfd); /* дескриптор или код ошибки */
        }
        nr -= 2;
    }
}
if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
    return(-1);
if (status >= 0) /* доставлены заключительные данные */
    return(newfd); /* дескриптор или код ошибки */
}
}

```

В FreeBSD при обмене идентификационной информацией используется константа `SCM_CREDS`, а в Linux — `SCM_CREDENTIALS`.

17.5. Сервер открытия файлов, версия 1

Теперь, используя возможность передачи дескрипторов файлов между процессами, напишем сервер открытия файлов — программу, которая запускается процессом и открывает один или более файлов. Но вместо содержимого файла сервер будет передавать вызывающему процессу открытый дескриптор. Это позволит

серверу работать с любыми типами файлов (такими, как устройства или сокеты), а не только с обычными файлами. Это также означает, что через механизмы IPC будет передаваться минимум информации — имя файла и режим открытия (от клиента серверу) и дескриптор открытого файла (от сервера клиенту). Содержимое файла передаваться не будет.

Такая архитектура, когда сервер работает в виде отдельного процесса (запускаемого клиентской программой, как в этом разделе, либо в виде демона — как в следующем разделе), имеет свои преимущества:

- Любой клиент может соединиться с сервером так же просто, как если бы он вызывал библиотечную функцию. Мы не «зашиваем» в программу службу с жестко заданным алгоритмом, а создаем универсальный инструмент, который может использоваться другими программами.
- Если потребуется внести изменения в сервер, они коснутся только одной программы, тогда как обновление одной библиотечной функции может потребовать внесения изменений во все программы, вызывающие эту функцию (точнее, потребуется заново скомпилировать приложения). Впрочем, подобное обновление можно упростить за счет использования динамических библиотек (раздел 7.7).
- Сервер может быть программой с установленным битом set-user-ID, что дает ему дополнительные права, которыми не обладает клиент. Обратите внимание, что библиотечные функции (или функции динамических библиотек) не дают такой возможности.

Клиентский процесс создает канал fd-pipe и затем с помощью функций `fork` и `exec` вызывает сервер. После этого клиент передает серверу запрос и через канал fd-pipe получает ответ.

Определим следующий протокол обмена данными между сервером и клиентом.

1. Клиент отправляет серверу через канал fd-pipe запрос вида «`open <pathname> <opemode>\0`», где `<opemode>` — строковое представление целого числа для второго аргумента функции `open`. Стока запроса завершается нулевым символом.
2. В ответ сервер посыпает вызывающему процессу дескриптор открытого файла или сообщение об ошибке, вызывая для этого функцию `send_fd` или `send_err` соответственно.

В данном примере дескриптор файла передается от дочернего процесса родительскому. В разделе 17.6 мы изменим этот пример так, чтобы сервер работал в виде отдельного процесса-демона, и тогда передача дескриптора будет осуществляться между независимыми процессами.

Для начала создадим заголовочный файл `open.h` (листинг 17.13), который подключает необходимые заголовочные файлы и содержит некоторые определения.

Листинг 17.13. Заголовочный файл open.h

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open" /* текст запроса, отправляемого серверу клиентом */

int csopen(char *, int);
```

Функция `main` клиента (листинг 17.14) — это цикл, который читает имя файла из стандартного ввода и копирует содержимое файла в стандартный вывод. Она вызывает функцию `csopen`, чтобы соединиться с сервером и получить от него дескриптор открытого файла.

Листинг 17.14. Функция `main` клиента, версия 1

```
#include "open.h"
#include <fcntl.h>

#define BUFFSIZE 8192

int
main(int argc, char *argv[])
{
    int      n, fd;
    char    buf[BUFFSIZE];
    char    line[MAXLINE];

    /* прочитать имя файла из стандартного ввода */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* заменить символ перевода строки */
                                         /* нулевым символом */

        /* открыть файл */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue; /* csopen() выведет сообщение, полученное от сервера */

        /* и вывести содержимое в стандартный вывод */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOOUT_FILENO, buf, n) != n)
                err_sys("ошибка вызова функции write");
        if (n < 0)
            err_sys("ошибка вызова функции read");
        close(fd);
    }

    exit(0);
}
```

Функция `csopen` (листинг 17.15) с помощью `fork` и `exec` запускает сервер, после чего создает канал `fd-pipe`.

Листинг 17.15. Функция `csopen`, версия 1

```
#include "open.h"
#include <sys/uio.h> /* struct iovec */

/*
 * Передает серверу аргументы name и oflag
 * и получает от него дескриптор открытого файла.
 */
int
csopen(char *name, int oflag)
{
    pid_t      pid;
    int       len;
```

```

char          buf[10];
struct iovec  iov[3];
static int    fd[2] = { -1, -1 };

if (fd[0] < 0) { /* при первом обращении запустить сервер */
    if (fd_pipe(fd) < 0){
        err_ret("ошибка вызова функции fd_pipe");
        return(-1);
    }
    if ((pid = fork()) < 0) {
        err_ret("ошибка вызова функции fork");
        return (-1);
    } else if (pid == 0) { /* дочерний процесс */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
            dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
            err_sys("ошибка переназначения stdin с помощью dup2");
        if (fd[1] != STDOUT_FILENO &&
            dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("ошибка переназначения stdout с помощью dup2");
        if (execl("./opend", "opend", (char *)0) < 0)
            err_sys("ошибка вызова функции execl");
    }
    close(fd[1]);           /* родительский процесс */
}
sprintf(buf, "%d", oflag); /* перевести oflag в строковое представление*/
iov[0].iov_base = CL_OPEN " "; /* конкатенация строк */
iov[0].iov_len  = strlen(CL_OPEN) + 1;
iov[1].iov_base = name;
iov[1].iov_len  = strlen(name);
iov[2].iov_base = buf;
iov[2].iov_len  = strlen(buf) + 1; /* +1 - для нулевого символа */
len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
if (writev(fd[0], &iov[0], 3) != len){
    err_ret("ошибка вызова функции writev");
    return(-1);
}

/* получить дескриптор, сообщение об ошибке обработать функцией write() */
return(recv_fd(fd[0], write));
}

```

Дочерний процесс закрывает один конец канала, а родительский процесс — другой. Кроме того, дочерний процесс перенаправляет стандартный ввод и стандартный вывод в канал. (Как вариант можно было бы передавать ASCII-представление дескриптора `fd[1]` в виде аргумента командной строки.)

Родительский процесс передает серверу запрос с именем файла и режимом открытия. В заключение родитель вызывает `recv_fd` и получает дескриптор или признак ошибки. Для вывода сообщения об ошибке вызывается функция `write`.

Теперь перейдем к реализации сервера. Эта программа, которую мы назвали `opend`, запускается клиентом из листинга 17.15. Сначала создадим заголовочный файл `opend.h` (листинг 17.16), который подключает необходимые заголовочные файлы и содержит ряд определений глобальных переменных и прототипов функций.

Листинг 17.16. Заголовочный файл opend.h, версия 1

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open" /* текст запроса, отправляемого серверу клиентом */

extern char errmsg[]; /* строка сообщения об ошибке, возвращаемая клиенту */
extern int oflag; /* флаги функции open(): O_xxx ... */
extern char *pathname; /* имя файла, полученное от клиента */

int cli_args(int, char **);
void request(char *, int, int);
```

Функция `main` (листинг 17.17) читает текст запроса из канала fd-pipe (из своего стандартного ввода) и вызывает функцию `handle_request`.

Листинг 17.17. Функция main сервера, версия 1

```
#include "opend.h"

char errmsg[MAXLINE];
int oflag;
char *pathname;

int
main(void)
{
    int nread;
    char buf[MAXLINE];

    for ( ; ; ) { /* прочитать аргументы в буфер и обработать запрос */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("ошибка чтения из канала");
        else if (nread == 0)
            break; /* клиент закрыл канал */
        handle_request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Основная работа выполняется в функции `handle_request` (листинг 17.18). Она вызывает функцию `buf_args` для извлечения и преобразования запроса клиента в `argv`-подобный список и передает его для обработки функции `cli_args`. Если ошибок не обнаружено, вызывается функция `open`, которая открывает файл, и затем функция `send_fd` отправляет клиенту дескриптор открытого файла через канал fd-pipe (стандартный вывод). Если возникла ошибка, вызывается функция `send_err`, которая отправляет клиенту сообщение об ошибке, используя описанный ранее протокол.

Листинг 17.18. Функция handle_request, версия 1

```
#include "opend.h"
#include <fcntl.h>

void
handle_request(char *buf, int nread, int fd)
{
```

```

int      newfd;

if (buf[nread-1] != 0) {
    snprintf(errmsg, MAXLINE-1,
        "текст запроса не завершается нулевым символом: %.*s\n",
        nread, nread, buf);
    send_err(fd, -1, errmsg);
    return;
}
if (buf_args(buf, cli_args) < 0) { /* разбор аргументов */
    send_err(fd, -1, errmsg);
    return;
}
if ((newfd = open(pathname, oflag)) < 0) {
    snprintf(errmsg, MAXLINE-1, "невозможно открыть файл %s: %s\n",
        pathname, strerror(errno));
    send_err(fd, -1, errmsg);
    return;
}
if (send_fd(fd, newfd) < 0) /* отправить дескриптор */
    err_sys("ошибка вызова функции send_fd");
close(newfd); /* сервер завершил работу с дескриптором */
}

```

Запрос клиента — это строка, завершающаяся нулевым символом, в которой все аргументы разделены пробельными символами. Функция `buf_args` из листинга 17.19 извлекает аргументы и передает их пользовательской функции в виде `argv`-подобного списка для дальнейшей обработки. Для извлечения аргументов из строки используется функция `strtok`, определяемая стандартом ISO C.

Листинг 17.19. Функция buf_args

```

#include "apue.h"

#define MAXARGC 50 /* максимальное количество аргументов в буфере */
#define WHITE " \t\n" /* пробельные символы, разделяющие аргументы */

/*
 * В buf[] содержатся аргументы, разделенные пробельными символами.
 * Содержимое буфера преобразуется в argv-подобный массив указателей
 * и передается пользовательской функции (optfunc) для дальнейшей обработки.
 * В случае ошибки при разборе содержимого буфера возвращается
 * значение -1, иначе возвращается результат работы функции optfunc().
 * Обратите внимание: содержимое буфера buf[] модифицируется
 * (после каждого аргумента вставляется нулевой символ).
 */
int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL) /* аргумент argv[0] обязателен */
        return(-1);
    argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1) /* -1 - предусмотреть место */
            /* для пустого указателя в конце списка */

```

```

        return(-1);
    argv[argc] = ptr;
}
argv[++argc] = NULL;

/*
 * Поскольку массив argv[] содержит указатели, ссылающиеся на строки
 * в массиве buf[], пользовательская функция может просто скопировать
указатели,
 * даже несмотря на то, что массив argv[] исчезнет после выхода из функции.
 */
return((*optfunc)(argc, argv));
}

```

Серверная функция, которую вызывает `buf_args`, называется `cli_args` (листинг 17.20). Она проверяет количество полученных аргументов и сохраняет имя файла и флаги режима открытия в глобальных переменных.

Листинг 17.20. Функция cli_args

```

#include "opend.h"

/*
 * Эта функция вызывается из buf_args(), которая, в свою очередь, вызывается
 * функцией handle_request(). Функция buf_args() преобразует содержимое буфера
 * в argv[]-подобный массив, который мы сейчас должны обработать.
 */

int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "Использование: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1]; /* сохранить указатель на имя файла */
    oflag = atoi(argv[2]);
    return(0);
}

```

На этом разработку сервера открытия файлов, запускаемого клиентской программой, можно считать завершенной. Перед вызовом функции `fork` клиент создает единственный канал fd-ріре, который используется для взаимодействия клиента и сервера. Благодаря такой архитектуре мы имеем по серверу для каждого клиента.

17.6. Сервер открытия файлов, версия 2

В предыдущем разделе мы разработали сервер открытия файлов, который запускается клиентским приложением с помощью функций `fork` и `exec`. Этот пример демонстрирует порядок передачи дескриптора от дочернего процесса родительскому. Теперь создадим сервер открытия файлов, который будет работать как демон. Один сервер будет обслуживать множество клиентов. Мы предполагаем, что такой вариант эффективнее, поскольку в нем отсутствует обращение к функции

ям `fork` и `exec`. Для взаимодействия клиента с сервером мы по-прежнему будем использовать сокеты домена UNIX и продемонстрируем возможность передачи дескриптора файла между независимыми процессами. В этом примере будут использоваться функции `serv_listen`, `serv_accept` и `cli_conn`, о которых мы говорили в разделе 17.3. Кроме того, эта версия демонстрирует возможность обслуживания множества клиентов единственным сервером с помощью функций `select` и `poll` (раздел 14.4).

Новый клиент очень похож на программу, приведенную в разделе 17.5. Функция `main` осталась без изменений (листинг 17.14). А в заголовочный файл `open.h` (листинг 17.13) мы добавили одну строку:

```
#define CS_OPEN "/tmp/opend.socket" /* предопределенное имя сервера */
```

Содержимое файла `open.c` (листинг 17.15) претерпело некоторые изменения, поскольку теперь вместо функций `fork` и `exec` вызывается функция `cli_conn`. Содержимое этого файла приводится в листинге 17.21.

Листинг 17.21. Функция `csopen`, версия 2

```
#include "open.h"
#include <sys/uio.h> /* struct iovec */

/*
 * Передает аргументы name и oflag серверу
 * и получает от него дескриптор открытого файла.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[12];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) { /* открыть соединение с сервером */
        if ((csfd = cli_conn(CS_OPEN)) < 0){
            err_ret("ошибка вызова функции cli_conn");
            return(-1);
        }
    }

    sprintf(buf, "%d", oflag); /* преобразовать oflag в строку ascii */
    iov[0].iov_base = CL_OPEN " "; /* конкатенация строк */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1; /* нулевой символ передается всегда */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len){
        err_ret("ошибка вызова функции writev");
        return(-1);
    }

    /* получить дескриптор, сообщение об ошибке обработать функцией write() */
    return(recv_fd(csfd, write));
}
```

Протокол взаимодействия клиента и сервера остался без изменений.

Теперь перейдем к реализации сервера. Заголовочный файл `opend.h` (листинг 17.22) подключает необходимые заголовочные файлы и содержит определения глобальных переменных и прототипов функций.

Листинг 17.22. Заголовочный файл `opend.h`, версия 2

```
#include "apue.h"
#include <errno.h>

#define CS_OPEN "/tmp/opend.socket" /* предопределеное имя сервера */
#define CL_OPEN "open"           /* текст запроса, отправляемого клиентом */

extern int debug;      /* ненулевое значение для запуска в
                         /* интерактивном режиме (не демон) */
extern char errmsg[]; /* строка сообщения об ошибке, возвращаемая клиенту */
extern int oflag;      /* флаги функции open: O_XXX ... */
extern char *pathname; /* имя файла, полученное от клиента */

typedef struct { /* по одной структуре Client для каждого клиента */
    int fd;        /* fd или -1, если элемент массива свободен */
    uid_t uid;
} Client;

extern Client *client; /* указатель на массив в динамической памяти */
extern int client_size; /* количество элементов в массиве client[] */

int cli_args(int, char **);
int client_add(int, uid_t);
void client_del(int);
void loop(void);
void handle_request(char *, int, int, uid_t);
```

Поскольку теперь сервер будет обслуживать сразу несколько клиентов, он должен отслеживать состояние соединения с каждым из них. Делать это он будет с помощью массива `client`, объявленного в заголовочном файле `opend.h`. В листинге 17.23 приводятся три функции, которые обслуживают этот массив.

Листинг 17.23. Функции обслуживания массива `client`

```
#include "opend.h"

#define NALLOC 10 /* количество структур в массиве client для alloc/realloc */

static void
client_alloc(void) /* разместить дополнительные элементы в массиве client[] */
{
    int i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("невозможно выделить пространство для массива клиентов");

    /* инициализировать новые элементы */
```

```

for (i = client_size; i < client_size + NALLOC; i++)
    client[i].fd = -1; /* fd = -1 означает, что элемент не занят */
client_size += NALLOC;
}

/*
 * Вызывается из функции loop() по прибытии нового запроса от клиента.
 */
int
client_add(int fd, uid_t uid)
{
    int      i;

    if (client == NULL) /* первое обращение к функции */
        client_alloc();

again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) { /* найти незанятый элемент */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i); /* вернуть индекс в массиве client[] */
        }
    }

    /* массив полон, выделить дополнительное пространство */
    client_alloc();
    goto again; /* и повторить поиск (на этот раз все будет в порядке) */
}

/*
 * Вызывается функцией loop() по завершении работы с клиентом.
 */
void
client_del(int fd)
{
    int      i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_quit("невозможно отыскать запись о клиенте по дескриптору %d", fd);
}

```

Функция `client_add` при первом вызове обращается к функции `client_alloc`, которая выделяет пространство для десяти записей с помощью функции `malloc`. Когда все десять записей будут заполнены, следующий вызов `client_add` приведет к выделению дополнительного пространства в массиве вызовом `realloc`. Используя такой способ хранения данных в динамической памяти, мы избежали необходимости ограничения размера массива во время компиляции и определения соответствующих значений в заголовочном файле. При появлении ошибок эти функции обращаются к функциям семейства `log_` (приложение B), поскольку предполагается, что сервер будет работать как демон.

В обычной ситуации сервер будет выполняться в режиме демона. Но нам необходимо предусмотреть возможность запускать его как обычное приложение, чтобы обеспечить вывод диагностических сообщений в стандартное устройство вывода сообщений об ошибках. Это упростит тестирование и отладку сервера, что особенно удобно, если у нас нет права на чтение файла журнала, куда обычно записываются диагностические сообщения. Для управления режимом работы сервера мы будем использовать параметр командной строки.

Очень важно, чтобы все команды следовали одним и тем же соглашениям по передаче аргументов, потому что это упрощает их использование. Если пользователь знаком с приемами формирования аргументов командной строки для какой-то одной команды, вероятность допустить ошибку возрастает, если другая команда следует иным соглашениям.

Эта проблема иногда проявляется, когда в командной строке присутствуют пробельные символы. Некоторые команды требуют отделять ключи от их аргументов пробелом, другие требуют, чтобы они писались слитно, без пробелов между ними. В отсутствие непротиворечивого набора правил пользователи вынуждены либо заучивать синтаксис вызова всех команд, либо идти путем проб и ошибок.

Стандарт Single UNIX Specification определяет ряд соглашений и рекомендаций, способствующих непротиворечивости синтаксиса командной строки. Они включают такие рекомендации, как «каждый ключ командной строки должен быть представлен единственным символом» и «все ключи должны начинаться с дефиса».

К счастью, в нашем распоряжении имеется функция `getopt`, помогающая обеспечить единообразную обработку ключей командной строки.

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *options);

extern int optind, optarg, opterr;
extern char *optarg;
```

Возвращает следующий символ, соответствующий параметру, или `-1`, если все параметры были обработаны

Аргументы `argc` и `argv` — это те же аргументы, что передаются функции `main` программы. Аргумент `options` — это строка, содержащая символы ключей, поддерживаемых командой (программой). Если символ ключа сопровождается двоеточием, такой ключ интерпретируется как принимающий дополнительный аргумент. Иначе ключ существует сам по себе. Например, если предположить, что синтаксис вызова команды имеет следующий вид:

`command [-i] [-u username] [-z] filename`

мы могли бы передать функции `getopt` строку "`iu:z`" в аргументе `option`.

Обычно функция `getopt` используется в цикле, который завершается, когда `getopt` возвращает `-1`. В каждой итерации такого цикла `getopt` будет возвращать следу-

ищий ключ для обработки. Однако разрешение конфликтов между ключами полностью возлагается на само приложение; `getopt` просто анализирует командную строку, пытаясь следовать принятым стандартам.

Встречая недопустимый ключ, `getopt` возвращает вместо символа знак вопроса. Если аргумент ключа отсутствует, `getopt` также возвращает знак вопроса, но если первым символом в строке *options* будет стоять двоеточие, `getopt` вернет двоеточие. Специальная последовательность `--` заставляет `getopt` остановить обработку ключей и вернуть `-1`. Это дает пользователям возможность передавать команде аргументы, начинающиеся со знака «минус», но не являющиеся ключами. Например, если имеется файл с именем `-bar`, его нельзя удалить командой

```
rm -bar
```

потому что `rm` попытается интерпретировать `-bar` как список ключей. Чтобы удалить этот файл, следует ввести команду

```
rm -- -bar
```

Функция `getopt` поддерживает четыре внешние переменные.

`optarg`

Если параметр принимает аргумент, `getopt` вернет в переменной `optarg` указатель на строку с аргументом ключа.

`opterr`

Встретив ошибочный параметр, `getopt` по умолчанию выводит сообщение об ошибке. Чтобы запретить такое поведение, приложение может присвоить переменной `opterr` значение `0`.

`optind`

Индекс строки в массиве `argv`, которая будет обработана следующей. Отсчет начинается с `1` и увеличивается по мере обработки каждого аргумента.

`optopt`

Столкнувшись с ошибкой в процессе обработки очередного ключа, `getopt` присвоит переменной `optopt` указатель на строку с ключом, вызвавшим ошибку.

Функция `main` сервера (листинг 17.24) определяет ряд глобальных переменных, обрабатывает аргументы командной строки и вызывает функцию `loop`. Если сервер вызван с ключом `-d`, он запускается в интерактивном режиме. Это может потребоваться для отладки сервера.

Листинг 17.24. Функция `main` сервера, версия 2

```
#include "opend.h"
#include <syslog.h>

int    debug, oflag, client_size, log_to_stderr;
char   errmsg[MAXLINE];
char   *pathname;

Client *client = NULL;

int
```

```

main(int argc, char *argv[])
{
    int      c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0; /* функция getopt() не должна выводить сообщения в stderr */

    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd': /* отладка */
            debug = log_to_stderr = 1;
            break;

        case '?':
            err_quit("недопустимая опция: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop(); /* никогда не вернет управление */
}

```

Функция `loop` выполняет бесконечный цикл. Мы продемонстрируем две версии этой функции. В листинге 17.25 приводится версия, реализованная на основе функции `select`, а в листинге 17.26 — на основе функции `poll`.

Листинг 17.25. Функция `loop` на основе функции `select`

```

#include "opend.h"
#include <sys/select.h>

void
loop(void)
{
    int      i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

    /* получить fd, на котором сервер будет ожидать поступления запросов */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("ошибка вызова функции serv_listen");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;

    for ( ; ; ) {
        rset = allset; /* rset модифицируется в каждой итерации */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("ошибка вызова функции select");

        if (FD_ISSET(listenfd, &rset)) {
            /* принять новый запрос на соединение с клиентом */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)

```

```

        log_sys("ошибка вызова функции serv_accept: %d", clifd);
        i = client_add(clifd, uid);
        FD_SET(clifd, &allset);
        if (clifd > maxfd)
            maxfd = clifd; /* максимальный номер fd для select() */
        if (i > maxi)
            maxi = i; /* максимальный индекс в массиве client[] */
        log_msg("новое соединение: uid %d, fd %d", uid, clifd);
        continue;
    }

    for (i = 0; i <= maxi; i++) { /* обход массива client[] */
        if ((clifd = client[i].fd) < 0)
            continue;
        if (FD_ISSET(clifd, &rset)) {
            /* прочитать содержимое буфера с аргументами */
            if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                log_sys("ошибка вызова функции read для fd %d", clifd);
            } else if (nread == 0) {
                log_msg("закрыто: uid %d, fd %d",
                        client[i].uid, clifd);
                client_del(clifd); /* клиент закрыл соединение */
                FD_CLR(clifd, &allset);
                close(clifd);
            } else { /* обработать запрос от клиента */
                handle_request(buf, nread, clifd, client[i].uid);
            }
        }
    }
}

```

Эта функция создает точку соединения на стороне сервера с помощью функции `serv_listen`. Остальная часть функции выполняет цикл, начинающийся с вызова функции `select`, после возврата из которой возможны два состояния.

1. Дескриптор `listenfd` готов для чтения. Это означает, что новый клиент вызвал функцию `cli_conn`. Для приема нового запроса на соединение вызывается функция `serv_accept` (листинг 17.6), а затем в массив `client` добавляется информация о клиенте. (Мы отслеживаем значения самого большого номера дескриптора для передачи его функции `select` в первом аргументе. Мы также отслеживаем значения самого большого индекса в массиве клиентов.)
 2. Дескриптор существующего соединения с клиентом готов для чтения. Это означает, что клиент либо закрыл соединение, либо прислал новый запрос. Если функция `read` вернула 0 (признак конца файла), значит, клиент закрыл соединение. Если функция `read` вернула значение больше 0, значит, клиент прислал новый запрос, который мы передаем функции `handle_request`.

Мы запоминаем используемые дескрипторы в наборе `allset`. Как только новый клиент соединится с сервером, мы включаем соответствующий разряд в наборе. После того как клиент закроет соединение, соответствующий разряд будет выключен.

Мы всегда знаем, когда клиент закрыл соединение (неважно, добровольно или в результате аварийного завершения), поскольку все дескрипторы клиента (включая

чая дескриптор, поддерживающий соединение с сервером) в этом случае будут закрыты ядром автоматически. В этом состоит одно из отличий дескрипторов от механизмов XSI IPC.

В листинге 17.26 приводится версия функции `loop`, реализованная на основе функции `poll`.

Листинг 17.26. Функция `loop` на основе функции `poll`

```
#include "opend.h"
#include <poll.h>

#define NALLOC 10 /* количество структур pollfd для alloc/realloc */

static struct pollfd *
grow_pollfd(struct pollfd *pfds, int *maxfd)
{
    int      i;
    int      oldmax = *maxfd;
    int      newmax = oldmax + NALLOC;

    if ((pfds = realloc(pfds, newmax * sizeof(struct pollfd))) == NULL)
        err_sys("realloc error");
    for (i = oldmax; i < newmax; i++) {
        pfd[i].fd = -1;
        pfd[i].events = POLLIN;
        pfd[i].revents = 0;
    }
    *maxfd = newmax;
    return(pfd);
}

void
loop(void)
{
    int          i, listenfd, clifd, nread;
    char         buf[MAXLINE];
    uid_t        uid;
    struct pollfd *pollfd;
    int          numfd = 1;
    int          maxfd = NALLOC;

    if ((pollfd = malloc(NALLOC * sizeof(struct pollfd))) == NULL)
        err_sys("ошибка вызова функции malloc");
    for (i = 0; i < NALLOC; i++) {
        pollfd[i].fd = -1;
        pollfd[i].events = POLLIN;
        pollfd[i].revents = 0;
    }

    /* получить fd, на котором сервер будет ожидать поступления запросов */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0); /* нулевой индекс используется для listenfd */
    pollfd[0].fd = listenfd;

    for ( ; ; ) {
        if (poll(pollfd, numfd, -1) < 0)
```

Чтобы иметь возможность обслуживать столько клиентов, сколько может быть открытых дескрипторов, мы динамически распределяем пространство под массивы структур `pollfd`, используя ту же стратегию, что и в функции `client_alloc` из листинга 17.23.

Первая запись в массиве `client` (с индексом 0) используется для хранения дескриптора `listenfd`. Поступление нового запроса на соединение определяется событием `POLLIN` дескриптора `listenfd`. Как и прежде, для приема запроса на соединение вызывается функция `serv.accept`.

Для существующего клиента мы должны обрабатывать два различных события функции `poll`: разрыв соединения (событие `POLLHUP`) и поступление нового запроса (событие `POLLIN`). Даже после закрытия канала со стороны клиента сервер сможет прочитать все посланные ему данные. Однако в этом случае при закрытии соединения со стороны клиента можно просто удалить все данные, находящиеся в потоке. Нет смысла обрабатывать запрос, если некому отправить ответ.

Как и в версии на основе функции `select`, запрос клиента обслуживается функцией `handle_request` (листинг 17.27). Эта версия функции похожа на предыдущую (листинг 17.18). Она вызывает ту же функцию `buf_args` (листинг 17.19), которая, в свою очередь, вызывает `cli_args` (листинг 17.20), но поскольку теперь она вызывается из демона, все сообщения вместо стандартного потока сообщений об ошибках выводятся в системный журнал.

Листинг 17.27. Функция `handle_request`, версия 2

```
#include "opend.h"
#include <fcntl.h>

void
handle_request(char *buf, int nread, int clifd, uid_t uid)
{
    int      newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
                 "строка запроса от uid %d не завершается нулевым символом: %.*s\n",
                 uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("запрос: %s, от uid %d", buf, uid);

    /* разбор аргументов */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "невозможно открыть %s: %s\n",
                 pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* передать дескриптор */
    if (send_fd(clifd, newfd) < 0)
        log_sys("ошибка вызова функции send_fd");
    log_msg("передан fd %d через fd %d для %s", newfd, clifd, pathname);
    close(newfd); /* работа с дескриптором завершена */
}
```

На этом мы завершаем разработку второй версии сервера открытия файлов, которая работает в режиме демона и может обслуживать запросы от множества клиентов.

17.7. Подведение итогов

Ключевые темы этой главы — передача дескрипторов открытых файлов между процессами и создание уникальных соединений между сервером и клиентами. Несмотря на то что все платформы поддерживают сокеты домена UNIX (табл. 15.1), их реализация на разных платформах различна, что существенно осложняет разработку переносимых приложений.

На протяжении всей главы мы использовали сокеты домена UNIX. Мы узнали, как на их основе реализовать дуплексный канал обмена данными и как с их помощью можно обеспечить применение функций мультиплексирования ввода/вывода, представленных в разделе 14.4, для работы с очередями сообщений XSI.

Мы представили две версии сервера открытия файлов. Первая версия запускается прямо из клиентского приложения с помощью функций `fork` и `exec`. Вторая версия реализована в виде демона, который способен обрабатывать запросы множества клиентов. В обеих версиях были использованы функции передачи и приема дескрипторов.

Кроме того, последняя версия использует функцию `getopt`, функции обслуживания соединений между клиентом и сервером, которые обсуждались в разделе 17.3, а также функции мультиплексирования ввода/вывода, о которых говорилось в разделе 14.4.

Упражнения

- 17.1** Для реализации программы из листинга 17.2 мы воспользовались сокетами домена UNIX типа `SOCK_DGRAM`, потому что они позволили избавиться от проверки границ сообщений. Опишите, какие изменения пришлось бы внести в программу, если бы вместо сокетов использовались обычные неименованные каналы. Как можно избавиться от двойного копирования сообщений?
- 17.2** Используя функции приема/передачи дескрипторов из этой главы и функции синхронизации родительского и дочернего процессов из раздела 8.9, напишите следующую программу. Программа вызывает функцию `fork`, дочерний процесс открывает существующий файл и передает дескриптор родительскому процессу. После этого дочерний процесс изменяет текущую позицию в файле с помощью функции `lseek` и извещает об этом родителя. Родительский процесс читает данные из файла, начиная с текущей позиции, и выводит их для проверки. Если дескриптор передан описанным нами способом, оба процесса должны совместно использовать одну и ту же запись в таблице файлов, поэтому изменение текущей позиции в дочернем

процессе должно отразиться на дескрипторе родительского процесса. После этого дочерний процесс должен переместить текущую позицию файла в другое место и опять сообщить об этом родительскому процессу.

- 17.3 В листингах 17.16 и 17.17 мы по-разному объявили и описали глобальные переменные. В чем суть этих различий?
- 17.4 Перепишите функцию `buf_args` (листинг 17.19), чтобы убрать ограничение времени компиляции на размер массива `argv`. Используйте динамическую память для размещения этого массива.
- 17.5 Подумайте, как можно оптимизировать функцию `loop` из листингов 17.25 и 17.26. Реализуйте оптимизированные версии.
- 17.6 В функции `serv_listen` (листинг 17.5) мы удаляем файл, представляющий сокет домена UNIX, если он уже существует. Чтобы избежать ошибочного удаления файла, не являющегося сокетом, можно было бы сначала вызвать функцию `stat` и проверить тип файла. Подумайте, какие две проблемы могут при этом возникнуть.
- 17.7 Опишите два возможных способа передачи более одного файлового дескриптора единственным вызовом функции `sendmsg`. Попробуйте реализовать их, чтобы узнать, поддерживаются ли они вашей операционной системой.

18

Терминальный ввод/вывод

18.1. Введение

Вопросы, связанные с терминальным вводом/выводом, относятся к разряду наиболее запутанных, независимо от типа операционной системы. UNIX не исключение. Самые объемные страницы справочного руководства обычно посвящены именно терминальному вводу/выводу.

Первые противоречия начали проявляться в конце 1970-х годов, когда при создании System III были разработаны процедуры для работы с терминалами, в корне отличавшиеся от тех же процедур в Version 7. Процедуры System III далее перекочевали в System V, а процедуры из Version 7 стали стандартом для BSD-систем. Как и в случае с сигналами, противоречия между этими двумя мирами были преодолены благодаря стандарту POSIX.1. В этой главе мы рассмотрим все функции, предназначенные для работы с терминалами, а также некоторые дополнительные функции, характерные для конкретных платформ.

Основная сложность системы терминального ввода/вывода связана с тем, что ее функции используются для выполнения самых разнообразных задач: для управления терминалами, для взаимодействия между компьютерами, соединенными кабелем, для работы с модемами, принтерами и т. п.

18.2. Обзор

Терминальный ввод/вывод имеет два режима работы.

1. Канонический режим обслуживания ввода. В этом режиме ввод с терминала обслуживается построчно. Драйвер терминала возвращает не более одной строки за один запрос.
2. Неканонический режим обслуживания ввода. Вводимые символы не собираются в строки.

Канонический режим действует по умолчанию, если мы не делаем что-то особенное. Например, если в командной оболочке стандартное устройство ввода связано с терминалом и мы копируем данные из стандартного ввода в стандартный вывод с помощью функций `read` и `write`, при работе терминала в каноническом режи-

ме функция `read` будет возвращать данные построчно. Программы, работающие в полноэкранном режиме, например редактор `vi`, используют неканонический режим, поскольку команды редактора могут состоять всего из одного символа и не содержать перевода строки. Кроме того, редактор не должен позволять системе обслуживать специальные символы, поскольку в самом редакторе они могут обозначать вполне определенные команды редактирования. Например, символ `Control-D`, который большинством терминалов воспринимается как признак конца файла, в редакторе `vi` обозначает команду прокрутки на пол-экрана вниз.

Драйверы терминалов в Version 7 и в первых версиях BSD поддерживали три режима обслуживания ввода с терминала: (a) подготовленный (cooked mode – вводимые символы собираются в строки и производится обработка специальных символов), (б) прозрачный (raw mode – вводимые символы не собираются в строки и обработка специальных символов не выполняется) и (в) посимвольный (cbreak mode – вводимые символы не собираются в строки, но обрабатываются некоторые специальные символы). В листинге 18.10 демонстрируется POSIX.1-совместимая функция, выполняющая перевод терминала в режим посимвольного и прозрачного ввода.

Стандарт POSIX.1 определяет 11 специальных символов, интерпретацию 9 из которых можно изменить. С некоторыми из них мы уже встречались в предыдущих главах, например символ конца файла (обычно `Control-D`) и символ приостановки (обычно `Control-Z`). В разделе 18.3 будут даны описания всех этих символов.

Терминал можно рассматривать как некоторое устройство, управляемое драйвером, обычно расположенным в ядре. Каждое терминальное устройство имеет входную и выходную очереди, как показано на рис. 18.1.

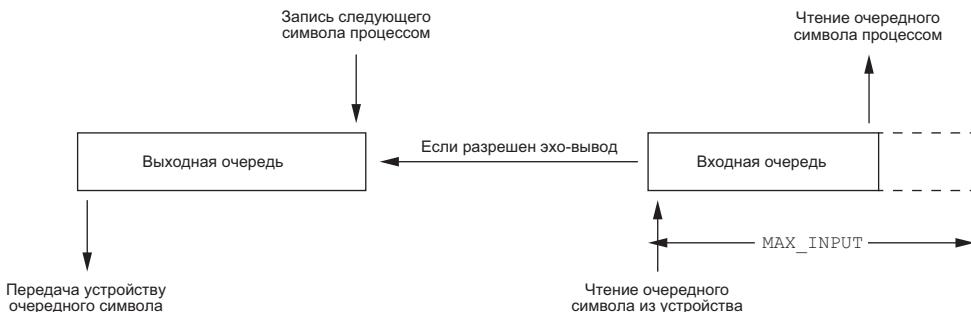


Рис. 18.1. Логическое изображение входной и выходной очередей устройства терминала

Обратите особое внимание на следующие моменты:

- Включенный режим эхо-вывода подразумевает связь между входной и выходной очередями.
- Размер входной очереди ограничивается значением `MAX_INPUT` (табл. 2.8). Реакция системы на переполнение входной очереди зависит от конкретной реализации. В большинстве версий UNIX, если это происходит, ввод последующих символов сопровождается звуковым сигналом.

- Существует еще один предел, ограничивающий размер входной очереди, который здесь не показан, — `MAX_CANON`. Этот предел определяет максимальный размер строки в байтах при работе терминала в каноническом режиме.
- Хотя размер выходной очереди также ограничен, константы, которые определяли бы конкретное значение, отсутствуют, потому что когда выходная очередь начинает переполняться, ядро просто приостанавливает процесс, выполняющий запись, пока в выходной очереди не освободится место.
- Позже мы увидим, как с помощью функции `tcflush` можно сбросить содержимое входной и выходной очередей. Аналогично, знакомясь с функцией `tcsetattr`, мы узнаем, как с ее помощью изменить характеристики терминала, но только после опустошения выходной очереди. (Это может понадобиться, например, для изменения параметров вывода.) Мы также можем заставить систему очистить очередь ввода при изменении параметров терминала. (Это может пригодиться при изменении параметров ввода или при переходе от канонического режима к неканоническому и обратно, чтобы предотвратить неверную интерпретацию ранее введенных символов.)

В большинстве версий UNIX реализация канонического режима выполнена в виде модуля, который называется *terminal line discipline* (дисциплина обслуживания линии связи с терминалом). Этот модуль можно рассматривать как некий черный ящик между универсальными функциями чтения/записи ядра и фактическим драйвером устройства (рис. 18.2).

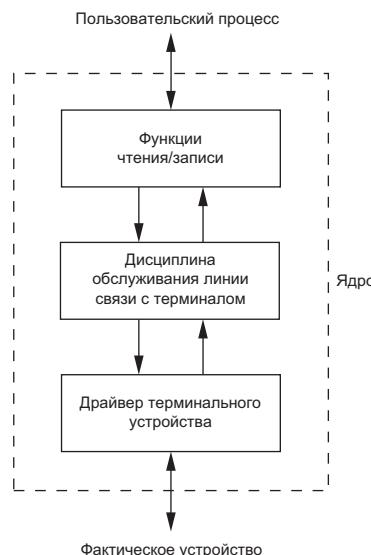


Рис. 18.2. Модуль, реализующий дисциплину обслуживания линии связи с терминалом

Благодаря выделению поддержки канонического режима в отдельный модуль, все драйверы могут единообразно обрабатывать этот режим. Мы еще вернемся к рис. 18.2 в главе 19, когда будем обсуждать псевдотерминалы.

Все характеристики терминала, которые можно узнать или изменить, хранятся в структуре `termios`. Определение этой структуры находится в заголовочном файле `<termios.h>`, который будет постоянно использоваться на протяжении всей главы:

```
struct termios {
    tcflag_t c_iflag;      /* флаги режима ввода */
    tcflag_t c_oflag;      /* флаги режима вывода */
    tcflag_t c_cflag;      /* флаги режима управления */
    tcflag_t c_lflag;      /* флаги локального режима */
    cc_t     c_cc[NCCS];   /* управляющие символы */
};
```

Можно сказать, что флаги режима ввода управляют вводом символов драйвером терминального устройства (очистка восьмого разряда, проверка разряда четности и т. д.), флаги режима вывода контролируют вывод драйвера (обработка выходного потока данных, замена символа перевода строки комбинацией CR/LF и т. п.), а флаги режима управления определяют параметры последовательного порта RS-232 (игнорировать строки состояния модема, количество стоповых битов и т. д.). И наконец, флаги локального режима оказывают влияние на интерфейс между драйвером и пользователем (включение/выключение эхо-вывода, отображение символа забоя, разрешение/запрет генерации сигналов терминалом и т. д.).

Тип `tcflag_t` достаточно велик, чтобы переменные этого типа могли хранить значения сразу всех флагов. Зачастую он определен как `unsigned int` или `unsigned long`. Массив `c_cc` хранит все специальные символы, которые можно изменить. Константа `NCCS` определяет количество элементов этого массива, обычно ее значение находится в диапазоне от 15 до 20 (поскольку большинство версий UNIX поддерживают более 11 управляющих символов, определяемых стандартом POSIX.1). Тип `cc_t` достаточно велик, чтобы переменные этого типа могли хранить любой из управляющих символов, и обычно определен как `unsigned char`.

В версиях System V, предшествовавших стандарту POSIX.1, имелись заголовочный файл `<termio.h>` и структура `termio`. Стандарт POSIX.1 добавил к именам букву `s`, чтобы отличить современные определения от их предшественников.

В табл. 18.1–18.4 перечисляются все флаги, с помощью которых можно воздействовать на характеристики терминального устройства. Обратите внимание, что хотя стандарт Single UNIX Specification определяет базовый набор, все платформы расширяют его собственными флагами. Большинство дополнительных флагов появились в результате исторически сложившихся различий между системами. Более подробно назначение каждого флага мы рассмотрим в разделе 18.5.

Итак, флаги нам известны, но как изменить те или иные характеристики терминального устройства? В табл. 18.5 перечисляются функции, определяемые стандартом Single UNIX Specification для взаимодействий с терминальными устройствами. (Все эти функции являются частью базовых спецификаций стандарта POSIX.1. Функции `tcgetpgrp`, `tcgetsid` и `tcsetpgrp` были описаны в разделе 9.7.)

Таблица 18.1. Флаги режима управления терминалом

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
CBAUDEXT	Расширенное управление скоростью					✓
CCAR_OFLOW	Использовать линию DCD для управления выходным потоком		✓		✓	
CCTS_OFLOW	Использовать линию CTS для управления выходным потоком		✓		✓	✓
CDSR_OFLOW	Использовать линию DSR для управления выходным потоком		✓		✓	
CDTR_IFLOW	Использовать линию DTR для управления входным потоком		✓		✓	
CIBAUDEXT	Расширенное управление скоростью приема					✓
CIGNORE	Игнорировать флаги управления режимами		✓		✓	
CLOCAL	Игнорировать линии состояния модема	✓	✓	✓	✓	✓
CMSPAR	Контроль четности по схеме mark или space			✓		
CREAD	Разрешить прием	✓	✓	✓	✓	✓
CRTSCTS	Разрешить аппаратное управление потоком данных		✓	✓	✓	✓
CRTS_IFLOW	Использовать линию RTS для управления входным потоком		✓		✓	✓
CRTSXOFF	Разрешить аппаратное управление входным потоком данных					✓
CSIZE	Маска размера символов	✓	✓	✓	✓	✓
CSTOPB	Передавать два стоповых бита или один	✓	✓	✓	✓	✓
HUPCL	Разорвать связь при закрытии устройства последним процессом	✓	✓	✓	✓	✓
MDMBUF	То же самое, что CCAR_OFLOW		✓		✓	

Таблица 18.1 (окончание)

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
PARENB	Разрешить контроль четности	✓	✓	✓	✓	✓
PAREXT	Контроль четности по схеме mark или space					✓
PARODD	Контроль четности по схеме odd или even	✓	✓	✓	✓	✓

Таблица 18.2. Флаги режима ввода

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
BRKINT	Генерировать сигнал SIGINT при получении символа BREAK	✓	✓	✓	✓	✓
ICRNL	Преобразовывать символ CR (возврат каретки) в символ NL (перевод строки) при вводе	✓	✓	✓	✓	✓
IGNBRK	Игнорировать символ BREAK	✓	✓	✓	✓	✓
IGNCR	Игнорировать символ CR	✓	✓	✓	✓	✓
IGNPAR	Игнорировать символы с ошибками контроля четности	✓	✓	✓	✓	✓
IMAXBEL	Выдавать звуковой сигнал при переполнении очереди ввода		✓	✓	✓	✓
INLCR	Преобразовывать символ NL в символ CR при вводе	✓	✓	✓	✓	✓
INPCK	Разрешить проверку бита четности при вводе	✓	✓	✓	✓	✓
ISTRIP	Сбрасывать восьмой бит во вводимых символах	✓	✓	✓	✓	✓
IUCLC	Преобразовывать при вводе символы верхнего регистра в нижний регистр			✓		✓
IUTF8	Входные данные поступают в кодировке UTF-8			✓	✓	
IXANY	Разрешить перезапуск вывода по любому символу	✓	✓	✓	✓	✓
IXOFF	Разрешить управление входным потоком данных с помощью символов START/STOP	✓	✓	✓	✓	✓

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
IXON	Разрешить управление выходным потоком данных с помощью символов START/STOP	✓	✓	✓	✓	✓
PARMRK	Отмечать ошибки контроля четности	✓	✓	✓	✓	✓

Таблица 18.3. Флаги локального режима

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
ALTWERASE	Использовать альтернативный алгоритм обработки символа WERASE		✓		✓	
ECHO	Разрешить эхо-вывод	✓	✓	✓	✓	✓
ECHOCTL	Выводить управляющие символы как ^(символ)		✓	✓	✓	✓
ECHOE	Отображать забой	✓	✓	✓	✓	✓
ECHOK	Отображать удаление строки	✓	✓	✓	✓	✓
ECHOKE	Отображать забой каждого символа при удалении строки		✓	✓	✓	✓
ECHONL	Отображать символ перевода строки	✓	✓	✓	✓	✓
ECHOPRT	Отображать удаление символов для вывода на принтер		✓	✓	✓	✓
EXTPROC	Внешний обработчик символов		✓	✓	✓	
FLUSHO	Сбрасывать очередь вывода		✓	✓	✓	✓
ICANON	Канонический режим ввода	✓	✓	✓	✓	✓
IEXTEN	Разрешить расширенную обработку вводимых символов	✓	✓	✓	✓	✓
ISIG	Разрешить генерацию сигналов терминалом	✓	✓	✓	✓	✓
NOFLSH	Запретить сброс очередей после получения сигналов прерывания и завершения (SIGINT и SIGQUIT)	✓	✓	✓	✓	✓

Таблица 18.3 (окончание)

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
NOKERNINFO	Не выводить информацию при вводе символа STATUS		✓		✓	
PENDIN	Вывод символов из очереди ввода		✓	✓	✓	✓
TOSTOP	Послать сигнал SIGTTOU фоновому источнику вывода	✓	✓	✓	✓	✓
XCASE	Каноническое представление символов верхнего и нижнего регистров			✓		✓

Таблица 18.4. Флаги режима вывода

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
BSDLY	Маска задержки символа забоя	XSI		✓		✓
CRDLY	Маска задержки символа CR (возврат каретки)	XSI		✓		✓
FFDLY	Маска задержки символа FF (перевод страницы)	XSI		✓		✓
NLDLY	Маска задержки символа NL (перевод строки)	XSI		✓		✓
OCRNL	Преобразовывать символ CR в NL при выводе	XSI	✓	✓		✓
OFDEL	Использовать символ DEL в качестве заполнителя, иначе использовать символ NUL	XSI		✓		✓
OFILL	Использовать для задержки символы заполнения	XSI		✓		✓
OLCUC	Преобразовывать символы нижнего регистра в верхний при выводе			✓		✓
ONLCR	Преобразовывать символы NL в последовательности символов CR-NL	XSI	✓	✓	✓	✓
ONLRET	Символ NL выполняет функции символа CR	XSI	✓	✓		✓
ONOCR	Не выводить символ CR в нулевой позиции строки	XSI	✓	✓		✓

Флаг	Описание	POSIX.1	FreeBSD 8.0	Linux 3.2.0	MacOS X 10.6.8	Solaris 10
ONOEOT	Не выводить символ EOT (^D)		✓		✓	
OPOST	Выполнять обработку вывода	✓	✓	✓	✓	✓
OXTABS	Заменять символы табуляции пробелами		✓		✓	
TABDLY	Маска задержки символа горизонтальной табуляции	XSI	✓	✓		✓
VTDLY	Маска задержки символа вертикальной табуляции	XSI		✓		✓

Обратите внимание, что стандарт Single UNIX Specification не предусматривает использования классической функции `ioctl` для работы с терминальными устройствами. Вместо нее должны использоваться 13 функций, перечисленных в табл. 18.5. Причина в том, что тип последнего аргумента функции `ioctl` зависит от выполняемой операции, что делает невозможным контроль соответствия типов.

Хотя для работы с терминальными устройствами определено всего 13 функций, первые две из табл. 18.5 (`tcgetattr` и `tcsetattr`) могут использоваться для управления почти 70 параметрами (табл. 18.1–18.4). Обслуживание терминальных устройств осложняется большим количеством параметров и необходимостью определять, какие из них требуются для работы с конкретным устройством (терминалом, модемом, принтером или любым другим).

Таблица 18.5. Перечень функций, предназначенных для работы с терминалами

Функция	Описание
<code>tcgetattr</code>	Получить характеристики терминала (структура <code>termios</code>)
<code>tcsetattr</code>	Изменить характеристики терминала (структура <code>termios</code>)
<code>cfgetispeed</code>	Получить скорость ввода
<code>cfgetospeed</code>	Получить скорость вывода
<code>cfsetispeed</code>	Установить скорость ввода
<code>cfsetospeed</code>	Установить скорость вывода
<code>tcdrain</code>	Ждать завершения отправки всех выходных данных
<code>tcflow</code>	Приостановить прием или передачу
<code>tcflush</code>	Сбросить содержимое очереди ввода или вывода
<code>tcsendbreak</code>	Отправить символ <code>BREAK</code>
<code>tcgetpgrp</code>	Получить идентификатор группы процессов переднего плана
<code>tcsetpgrp</code>	Перевести группу процессов с заданным идентификатором на передний план
<code>tcgetsid</code>	Получить идентификатор группы процессов лидера сеанса для заданного управляющего терминала

Взаимосвязь функций из табл. 18.5 показана на рис. 18.3.

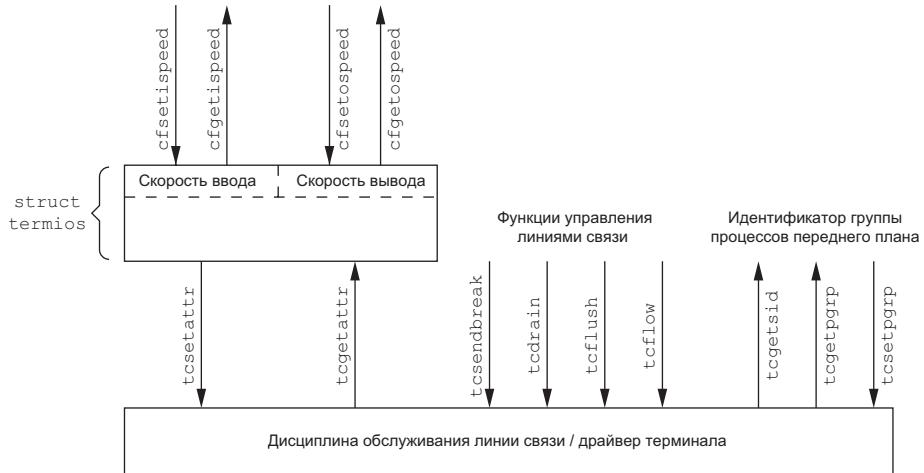


Рис. 18.3. Взаимосвязь функций, предназначенных для работы с терминалами

Стандарт POSIX.1 не оговаривает, в каком поле структуры `termios` хранится информация о скорости обмена, — это оставлено на усмотрение реализации. Некоторые системы, такие как Solaris, хранят сведения о скорости в поле `c_cflag`. Linux и системы, производные от BSD, такие как FreeBSD и Mac OS X, предусматривают в структуре два отдельных поля: одно для скорости ввода, другое для скорости вывода.

18.3. Специальные символы ввода

Стандарт POSIX.1 определяет 11 специальных (или служебных) символов ввода. Каждая реализация может дополнять этот список своими символами. В табл. 18.6 приводится список таких специальных символов.

Из 11 специальных символов, определяемых стандартом POSIX.1, 9 мы можем заменить практически любыми символами по своему желанию. Исключение составляют символы возврата каретки и перевода строки (`\r` и `\n` соответственно) и, возможно, символы STOP и START (зависит от реализации). Чтобы выполнить замену, нужно изменить соответствующие элементы массива `c_cc` в структуре `termios`. Элементы этого массива индексируются константами, имена которых начинаются с буквы V (третья колонка в табл. 18.6).

Стандарт POSIX.1 позволяет запретить действие этих символов. Для этого в соответствующий элемент массива нужно записать значение `_POSIX_VDISABLE`.

Таблица 18.6. Список специальных символов ввода

Символ	Описание	Индекс в массиве <code>c_cc</code>	Разрешается поле флаг	Типичное значение	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CR	Возврат каретки (нельзя изменить)	<code>c_lflag</code> <code>ICANON</code>	<code>\r</code>	<code>\r</code>	<code>\r</code>	<code>\r</code>	<code>\r</code>	<code>\r</code>	<code>\r</code>
DISCARD	Отменить вывод	<code>VDISCARD</code>	<code>c_lflag</code> <code>IEXTEN</code>	<code>\^O</code>		<code>\^O</code>	<code>\^O</code>	<code>\^O</code>	<code>\^O</code>
DSUSP	Огложенная приостановка (SIGSTP)	<code>VDSUSP</code>	<code>c_lflag</code> <code>ISIG</code>	<code>\^Y</code>		<code>\^Y</code>	<code>\^Y</code>	<code>\^Y</code>	<code>\^Y</code>
EOF	Конец файла	<code>VEOF</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>
EOL	Конец строки	<code>VEOL</code>	<code>c_lflag</code> <code>ICANON</code>			<code>\^D</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>
EOL2	Альтернативный конец строки	<code>VEOL2</code>	<code>c_lflag</code> <code>ICANON</code>			<code>\^D</code>	<code>\^D</code>	<code>\^D</code>	<code>\^D</code>
ERASE	Забой	<code>VERASE</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^H, \?</code>					
ERASE2	Альтернативный забой	<code>VERASE2</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^H, \?</code>					
INTR	Сигнал прерывания (SIGINT)	<code>VINTR</code>	<code>c_lflag</code> <code>ISIG</code>	<code>\^?, \^C</code>					
KILL	Стирание строки	<code>VKILL</code>	<code>c_lflag</code> <code>ISIG</code>	<code>\^U</code>	<code>\^U</code>	<code>\^U</code>	<code>\^U</code>	<code>\^U</code>	<code>\^U</code>
LNEXT	Экранирует следующий символ	<code>VLNEXT</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^V</code>	<code>\^V</code>	<code>\^V</code>	<code>\^V</code>	<code>\^V</code>	<code>\^V</code>
NL	Перевод строки	(нельзя изменить)	<code>c_lflag</code> <code>ICANON</code>	<code>\n</code>	<code>\n</code>	<code>\n</code>	<code>\n</code>	<code>\n</code>	<code>\n</code>
QUIT	Сигнал завершения (SIGQUIT)	<code>VQUIT</code>	<code>c_lflag</code> <code>ISIG</code>	<code>\^\\</code>	<code>\^\\</code>	<code>\^\\</code>	<code>\^\\</code>	<code>\^\\</code>	<code>\^\\</code>
REPRINT	Перепечатать входную строку	<code>VREPRINT</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^R</code>		<code>\^R</code>	<code>\^R</code>	<code>\^R</code>	<code>\^R</code>
START	Продолжить вывод	<code>VSTART</code>	<code>c_lflag</code> <code>IXON/IXOFF</code>	<code>\^Q</code>	<code>\^Q</code>	<code>\^Q</code>	<code>\^Q</code>	<code>\^Q</code>	<code>\^Q</code>
STATUS	Запрос состояния	<code>VSTATUS</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^T</code>		<code>\^T</code>	<code>\^T</code>	<code>\^T</code>	<code>\^T</code>
STOP	Остановить вывод	<code>VSTOP</code>	<code>c_lflag</code> <code>IXON/IXOFF</code>	<code>\^S</code>	<code>\^S</code>	<code>\^S</code>	<code>\^S</code>	<code>\^S</code>	<code>\^S</code>
SUSP	Сигнал приостановки (SIGSTP)	<code>VVSUSP</code>	<code>c_lflag</code> <code>ISIG</code>	<code>\^Z</code>	<code>\^Z</code>	<code>\^Z</code>	<code>\^Z</code>	<code>\^Z</code>	<code>\^Z</code>
WERASE	Стереть одно слово	<code>VWERASE</code>	<code>c_lflag</code> <code>ICANON</code>	<code>\^W</code>		<code>\^W</code>	<code>\^W</code>	<code>\^W</code>	<code>\^W</code>

В ранних версиях стандарта Single UNIX Specification поддержка константы _POSIX_VDISABLE была необязательной. Современная версия стандарта требует, чтобы эта константа поддерживалась всеми реализациями.

Все четыре платформы, обсуждаемые в этой книге, поддерживают такую возможность. В Linux 3.2.0 и Solaris 10 константа _POSIX_VDISABLE определена со значением 0, в FreeBSD 8.0 и Mac OS X 10.6.8 – со значением 0xff.

В некоторых ранних версиях UNIX действие того или иного специального символа можно было запретить, записав 0 в соответствующий элемент массива.

Пример

Прежде чем приступить к подробному описанию специальных символов, рассмотрим небольшую программу, которая изменяет их. Программа в листинге 18.1 запрещает символ прерывания и устанавливает символ Control-B в качестве символа конца файла.

Листинг 18.1. Запрет символа прерывания и изменение символа конца файла

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;
    long          vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("стандартное устройство ввода не является терминалом");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("ошибка fpathconf или _POSIX_VDISABLE не поддерживается");

    /* получить характеристики терминала */
    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("ошибка вызова функции tcgetattr");

    term.c_cc[VINTR] = vdisable; /* запретить действие символа INTR */
    term.c_cc[VEOF] = 2;          /* символ конца файла теперь Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("ошибка вызова функции tcsetattr");

    exit(0);
}
```

Обратите внимание на следующие аспекты:

- Изменение значений служебных символов выполняется, только если стандартное устройство ввода является терминалом. Для проверки вызывается функция `isatty` (раздел 18.9).
- Значение константы `_POSIX_VDISABLE` мы получаем с помощью функции `fpathconf`.

- Сначала функция `tcgetattr` (раздел 18.4) получает от ядра структуру `termios`. После модификации ее содержимого вызывается функция `tcsetattr`, которая устанавливает новые значения. Изменятся только те значения, которые были модифицированы явно.
- Запрет действия клавиши прерывания имеет иной смысл, нежели изменение диспозиции сигнала `SIGINT`. Программа в листинге 18.1 просто запрещает действие служебного символа, который заставляет драйвер терминала генерировать сигнал `SIGINT`. Но мы по-прежнему можем прервать работу процесса, послав ему сигнал с помощью функции `kill`.

Теперь подробнее опишем каждый служебный символ. Мы называем эти символы служебными символами ввода, но два из них — `START` и `STOP` (`Control-Q` и `Control-S`) — также имеют специальное назначение при выводе. Обратите внимание, что эти символы распознаются драйвером терминала и обрабатываются отдельно, после чего большинство из них уничтожается — они не передаются процессу при выполнении операции чтения. Исключение из этого правила составляют символы перевода строки (`NL`, `EOL`, `EOL2`) и возврата каретки (`CR`).

CR Символ возврата каретки. Мы не можем изменить его значение. Символ распознается в каноническом режиме ввода. Когда одновременно установлены флаги `ICANON` (канонический режим) и `ICRNL` (преобразование `CR` в `NL`) и сброшен флаг `IGNCR` (игнорировать `CR`), символ `CR` преобразуется в символ `NL` и воспринимается как символ `NL`. Этот символ передается читающему процессу (возможно, преобразованный в символ `NL`).

DISCARD Стандарт POSIX.1 позволяет запретить действие этих символов. Для этого в соответствующий элемент массива нужно записать значение `_POSIX_VDISABLE`.

Символ распознается в расширенном режиме ввода (`IEXTEN`) и уничтожает все вводимые символы, пока не будет встречен другой символ `DISCARD` или пока состояние терминала не будет изменено (флаг `FLUSHO`). После обработки этот символ уничтожается (то есть не передается процессу).

DSUSP Символ отложенной приостановки выполнения задания. Распознается в расширенном режиме ввода (`IEXTEN`), если поддерживается управление заданиями и установлен флаг `ISIG`. Аналогично символу `SUSP`, символ отложенной приостановки генерирует сигнал `SIGTSTP`, который передается всем процессам из группы процессов переднего плана (рис. 9.7). Единственное отличие — сигнал посыпается не когда будет введен символ, а когда процесс начнет чтение из управляющего терминала. Этот символ в процессе обработки уничтожается (то есть не передается процессу).

EOF Символ конца файла. Распознается в каноническом режиме ввода (`ICANON`). При вводе этого символа все данные во входной очереди немедленно передаются читающему процессу. Если очередь ввода была пуста, процессу возвращается счетчик прочитанных байтов, равный нулю. Как правило, чтобы передать программе признак конца файла, символ `EOF` вводится в начале новой строки. Когда этот символ вводится в каноническом режиме, он уничтожается после обработки (то есть не передается процессу).

EOL Дополнительный символ — разделитель строк, подобный символу **NL**. Распознается при вводе в каноническом режиме (**ICANON**) и передается читающему процессу. Обычно этот символ не используется.

EOL2 Еще один символ — разделитель строк, подобный символу **NL**. Интерпретируется так же, как символ **EOL**.

ERASE Символ забоя. Распознается в каноническом режиме ввода (**ICANON**). Стирает предыдущий символ в строке, но не переходит через начало строки. Когда этот символ вводится в каноническом режиме, он уничтожается после обработки (то есть не передается процессу).

ERASE2 Альтернативный символ забоя. Интерпретируется так же, как символ **ERASE**.

INTR Символ прерывания. Распознается при вводе, если установлен флаг **ISIG**, и приводит к генерации сигнала **SIGINT**, который посыпается всем процессам в группе процессов переднего плана (рис. 9.7). Этот символ уничтожается после обработки (то есть не передается процессу).

KILL Символ стирания строки. (Имя «kill» подобрано не совсем правильно, потому что напоминает имя функции **kill**, которая используется для посылки сигналов процессу. Этот символ лучше было бы назвать **line-erase** (стирание строки), так как он не имеет никакого отношения к сигналам.) Распознается в каноническом режиме ввода (**ICANON**). Удаляет всю строку и уничтожается после обработки (то есть не передается процессу).

LNEXT Экранирует следующий символ. Распознается в расширенном режиме ввода (**IEXTEN**); отменяет специальное назначение следующего за ним символа. Это относится к любым специальным символам из тех, что описываются в этом разделе. С помощью этого символа программе можно передать любой символ. После обработки символ **LNEXT** уничтожается, но следующий за ним символ передается процессу.

NL Символ перевода строки, который служит разделителем строк. Распознается в каноническом режиме ввода (**ICANON**). Передается процессу, выполняющему чтение.

QUIT Символ завершения. Распознается при вводе, если установлен флаг **ISIG**. Символ **QUIT** приводит к генерации сигнала **SIGQUIT**, который посыпается всем процессам из группы процессов переднего плана (рис. 9.7). После обработки этот символ уничтожается (то есть не передается процессу).

В табл. 10.1 указано, что различие между символами **INTR** и **QUIT** заключается в том, что при вводе **QUIT** по умолчанию процесс не просто завершается, а создает файл с дампом памяти (**core**).

REPRINT Символ перепечатки. Распознается в расширенном каноническом режиме ввода (установлены оба флага, **IEXTEN** и **ICANON**) и заставляет терминал вывести все символы из очереди ввода (повторный вывод). После обработки символ уничтожается (то есть не передается процессу).

START Символ запуска. Распознается при вводе, если установлен флаг **IXON**, и автоматически отправляется на вывод, если установлен флаг **IXOFF**. Прием символа **START** при установленном флаге **IXON** возобновляет ввод, который был приостановлен.

новлен введенным ранее символом **STOP**. В этом случае символ **START** уничтожается после обработки (то есть не передается процессу).

При установленном флаге **IHOFF** драйвер терминала автоматически генерирует символ **START**, чтобы продолжить ввод, который ранее был приостановлен из-за переполнения очереди ввода.

STATUS Символ запроса состояния терминала в BSD-системах. Распознается в расширенном каноническом режиме ввода (установлены оба флага, **TEXTEN** и **ICANON**) и генерирует сигнал **SIGINFO**, который передается всем процессам в группе процессов переднего плана (рис. 9.7). Дополнительно, если не установлен флаг **NOKERNINFO**, информация о состоянии группы процессов переднего плана выводится на терминал. После обработки этот символ уничтожается (то есть не передается процессу).

STOP Символ останова. Распознается при вводе, если установлен флаг **IXON**, и автоматически отправляется на вывод, если установлен флаг **IHOFF**. Прием символа **STOP** при установленном флаге **IXON** приостанавливает вывод данных. В этом случае после обработки символ **STOP** уничтожается (то есть не передается процессу). Приостановленный вывод возобновляется после ввода символа **START**.

Если установлен флаг **IHOFF**, драйвер терминала автоматически генерирует символ **STOP**, когда возникает угроза переполнения очереди ввода.

SUSP Символ приостановки выполнения задания. Распознается при вводе, если поддерживается управление заданиями и установлен флаг **SIG**. Символ **SUSP** приводит к генерации сигнала **SIGTSTP**, который передается всем процессам из группы процессов переднего плана (рис. 9.7). Уничтожается в процессе обработки (то есть не передается процессу).

WERASE Символ удаления слова. Распознается в расширенном каноническом режиме ввода (установлены оба флага, **TEXTEN** и **ICANON**) и приводит к удалению предыдущего слова. Сначала стираются любые предшествующие пробельные символы (пробелы или символы табуляции), затем символы предшествующей лексемы. Курсор ввода останавливается на месте первого символа стертой лексемы. Обычно границами лексем служат пробельные символы. Однако этот порядок распознавания границ лексем можно изменить, установив флаг **ALTWERASE**. Тогда границами лексем будут считаться любые не алфавитно-цифровые символы. Этот символ уничтожается в процессе обработки (то есть не передается процессу).

Еще один «символ», который мы должны определить, — это символ **BREAK** (прерывание передачи связи). В действительности **BREAK** не является символом, это скорее состояние, которое возникает в процессе асинхронной последовательной передачи данных. Драйвер терминала может быть извещен о наступлении состояния **BREAK** различными способами в зависимости от типа последовательного интерфейса.

*Большинство старых терминалов имели специальную клавишу с надписью **BREAK**, нажатие которой порождало состояние **BREAK**. По этой причине многие думают, что **BREAK** — это символ. На клавиатурах современных терминалов эта клавиша отсутствует. На клавиатурах персональных компьютеров клавиша **BREAK** несет совсем другую смысловую нагрузку. Например, с помощью комбинации клавиш **Control-BREAK** в ОС Windows можно прервать работу командного интерпретатора.*

При использовании асинхронного режима последовательной передачи данных **BREAK** представляет последовательность нулевых битов, которые продолжают передаваться дольше, чем требуется для передачи одного байта. Вся последовательность нулевых битов рассматривается как один «символ» **BREAK**. В разделе 18.8 мы узнаем, как можно передать «символ» **BREAK** с помощью функции `tcsendbreak`.

18.4. Получение и изменение характеристик терминала

Чтобы получить и установить структуру `termios`, можно воспользоваться двумя функциями: `tcgetattr` и `tcsetattr`. С их помощью можно проверить и изменить различные характеристики терминала и специальные символы, чтобы терминал действовал так, как нам требуется.

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termptr);

int tcsetattr(int fd, int opt, const struct termios *termptr);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Обе принимают указатель на структуру `termios` и либо возвращают текущие характеристики терминала, либо изменяют их. Обе функции могут работать только с терминальными устройствами, поэтому, когда дескриптор `fd` не является терминалом, они возвращают признак ошибки с кодом `ENOTTY` в переменной `errno`.

Аргумент `opt` функции `tcsetattr` позволяет определить, когда новые характеристики терминала должны вступить в силу. В этом аргументе можно передать одну из следующих констант.

TCSANOW Изменения вступают в силу немедленно.

TCSADRAIN Изменения вступают в силу после отправки всех данных в очереди вывода. Эта константа используется в случае, если изменяются характеристики вывода.

TCSAFLUSH Изменения вступают в силу после отправки всех данных в очереди вывода. Кроме того, когда изменения вступают в силу, все непрочитанные данные в очереди ввода уничтожаются (сбрасываются).

Возвращаемое значение функции `tcsetattr` может ввести в заблуждение. Дело в том, что она возвращает признак успешного завершения, если ей удалось выполнить изменение *хотя бы одной* характеристики, а не всех. Поэтому если функция `tcsetattr` вернула признак успешного выполнения, мы должны убедиться, что были выполнены все запрошенные изменения. Это означает, что после вызова `tcsetattr` следует вызвать функцию `tcgetattr` и сравнить фактические характеристики терминала с желаемыми.

Характеристики только что открытого терминала зависят от системы. Некоторые системы могут инициализировать терминал значениями по умолчанию, определяемыми реализацией. Другие сохраняют значения, полученные при последнем использовании терминала. Если необходимо обеспечить стандартное поведение терминала, терминальное устройство можно открыть с флагом `O_TTY_INIT` (раздел 3.3). Этот прием гарантирует инициализацию всех нестандартных полей структуры `termios` при вызове `tcgetattr` так, что терминал будет показывать ожидаемое поведение после изменения характеристик и вызова `tcsetattr`.

18.5. Флаги режимов терминала

В этом разделе мы подробно рассмотрим все флаги режимов терминала, перечисленные в табл. 18.1–18.4. Описания расположены в алфавитном порядке. Для каждого флага указывается, в каком из четырех полей он передается. (Обычно из названия флага трудно определить, для какого поля он предназначен.) Кроме того, для каждого флага указано, определен ли он в стандарте Single UNIX Specification, и перечисляются платформы, которые его поддерживают.

Каждому из перечисленных флагов соответствует один или более разрядов, если только флаг не является маской. Флаг-маска определяет набор сгруппированных разрядов, которые можно установить или сбросить. Мы перечислим имена всех масок и имена всех значений для каждой из них. Например, чтобы изменить размер символа, прежде всего нужно сбросить разряды, используя маску `CSIZE`, и затем установить одно из значений: `CS5`, `CS6`, `CS7` или `CS8`.

Шесть значений задержек, которые поддерживаются в Linux и Solaris, также являются масками: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY` и `VTDLY`. Значение каждой из них вы найдете на странице справочного руководства `termio(7I)` в Solaris. В любом случае нулевое значение маски означает отсутствие задержки. Если задержка определена, флаги `OFILL` и `OFDEL` определяют, должен ли драйвер терминала действительно выполнять задержку или вместо этого он должен посыпать символы-заполнители.

Пример

Программа в листинге 18.2 демонстрирует получение и изменение значений с помощью маски.

Листинг 18.2. Пример использования функций `tcgetattr` и `tcsetattr`

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("ошибка вызова функции tcgetattr");

    /* ... */
```

```

switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 бит на байт\n");
        break;
    case CS6:
        printf("6 бит на байт \n");
        break;
    case CS7:
        printf("7 бит на байт \n");
        break;
    case CS8:
        printf("8 бит на байт \n");
        break;
    default:
        printf("неизвестное количество битов на байт\n");
}
term.c_cflag &= .CSIZE; /* обнулить биты */
term.c_cflag |= CS8; /* установить 8 бит на байт */
if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
    err_sys("ошибка вызова функции tcsetattr");

exit(0);
}

```

А теперь опишем каждый из флагов.

ALTWERASE (*c_lflag*, FreeBSD, Mac OS X) Если флаг установлен, используется альтернативный алгоритм стирания слова при вводе символа **WERASE**. Предыдущее слово стирается не до первого пробельного символа, а до первого символа, не являющегося алфавитно-цифровым.

BRKINT (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если этот флаг установлен, а **IGNBRK** — нет, при появлении символа **BREAK** производится сброс очередей ввода и вывода и генерируется сигнал **SIGINT**. Этот сигнал посыпается группе процессов переднего плана, если терминальное устройство является управляющим терминалом.

Если оба флага, **BRKINT** и **IGNBRK**, сброшены, символ **BREAK** будет прочитан как символ **\0**, если флаг **PARMRK** сброшен, или как последовательность символов **\377, \0, \0**, если флаг **PARMRK** установлен.

BSDLY (*c_oflag*, XSI, Linux, Solaris) Маска задержки символа забоя. Маска может иметь два значения: **BS0** и **BS1**.

CBAUDEXT (*c_cflag*, Solaris) Расширенный диапазон скоростей передачи. Применяется, чтобы иметь возможность использовать скорости выше **B38400**. (Скорость приема/передачи мы рассмотрим в разделе 18.7.)

CCAR_OFLOW (*c_cflag*, FreeBSD, Mac OS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала **DCD** (Data-Carrier-Detect — обнаружение несущего сигнала) интерфейса RS-232. То же самое, что устаревший флаг **MDMBUF**.

CCTS_OFLOW (*c_cflag*, FreeBSD, Mac OS X, Solaris) Разрешает аппаратное управление выходным потоком данных с использованием сигнала **CTS** (Clear-To-Send — разрешение на передачу) интерфейса RS-232.

CDSR_OFLOW (*c_cflag*, FreeBSD, Mac OS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала RS-232 DSR (Data-Send-Ready — готовность к передаче).

CDTR_IFLOW (*c_cflag*, FreeBSD, MacOS X) Разрешает аппаратное управление выходным потоком данных с использованием сигнала DTR (Data-Terminal-Ready — готовность терминала) интерфейса RS-232.

CIBAUDEXT (*c_cflag*, Solaris) Расширенный диапазон скоростей приема. Применяется, чтобы иметь возможность использовать скорости приема данных выше B38400. (Скорость приема/передачи мы рассмотрим в разделе 18.7.)

CIGNORE (*c_cflag*, FreeBSD, Mac OS X) Игнорировать флаги режима управления.

CLOCAL (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, линии состояния модема игнорируются. Обычно это означает, что терминальное устройство подключено непосредственно к компьютеру. Если флаг не установлен, операция открытия терминального устройства блокируется, пока удаленный modem не ответит на звонок и не установит соединение.

CMSPAR (*c_oflag*, Linux) Выбор режима контроля четности по схеме MARK или SPACE. Если установлен флаг PARODD, бит четности всегда будет равен 1 (схема MARK). Иначе бит четности всегда будет равен 0 (схема SPACE).

CRDLY (*c_oflag*, XSI, Linux, Solaris) Мaska задержки символа CR. Возможные значения маски: CR0, CR1, CR2 и CR3.

CREAD (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Установка этого флага разрешает прием данных.

CRTSCTS (*c_cflag*, FreeBSD, Linux, Mac OS X, Solaris) Назначение этого флага зависит от платформы. В Solaris разрешает аппаратное управление исходящим потоком данных. На остальных трех платформах разрешает аппаратное управление как исходящим, так и входящим потоком данных (эквивалент CCTS_OFLOW|CRTS_IFLOW).

CRTS_IFLOW (*c_cflag*, FreeBSD, Mac OS X, Solaris) Разрешает аппаратное управление входным потоком данных с использованием сигнала RTS (Request-To-Send — запрос на передачу) интерфейса RS-232.

CRTSXOFF (*c_cflag*, Solaris) Разрешает аппаратное управление входным потоком данных. Проверяется состояние управляющего сигнала RTS интерфейса RS-232.

CSIZE (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Является маской, которая определяет количество битов на символ при приеме/передаче, не включая бит четности. Возможные значения маски: CS5, CS6, CS7 и CS8, которые соответствуют 5, 6, 7 и 8 битам на символ соответственно.

CSTOPB (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, используются два стоповых бита, иначе — один.

ECHO (*c_lflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, производится эхо-вывод введенных символов. Эхо-вывод может работать как в каноническом, так и в неканоническом режиме.

ECHOCTL (*c_lflag*, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ECHO, управляющие символы ASCII (символы с вось-

меричными кодами от 0 до 37 включительно), кроме символов ASCII TAB, ASCII NL, START и STOP, выводятся в форме X , где X — символ, сформированный из кода управляющего символа путем добавления к нему восьмеричного числа 100. Это означает, например, что управляющий символ Control-A (восьмеричный код 1) будет выведен как A . Кроме того, символ ASCII DELETE (восьмеричный код 177) будет выводиться как $^?$. Если флаг не установлен, управляющие символы ASCII выводятся как есть. Как и в случае с флагом ECHO, этот флаг воздействует на вывод управляющих символов как в каноническом, так и в неканоническом режиме. Следует отметить, что в некоторых системах символ EOF выводится несколько иначе, так как обычное его значение — Control-D. (Control-D — это ASCII-символ EOT, вызывающий на некоторых терминалах разрыв связи.) Подробнее см. в справочном руководстве.

ECHOE (*c_1flag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, при вводе символа ERASE производится стирание последнего символа в текущей строке на дисплее. Обычно это осуществляется драйвером терминала путем записи последовательности трех символов: шаг назад (backspace), вывод пробела (space), шаг назад (backspace).

Если драйвер терминала поддерживает символ WERASE, при установленном флаге ECHOE ввод WERASE выполняет стирание последнего слова за счет записи одной или нескольких последовательностей из этих же трех символов.

Если поддерживается флаг ECHOPRT, данное описание подразумевает, что флаг ECHOPRT не установлен.

ECHOK (*c_1flag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, символ KILL стирает текущую строку на дисплее или выводит символ NL (чтобы показать, что строка стерта).

Если поддерживается флаг ECHOKE, данное описание подразумевает, что флаг ECHOKE не установлен.

ECHOKE (*c_1flag*, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, символ KILL стирает каждый символ в текущей строке на дисплее. Способ, которым это достигается, зависит от установки флагов ECHOE и ECHOPRT.

ECHONL (*c_1flag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлен флаг ICANON, эхо-вывод символа NL производится, даже когда флаг ECHO не установлен.

ECHOPRT (*c_1flag*, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом установлены флаги ICANON и ECHO, ввод символа ERASE (и символа WERASE, если поддерживается) приводит к тому, что все удаляемые символы выводятся на печать. Это бывает удобно при работе с печатающими терминалами, так как позволяет точно увидеть, какие символы стерты.

EXTPROC (*c_1flag*, FreeBSD, Linux, Mac OS X) Если флаг установлен, каноническая обработка символов выполняется независимо от ОС. Например, когда устройство связи может само производить некоторую обработку данных, связанную с дисциплиной обслуживания линии связи. Аналогичная обработка производится при работе с псевдотерминалами (глава 19).

FFDLY (*c_oflag*, XSI, Linux, Solaris) Маска задержки символа FF (перевод страницы). Возможные значения маски: **FF0** и **FF1**.

FLUSHO (*c_lflag*, FreeBSD, Linux, Mac OS X, Solaris) При установке этого флага производится сброс очереди вывода. Этот флаг устанавливается при вводе символа **DISCARD** и сбрасывается при повторном вводе символа **DISCARD**. Этот флаг можно также установить напрямую.

HUPCL (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, когда последний процесс закроет терминальное устройство, модемное соединение будет разорвано.

ICANON (*c_lflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, вступает в силу канонический режим (раздел 18.10). В этом режиме разрешена обработка символов: **EOF**, **EOL**, **EOL2**, **ERASE**, **KILL**, **REPRINT**, **STATUS** и **WERASE**. Вводимые символы собираются в строки.

В каноническом режиме запрос на чтение из очереди ввода не может быть удовлетворен немедленно, если не получено хотя бы **MIN** байт или не истек тайм-аут **TIME** после приема последнего байта. Подробности см. в разделе 18.11.

ICRNL (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом сброшен флаг **IGNCR**, принимаемые символы **CR** преобразуются в символы **NL**.

IEXTEN (*c_lflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, распознаются и обрабатываются дополнительные специальные символы, определяемые реализацией.

IGNBRK (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной псевдосимвол **BREAK** игнорируется. Из описания флага **BRKINT** вы узнаете, когда псевдосимвол **BREAK** генерирует сигнал **SIGINT**, а когда может быть прочитан как обычные данные.

IGNCR (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной символ **CR** игнорируется. Если этот флаг сброшен, становится возможным прием символа **CR** или его преобразование в символ **NL** при установленном флаге **ICRNL**.

IGNPAR (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, входной байт, принятый с ошибкой в кадровой синхронизации (за исключением псевдосимвола **BREAK**) или с ошибкой контроля четности, игнорируется.

IMAXBEL (*c_iflag*, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при переполнении очереди ввода выдается звуковой сигнал.

INLCR (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, принимаемые символы **NL** преобразуются в символы **CR**.

INPCK (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешается проверка бита четности при вводе. Если сброшен, проверка бита четности при вводе не производится.

«Контроль четности» и «проверка бита четности» — разные понятия. За контроль четности отвечает флаг **PARENB**. Установка этого флага обычно приводит к тому,

что драйвер последовательного интерфейса генерирует биты четности для исходящих символов и проверяет для входящих. Флаг **PARODD** определяет схему контроля четности — ODD (чет) или EVEN (нечет). Если входящий символ поступает с неверным значением бита четности, проверяется состояние флага **TNPCK**. Если он установлен, проверяется состояние флага **IGNPAR** (чтобы определить, следует ли игнорировать символ, поступивший с ошибкой контроля четности). Если байт не должен игнорироваться, проверяется состояние флага **PARMRK**, чтобы узнать, следует ли передавать процессу символы, принятые с ошибкой.

ISIG (*c_lflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при обработке входящих символов выполняется проверка необходимости сгенерировать сигнал (символы INTR, QUIT, SUSP и DSUSP). Если принят один из этих символов, будет сгенерирован соответствующий сигнал.

ISTRIP (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, в принятых байтах сбрасывается 8-й бит. Если флаг сброшен, обрабатываются все 8 битов.

IUCLC (*c_iflag*, Linux, Solaris) Если флаг установлен, символы верхнего регистра при вводе преобразуются в символы нижнего регистра.

IUTF8 (*c_iflag*, Linux, Mac OS X) Включает поддержку многобайтных символов UTF-8 для функции стирания.

IXANY (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешается возобновление вывода по любому символу.

IXOFF (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешено управление входным потоком с помощью символов START/STOP. Когда возникает угроза переполнения очереди ввода, драйвер терминала отправляет символ STOP. Этот символ должен распознаваться устройством, отправляющим данные, и вызывать приостановку передачи. Позднее, когда очередь ввода освободится, драйвер терминала отправит символ START, в результате передающее устройство продолжит передачу данных.

IXON (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, разрешено управление выходным потоком с помощью символов START/STOP. Получив символ STOP, драйвер терминала приостанавливает вывод данных. Когда драйвер получит символ START, он возобновит вывод данных. Если этот флаг сброшен, символы START и STOP будут передаваться читающему процессу.

MDMBUF (*c_cflag*, FreeBSD, Mac OS X) Разрешает аппаратное управление потоком данных с использованием сигнала DCD модема. Это устаревшее название флага **CCAR_OFLOW**.

NLDLY (*c_oflag*, XSI, Linux, Solaris) Маска задержки символа NL. Возможные значения маски: NL0 и NL1.

NOFLSH (*c_lflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) По умолчанию, когда драйвер терминала генерирует сигнал **SIGINT** или **SIGQUIT**, обе очереди (ввода и вывода) сбрасываются. Кроме того, когда генерируется сигнал **SIGSUSP**, сбрасывается очередь ввода. Если установлен флаг **NOFLSH**, то при генерации сигналов содержимое очередей не сбрасывается.

NOKERNINFO (*c_lflag*, FreeBSD, Mac OS X) Установка флага предотвращает вывод информации о группе процессов переднего плана при вводе символа **STATUS**. Независимо от состояния флага, символ **STATUS** вызывает генерацию сигнала **SIGINFO**, который посыпается группе процессов переднего плана.

OCRNL (*c_oflag*, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, символы **CR** при выводе преобразуются в символы **NL**.

OFDEL (*c_oflag*, XSI, Linux, Solaris) Если флаг установлен, в качестве символа-заполнителя выводится символ ASCII **DEL**, иначе — ASCII **NUL**. Подробности см. в описании флага **OFILL**.

OFILL (*c_oflag*, XSI, Linux, Solaris) Если флаг установлен, вместо временной задержки будут передаваться символы-заполнители (ASCII **DEL** или ASCII **NUL**). Существует шесть масок задержки: **BSDLY**, **CRDLY**, **FFDLY**, **NLDLY**, **TABDLY** и **VTDLY**.

OLCUC (*c_oflag*, Linux, Solaris) Если флаг установлен, символы нижнего регистра при выводе преобразуются в символы верхнего регистра.

ONLCR (*c_oflag*, XSI, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, на выходе символы **NL** преобразуются в последовательности символов **CR-NL**.

ONLRET (*c_oflag*, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, предполагается, что на выходе символ **NL** должен выполнять функцию символа возврата каретки.

ONOCR (*c_oflag*, XSI, FreeBSD, Linux, Solaris) Если флаг установлен, символ **CR**, находящийся в начале строки, не выводится.

ONOEOF (*c_oflag*, FreeBSD, Mac OS X) Если флаг установлен, символ **EOT** (^D) при выводе уничтожается. Это может потребоваться при работе с некоторыми терминалами, интерпретирующими символ **Control-D** как разрыв соединения.

OPOST (*c_oflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, выполняется дополнительная обработка выходных данных, зависящая от реализации. В табл. 18.4 перечисляются различные флаги, определяемые отдельными реализациями.

OXTABS (*c_oflag*, FreeBSD, Mac OS X) Если флаг установлен, при выводе символы табуляции заменяются пробелами. При использовании этого флага возникает тот же эффект, что и при установке маски задержки символа горизонтальной табуляции (**TABDLY**) в значение **XTABS** или **TAB3**.

PARENB (*c_cflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Разрешает генерацию бита четности для исходящих символов и его проверку для входящих символов. Если установлен флаг **PARODD**, контроль ведется по четности, иначе — по нечетности. Дополнительные сведения по этой теме вы найдете в описаниях флагов **INPCK**, **IGNPAR** и **PARMRK**.

PAREXT (*c_cflag*, Solaris) Выбор схемы контроля четности MARK/SPACE. Если установлен флаг **PARODD**, бит четности всегда будет равен 1 (MARK), иначе — 0 (SPACE).

PARMRK (*c_iflag*, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и при этом сброшен флаг **IGNPAR**, входной байт, принятый с ошибкой кадровой синхронизации (за исключением псевдосимвола **BREAK**) или с ошибкой контроля четности, передается процессу в виде последовательности из трех сим-

волов: \377, \0, X, где X – байт, принятый с ошибкой. Если флаг `ISTRIP` сброшен, обычный символ \377 передается процессу в виде двух символов: \377, \377. Если флаги `IGNPAR` и `PARMRK` не установлены, входной байт, принятый с ошибкой кадровой синхронизации (за исключением псевдосимвола `BREAK`) или с ошибкой контроля четности, передается процессу в виде одного байта \0.

PARODD (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, выбирается схема контроля бита паритета по четности исходящих и входящих данных. Иначе бит четности проверяется на нечетность. Обратите внимание: управление контролем четности производится с помощью флага `PARENB`.

Флаг `PARODD` используется также для выбора схемы контроля `MARK` или `SPACE`, если установлен один из флагов: `CMSPAR` или `PAREXT`.

PENDIN (`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен, при вводе очередного символа будут напечатаны все символы, которые еще не были прочитаны из очереди ввода. Действие этого флага напоминает то, что происходит при нажатии клавиши `REPRINT`.

TABDLY (`c_oflag`, XSI, Linux, Mac OS X, Solaris) Маска задержки символа горизонтальной табуляции. Возможные значения маски: `TAB0`, `TAB1`, `TAB2` и `TAB3`.

Значение маски `XTABS` эквивалентно `TAB3`. Оно заставляет систему заменять символы табуляции пробелами. При этом предполагается, что расстояние между соседними позициями табуляции составляет восемь пробелов, но его можно изменить.

TOSTOP (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Если флаг установлен и реализация поддерживает управление заданиями, при попытке вывода в управляющий терминал группе фоновых процессов посыпается сигнал `SIGTTOU`. По умолчанию этот сигнал приостанавливает работу процессов в группе. Этот сигнал не генерируется драйвером, если фоновый процесс, который произвел попытку записи в управляющий терминал, либо заблокировал сигнал, либо игнорирует его.

VTDLY (`c_oflag`, XSI, Linux, Solaris) Маска задержки символа вертикальной табуляции. Возможные значения маски: `VT0` и `VT1`.

XCASE (`c_lflag`, Linux, Solaris) Если флаг установлен и при этом установлен флаг `ICANON`, все исходящие символы преобразуются в верхний регистр, а входящие – в нижний. В этом случае ввод символа верхнего регистра необходимо предварять символом обратного слеша. Аналогично, при выводе символов верхнего регистра система также предваряет их символом обратного слеша. (Этот флаг считается устаревшим, поскольку на сегодняшний день терминалы, которые могут отображать только символы верхнего регистра, практически не используются.)

18.6. Команда `stty`

Состояние всех флагов, описанных в предыдущем разделе, можно проверить и изменить из программы, с помощью функций `tcgetattr` и `tcsetattr` (раздел 18.4), и из командной строки (или из сценариев командной оболочки), с помощью команды `stty(1)`. Эта команда представляет собой упрощенный интерфейс к первым шести функциям из табл. 18.5. Если запустить команду с ключом `-a`, она выведет все характеристики терминала:

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscs
        -dsrflow -dtrflow -mdmbuf
cchars: discard = .O; dsusp = .Y; eof = .D; eol = <undef>;
        eol2 = <undef>; erase = .H; erase2 = ?.; intr = .C; kill = .U;
        lnext = .V; min = 1; quit = .; reprint = .R; start = .Q;
        status = .T; stop = .S; susp = .Z; time = 0; werase = .W;
```

Дефис, предшествующий имени флага, означает, что флаг сброшен. В последних четырех строках выводятся текущие значения специальных символов (раздел 18.3). В первой строке выводится количество строк и символов в строке для текущего терминала — более подробно мы обсудим эти величины в разделе 18.12.

Для получения и изменения характеристик терминала команда stty использует стандартное устройство ввода. Некоторые старые версии команды использовали для этих целей стандартное устройство вывода, однако стандарт POSIX.1 явно требует, чтобы использовалось стандартное устройство ввода. Все четыре реализации, обсуждаемые в этой книге, предоставляют версию stty, которая работает со стандартным устройством ввода. Это означает, что если нас интересуют характеристики терминала tty1a, можно ввести следующую команду:

```
stty -a </dev/tty1a
```

18.7. Функции для работы со скоростью передачи

Традиционно *скорость передачи* измеряется в бодах, что в наши дни можно трактовать как «биты в секунду». Хотя большинство терминалов используют одно и то же значение скорости как для ввода, так и для вывода, тем не менее ввод и вывод можно производить на разных скоростях, если аппаратура это позволяет.

```
#include <termios.h>
speed_t cfgetispeed(const struct termios *termptr);
speed_t cfgetospeed(const struct termios *termptr);
```

Обе возвращают значение скорости в бодах

```
int cfsetispeed(struct termios *termptr, speed_t speed);
int cfsetospeed(struct termios *termptr, speed_t speed);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Значение скорости, возвращаемое функциями `cfget` и передаваемое в виде аргументов *speed* функциям `cfset`, представляет собой одну из следующих констант: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200` или `B38400`. Константа `B0` обозначает «разрыв соединения». Если с помощью функции `tcsetattr` устанавливается скорость вывода `B0`, линии управления модемом не задействуются.

Большинство систем определяют две дополнительные константы: B57600 и B115200.

При использовании этих функций необходимо понимать, что скорости ввода и вывода хранятся в структуре `termios`, как показано на рис. 18.3. Прежде чем вызвать какую-либо из функций `cfget`, необходимо сначала получить содержимое структуры `termios` устройства с помощью функции `tcgetattr`. Аналогично, после установки значения скорости в структуре `termios` функциями `cfset` необходимо сохранить эту структуру с помощью функции `tcsetattr`. Если было установлено ошибочное значение скорости, мы не узнаем об этом, пока не вызовем функцию `tcsetattr`.

Четыре функции, предназначенные для работы со значениями скорости, скрывают от прикладных программ различные способы представления скорости в разных реализациях. Так, например, системы, производные от BSD, сохраняют значения скорости в числовом виде (то есть скорость 9600 хранится как число 9600), тогда как Linux и производные от System V представляют скорость в виде битовой маски. Функции `cfget` возвращают, а функции `cfset` принимают значения скорости в том виде, в каком они хранятся в структуре `termios`.

18.8. Функции управления линией связи

Следующие четыре функции позволяют управлять процессом обмена между терминалами. Все четыре требуют, чтобы аргумент *fd* представлял дескриптор терминального устройства, иначе они будут возвращать признак ошибки с кодом `ENOTTY` в переменной `errno`.

```
#include <termios.h>

int tcdrain(int fd);
int tcflow(int fd, int action);
int tcflush(int fd, int queue);
int tcsendbreak(int fd, int duration);
```

Все четыре возвращают 0 в случае успеха, -1 — в случае ошибки

Функция `tcdrain` ожидает, пока будут отправлены все выходные данные. Функция `tcflow` позволяет управлять входным и выходным потоками. В аргументе *action* допускается передавать одно из следующих значений.

TCOFF Приостановить вывод.

TCON Возобновить ранее приостановленный вывод.

TCIOFF Система отправляет символ STOP, который должен заставить терминал приостановить передачу.

TCION Система отправляет символ START, который должен заставить терминал возобновить передачу.

Функция `tcflush` позволяет сбросить (удалить) данные из очереди ввода (принятые драйвером терминала, но еще не прочитанные процессом) или немедленно отправить данные из очереди вывода (записанные процессом, но еще не отправленные). В аргументе *queue* допускается передавать одно из следующих значений:

TCIFLUSH Сбросить данные из очереди ввода.

TCOFLUSH Сбросить данные из очереди вывода.

TCIOFLUSH Сбросить данные из обеих очередей.

Функция `tcsendbreak` посыпает последовательность нулевых битов в течение заданного времени. Если в аргументе *duration* передать 0, продолжительность передачи будет находиться в диапазоне от 0,25 до 0,5 секунды. Стандарт POSIX.1 указывает, что продолжительность передачи при ненулевом значении аргумента *duration* определяется самой реализацией.

18.9. Идентификация терминала

Традиционно управляющий терминал в большинстве версий UNIX соответствует устройству `/dev/tty`. Стандарт POSIX.1 определяет функции, которые могут использоваться для получения имени управляющего терминала во время выполнения.

```
#include <stdio.h>
char *ctermid(char *ptr);
```

Возвращает указатель на строку с именем управляющего терминала
в случае успеха, указатель на пустую строку — в случае ошибки

Если в аргументе *ptr* передается непустой указатель, предполагается, что он указывает на буфер длиной не менее `L_ctermid` байт. В этом буфере будет сохранено имя управляющего терминала вызывающего процесса. Константа `L_ctermid` определена в файле `<stdio.h>`. Если в аргументе *ptr* передать пустой указатель, функция выделит место для буфера (обычно в статической области памяти) и сохранит строку с именем управляющего терминала вызывающего процесса в этом буфере.

В обоих случаях функция передает адрес буфера вызывающему процессу в виде возвращаемого значения. Поскольку большинство версий UNIX используют в качестве имени управляющего терминала `/dev/tty`, эта функция предназначена для обеспечения переносимости приложений в другие операционные системы.

На всех четырех платформах, описываемых в данной книге, функция ctermid возвращает имя /dev/tty.

Пример — функция ctermid

В листинге 18.3 приводится реализация функции ctermid стандарта POSIX.1.

Листинг 18.3. Функция ctermid стандарта POSIX.1

```
#include <stdio.h>
#include <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strncpy(str, "/dev/tty")); /* функция strncpy() вернет str */
}
```

Обратите внимание: мы никак не защищены от переполнения буфера, предоставляемого вызывающим процессом, поскольку у нас нет возможности определить его размер.

Для приложений UNIX больший интерес представляют две другие функции: **isatty**, которая возвращает истинное значение, если дескриптор является дескриптором терминального устройства, и **ttynname**, которая возвращает полное имя файла устройства терминала.

```
#include <unistd.h>

int isatty(int fd);
```

Возвращает 1 (истина), если *fd* представляет терминальное устройство, 0 (ложь) — в противном случае

```
char *ttynname(int fd);
```

Возвращает указатель на строку с полным именем специального файла устройства, соответствующего терминалу, или NULL — в случае ошибки

Пример — функция isatty

Функция **isatty** тривиальна в реализации, что хорошо видно из листинга 18.4. Она просто пытается вызвать одну из терминальных функций (которая в случае успеха ничего особенного не делает) и проверяет возвращаемое значение.

Листинг 18.4. Функция `isatty` стандарта POSIX.1

```
#include <termios.h>

int
isatty(int fd)
{
    struct termios ts;

    return(tcgetattr(fd, &ts) != -1); /* истина, если нет ошибки (терминал) */
}
```

Протестируем работу нашей функции `isatty` с помощью программы из листинга 18.5.

Листинг 18.5. Тест функции `isatty`

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "не tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "не tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "не tty");
    exit(0);
}
```

Запустив эту программу, мы получили:

```
$ ./a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: не tty
fd 1: tty
fd 2: не tty
```

Пример — функция `ttynname`

Функция `ttynname` (листинг 18.6) гораздо сложнее, так как она должна просмотреть весь список устройств и отыскать совпадение.

Листинг 18.6. Функция `ttynname` стандарта POSIX.1

```
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>

struct devdir {
    struct devdir *d_next;
    char         *d_name;
};

static struct devdir *head;
```

```
static struct devdir    *tail;
static char          pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir    *ddp;
    int             len;

    len = strlen(dirname);

    /*
     * Пропустить каталоги ., .. и /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/') ||
        (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    if((ddp = malloc(sizeof(struct devdir))) == NULL)
        return;
    if((ddp->d_name = strdup(dirname)) == NULL) {
        free(ddp);
        return;
    }

    ddp->d_next = NULL;
    if (tail == NULL) {
        head = ddp;
        tail = ddp;
    } else {
        tail->d_next = ddp;
        tail = ddp;
    }
}

static void
cleanup(void)
{
    struct devdir    *ddp, *nddp;

    ddp = head;

    while (ddp != NULL) {
        nddp = ddp->d_next;
        free(ddp->d_name);
        free(ddp);
        ddp = nddp;
    }
    head = NULL;
    tail = NULL;
}

static char *
searchdir(char *dirname, struct stat *fdstatp)
{
    struct stat      devstat;
    DIR            *dp;
    int             devlen;
```

```
struct dirent *dirp;

strcpy(pathname, dirname);
if ((dp = opendir(dirname)) == NULL)
    return(NULL);
strcat(pathname, "/");
devlen = strlen(pathname);
while ((dirp = readdir(dp)) != NULL) {
    strncpy(pathname + devlen, dirp->d_name,
            _POSIX_PATH_MAX - devlen);

    /*
     * Пропустить псевдонимы.
     */
    if (strcmp(pathname, "/dev/stdin") == 0 ||
        strcmp(pathname, "/dev/stdout") == 0 ||
        strcmp(pathname, "/dev/stderr") == 0)
        continue;
    if (stat(pathname, &devstat) < 0)
        continue;
    if (S_ISDIR(devstat.st_mode)) {
        add(pathname);
        continue;
    }
    if (devstat.st_ino == fdstatp->st_ino &&
        devstat.st_dev == fdstatp->st_dev) { /* совпадение найдено */
        closedir(dp);
        return(pathname);
    }
}
closedir(dp);
return(NULL);
}

char *
ttynname(int fd)
{
    struct stat    fdstat;
    struct devdir  *ddp;
    char          *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    rval = searchdir("/dev", &fdstat);
    if (rval == NULL) {
        for (ddp = head; ddp != NULL; ddp = ddp->d_next)
            if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                break;
    }
    cleanup();
    return(rval);
}
```

Функция просматривает каталог `/dev` и отыскивает запись с указанным номером устройства и индексным узлом. Мы уже говорили в разделе 4.24, что каждая файловая система обладает уникальным номером устройства (поле `st_dev` структуры `stat`, раздел 4.2), а каждая запись в каталоге — уникальным номером индексного узла (поле `st_ino` структуры `stat`). Предполагается, что когда функция обнаружит запись с соответствующим номером устройства и номером индексного узла, можно сделать вывод, что найдено требуемое устройство. Функция также могла бы проверить, совпадает ли содержимое поля `st_rdev` с заданными старшим и младшим номерами устройства и является ли найденный файл специальным файлом символьного устройства. Но поскольку она уже убедилась, что переданный ей дескриптор является терминальным устройством и специальным файлом символьного устройства (а номер устройства и номер индексного узла в системе являются уникальными), в дополнительных проверках нет необходимости.

Специальный файл устройства терминала может находиться в одном из подкаталогов каталога `/dev`. Поэтому необходимо обойти все дерево подкаталогов в `/dev`. Мы пропускаем специальные каталоги `/dev/..`, `/dev/..` и `/dev/fd`. Мы также не рассматриваем псевдонимы `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, поскольку они являются символическими ссылками, ведущими в каталог `/dev/fd`.

Работоспособность нашей функции `ttynname` можно проверить с помощью программы в листинге 18.7.

Листинг 18.7. Проверка функции `ttynname`

```
#include "apue.h"

int
main(void)
{
    char    *name;

    if (isatty(0)) {
        name = ttynname(0);
        if (name == NULL)
            name = "не определено";
    } else {
        name = "не tty";
    }
    printf("fd 0: %s\n", name);

    if (isatty(1)) {
        name = ttynname(1);
        if (name == NULL)
            name = "не определено";
    } else {
        name = "не tty";
    }
    printf("fd 1: %s\n", name);

    if (isatty(2)) {
        name = ttynname(2);
        if (name == NULL)
            name = "не определено";
    } else {
```

```
        name = "не tty";
    }
    printf("fd 2: %s\n", name);

    exit(0);
}
```

Запустив программу из листинга 18.7, мы получили следующие результаты:

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: не tty
```

18.10. Канонический режим

Канонический режим очень прост: мы запускаем операцию чтения, а драйвер терминала возвращает строку, когда она будет введена. Операция чтения завершается в следующих ситуациях:

- Когда прочитано запрошенное количество байтов. Стока при этом может быть прочитана не до конца. Если прочитана только часть строки, оставшаяся ее часть не будет потеряна; ее можно прочитать следующей операцией чтения.
- Когда достигнут разделитель строк. В разделе 18.3 мы уже говорили, что в каноническом режиме разделителями строк служат символы NL, EOL, EOL2 и EOF. Кроме того, в разделе 18.5 говорилось, что символ CR также рассматривается как разделитель строк, если установлен флаг ICRNL, а флаг IGNCR сброшен. Помните, что из этих пяти разделителей только один (EOF) уничтожается драйвером терминала в процессе обработки. Остальные четыре передаются читающему процессу в качестве последнего символа строки.
- Операция чтения также может завершиться, если перехвачен сигнал и системный вызов не перезапускается автоматически (раздел 10.5).

Пример — функция `getpass`

Теперь продемонстрируем реализацию функции `getpass`, которая читает пароль, вводимый пользователем с терминала. Эта функция вызывается программами `login(1)` и `crypt(1)`. Чтобы прочитать пароль, функция должна отключить эхоВывод, но оставить терминал в каноническом режиме, поскольку пароль представляет собой полноценную строку. В листинге 18.8 приводится типичная реализация этой функции в UNIX.

Несколько замечаний к данному примеру:

- Вместо того чтобы жестко «зашивать» в программу имя управляющего терминала (`/dev/tty`), мы воспользуемся функцией `ctermid`.
- Управляющий терминал необходим для выполнения операций записи/чтения, поэтому функция будет возвращать признак ошибки, если ей не удастся открыть соответствующее устройство для чтения и записи. Функция `getpass` в версии для BSD читает данные из стандартного ввода и выводит сообщение

в стандартный вывод сообщений об ошибках, если ей не удалось открыть терминал для чтения и записи. В версии для Solaris вывод производится только в стандартный вывод сообщений об ошибках, а ввод — только из управляющего терминала.

Листинг 18.8. Реализация функции getpass

```
#include <signal.h>
#include <stdio.h>
#include <termios.h>

#define MAX_PASS_LEN 8 /* максимальное количество символов в пароле */

char *
getpass(const char *prompt)
{
    static char      buf[MAX_PASS_LEN + 1]; /* нулевой байт в конце */
    char            *ptr;
    sigset(SIGPOLL, 0);
    struct termios ts, ots;
    FILE           *fp;
    int             c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(SIGPOLL);
    sigaddset(SIGPOLL, SIGPOLLIN); /* заблокировать SIGPOLL */
    sigaddset(SIGPOLL, SIGPOLLHUP); /* заблокировать SIGPOLLHUP */
    sigprocmask(SIG_BLOCK, &SIGPOLL, &osig); /* сохранить маску */

    tcgetattr(fileno(fp), &ts); /* сохранить состояние терминала */
    ots = ts; /* скопировать структуру */
    ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getchar(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    *ptr = 0; /* завершающий нулевой символ */
    putc('\n', fp); /* вывести символ перевода строки */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* восстановить состояние терминала */
    sigprocmask(SIG_SETMASK, &osig, NULL); /* восстановить маску */
    fclose(fp); /* завершить работу с /dev/tty */
    return(buf);
}
```

- Функция блокирует сигналы **SIGINT** и **SIGTSTP**. Если этого не сделать, ввод символа **INTR** может завершить работу программы и оставить терминал в состоянии запрещенного эхо-вывода. Аналогично, ввод символа **SUSP** может приостановить работу программы и вернуть управление командной оболочке при запрещенном эхо-выводе. Сигналы остаются заблокированными, пока не будет восстановлено прежнее состояние терминала. Если эти сигналы будут

сгенерированы во время чтения пароля, они останутся в состоянии ожидания обработки, пока функция не вернет управление. Существуют и другие способы обработки этих сигналов. Некоторые версии просто игнорируют сигнал `SIGINT` (сохранив его предыдущую диспозицию) во время работы функции `getpass`, восстанавливая диспозицию сигнала в исходное состояние перед выходом из функции. Другие перехватывают сигнал `SIGINT` (сохранив его предыдущую диспозицию) и после восстановления состояния терминала и диспозиции сигнала посыпают его себе с помощью функции `kill`. Но ни одна версия функции `getpass` не игнорирует, не блокирует и не перехватывает сигнал `SIGQUIT` — то есть ввод символа `QUIT` может прервать работу программы и, скорее всего, оставить терминал в состоянии отключенного эхо-вывода.

- Следует помнить, что некоторые командные оболочки (в первую очередь Korn shell) включают эхо-вывод, когда ожидают интерактивного взаимодействия с пользователем. Эти командные оболочки предоставляют возможность редактирования командной строки и поэтому корректируют состояние терминала всякий раз, когда вводится очередная команда. То есть если запустить эту программу в одной из таких командных оболочек и затем прервать ее выполнение вводом символа `QUIT`, режим эхо-вывода будет восстановлен. Другие командные оболочки, такие как Bourne shell, при аварийном завершении программы не восстанавливают состояние терминала и оставляют его с отключенным эхо-выводом. В этом случае эхо-вывод можно восстановить с помощью команды `stty`.
- Наша версия функции `getpass` для работы с управляющим терминалом использует функции стандартной библиотеки ввода/вывода. Мы специально назначаем небуферизованный режим работы потока — иначе могут возникнуть взаимовлияния между операциями чтения и записи, производимыми над потоком (нам потребовалось бы добавить несколько вызовов функции `fflush`). Можно было бы использовать функции небуферизованного ввода/вывода (глава 3), но тогда пришлось бы эмулировать поведение функции `getc` через функцию `read`.
- Мы читаем только первые восемь символов пароля. Любые последующие символы просто игнорируются.

Программа в листинге 18.9 вызывает функцию `getpass` и выводит то, что было введено, позволяя убедиться, что символы `ERASE` и `KILL` обрабатываются должным образом (как и следует ожидать при работе в каноническом режиме).

Листинг 18.9. Вызов функции `getpass`

```
#include "apue.h"

char *getpass(const char *);

int
main(void)
{
    char     *ptr;

    if ((ptr = getpass("Введите пароль:")) == NULL)
        err_sys("ошибка вызова функции getpass");
```

```

printf("пароль: %s\n", ptr);

/* здесь можно работать с паролем (например, зашифровать его) ... */

while (*ptr != 0)
    *ptr++ = 0; /* забить нулями, когда он стал больше не нужен */
exit(0);
}

```

Всякий раз, когда программа завершает работу с паролем в виде открытого текста, она должна забить соответствующую область памяти нулями — просто для безопасности. Если программа завершится аварийно с созданием файла core, доступного для чтения для всех, или если другой процесс сможет просмотреть содержимое памяти нашего процесса, пароль может быть прочитан. (Под «паролем в виде открытого текста» мы подразумеваем строку, которая вводится с клавиатуры в ответ на запрос функции `getpass`. В большинстве случаев, получив пароль, программы UNIX тут же шифруют его. Так, например, поле `pw_passwd` в файле паролей хранит пароль в зашифрованном, а не в открытом виде.)

18.11. Неканонический режим

Переход в неканонический режим осуществляется сбросом флага `ICANON` в поле `c_lflag` структуры `termios`. В неканоническом режиме принимаемые символы не собираются в строки, а служебные символы `ERASE`, `KILL`, `EOF`, `NL`, `EOL`, `EOL2`, `CR`, `REPRINT`, `STATUS` и `WERASE` не обрабатываются.

Как мы уже говорили, канонический режим очень прост в использовании: система возвращает одну строку символов за раз. Но как узнать, когда система сможет вернуть нам данные при использовании неканонического режима? Если читать данные по одному байту, это повлечет непроизводительное расходование системных ресурсов. (Вспомните табл. 3.3, где приводились экспериментальные данные, наглядно показывающие, что при удвоении объема читаемых данных в два раза снижаются накладные расходы.) Не всегда можно заранее сказать, какое количество данных находится в очереди ввода.

Решение состоит в том, чтобы сообщить системе, когда она должна возвращать управление — по прочтении заданного объема данных или по прошествии определенного времени. Для этих целей в массиве `c_cc` структуры `termios` предусмотрены два элемента, `MIN` и `TIME`, с индексами `VMIN` и `VTIME`.

Элемент `MIN` определяет минимальное количество байтов, по прочтении которого функция `read` должна возвращать управление. Элемент `TIME` задает количество десятых долей секунды, в течение которых следует ожидать поступления данных. Соответственно имеется четыре возможных случая.

Случай А: $\text{MIN} > 0$, $\text{TIME} > 0$

Элемент `TIME` определяет время таймера, который запускается только после приема первого байта. Если `MIN` байт будет принято раньше, чем истечет время таймера, функция `read` вернет `MIN` байт. Если время таймера истечет до того, как будет принято `MIN` байт, функция `read` вернет столько байтов, сколько было при-

нято. (Будет возвращен по меньшей мере один байт, поскольку таймер запускается только после приема первого байта.) В этом случае вызывающий процесс блокируется, пока не будет принят первый байт. Если во время вызова функции `read` в очереди уже имеются данные, считается, что эти данные приняты сразу же после входа в функцию `read`.

Случай Б: `MIN > 0, TIME == 0`

Функция `read` не вернет управление, пока не прочитает `MIN` байт. В результате процесс может оказаться заблокированным на неопределенное время.

Случай В: `MIN == 0, TIME > 0`

Элемент `TIME` задает время таймера чтения, который запускается в момент вызова функции `read`. (Сравните со случаем А, когда таймер запускается только после приема первого байта.) Функция `read` вернет управление после приема первого байта или по истечении времени таймера. Если время таймера истечет после приема хотя бы одного байта, функция `read` вернет значение 0.

Случай Г: `MIN == 0, TIME == 0`

Если в очереди имеются какие-либо данные, функция `read` вернет запрошенное количество байтов или столько, сколько доступно в очереди. Если очередь пуста, функция `read` сразу же вернет 0.

Важно понимать, что `MIN` определяет лишь минимальный объем данных. Если программа запрашивает большее количество байтов, она вполне может получить объем вплоть до запрошенного количества. То же относится и к случаям В и Г, когда значение `MIN` равно нулю.

Таблица 18.7 обобщает все четыре случая неканонического ввода. В этой таблице число `nbytes` соответствует третьему аргументу функции `read` (максимальное количество байтов, которое она может вернуть).

Таблица 18.7. Четыре случая неканонического ввода

	<code>MIN > 0</code>	<code>MIN == 0</code>
<code>TIME > 0</code>	<p>A: <code>read</code> возвращает <code>[MIN, nbytes]</code> до того, как истечет время таймера <code>read</code> возвращает <code>[1, MIN]</code> по истечении времени таймера. (TIME = время таймера, который запускается после приема первого байта. Вызывающий процесс может оказаться заблокированным на неопределенное время.)</p>	<p>B: <code>read</code> возвращает <code>[1, nbytes]</code> до того, как истечет время таймера <code>read</code> возвращает 0 по истечении времени таймера. (TIME = время таймера чтения, который запускается в момент вызова функции <code>read</code>)</p>
<code>TIME == 0</code>	<p>C: <code>read</code> возвращает <code>[MIN, nbytes]</code>, если в очереди имеются данные. (Вызывающий процесс может оказаться заблокированным на неопределенное время.)</p>	<p>D: <code>read</code> возвращает <code>[0, nbytes]</code> немедленно</p>

Помните, что стандарт POSIX.1 допускает совпадение индексов *VMIN* и *VTIME* с индексами *VEOF* и *VEOL* соответственно. В Solaris за счет этого обеспечивается обратная совместимость с устаревшими версиями System V. Однако это порождает проблему переносимости. При переходе из неканонического в канонический режим мы вынуждены восстанавливать значения элементов с индексами *VEOF* и *VEOL*. Так, если *VMIN* совпадает с *VEOF* и при переходе из неканонического режима в канонический мы не восстановим значение этого элемента, который в случае *VMIN* обычно равен 1, признаком конца файла станет символ *Control-A*. Самый простой способ решения этой проблемы — сохранять все содержимое структуры *termios* при переходе в неканонический режим и восстанавливать ее при возврате к каноническому режиму.

Пример

Программа в листинге 18.10 определяет функции *tty_cbreak* и *tty_raw*, которые служат для перевода терминала в режимы посимвольного (*cbreak*) и прозрачного (*raw*) ввода. (Термины *cbreak* и *raw* пришли из драйвера терминала Version 7.) Вернуть терминал в первоначальное состояние (предшествовавшее вызову любой из этих двух функций) можно с помощью функции *tty_reset*.

После вызова функции *tty_cbreak* нужно обратиться к функции *tty_reset*, прежде чем вызывать функцию *tty_raw*. То же относится к вызову функции *tty_cbreak* после вызова *tty_raw*. Это повышает вероятность, что терминал останется в состоянии, пригодном к работе, если мы столкнемся с непредвиденными ошибками.

Дополнительно в листинге 18.10 определены две вспомогательные функции: *tty_atexit*, которая может использоваться как обработчик выхода, обеспечивая возврат терминала в первоначальное состояние при вызове функции *exit*, и *tty_termios*, возвращающая указатель на оригинальную структуру *termios*, соответствующую каноническому режиму терминала.

Листинг 18.10. Установка режимов прозрачного и посимвольного ввода

```
#include "apue.h"
#include <termios.h>
#include <errno.h>

static struct termios      save_termios;
static int                  ttysavefd = -1;
static enum { RESET, RAW, CBREAK } ttystate = RESET;

int
tty_cbreak(int fd) /* переводит терминал в режим посимвольного ввода */
{
    int          err;
    struct termios buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* копия структуры */
```

```
/*
 * Отключить эхо-вывод и выйти из канонического режима.
 */
buf.c_lflag &= ~(ECHO | ICANON);

/*
 * Случай Б: минимум 1 байт, время ожидания не ограничено.
 */
buf.c_cc[VMIN] = 1;
buf.c_cc[VTIME] = 0;
if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
    return(-1);

/*
 * Убедиться, что были произведены все изменения. Функция tcsetattr может
 * вернуть 0, даже если выполнена лишь часть изменений.
 */
if (tcgetattr(fd, &buf) < 0) {
    err = errno;
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = err;
    return(-1);
}
if (((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
    buf.c_cc[VTIME] != 0) {

    /*
     * Были произведены лишь некоторые изменения.
     * Восстановить начальные настройки.
     */
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = EINVAL;
    return(-1);
}

ttystate = CBREAK;
ttysavefd = fd;
return(0);
}

int
tty_raw(int fd) /* переводит терминал в режим прозрачного ввода (raw) */
{
    int          err;
    struct termios buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* копия структуры */

    /*
     * Отключить эхо-вывод, выйти из канонического режима, отключить расширенную
     * обработку ввода, отключить обработку символов, генерирующих сигналы.
     */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
```

```

/*
 * Не выдавать сигнал SIGINT по псевдосимволу BREAK, отключить
 * преобразование CR->NL, отключить проверку паритета ввода,
 * не сбрасывать 8-й бит, отключить управление выводом.
 */
buf.c_iflag &= .(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

/*
 * Сбросить маску управления размером, отключить контроль четности.
 */
buf.c_cflag &= .(CSIZE | PARENB);

/*
 * Установить размер символа 8 бит/символ.
 */
buf.c_cflag |= CS8;

/*
 * Отключить обработку вывода.
 */
buf.c_oflag &= .(OPOST);

/*
 * Случай Б: минимум 1 байт, время ожидания не ограничено.
 */
buf.c_cc[VMIN] = 1;
buf.c_cc[VTIME] = 0;
if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
    return(-1);

/*
 * Убедиться, что были произведены все изменения. Функция tcsetattr может
 * вернуть 0, даже если выполнена лишь часть изменений.
 */
if (tcgetattr(fd, &buf) < 0) {
    err = errno;
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = err;
    return(-1);
}

if (((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
    (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
    (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
    (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
    buf.c_cc[VTIME] != 0) {
    /*
     * Были произведены лишь некоторые изменения.
     * Восстановить начальные настройки.
     */
    tcsetattr(fd, TCSAFLUSH, &save_termios);
    errno = EINVAL;
    return(-1);
}

ttystate = RAW;
ttysavefd = fd;
return(0);
}

```

```
int
tty_reset(int fd) /* восстанавливает состояние терминала */
{
    if (ttystate == RESET)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;

    return(0);
}

void
tty_atexit(void) /* может устанавливаться вызовом atexit(tty_atexit) */
{
    if (ttypsavefd >= 0)
        tty_reset(ttypsavefd);
}

struct termios *
tty_termios(void) /* позволяет вызывающему процессу */
/* узнать начальное состояние терминала */
{
    return(&save_termios);
}
```

Мы определили режим посимвольного (cbreak) ввода следующим образом:

- Неканонический режим. Как уже упоминалось в начале главы, в этом режиме отключена обработка некоторых служебных символов при вводе. Генерация сигналов не запрещена, поэтому пользователь всегда сможет послать сигнал вводом соответствующих символов. Необходимо понимать, что вызывающий процесс должен предусмотреть их обработку, иначе есть вероятность, что сигнал приведет к завершению процесса и терминал останется в режиме посимвольного ввода.

Как правило, при написании программ, изменяющих состояние терминала, нужно предусматривать обработку большинства сигналов. Это позволяет восстановить состояние терминала перед завершением приложения.

- Эхо-вывод отключен.
- За один раз читается как минимум один байт. Для этого мы записываем в элемент **MIN** значение 1, а в элемент **TIME** — значение 0. Это случай Б из табл. 18.7. Функция **read** не вернет управление, пока не будет доступен для чтения хотя бы один байт.

Мы определили режим прозрачного (raw) ввода следующим образом:

- Неканонический режим. Отключаются: обработка символов, генерирующих сигналы (**ISIG**), и расширенная обработка символов при вводе (**IEXTEN**). Дополнительно запрещается генерация сигнала **SIGINT** при получении псевдо-символа **BREAK** выключением флага **BRKINT**.
- Эхо-вывод выключен.
- Запрещены преобразование CR->NL при вводе (**ICRNL**), проверка четности (**INPCK**), сброс 8-го бита (**ISTRIP**) при вводе и управление выходным потоком (**IXON**).

○ Размер символа 8 бит (CS8), запрещен контроль четности (PARENB).

○ Запрещена обработка вывода (OPOST).

○ За один раз читается как минимум один байт (MIN = 1, TIME = 0).

Программа в листинге 18.11 тестирует режимы прозрачного и посимвольного ввода.

Листинг 18.11. Тест режимов raw и cbreak

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("перехвачен сигнал\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int      i;
    char     c;

    if (signal(SIGINT, sig_catch) == SIG_ERR) /* включить обработку сигналов */
        err_sys("ошибка вызова функции signal(SIGINT)");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGQUIT)");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("ошибка вызова функции signal(SIGTERM)");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_raw");
    printf("Переход в режим raw, выход из режима по нажатии DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177) /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_reset");
    if (i <= 0)
        err_sys("ошибка чтения");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_cbreak");
    printf("\nПереход в режим cbreak, выход из режима по сигналу SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("ошибка вызова функции tty_reset");
    if (i <= 0)
        err_sys("ошибка чтения");

    exit(0);
}
```

Запустив программу из листинга 18.11, мы сможем наблюдать за поведением терминала в этих двух режимах:

```
$ ./a.out
Переход в режим raw, выход из режима по нажатии DELETE
4
33
133
61
70
176
нажата клавиша DELETE
Переход в режим cbreak, выход из режима по сигналу SIGINT
1
10
перехвачен сигнал
нажата клавиша Control-A
нажата клавиша Backspace
нажата клавиша прерывания
```

В режиме прозрачного ввода (raw) были нажаты клавиши Control-D (04) и функциональная клавиша F7. На данном терминале эта функциональная клавиша генерирует пять символов: ESC (033), [(0133), 1 (061), 8 (070) и ~ (0176). Обратите внимание: когда отключена обработка вывода (~OPOST), возврат каретки после ввода каждого символа не производится. Отметьте также, что в режиме посимвольного ввода (cbreak) запрещена обработка некоторых служебных символов (таких, как символ конца файла (Control-D) и символ забоя (Backspace)), тогда как символы, генерирующие сигналы, по-прежнему обрабатываются.

18.12. Размер окна терминала

Большинство версий UNIX позволяют определить размеры окна терминала и сообщить процессам из группы процессов переднего плана об их изменении. Каждому терминалу и псевдотерминалу ядро ставит в соответствие структуру `winsize`:

```
struct winsize {
    unsigned short ws_row;      /* количество строк */
    unsigned short ws_col;      /* количество символов в строке */
    unsigned short ws_xpixel;   /* горизонтальный размер в пикселях */
                                /* (не используется) */
    unsigned short ws_ypixel;   /* вертикальный размер в пикселях (не используется) */
};
```

Правила работы со структурой:

- Текущее содержимое структуры можно получить с помощью команды `TIOCGWINSZ` функции `ioctl` (раздел 3.15).
- Записать новое содержимое структуры в ядро можно с помощью команды `TIOCSWINSZ` функции `ioctl`. Если новые размеры окна отличаются от текущих, группе процессов переднего плана будет послан сигнал `SIGWINCH`. (Обратите внимание, что, согласно табл. 10.1, по умолчанию этот сигнал игнорируется.)
- Кроме хранения текущих значений и посылки сигнала при ее изменении, ядро больше ничего не делает с этой структурой. Интерпретация структуры полностью возлагается на прикладные программы.

Основное назначение этой функциональной возможности — извещать приложения (такие, как редактор `vi`) об изменении размеров окна терминала. Когда сигнал будет доставлен, приложение сможет узнать новые размеры окна и перерисовать экран.

Пример

В листинге 18.12 приводится исходный текст программы, которая выводит текущие размеры окна и приостанавливается. Каждый раз, когда изменяется размер окна, программа перехватывает сигнал `SIGWINCH` и выводит новые значения размеров. Чтобы завершить работу программы, необходимо сгенерировать сигнал.

Листинг 18.12. Вывод информации о размерах окна

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

static void
pr_winsize(int fd)
{
    struct winsize    size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("ошибка выполнения команды TIOCGWINSZ");
    printf("%d строк, %d символов в строке\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("доставлен сигнал SIGWINCH\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("ошибка вызова функции signal");
    pr_winsize(STDIN_FILENO); /* вывести начальные размеры окна */
    for ( ; ; ) /* и приостановиться */
        pause();
}
```

Запустив эту программу на терминале с изменяемым размером окна, мы получили следующие результаты:

```
$ ./a.out
35 строк, 80 символов в строке      начальный размер окна
доставлен сигнал SIGWINCH          изменен размер окна: перехвачен сигнал
40 строк, 123 символа в строке
```

доставлен сигнал SIGWINCH
42 строки, 33 символа в строке
^C \$

и еще раз

нажата клавиша прерывания, чтобы завершить
программу

18.13. termcap, terminfo и curses

Схема хранения информации о терминалах под названием `termcap` (расшифровывается как «terminal capability» — «характеристики терминала») была разработана в Беркли для поддержки текстового редактора `vi`. Она включает текстовый файл `/etc/termcap` и набор процедур для работы с ним. Файл `termcap` содержит характеристики различных терминалов: какие возможности поддерживаются терминалом (количество строк и символов в строке, поддержка символа забоя и т. п.) и как заставить терминал выполнять определенные операции (очистку экрана, перемещение курсора в заданную позицию и пр.). Убрав эту информацию из кода программы и поместив ее в обычный текстовый файл, который легко можно отредактировать, разработчики сделали возможным использование редактора `vi` на самых разных терминалах.

Процедуры поддержки `termcap` также были извлечены из редактора `vi` и размещены в отдельной библиотеке под названием `curses`. В эту библиотеку было добавлено много новых функций, что сделало ее пригодной для работы в составе любой программы, которая должна управлять выводом информации на экран.

Но у схемы `termcap` были недостатки. Все больше описаний терминалов добавлялось в файл `termcap` и все больше времени требовалось программам, чтобы отыскать в нем описание какого-либо терминала. Кроме того, для обозначения различных характеристик терминалов использовались двухсимвольные имена. Эти недостатки привели к появлению новой схемы `terminfo` и связанной с ней библиотеки `curses`. Описания терминалов в схеме `terminfo` хранятся в скомпилированном виде, что значительно ускоряет поиск нужной информации. Впервые схема `terminfo` появилась в SVR2 и с тех пор используется во всех версиях System V.

Системы, основанные на System V, традиционно используют схему terminfo, а BSD-системы — termcap, но современные системы обычно поддерживают обе схемы. Однако Mac OS X поддерживает только terminfo.

Описание `terminfo` и библиотеки `curses` можно найти в [Goodheart, 1991], но весь тираж этой книги уже распродан. В книге [Strang, 1986] описывается версия библиотеки `curses` из Беркли. В книге [Strang, Mui, and O'Reilly, 1988] содержится описание `termcap` и `terminfo`.

Библиотеку ncurses — свободно распространяемую версию, совместимую с интерфейсом curses SVR4, — вы найдете по адресу <http://invisibleisland.net/ncurses/ncurses.html>. Ее также можно найти по адресу <http://www.gnu.org/software/ncurses>.

Ни `termcap`, ни `terminfo` сами по себе не имеют отношения к задачам, которые мы рассматривали в этой главе (изменение режима терминала, изменение значений управляющих символов, обслуживание размеров окна и т. п.). На самом деле

они предоставляют средства выполнения типичных операций (очистка экрана, перемещение курсора) для различных терминалов. С другой стороны, библиотека *curses* действительно помогает при решении некоторых задач, обсуждавшихся в этой главе. Она предоставляет функции для перевода терминала в режим посимвольного и прозрачного ввода, включения и отключения эхо-вывода и т. п. Но изначально библиотека *curses* была разработана для простых алфавитно-цифровых терминалов, которые сегодня в большинстве своем заменены графическими.

18.14. Подведение итогов

Терминалы обладают множеством свойств и возможностей, большинство из которых можно контролировать и подстраивать под свои нужды. В этой главе мы описали большое количество функций, которые изменяют характеристики терминалов — флаги режимов и значения служебных символов. Мы подробно рассмотрели все специальные символы и флаги, которые можно сбросить или установить.

Терминалы могут работать в двух режимах ввода — каноническом (построчный ввод) и неканоническом. Мы продемонстрировали примеры обоих режимов и показали функции для переключения терминала в устаревшие режимы прозрачного (*raw*) и посимвольного (*cbreak*) ввода. Также мы рассказали, как получить и изменить размеры окна терминала.

Упражнения

- 18.1** Напишите программу, которая вызывала бы функцию *tty_raw* и завершала работу (без восстановления канонического режима терминала). Если ваша система поддерживает команду *reset(1)* (она доступна на всех четырех платформах, обсуждаемых в этой книге), попробуйте с ее помощью восстановить режим ввода терминала.
- 18.2** Схему контроля четности — ODD или EVEN — можно задать с помощью флага *PARODD* в поле *c_cflag*. Программа *tip* в BSD, кроме того, позволяет задать значение бита четности 0 или 1. Как она это делает?
- 18.3** Если в вашей системе команда *stty(1)* поддерживает элементы *MIN* и *TIME*, выполните следующее упражнение. Войдите в систему с двух терминалов и запустите редактор *vi* на одном из них. С помощью команды *stty* с другого терминала определите, какие значения *MIN* и *TIME* устанавливает редактор *vi* (так как этот редактор переводит терминал в неканонический режим). (Если ваш терминал работает под управлением оконной системы, то же самое можно сделать, открыв два терминала в отдельных окнах.)

19

Псевдотерминалы

19.1. Введение

В главе 9 мы видели, что вход в систему осуществляется через терминальное устройство, которое автоматически реализует семантику терминала. Управление взаимодействием запускаемых программ с терминалом осуществляется модулем дисциплины обслуживания линии связи (рис. 18.2),¹ что позволяет назначить специальные символы терминала (символы забоя, стирания строки, прерывания и пр.) и изменить другие его характеристики. Однако при входе в систему через сетевое соединение модуль дисциплины обслуживания линии связи между сетевым соединением и оболочкой входа не предоставляется автоматически. На рис. 9.5 показано, что семантика терминала в этом случае реализуется драйвером псевдотерминала.

Помимо входа в систему через сетевое соединение псевдотерминалы используются и в других случаях, которые мы будем рассматривать в этой главе. Обсуждение псевдотерминалов мы начнем с краткого обзора их применения, который завершится описанием некоторых особых случаев. После этого мы рассмотрим функции создания псевдотерминалов, предоставляемые различными платформами, и воспользуемся ими при написании программы, которую мы назвали `pty`. Мы покажем различные способы использования этой программы: создание журнала ввода/вывода терминала (программа `script(1)`) и запуск сопроцессов, не подверженных проблемам с буферизацией, с которыми мы столкнулись в программе из листинга 15.10.

19.2. Обзор

Термином *псевдотерминал* обозначается программное устройство, похожее на терминал, но не являющееся им. На рис. 19.1 показана типичная схема использования псевдотерминала процессами. Вот ключевые моменты, на которые следует обратить особое внимание:

- Обычно процесс открывает ведущий (master) псевдотерминал, затем вызывается функция `fork`. Дочерний процесс создает новый сеанс, открывает соответ-

¹ Было бы правильнее назвать протоколом передачи данных, но так как данный термин уже устоялся, мы будем использовать его. — Примеч. пер.

ствующий ведомый (slave) псевдотерминал, создает дубликаты дескрипторов стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках и вызывает функцию `exec`. Ведомый псевдотерминал становится управляющим терминалом дочернего процесса.

- Пользовательский процесс, расположенный на рис. 19.1 над ведомым терминалом, считает, что его стандартный ввод, стандартный вывод и стандартный вывод ошибок связаны с терминальным устройством. Процесс может использовать любые функции из главы 18, предназначенные для работы с терминалом. Но поскольку ведомый терминал не является настоящим терминальным устройством, функции, которые не будут иметь смысла (изменение скорости передачи, отправка псевдосимвола `BREAK`, проверка бита четности и подобные), просто игнорируются.
- Все, что будет записано в ведущий псевдотерминал, появится на входе ведомого псевдотерминала, и наоборот. То есть вывод процесса, владеющего ведущим псевдотерминалом, передается на вход процесса, владеющего ведомым псевдотерминалом. Это очень напоминает двунаправленный канал, но благодаря наличию промежуточного модуля, реализующего дисциплину обслуживания линии связи, мы получаем дополнительные преимущества перед обычными каналами.

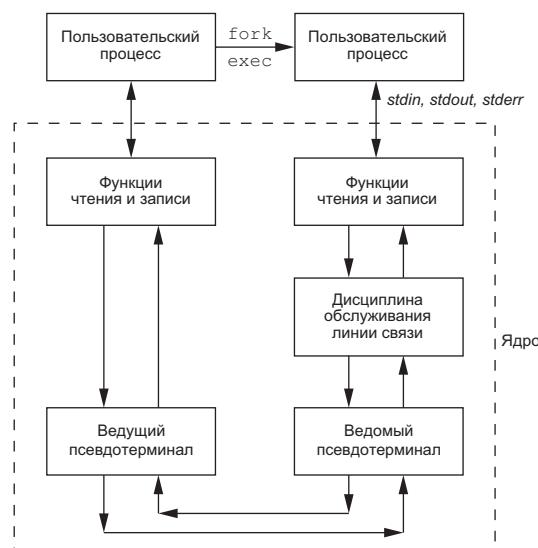


Рис. 19.1. Типичная схема взаимоотношения процессов, использующих псевдотерминал

На рис. 19.1 показано, как реализованы псевдотерминалы в FreeBSD, Mac OS X и Linux. В разделе 19.3 обсуждается, как открывать эти устройства.

В Solaris псевдотерминалы построены на базе подсистемы STREAMS. Их устройство показано на рис. 19.2. Два модуля, изображенные в виде пунктирных прямоугольников, являются необязательными. Модули `pkct` и `rtem` обеспечивают семантику псевдотерминала. Другие два модуля (`ldterm` и `ttcompat`) реализуют

дисциплину обслуживания потока данных. В разделе 19.3 мы покажем, как построить такую схему расположения модулей STREAMS.

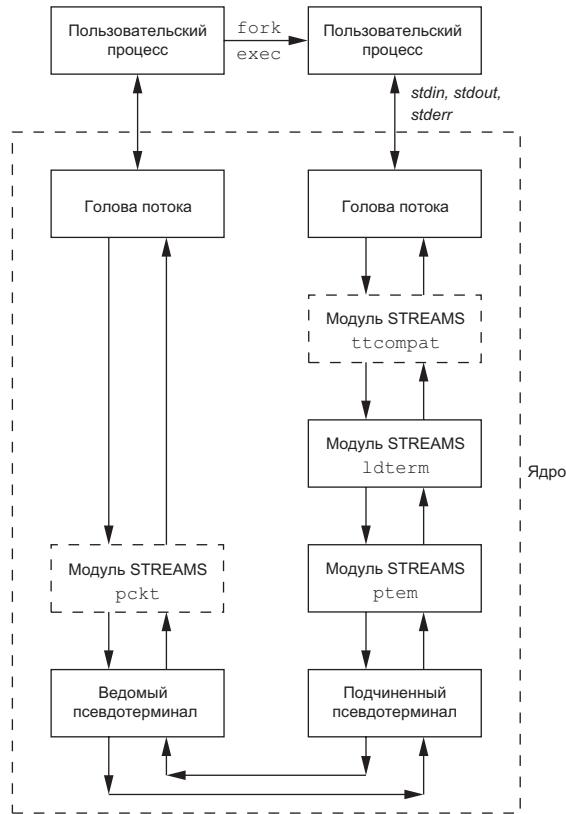


Рис. 19.2. Схема реализации псевдотерминалов в Solaris

Чтобы упростить последующие рисунки, мы не будем показывать на них «функции чтения и записи» (рис. 19.1) и «голову потока» (рис. 19.2). Кроме того, псевдотерминал мы будем обозначать аббревиатурой PTY, а все модули STREAMS, расположенные выше ведомого PTY на рис. 19.2, будем объединять в один блок с названием «дисциплина обслуживания терминала» как на рис. 19.1.

А теперь рассмотрим типичные области применения псевдотерминалов.

Серверы сетевого входа в систему

Псевдотерминалы встроены в серверы, обеспечивающие возможность сетевого входа в систему. Примерами таких серверов являются `telnetd` и `rlogind`.

Детальное описание службы `rlogin` вы найдете в главе 15 [Stevens, 1990]. После запуска оболочки входа на удаленной машине мы получим схему, которая изображена на рис. 19.3. Аналогичные результаты будут получены при использовании сервера `telnetd`.

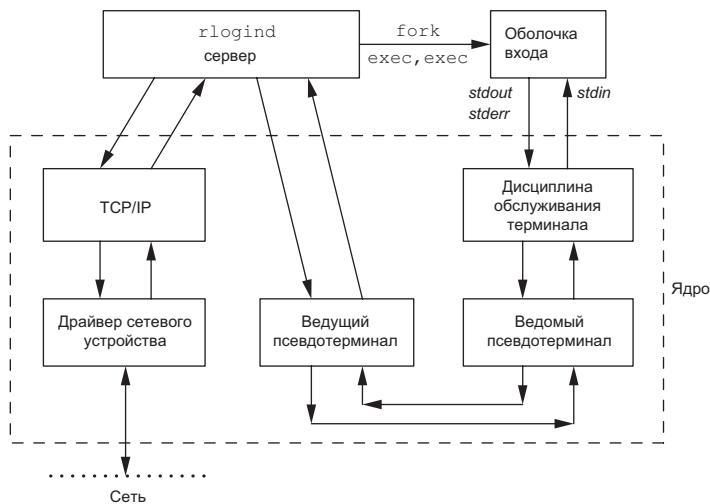


Рис. 19.3. Типичная схема взаимоотношения процессов при использовании службы rlogind

Между оболочкой входа и сервером rlogind показаны два вызова функции `exec`, потому что для идентификации пользователя обычно вызывается программа `login`. Ключевым моментом в этой схеме является то, что программа, управляющая ведущим терминалом, параллельно производит чтение и запись в другой поток ввода/вывода. В данном примере этот поток ввода/вывода показан как блок TCP/IP. Это означает, что процесс должен использовать ту или иную форму мультиплексирования ввода/вывода (раздел 14.4), например `select` или `poll`, или разделиться на два процесса или потока.

Эмулятор терминала оконной системы

Оконные системы обычно предоставляют эмуляторы терминалов, давая возможность запускать программы в знакомой среде командной строки. Эмулятор терминала действует как промежуточное звено между командной оболочкой и диспетчером окон. Каждый экземпляр командной оболочки выполняется в отдельном окне. Данная схема (с двумя экземплярами командной оболочки, выполняющимися в разных окнах) изображена на рис. 19.4. Стандартный ввод, стандартный вывод и стандартный вывод сообщений об ошибках командной оболочки подключаются к ведомому PTY. Ведущий PTY открывается программой-эмодулятором терминала. Помимо выполнения функций интерфейса к оконной системе, эмулятор терминала отвечает также за эмуляцию поведения терминала определенного типа, то есть он должен обрабатывать управляющие последовательности, как это делает устройство, которое он эмулирует. Эти последовательности перечислены в базах данных `termcap` и `terminfo`.

Когда пользователь изменяет размеры окна эмулятора терминала, диспетчер окон информирует эмулятор об этом. В ответ эмулятор терминала вызывает команду `TIOCSWINSZ` функции `ioctl` на стороне ведущего PTY, чтобы установить размер окна

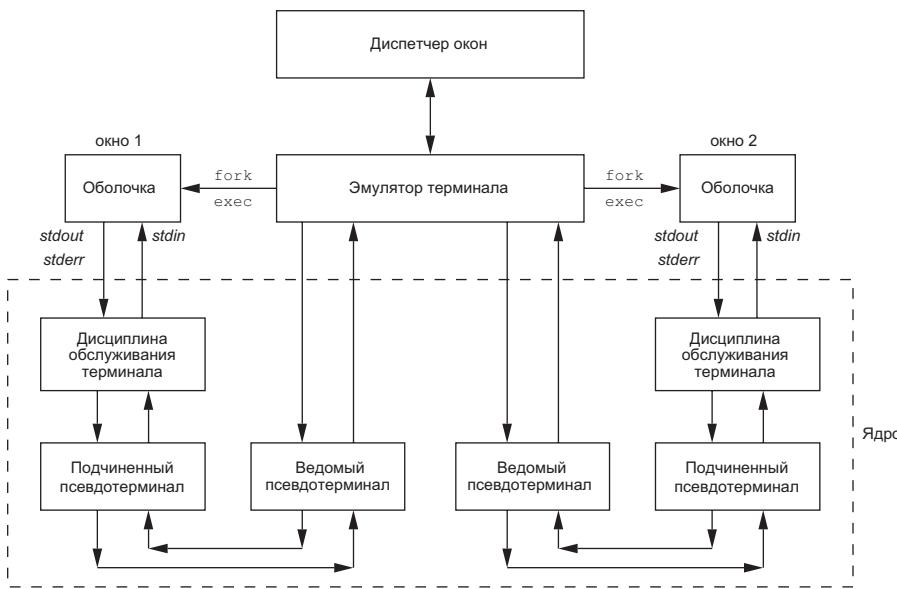


Рис. 19.4. Схема процессов в оконной системе

для ведомой стороны. Если новый размер отличается от текущего, ядро посылает сигнал `SIGWINCH` группе процессов переднего плана ведомого PTY. Если приложение должно перерисовывать изображение на экране при изменении размеров окна, оно может перехватить сигнал `SIGWINCH`, выполнить команду `TIOCGWINSZ` функции `ioctl`, чтобы получить новые размеры окна, и перерисовать изображение.

Программа `script`

В большинстве версий UNIX имеется программа `script(1)`, которая копирует входные и выходные данные терминала в файл. Для этого программа размещает себя между терминалом и вызовом новой командной оболочки. На рис. 19.5 подробно показаны все взаимодействия между процессами при запуске программы `script`. В частности, рисунок показывает, что программа `script` обычно запускается из оболочки входа, которая затем просто ожидает ее завершения.

Во время работы программы `script` весь вывод с терминала, который идет от модуля дисциплины обслуживания терминала, расположенного выше ведомого PTY, копируется в файл журнала (который обычно называется `typescript`). Поскольку весь ввод с клавиатуры обычно выводится модулем дисциплины обслуживания, в файл журнала попадает все, что введено с клавиатуры. Пароли, вводимые с клавиатуры, не могут попасть в файл журнала, поскольку во время ввода пароля эхо-вывод отключен.

При работе над первым изданием этой книги Ричард Стивенс использовал программу `script` для захвата вывода программ-примеров, чтобы избежать опечаток, которые на-верняка возникли бы при ручном наборе. Недостаток такого использования программы `script` заключался в том, что приходилось разбираться с представлением управляющих символов в файле журнала.

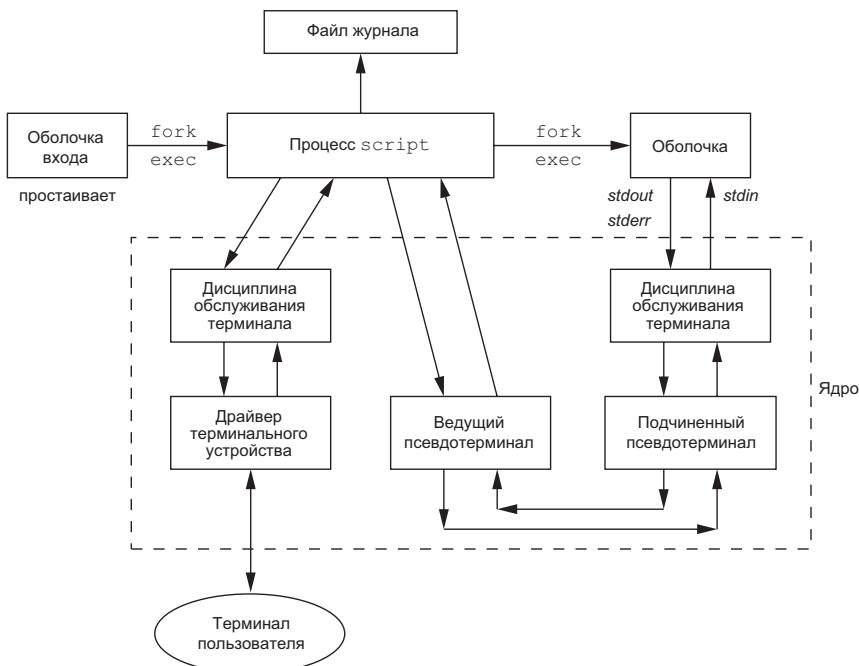


Рис. 19.5. Программа script

После разработки универсальной программы `pty` в разделе 19.5 мы увидим, что из нее легко можно сделать версию программы `script` с помощью простенького сценария на языке командной оболочки.

Программа `expect`

Псевдотерминалы можно использовать для управления интерактивными программами в неинтерактивном режиме. Множество программ требуют взаимодействия с терминалом. Одна из них — программа `passwd(1)`, которая ожидает ввода пароля в ответ на приглашение.

Вместо того чтобы добавлять во все интерактивные программы поддержку работы в пакетном режиме, удобнее управлять ими из сценариев. Такую возможность предоставляет программа `expect` (см. [Libes, 1990, 1991, 1994]). Она использует псевдотерминалы, подобные программе `pty` из раздела 19.5, для запуска других программ. Кроме того, программа `expect` поддерживает свой язык программирования для проверки вывода запущенных программ, что позволяет на основе анализа выводимых данных принимать решение о том, какие данные программа ожидает получить. Когда интерактивная программа запускается из сценария, мы не можем просто скопировать данные из сценария в программу и обратно. Вместо этого мы должны отправить программе некоторые данные, проанализировать полученный вывод и принять решение о том, что ввести в следующий раз.

Запуск сопроцессов

В примере сопроцесса из листинга 15.10 мы не могли использовать для взаимодействия с сопроцессом функции стандартной библиотеки ввода/вывода, потому что при работе с неименованными каналами стандартная библиотека ввода/вывода устанавливает режим полной буферизации для стандартных потоков ввода и вывода, что приводит к тупиковой ситуации. Если сопроцесс представляет собой скомпилированную программу, исходный код которой недоступен, мы не сможем добавить дополнительные вызовы функции `fflush`, чтобы решить эту проблему. Рисунок 15.8 иллюстрирует управление сопроцессом. Все, что нам необходимо, — это поместить псевдотерминал между двумя процессами, как показано на рис. 19.6, чтобы «обмануть» сопроцесс и заставить его думать, что он взаимодействует с терминалом, а не с другим процессом.



Рис. 19.6. Управление сопроцессом с помощью псевдотерминала

Теперь стандартный ввод и стандартный вывод сопроцесса с его точки зрения выглядят так, будто они связаны с терминальным устройством, поэтому стандартная библиотека ввода/вывода установит для них построчный режим буферизации. Родительский процесс может вставить псевдотерминал между собой и сопроцессом двумя способами. (В этом случае родительским процессом может быть программа из листинга 15.9, которая использует для взаимодействия с сопроцессом два неименованных канала.) Один из них — запустить дочерний процесс функцией `pty_fork` (раздел 19.4) вместо `fork`. Другой способ — запустить с помощью функции `exec` программу `pty` (раздел 19.5) и передать ей имя программы сопроцесса в качестве аргумента. Мы продемонстрируем оба способа после того, как рассмотрим программу `pty`.

Отслеживание вывода программ, работающих продолжительное время

Если программа должна работать продолжительное время, мы можем просто запустить ее в фоновом режиме средствами любой стандартной командной оболочки. Но если стандартный вывод программы перенаправлен в файл, а объем генерируемых программой данных невелик, отслеживать ход выполнения программы будет очень непросто, потому что стандартная библиотека ввода/вывода назначит режим полной буферизации для потока стандартного вывода, причем размер буфера может превышать 8192 байт.

Если доступен исходный код программы, можно вставить дополнительные вызовы функции `fflush`. В качестве альтернативы можно запустить программу под управлением программы `pty`, которая заставит стандартную библиотеку ввода/вывода думать, что стандартный вывод связан с терминалом. На рис. 19.7 приво-

дится схема взаимодействия процессов, соответствующая такому случаю. Здесь `slowout` — это имя программы, которая выводит данные редко и в небольшом объеме. Стрелка, соответствующая запуску программы `pty`, нарисована пунктиром, чтобы подчеркнуть, что она запускается как фоновое задание.

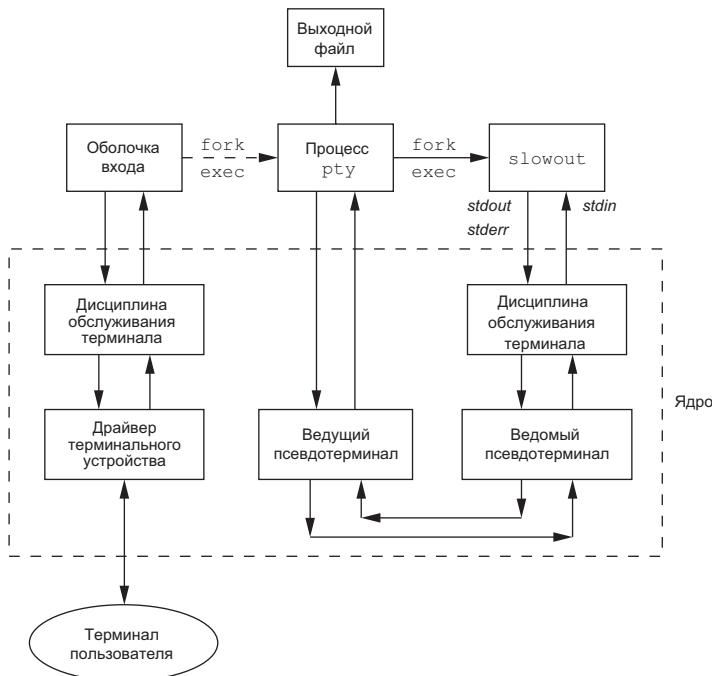


Рис. 19.7. Запуск программы, которая выводит данные редко и в небольшом объеме с помощью псевдотерминала

19.3. Открытие устройств псевдотерминалов

Псевдотерминалы работают подобно физическим терминальным устройствам, поэтому тип используемого устройства для приложений не имеет значения. Однако при открытии файлов устройств PTY приложения не должны устанавливать флаг `O_TTY_INIT`. Стандарт Single UNIX Specification уже требует, чтобы реализации инициализировали ведомую сторону PTY при первом открытии и устанавливали все нестандартные флаги `termios`, необходимые для нормальной работы устройства. Это требование было выработано с целью обеспечить корректную работу устройства PTY с POSIX-совместимыми приложениями, вызывающими функции `tcgetattr` и `tcsetattr`.

На разных платформах открытие устройств псевдотерминалов осуществляется разными способами. Чтобы упорядочить их, стандарт Single UNIX Specification определяет ряд функций в качестве расширений XSI. Эти расширения основаны

на функциях, изначально предназначенных для управления псевдотерминалами STREAMS в System V Release 4. Функция `posix_openpt` предоставляет переносимый способ открытия устройства ведущего псевдотерминала.

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

Возвращает дескриптор следующего доступного ведущего PTY
в случае успеха, -1 — в случае ошибки

Аргумент *oflag* похож на аналогичный аргумент функции `open(2)` и представляет битовую маску режима открытия устройства. Однако эта функция поддерживает не все флаги режимов. С функцией `posix_openpt` можно использовать флаг `O_RDWR`, чтобы открыть устройство для чтения и записи, и `O_NOCTTY`, чтобы открываемое устройство не стало управляющим терминалом для вызывающего процесса. Поведение остальных флагов не определено.

Прежде чем использовать ведомый псевдотерминал, необходимо установить права доступа к нему так, чтобы он стал доступен для приложений. Сделать это можно с помощью функции `grantpt`. Идентификатор пользователя ведомого устройства она устанавливает равным реальному идентификатору пользователя вызывающего процесса, а идентификатор группы устройства — в неопределенное состояние; обычно это идентификатор некоторой группы, имеющей право доступа к терминальным устройствам. Биты прав доступа устанавливаются так, чтобы разрешить доступ на чтение и на запись владельцу и право на запись группы владельца (0620).

Реализации часто назначают в качестве группы владельца ведомого устройства PTY группу `tty`. Для программ, которым требуется право на запись во все активные терминалы в системе, устанавливается бит `set-group-ID` для группы `tty`. Примерами таких программ являются `wall(1)` и `write(1)`. Так как ведомые устройства PTY дают право на запись для группы, эти программы оказываются в состоянии писать в них.

```
#include <stdlib.h>

int grantpt(int fd);
int unlockpt(int fd);
```

Обе возвращают 0 в случае успеха, -1 — в случае ошибки

Чтобы изменить права доступа к ведомому устройству, функции `grantpt` может потребоваться запустить программу с установленным битом `set-user-ID` (например, `/usr/lib/pt_chmod` в Solaris). То есть поведение функции `grantpt` окажется непредсказуемым, если процесс предусматривает перехват сигнала `SIGCHLD`.

Функция `unlockpt` разблокирует доступ к ведомому псевдотерминалу, что позволит другим приложениям открывать устройство. Пока устройство заблокировано, приложению-владельцу предоставляется удобная возможность инициализировать и должным образом настроить ведущее и ведомые устройства прежде, чем они будут использованы.

Обратите внимание, что обе функции принимают аргумент `fd` с файловым дескриптором, связанным с ведущим псевдотерминалом.

Функция `ptsname` возвращает полное имя ведомого устройства псевдотерминала по заданному дескриптору ведущего псевдотерминала. Это позволяет приложениям идентифицировать ведомое устройство независимо от соглашений, принятых на той или иной платформе. Заметьте, что возвращаемая строка может храниться статически, так что она может быть затерта в результате следующего вызова `ptsname`.

```
#include <stdlib.h>
char *ptsname(int fd);
```

Возвращает указатель на строку с именем ведомого PTY в случае успеха, NULL — в случае ошибки

В табл. 19.1 перечислены функции, определяемые стандартом Single UNIX Specification для работы с псевдотерминалами, и указано, какие из них поддерживаются четырьмя платформами, обсуждаемыми в этой книге.

В ОС FreeBSD функции `unlockpt` и `unlockpt` не выполняют никаких действий, кроме проверки аргументов; устройства PTY создаются динамически, с корректно установленными правами доступа. Имейте в виду, что FreeBSD определяет флаг `O_NOCTTY` только для совместимости с программами, вызывающими функцию `posix_openpt`. Эта операционная система не назначает открываемое устройство управляющим терминалом, поэтому флаг `O_NOCTTY` просто игнорируется.

Таблица 19.1. Функции XSI для работы с псевдотерминалами

Функция	Описание	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>grantpt</code>	Изменяет права доступа к ведомому PTY	✓	✓	✓	✓	✓
<code>posix_openpt</code>	Открывает ведущее устройство PTY	✓	✓	✓	✓	✓
<code>ptsname</code>	Возвращает имя ведомого устройства PTY	✓	✓	✓	✓	✓
<code>unlockpt</code>	Разрешает открытие ведомого устройства PTY	✓	✓	✓	✓	✓

Несмотря на то что стандарт Single UNIX Specification способствовал существенному повышению переносимости приложений, различия все еще остаются. Мы

разработали две функции, предназначенные для открытия псевдотерминалов: `ptym_open` открывает следующее доступное ведущее устройство PTY, а `ptys_open` — соответствующее ведомое устройство.

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);
```

Возвращает дескриптор ведущего устройства PTY
в случае успеха, -1 — в случае ошибки

```
int ptys_open(char *pts_name);
```

Возвращает дескриптор ведомого устройства PTY
в случае успеха, -1 — в случае ошибки

Обычно эти функции не вызываются непосредственно; это делает функция `pty_fork` (раздел 19.4), которая одновременно запускает дочерний процесс.

Функция `ptym_open` определяет следующий доступный ведущий PTY и открывает его. Вызывающий процесс должен разместить в памяти буфер для имени ведущего или ведомого устройства; в случае успеха в этом буфере (аргумент `pts_name`) будет возвращено имя ведомого PTY. Это имя затем передается функции `ptys_open`, которая открывает ведомое устройство. Размер буфера в байтах передается в аргументе `pts_namesz`, чтобы функция `ptym_open` не создавала копию строки длиннее, чем размер буфера.

Причина, по которой мы предоставляем две разные функции для открытия двух типов устройств, станет понятна после того, как мы рассмотрим функцию `pty_fork`. Обычно процесс вызывает функцию `ptym_open`, чтобы открыть ведущее устройство и получить имя ведомого. Затем процесс вызывает функцию `fork` и дочерний процесс с помощью `ptys_open` открывает ведомое устройство, после чего открывает новый сеанс обращением к функции `setsid`. Благодаря этому ведомый псевдотерминал становится управляющим терминалом дочернего процесса.

Листинг 19.1. Функции открытия псевдотерминала

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#if defined(SOLARIS)
#include <stropts.h>
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char     *ptr;
    int      fdm, err;

    if ((fdm = posix_openpt(O_RDWR)) < 0)
        return(-1);
```

```

if (grantpt(fdm) < 0) /* разрешить доступ к ведомому */
    goto errout;
if (unlockpt(fdm) < 0) /* сбросить флаг блокировки ведомого */
    goto errout;
if ((ptr = ptsname(fdm)) == NULL) /* получить имя ведомого */
    goto errout;

/*
 * Вернуть имя ведомого устройства.
 * Завершить нулевым символом, чтобы обработать ситуацию, когда
 * strlen(ptr) > pts_namesz.
 */
strncpy(pts_name, ptr, pts_namesz);
pts_name[pts_namesz - 1] = '\0';
return(fdm); /* вернуть дескриптор ведущего */

errout:
    err = errno;
    close(fdm);
    errno = err;
    return(-1);
}

int
pty_open(char *pts_name)
{
    int      fds;
#ifndef SOLARIS
    int      err, setup;
#endif

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-1);
#ifndef SOLARIS
    /*
     * Проверить: возможно, поток уже настроен должным образом
     * благодаря автоматической вставке модулей.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0)
        goto errout;

    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0)
            goto errout;
        if (ioctl(fds, I_PUSH, "ldterm") < 0)
            goto errout;
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {

```

errout:

```

            err = errno;
            close(fds);
            errno = err;
            return(-1);
        }
    }
#endif
    return(fds);
}

```

Функция `ptym_open`, используя функции управления псевдотерминалом из расширений XSI, отыскивает и открывает неиспользуемый ведущий псевдотерминал, после чего инициализирует соответствующий ведомый псевдотерминал. Функция `ptys_open` открывает ведомый псевдотерминал. Однако в Solaris может потребоваться выполнить дополнительные операции, чтобы ведомый псевдотерминал действовал как терминал.

В Solaris после открытия ведомого псевдотерминала может также понадобиться добавить в поток ведомого устройства три модуля STREAMS. Модуль эмуляции псевдотерминала (`ptem`) и модуль дисциплины обслуживания терминала (`ldterm`) совместно работают как настоящий терминал. Модуль `ttcompat` обеспечивает совместимость с устаревшими версиями системного вызова `ioctl` в V7, 4BSD и Xenix. Этот модуль относится к разряду необязательных, но поскольку он автоматически добавляется при входе в систему через сетевое соединение, мы также помещаем его в поток ведомого устройства.

Эти три модуля могут размещаться в потоке автоматически. Система STREAMS поддерживает функциональную возможность, известную как *autopush* (автоматическое добавление). Она позволяет администратору создать список модулей, которые должны автоматически размещаться в потоке того или иного устройства при его открытии (более подробное описание вы найдете в [Rago, 1993]). С помощью команды `I_FIND` функции `ioctl` мы проверяем наличие модуля `ldterm` в потоке. Если модуль найден, следовательно, поток уже был сконфигурирован механизмом автоматического добавления модулей и нам не нужно вставлять его повторно.

Linux, Mac OS X и Solaris следуют исторически сложившемуся в System V правилу: если вызывающий процесс является лидером сеанса, который еще не имеет управляющего терминала, вызов функции `open` назначает открываемый ведомый PTY управляющим терминалом процесса. Если нужно избежать этого, следует передать функции `open` флаг `O_NOCTTY`. Однако в FreeBSD вызов `open` не приводит к назначению ведомого PTY управляющим терминалом процесса. Как назначать управляющий терминал при выполнении в FreeBSD, будет показано в следующем разделе.

19.4. Функция pty_fork

Теперь, используя функции открытия псевдотерминала из предыдущего раздела, `ptym_open` и `ptys_open`, напишем новую функцию `pty_fork`. Она будет совмещать открытие ведущего и ведомого устройств, запуск дочернего процесса и назначение его лидером сеанса со своим управляющим терминалом.

```
#include "apue.h"
#include <termios.h>

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса — в родительском процессе, -1 — в случае ошибки

Дескриптор ведущего PTY возвращается в переменной по адресу `ptrfdm`.

Если в аргументе `slave_name` передается непустой указатель, по заданному адресу сохраняется имя ведомого устройства. Разумеется, вызывающий процесс должен выделить память для хранения строки, на которую указывает этот аргумент.

Если в аргументе `slave_termios` передается непустой указатель, система использует структуру, на которую ссылается этот аргумент, для инициализации ведомого терминала. Если в этом аргументе передается `NULL`, система инициализирует структуру `termios` ведомого устройства значениями по умолчанию, зависящими от реализации. Аналогично инициализируется структура с размером окна ведомого устройства, если в аргументе `slave_winsize` передается непустой указатель. Если в этом аргументе передается `NULL`, структура `winsize`, как правило, инициализируется нулями.

В листинге 19.2 приводится исходный код этой функции. Она будет работать на всех четырех платформах, рассматриваемых в этой книге, обращаясь к функциям `ptym_open` и `ptys_open`.

После открытия ведущего PTY вызывается функция `fork`. Как уже говорилось, функция `ptys_open` должна вызываться только после того, как дочерний процесс откроет новый сеанс вызовом функции `setsid`. К моменту вызова `setsid` дочерний процесс не является лидером группы, поэтому выполняются три действия, описанные в разделе 9.5: (а) открывается новый сеанс, в котором дочерний процесс выступает в роли лидера, (б) создается новая группа процессов для дочернего процесса и (в) дочерний процесс теряет любую установленную связь с предыдущим управляющим терминалом. В Linux, Mac OS X и Solaris ведомое устройство становится управляющим терминалом этого нового сеанса при вызове `ptys_open`. В FreeBSD мы должны назначить управляющий терминал с помощью команды `TIOCSETTY` функции `ioctl`. (Другие три платформы также поддерживают команду `TIOCSETTY` функции `ioctl`, но вызвать ее требуется только в FreeBSD.)

После этого в дочернем процессе инициализируются структуры `termios` и `winsize`. В заключение дочерний процесс дублирует дескриптор ведомого PTY на стандартный ввод, стандартный вывод и стандартный вывод сообщений об ошибках. Это означает, что в любом процессе, запускаемом из дочернего процесса с помощью функции `exec`, эти три дескриптора останутся связаны с ведомым PTY (с его управляющим терминалом).

После вызова функции `fork` родительскому процессу возвращается дескриптор ведущего PTY и идентификатор дочернего процесса. В следующем разделе мы будем использовать функцию `pty_fork` при разработке программы `pty`.

Листинг 19.2. Функция `pty_fork`

```
#include "apue.h"
#include <termios.h>

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int      fdm, fds;
```

```
pid_t pid;
char pts_name[20];

if ((fdm = ptym_open(pts_name, sizeof(pts_name))) < 0)
    err_sys("невозможно открыть ведущий pty: %s, ошибка %d", pts_name, fdm);

if (slave_name != NULL) {
    /*
     * Вернуть имя ведомого устройства.
     * Завершить нулевым символом, чтобы обработать ситуацию, когда
     * strlen(ptr) > pts_namesz.
     */
    strncpy(slave_name, pts_name, slave_namesz);
    slave_name[slave_namesz - 1] = '\0';
}

if ((pid = fork()) < 0) {
    return(-1);
} else if (pid == 0) { /* дочерний процесс */
    if (setsid() < 0)
        err_sys("ошибка вызова функции setsid");

    /*
     * System V автоматически назначает управляющий терминал при открытии.
     */
    if ((fds = ptys_open(pts_name)) < 0)
        err_sys("невозможно открыть ведомый pty");
    close(fdm); /* работа с ведущим pty в дочернем процессе завершена */

#endif
/* Команда TIOCSCTTY — способ назначения управляющего терминала в BSD.
 */
if (ioctl(fds, TIOCSCTTY, (char *)0) < 0)
    err_sys("ошибка выполнения команды TIOCSCTTY");

/*
 * Инициализировать структуры termios и winsize ведомого pty.
 */
if (slave_termios != NULL) {
    if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
        err_sys("ошибка вызова tcsetattr для ведомого pty");
}
if (slave_winsize != NULL) {
    if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
        err_sys("ошибка выполнения TIOCSWINSZ для ведомого pty");
}

/*
 * Связать stdin/stdout/stderr с терминалом в дочернем процессе.
 */
if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
    err_sys("ошибка вызова функции dup2 для stdin");
if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
    err_sys("ошибка вызова функции dup2 для stdout");
if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
    err_sys("ошибка вызова функции dup2 для stderr");
if (fds != STDIN_FILENO && fds != STDOUT_FILENO &&
    fds != STDERR_FILENO)
```

```

        close(fds);
    return(0); /* вернуть 0 дочернему процессу, как это делает fork() */
} else {
    /* родительский процесс */
    *ptrfdm = fdm; /* вернуть fd ведущего pty */
    return(pid); /* вернуть pid дочернего процесса родителю */
}
}

```

19.5. Программа pty

Цель программы `pty` — дать возможность вводить команды в виде

`pty prog arg1 arg2`

вместо

`prog arg1 arg2`

Когда для запуска программы используется `pty`, эта программа работает в рамках своего собственного сеанса, связанного с псевдотерминалом.

Рассмотрим исходный код программы `pty`. Первый файл содержит функцию `main` (листинг 19.3). Она обращается к функции `pty_fork`, которая была описана в предыдущем разделе.

Листинг 19.3. Функция `main` программы `pty`

```

#include "apue.h"
#include <termios.h>

#ifndef LINUX
#define OPTSTR "+d:einv"
#else
#define OPTSTR "d:einv"
#endif

static void set_noecho(int); /* реализация находится в конце этого файла */
void      do_driver(char *); /* в файле driver.c */
void      loop(int, int); /* в файле loop.c */

int
main(int argc, char *argv[])
{
    int          fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t        pid;
    char        *driver;
    char        slave_name[20];
    struct termios orig_termios;
    struct winsize size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;
}

```

```
opterr = 0; /* нежелательно, чтобы getopt() выводила на stderr */
while ((c = getopt(argc, argv, OPTSTR)) != EOF) {
    switch (c) {
        case 'd': /* драйвер для stdin/stdout */
            driver = optarg;
            break;
        case 'e': /* отключить эхо-вывод для ведомого pty */
            noecho = 1;
            break;
        case 'i': /* игнорировать символ EOF для стандартного ввода */
            ignoreeof = 1;
            break;
        case 'n': /* неинтерактивный режим */
            interactive = 0;
            break;
        case 'v': /* вывод подробных сообщений */
            verbose = 1;
            break;
        case '?':
            err_quit("недопустимая опция: -%c", optopt);
    }
}
if (optind >= argc)
    err_quit("Использование: "
             "pty [ -d driver -einv ] program [ arg ... ]");

if (interactive) { /* получить текущие termios и winsize */
    if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
        err_sys("ошибка вызова функции tcgetattr для stdin");
    if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("ошибка выполнения команды TIOCGWINSZ");
    pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
                   &orig_termios, &size);
} else {
    pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
                   NULL, NULL);
}

if (pid < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid == 0) { /* дочерний процесс */
    if (noecho)
        set_noecho(STDIN_FILENO); /* stdin - ведомый pty */

    if (execvp(argv[optind], &argv[optind]) < 0)
        err_sys("невозможно запустить: %s", argv[optind]);
}

if (verbose) {
    fprintf(stderr, "имя ведомого = %s\n", slave_name);
    if (driver != NULL)
        fprintf(stderr, "драйвер = %s\n", driver);
}

if (interactive && driver == NULL) {
    if (tty_raw(STDIN_FILENO) < 0) /* перевести tty в прозрачный режим */
        err_sys("ошибка вызова функции tty_raw");
    if (atexit(tty_atexit) < 0) /* восстановление настроек при выходе */
        err_sys("ошибка вызова функции atexit");
```

```

    }

    if (driver)
        do_driver(driver); /* изменить наши stdin/stdout */

    loop(fdm, ignoreeof); /* копировать stdin -> pty, pty -> stdout */

    exit(0);
}

static void
set_noecho(int fd) /* отключить эхо-вывод (для ведомого pty) */
{
    struct termios stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("ошибка вызова функции tcgetattr");

    stermios.c_lflag &= .(ECHO | ECHOE | ECHOK | ECHONL);

    /*
     * Кроме того, отключить преобразование NL в CR/NL при выводе.
     */
    stermios.c_oflag &= .(ONLCR);

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("ошибка вызова функции tcsetattr");
}

```

Различные параметры командной строки мы рассмотрим в следующем разделе, когда будем экспериментировать с программой `pty`. Функция `getopt` помогает разобрать аргументы командной строки. Чтобы гарантировать совместимость со стандартом POSIX в Linux, мы добавляем в начало строки параметров знак «плюс».

Перед обращением к функции `pty_fork` мы получаем текущие значения структур `termios` и `winsize` и передаем их функции `pty_fork` в виде аргументов. Благодаря этому ведомый PTY будет иметь точно такие же настройки, что и текущий терминал.

После возврата из `pty_fork` дочерний процесс отключает эхо-вывод для ведомого PTY (если задан соответствующий параметр) и затем с помощью `execvp` запускает программу, указанную в командной строке. Все остальные аргументы командной строки передаются этой программе.

Родительский процесс переводит пользовательский терминал в прозрачный (`raw`) режим (если выбран соответствующий параметр) и при необходимости устанавливает обработчик выхода, который восстановит настройки терминала в случае вызова функции `exit`. Функция `do_driver` будет описана в следующем разделе.

После этого родительский процесс вызывает функцию `loop` (листинг 19.4), которая копирует все, что принято со стандартного ввода и стандартного вывода ведущего PTY. Для разнообразия мы предусмотрели выполнение этих операций двумя процессами, хотя в данной ситуации эту задачу вполне можно решить с помощью функций мультиплексирования ввода/вывода `select` и `poll` или с помощью нескольких потоков.

Листинг 19.4. Функция loop

```
#include "apue.h"

#define BUFFSIZE 512

static void sig_term(int);
static volatile sig_atomic_t sigcaught; /* изменяется обработчиком сигнала */

void
loop(int ptym, int ignoreeof)
{
    pid_t    child;
    int      nread;
    char     buf[BUFFSIZE];

    if ((child = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (child == 0) { /* дочерний процесс копирует stdin в ptym */
        for ( ; ; ) {
            if ((nread = read(STDIN_FILENO, buf, BUFFSIZE)) < 0)
                err_sys("ошибка чтения из stdin");
            else if (nread == 0)
                break; /* EOF в stdin означает конец ввода */
            if (writen(ptym, buf, nread) != nread)
                err_sys("ошибка записи в ведущий pty");
        }
        /*
         * Мы всегда завершаем работу, когда обнаруживаем EOF в stdin,
         * но извещаем родителя только тогда, когда ignoreeof == 0.
         */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM); /* известить родительский процесс */
        exit(0); /* и завершить работу; дочерний процесс */
        /* не может вернуть управление */
    }
    /*
     * Родительский процесс копирует ptym в stdout.
     */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("ошибка вызова функции signal_intr для SIGTERM");
    for ( ; ; ) {
        if ((nread = read(ptym, buf, BUFFSIZE)) <= 0)
            break; /* перехвачен сигнал, ошибка или получен EOF */
        if (writen STDOUT_FILENO, buf, nread) != nread)
            err_sys("ошибка записи в stdout");
    }
    /*
     * В этой точке мы оказываемся в трех случаях: функция sig_term()
     * (ниже) перехватила сигнал SIGTERM от дочернего процесса,
     * был прочитан символ EOF из ведущего pty (это означает,
     * что мы должны известить об этом потомку) или в случае ошибки.
     */
    if (sigcaught == 0)
        /* послать сигнал потомку, */
        kill(child, SIGTERM); /* если от него не был получен сигнал */

    /*
     * Родительский процесс возвращает управление вызывающему.
     */
}
```

```

        */
}

/*
 * Потомок посыпает сигнал SIGTERM, когда получает EOF из ведомого pty или
 * когда функция read() терпит неудачу. Вероятно, было прервано чтение из ptym.
 */
static void
sig_term(int signo)
{
    sigcaught = 1; /* просто установить флаг и вернуться */
}

```

Обратите внимание: в случае с двумя процессами, когда один завершает работу, он сообщает об этом другому с помощью сигнала SIGTERM.

19.6. Использование программы pty

Теперь рассмотрим некоторые примеры использования программы `pty` и попутно объясним назначение различных параметров.

Если в качестве командной оболочки используется Korn shell, можно запустить команду

`pty ksh`

и получить совершенно новый экземпляр командной оболочки, работающей под управлением псевдотерминала.

Если предположить, что программа из листинга 18.7 называется `ttyname`, можно ее запустить так:

```

$ who
sar  console May 19 16:47
sar  ttys000 May 19 16:47
sar  ttys001 May 19 16:48
sar  ttys002 May 19 16:48
sar  ttys003 May 19 16:49
sar  ttys004 May 19 16:49          ttys004 - наибольший номер PTY, используемый
                                    в настоящее время
$ pty ttyname                  запуск программы из листинга 18.7 из PTY
fd 0: /dev/ttys005              ttys005 - следующий доступный PTY
fd 1: /dev/ttys005
fd 2: /dev/ttys005

```

Файл utmp

В разделе 6.8 мы рассматривали файл `utmp`, в котором хранятся сведения обо всех работающих в системе пользователях. Запуск пользовательской программы на псевдотерминале рассматривается как вход в систему. В случае удаленного входа в систему через `telnetd` или `rlogin` в файл `utmp` должна помещаться соответствующая запись о входе с псевдотерминала. Однако в случае запуска командной оболочки на псевдотерминале из оконной системы или из таких программ, как `script`, это соглашение соблюдается не всегда. Некоторые системы в этом случае добавляют записи в файл `utmp`, а некоторые — нет. Если система не производит

запись в файл `utmp`, команда `who(1)`, как правило, не сообщает, что соответствующие псевдотерминалы используются.

Если для файла `utmp` не установлен бит права на запись для остальных (что, строго говоря, рассматривается как уязвимость в системе безопасности), не все программы, использующие псевдотерминалы, смогут добавлять записи в этот файл.

Взаимодействие с механизмом управления заданиями

При запуске под управлением `pty` командной оболочки, поддерживающей управление заданиями, мы не заметим ничего необычного. Например, команда

```
pty ksh
```

запустит Korn shell под управлением `pty`. Мы сможем запускать программы в этой новой командной оболочке и использовать механизм управления заданиями, как это делается в командной оболочке входа. Но когда под управлением `pty` запускается не командная оболочка, которая поддерживает управление заданиями, а любая другая интерактивная программа, как, например,

```
pty cat
```

все будет работать прекрасно, пока мы не введем символ приостановки задания. В этом случае управляющий символ выводится как `^Z` и игнорируется. В ранних версиях BSD это приводило к завершению процессов `cat` и `pty` и возвращению в первоначальную командную оболочку. Чтобы разобраться в этой ситуации, нам необходимо исследовать все задействованные процессы, их группы процессов и сеансы. На рис. 19.8 приводится схема состояния процессов, которая соответствует запуску команды `pty cat`.

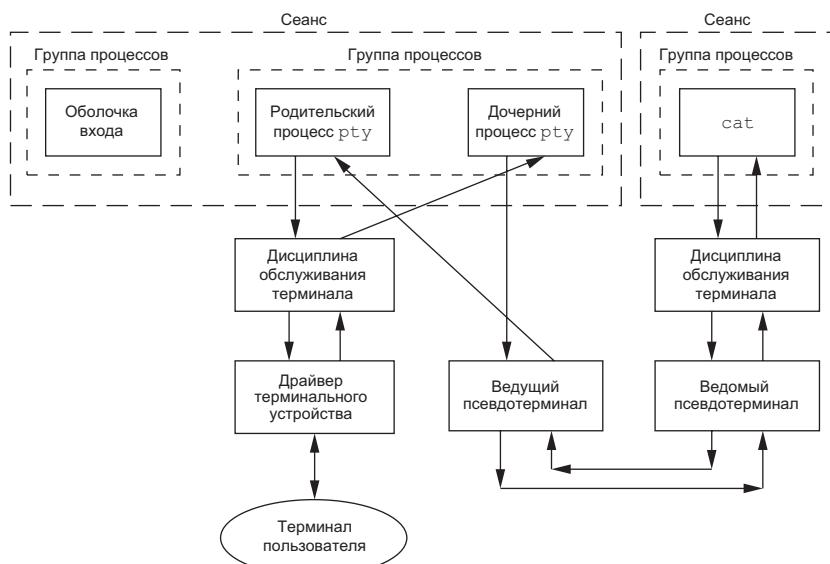


Рис. 19.8. Группы процессов и сеансы, создаваемые командой `pty cat`

Когда мы вводим символ приостановки задания (Control-Z), он распознается модулем дисциплины обслуживания терминала (на рис. 19.8 расположен ниже процесса `cat`), поскольку `pty` переводит терминал (на рисунке — ниже родительского процесса `pty`) в прозрачный режим. Но ядро не будет приостанавливать процесс `cat`, поскольку он принадлежит осиротевшей группе процессов (раздел 9.10). Родительским процессом для `cat` является `pty`, а он принадлежит другому сеансу.

Исторически разные реализации по-разному обрабатывали эту ситуацию. Стандарт POSIX.1 утверждает лишь, что сигнал `SIGTSTP` не может доставляться процессу. Системы, производные от 4.3BSD, вместо него доставляли сигнал `SIGKILL`, который не может быть перехвачен процессом. В 4.4BSD такое поведение системы было изменено в соответствии со стандартом POSIX.1. Вместо посылки сигнала `SIGKILL` ядро 4.4BSD просто уничтожает сигнал `SIGTSTP`, если для него выбрана диспозиция по умолчанию и получатель находится в осиротевшей группе процессов. Большинство современных реализаций придерживаются именно такого поведения.

Когда мы запускаем под управлением `pty` командную оболочку, поддерживающую управление заданиями, задания, запускаемые из этой новой оболочки, уже не принадлежат осиротевшей группе процессов, поскольку командная оболочка сама принадлежит тому же сеансу. В этом случае символ Control-Z передается процессу, запущенному из оболочки, а не самой оболочки.

Единственный способ ликвидировать неспособность процесса, вызываемого программой `pty`, обрабатывать символы управления заданиями — добавить в программу `pty` еще один параметр командной строки, который будет заставлять ее самостоятельно распознавать символ приостановки задания (в дочернем процессе `pty`), вместо того чтобы передавать его на обработку другим модулям дисциплины обслуживания.

Отслеживание вывода программ, работающих продолжительное время

Другой пример взаимодействия с механизмом управления заданиями приводится на рис. 19.7. Если запустить программу, которая выводит данные достаточно редко и небольшими порциями, например

```
pty slowout > file.out &
```

выполнение процесса `pty` будет приостановлено, как только дочерний процесс попытается прочитать данные со своего стандартного ввода (с терминала). Дело в том, что задание выполняется в фоновом режиме и будет приостановлено механизмом управления заданиями, как только попытается получить доступ к терминалу. Если перенаправить стандартный ввод так, чтобы `pty` не пыталась читать из терминала, например:

```
pty slowout < /dev/null > file.out &
```

программа `pty` сразу же приостановится, потому что прочитает со стандартного ввода признак конца файла, и завершится. Решение этой проблемы заключается

в передаче программе ключа `-i`, который сообщает, что программа должна игнорировать признак конца файла, полученный со стандартного ввода:

```
pty -i slowout < /dev/null > file.out &
```

Передача этого ключа приводит к тому, что дочерний процесс `pty` из листинга 19.4 завершается при получении признака конца файла, но не сообщает об этом родительскому процессу. Благодаря этому родительский процесс продолжает копировать вывод из ведомого PTY на стандартный вывод (в данном примере — файл `file.out`).

Программа script

С помощью программы `pty` можно реализовать программу `script(1)` в виде простого сценария на языке командной оболочки:

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

После запуска этого сценария можно проследить взаимоотношения между процессами с помощью команды `ps`. Схема процессов, описывающая эти взаимоотношения, приводится на рис. 19.9.

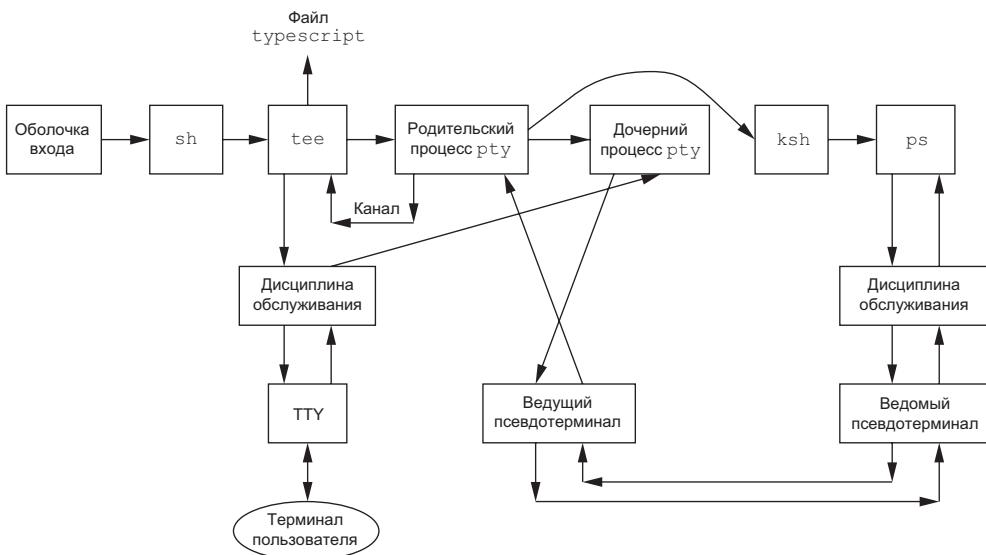


Рис. 19.9. Схема процессов, запущенных сценарием `script`

В этом примере предполагается, что содержимое переменной `SHELL` соответствует командной оболочке Korn shell (скорее всего, `/bin/ksh`). Как уже говорилось, программа `script` копирует только то, что выводится новой оболочкой (и любыми процессами, которые из нее запущены). Но благодаря тому, что модуль дис-

циплины обслуживания, расположенный на рис. 19.9 выше ведомого PTY, обычно разрешает эхо-вывод, большая часть ввода с клавиатуры также попадает в файл `typescript`.

Запуск сопроцессов

Сопроцессы из листинга 15.9 не могли использовать функции стандартной библиотеки ввода/вывода, потому что для стандартных потоков ввода и вывода, не связанных с терминалом, выбирается режим полной буферизации. Если запустить сопрощес под управлением программы `pty`, заменив строку

```
if (exec1("./add2", "add2", (char *)0) < 0)
```

на

```
if (exec1("./pty", "pty", "-e", "add2", (char *)0) < 0)
```

программа будет работать, даже если сопрощес использует функции стандартной библиотеки ввода/вывода.

На рис. 19.10 приводится схема состояния процессов при запуске сопрощеса с вводом/выводом через псевдотерминал. Эта расширенная версия рис. 19.6 показывает все связанные процессы и потоки движения данных. Под блоком «управляющая программа» подразумевается программа из листинга 15.9, для которой мы изменили вызов функции `exec1`.

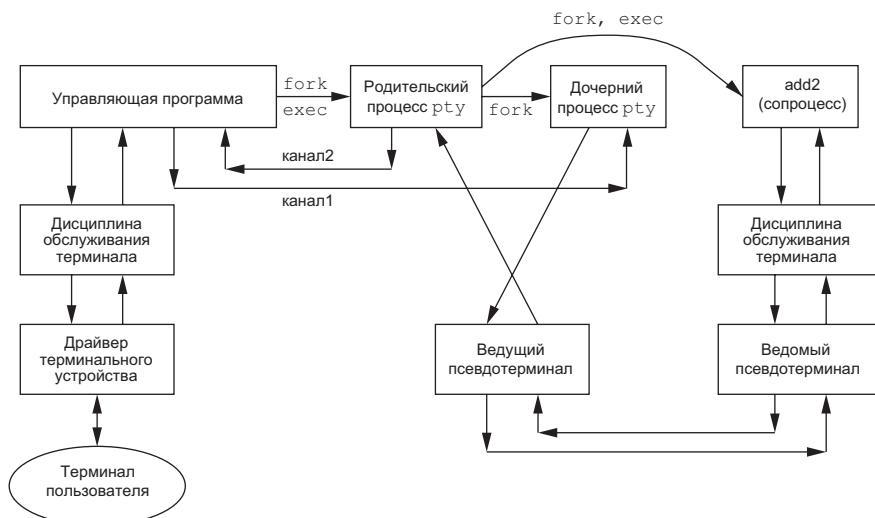


Рис. 19.10. Запуск сопрощеса с вводом/выводом через псевдотерминал

Из этого примера видно, что мы должны запускать программу `pty` с ключом `-e` (запрет эхо-вывода). Программа `pty` запускается в неинтерактивном режиме, потому что ее стандартный ввод не связан с терминалом. В листинге 19.3 флаг `interactive` будет по умолчанию сброшен, потому что функция `isatty` вернет

ложное значение. Это означает, что дисциплина обслуживания терминала, расположенная на рисунке выше фактического терминала, останется в каноническом режиме с разрешенным эхо-выводом. Ключом `-e` мы отключаем эхо-вывод в модуле дисциплины обслуживания, расположенном на рис. 19.10 выше ведомого PTY. Если этого не сделать, весь ввод с клавиатуры будет выведен дважды — обеими модулями дисциплины обслуживания.

Кроме того, ключ `-e` сбрасывает флаг `ONLCR` в структуре `termios`, что предотвращает преобразование символов перевода строки, выводимых сопроцессом, в последовательности символов `CR-NL`.

Тестирование этого примера на разных платформах выявило еще одну проблему, о которой упоминалось в разделе 14.7 при описании функций `readn` и `writen`. Объем данных, возвращаемых функцией `read` при работе с дескриптором, соответствующим файлу, отличному от обычного дискового файла, может различаться в разных реализациях. В программе из листинга 15.9 запуск сопроцесса с использованием `pty` дал неожиданные результаты при использовании функции `read` с неименованными каналами, когда `read` возвращала неполную строку. Решение проблемы заключается в том, чтобы вместо программы из листинга 15.9 задействовать программу из упражнения 15.5, использующую функции стандартной библиотеки ввода/вывода для работы с каналами, для которых устанавливается режим построчной буферизации. Благодаря этому функция `fgets` будет вызывать функцию `read` столько раз, сколько потребуется для получения полной строки. Цикл `while` в листинге 15.9 построен в предположении, что каждой строке, переданной сопроцессу, будет соответствовать одна строка, полученная от него.

Неинтерактивное управление интерактивными программами

Хотя все вышеизложенное заставляет думать, что посредством программы `pty` можно запускать любые сопроцессы, в действительности `pty` не может взаимодействовать с интерактивными сопроцессами. Проблема в том, что `pty` просто копирует данные, полученные со своего стандартного ввода, в PTY и выводит на свой стандартный вывод все, что поступает от PTY, не анализируя при этом, что было получено и что следует отправить.

Например, мы можем запустить команду `telnet` под управлением `pty`, передав ей адрес удаленного хоста:

```
pty telnet 192.168.1.3
```

Такая последовательность команд не дает никаких преимуществ перед простой командой `telnet 192.168.1.3`, но нам может потребоваться запускать команду `telnet` из сценария, чтобы с ее помощью проверить состояние удаленного хоста. Представим себе, что у нас есть файл `telnet.cmd`, который содержит четыре строки:

```
sar
пароль
uptime
exit
```

где первая строка — имя пользователя, вторая строка — пароль, третья — команда, запускаемая на удаленной машине, и четвертая — команда закрытия сеанса. Но если запустить сценарий как

```
pty -i < telnet.cmd telnet 192.168.1.3
```

мы не получим ожидаемого результата. Дело в том, что содержимое файла `telnet.cmd` будет отправлено удаленной стороне еще до запроса на ввод имени пользователя и пароля. Для отключения эхо-вывода при вводе пароля программа `login` вызывает функцию `tcsetattr`, которая сбрасывает все данные, находившиеся в очереди. То есть отправленные нами данные будут просто потеряны.

Когда команда `telnet` запускается в интерактивном режиме, мы ждем приглашения от удаленной стороны, прежде чем вводить пароль, но программа `pty` ничего не знает об этом. По этой причине для управления интерактивными программами из сценария необходимо использовать программу более сложную, чем `pty`, например `expect`.

Даже запуск `pty` из программы, приведенной в листинге 15.9, нам не поможет, потому что эта программа построена на предположении, что каждой строке, записываемой в один канал, соответствует только одна строка, получаемая из другого канала. При работе с интерактивными программами нередки случаи, когда ввод одной строки приводит к выводу нескольких строк. Кроме того, программа из листинга 15.9 всегда отправляет строку сопроцессу, прежде чем прочитать ответ от него. Это не подходит для случая, когда мы должны получить какие-либо данные от сопроцесса, прежде чем отправить ему что-нибудь.

Существует несколько способов организовать взаимодействие с интерактивной программой из сценария. Можно придумать для `pty` язык команд и его интерпретатор, но такой «довесок» наверняка будет в десятки раз превышать размер самой программы `pty`. Другой вариант — взять существующий командный интерпретатор, который мог бы управлять интерактивной программой, запуская ее с помощью функции `pty_fork`. Именно так и работает программа `expect`.

Мы выберем иной путь и просто добавим возможность соединить ввод и вывод программы `pty` с управляющим процессом с помощью ключа `-d`. Стандартный вывод управляющего процесса (драйвера) соединяется со стандартным вводом программы `pty`, и наоборот. Это очень похоже на работу с сопроцессом. Результат напоминает рис. 19.10, но в данной ситуации процесс-драйвер запускается программой `pty`. Кроме того, вместо двух полудуплексных каналов мы используем для взаимодействия `pty` и драйвера один двунаправленный канал.

В листинге 19.5 приводится исходный код функции `do_driver`, которая вызывается из функции `main` (листинг 19.3), если указан ключ `-d`.

Листинг 19.5. Функция `do_driver` для программы `pty`

```
#include "apue.h"

void
do_driver(char *driver)
{
    pid_t child;
    int pipe[2];
```

```
/*
 * Создать канал для взаимодействия с драйвером.
 */
if (fd_pipe(pipe) < 0)
    err_sys("невозможно создать канал");

if ((child = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (child == 0) { /* дочерний процесс */
    close(pipe[1]);

    /* stdin драйвера */
    if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
        err_sys("ошибка вызова функции dup2 для stdin");

    /* stdout драйвера */
    if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("ошибка вызова функции dup2 для stdout");
    if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
        close(pipe[0]);

    /*оставить stderr драйвера неизменным*/
    execlp(driver, driver, (char *)0);
    err_sys("ошибка вызова функции execlp для: %s", driver);
}

close(pipe[0]); /* родительский процесс */
if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
    err_sys("ошибка вызова функции dup2 для stdin");
if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
    err_sys("ошибка вызова функции dup2 для stdout");
if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
    close(pipe[1]);

/*
 * Родительский процесс возвращает управление, но stdin и stdout
 * остаются связанными с драйвером.
 */
}
```

Написав свой драйвер, который будет вызываться программой `pty`, мы можем управлять интерактивными программами по своему желанию. Даже при том, что стандартный ввод и стандартный вывод драйвера связаны с программой `pty`, он по-прежнему может взаимодействовать с пользователем, обращаясь к устройству `/dev/tty`. Наше решение не такое универсальное, как программа `expect`, зато мы дополнили программу `pty` очень полезной возможностью, добавив всего 50 строк кода.

19.7. Дополнительные возможности

Псевдотерминалы обладают рядом дополнительных возможностей, о которых мы коротко расскажем в этом разделе. Эти возможности описываются в [Sun Microsystems, 2002] и на страницах справочного руководства BSD к программе `pts(4)`.

Пакетный режим

Пакетный режим позволяет ведущему PTY узнавать об изменении состояния ведомого PTY. В Solaris этот режим устанавливается размещением модуля `pckt` в потоке STREAMS со стороны ведущего PTY. Мы показали этот модуль на рис. 19.2. В FreeBSD, Linux и Mac OS X этот режим устанавливается командой `TIOCSPKT` функции `ioctl`.

Внутренняя реализация пакетного режима в Solaris отличается от реализации на других платформах. В Solaris процесс, получающий данные из ведущего PTY, должен вызывать функцию `getmsg`, чтобы получать сообщения из головы потока, так как модуль `pckt` преобразует отдельные события в сообщения STREAMS, не содержащие данных. На других платформах операция чтения из ведущего PTY возвращает байт статуса, сопровождаемый необязательными данными.

Вне зависимости от внутренней реализации назначение пакетного режима заключается в том, чтобы информировать процесс, читающий данные из ведущего PTY, о наступлении определенных событий, которые происходят в модуле дисциплины обслуживания терминала, расположенного выше ведомого PTY: сбрасывается очередь чтения, сбрасывается очередь записи, приостанавливается вывод (например, по `Control-S`), возобновляется вывод, разрешается управление потоком данных `XON/XOFF` после того, как оно было запрещено, запрещается управление потоком данных `XON/XOFF` после того, как оно было разрешено. Эти события используются, например, клиентом `rlogin` и сервером `rlogind`.

Дистанционный режим

Ведущий PTY может перевести ведомый PTY в дистанционный режим, запустив функцию `ioctl` с командой `TIOCREMOTE`. Хотя Mac OS X 10.6.8 и Solaris 10 используют для входа и выхода из этого режима одну и ту же команду, в Solaris в третьем аргументе функции `ioctl` передается целое число, в то время как в Mac OS X – указатель на целое число. (FreeBSD 8.0 и Linux 3.2.0 не поддерживают эту команду.)

Установкой этого режима ведущий PTY сообщает модулю дисциплины обслуживания ведомого PTY, что он не должен производить какую-либо обработку данных, поступающих от ведущего PTY, вне зависимости от состояния флага канонического режима в структуре `termios` ведомого PTY. Дистанционный режим предназначен для таких приложений, как диспетчер окон, который реализует свою процедуру обработки строк.

Изменение размеров окна

Процесс, расположенный над ведущим PTY, может командой `TIOCWINSZ` функции `ioctl` изменить размер окна ведомого PTY. Если новый размер окна отличается от текущего, группе процессов переднего плана ведомого PTY будет послан сигнал `SIGWINCH`.

Генерация сигналов

Процесс, имеющий доступ к ведущему PTY, может посыпать сигналы группе процессов ведомого PTY. В Solaris 10 это можно сделать с помощью команды

TIOCSIGNAL функции `ioctl`, в третьем аргументе которой передается номер сигнала. В FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 для этих целей используется команда TIOCSIG функции `ioctl`. В обоих случаях в третьем аргументе передается номер сигнала.

19.8. Подведение итогов

Эту главу мы начали с краткого обзора области применения псевдотерминалов и рассмотрели ряд примеров их использования. После этого мы исследовали код открытия псевдотерминала под управлением каждой из четырех платформ, обсуждаемых в этой книге. Затем мы использовали этот код для реализации универсальной функции `pty_fork`, которая может применяться различными приложениями. Эта функция легла в основу небольшой программы `pty`, с помощью которой мы смогли исследовать многочисленные свойства псевдотерминалов.

Псевдотерминалы широко применяются в большинстве современных версий UNIX для организации сетевого входа в систему. Мы рассмотрели и другие способы использования псевдотерминалов — от простой программы `script` до управления интерактивными программами из сценариев пакетной обработки данных.

Упражнения

- 19.1 В процессе удаленного входа в BSD-систему посредством программы `telnet` или `rlogin` идентификатор владельца и права доступа для ведомого PTY устанавливаются согласно правилам, описанным в разделе 19.3. Как это происходит?
- 19.2 С помощью программы `pty` определите, какими значениями инициализируются структуры `termios` и `winsize` ведомого PTY в вашей системе.
- 19.3 Реализуйте функцию `loop` (листинг 19.4) в виде отдельного процесса, чтобы она использовала функцию `select` или `poll`.
- 19.4 В дочернем процессе после возврата из функции `pty_fork` стандартные устройства ввода, вывода и сообщений об ошибках открыты для чтения и записи. Можно ли изменить права доступа к ним, чтобы стандартное устройство ввода было доступно только для чтения, а остальные два устройства — только для записи?
- 19.5 На рис. 19.8 попытайтесь определить, какая группа процессов выполняется на переднем плане, а какая — в фоновом режиме. Определите лидеров сессий.
- 19.6 В каком порядке завершатся процессы из рис. 19.8, если ввести символ конца файла? Если возможно, проверьте это с помощью механизма учета ресурсов, потребляемых процессами.
- 19.7 Обычно программа `script(1)` добавляет в начало выходного файла строку, содержащую время начала работы, а в конец файла — время окончания.

Добавьте эту возможность в сценарий на языке командной оболочки, который мы продемонстрировали.

- 19.8 Объясните, почему содержимое файла `data` в следующем примере выводится на терминал, если программа `ttyname` только выводит данные и никогда не читает их?

```
$ cat data          файл с двумя строками текста
hello,
world
$ pty -i < data ttyname    ключ -i говорит о том, что символ EOF
                           должен игнорироваться
hello,                      откуда появились эти две строки?
world
fd 0: /dev/ttys005          эти три строки мы ожидали
fd 1: /dev/ttys005          получить от ttyname
fd 2: /dev/ttys005
```

- 19.9 Напишите программу, которая вызывала бы функцию `pty_fork` и дочерний процесс запускал бы другую программу, которую вы также должны написать. Новая программа, запускаемая из дочернего процесса, должна перехватывать сигналы `SIGTERM` и `SIGWINCH`. При получении сигналов программа должна выводить соответствующие сообщения, причем при получении сигнала `SIGWINCH` она должна дополнительно выводить размеры окна терминала. После запуска программы родительский процесс должен послать сигнал группы процессов ведомого PTY с помощью функции `ioctl`, как это было описано в разделе 19.7, прочитать строку с сообщением от ведомого PTY и убедиться, что дочерний процесс получил сигнал. Затем родительский процесс должен изменить размер окна ведомого PTY и опять прочитать строку с сообщением от ведомого PTY. Далее завершите родительский процесс и посмотрите, завершился ли дочерний. Если это произошло, то объясните почему.

20

Библиотека базы данных

20.1. Введение

В начале 1980-х годов UNIX считалась недружественной средой для много-пользовательских систем управления базами данных. (См. [Stonebraker, 1981] и [Weinberger, 1982].) Для ранних версий UNIX, таких как Version 7, подобные утверждения действительно представлялись обоснованными, поскольку какие-либо разновидности механизмов межпроцессных взаимодействий (исключая полудуплексные каналы) отсутствовали и механизм блокировок отдельных записей в файлах еще не был реализован. Однако со временем большинство из этих недостатков было устранено. К концу 80-х UNIX достигла такого уровня развития, который позволил обеспечить подходящую среду для работы надежных много-пользовательских систем управления базами данных. С тех пор коммерческими фирмами были разработаны самые различные системы баз данных.

В этой главе мы создадим простую библиотеку функций на языке C, которая может использоваться любыми программами для получения и хранения записей в базе данных. (Такие базы данных часто называют *хранилищами типа ключ/значение*.) Подобные библиотеки функций являются лишь частью полной системы управления базами данных. Мы не будем заниматься разработкой других ее компонентов, таких как язык запросов, оставляя освещение этих тем многочисленным учебникам по базам данных. Основной интерес для нас будут представлять интерфейсы UNIX, необходимые для реализации библиотеки, и то, как эти интерфейсы связаны с уже рассмотренными темами (такими, как блокировка записей в файлах, раздел 14.3).

20.2. Предыстория

Одной из популярных библиотек функций для работы с базами данных в UNIX является библиотека `dbm(3)`. Эта библиотека, разработанная Кеном Томпсоном, использует схему динамического хеширования. Изначально она распространялась вместе с Version 7, затем была перенесена во все выпуски BSD, а также поставлялась вместе с SVR4 для совместимости с BSD [AT&T 1990c]. Разработчики BSD расширили библиотеку `dbm` и назвали ее `ndbm`. Библиотека `ndbm` вошла в состав BSD и SVR4. Функции `ndbm` стандартизированы в виде расширений XSI стандарта Single UNIX Specification.

Подробная история развития алгоритма динамического хеширования, используемого библиотекой `dbm` и последующими ее реализациями, включая `gdbm` (GNU-версия библиотеки `dbm`), приводится в книге [Seltzer and Yigit, 1991]. К сожалению, основное ограничение всех этих реализаций заключается в том, что ни одна из них не допускает одновременного обновления данных из нескольких процессов. Эти реализации не предусматривают никаких средств управления одновременным доступом (например, блокировку записей).

В 4.4BSD была реализована новая библиотека `db(3)`, которая поддерживала три формы выборки: (а) ориентированную на записи, (б) хеширование и (в) двоичные деревья (B-tree). Но она также не предоставляла возможности одновременного доступа (о чем прямо заявлено в разделе BUGS страницы справочного руководства `db(3)`).

Компания Oracle (<http://www.oracle.com>) разработала версии библиотеки db, которые поддерживают одновременную работу нескольких пользователей, а также механизмы блокировки записей и транзакций.

Большинство коммерческих библиотек баз данных реализуют механизмы управления одновременным доступом к данным из нескольких процессов. Для этого они обычно применяют рекомендательные блокировки, описанные в разделе 14.3, но нередко реализуют собственные примитивы, чтобы избежать накладных расходов, которые неизбежно возрастают, когда системные вызовы не могут установить уже захваченную блокировку. Чаще всего эти коммерческие системы реализуют базы данных на основе сбалансированных двоичных деревьев (B+ tree) [Comer, 1979] или алгоритмов динамического хеширования — например, линейного хеширования [Litwin, 1980] или расширяемого хеширования [Fagin et al., 1979].

В табл. 20.1 приводится список библиотек, которые обычно поставляются в составе четырех платформ, рассматриваемых в этой книге. Обратите внимание, что в Linux поддержка функций `dbm` и `ndbm` предоставляется библиотекой `gdbm`.

Таблица 20.1. Библиотеки баз данных, поддерживаемые различными платформами

Библиотека	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
<code>dbm</code>			<code>gdbm</code>		✓
<code>ndbm</code>	XSI	✓	<code>gdbm</code>	✓	✓
<code>db</code>		✓	✓	✓	

20.3. Библиотека

Библиотека, которую мы разработаем в этой главе, похожа на библиотеку `ndbm`, но мы добавим к ней механизм одновременного доступа к данным из нескольких процессов. В первую очередь мы рассмотрим интерфейс к библиотеке на языке C, а в следующем разделе перейдем к фактической реализации.

При открытии базы данных мы получаем некоторый дескриптор (обычный указатель), который представляет базу данных. Этот дескриптор будет передаваться всем функциям, работающим с базой данных.

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);
```

Возвращает дескриптор базы данных в случае успеха,
NULL — в случае ошибки

```
void db_close(DBHANDLE db);
```

Если вызов функции `db_open` завершается успехом, она создает два файла: индексный файл `pathname.idx` и файл с данными `pathname.dat`. Аргумент `oflag` используется так же, как второй аргумент функции `open` (раздел 3.3), — он определяет режим открытия файлов (только для чтения, для записи и для чтения, создание файла, если он не существует, и т. д.). Аргумент `mode` используется при создании файлов базы данных подобно третьему аргументу функции `open` (он определяет права доступа).

По завершении работы с базой данных ее следует закрыть вызовом функции `db_close`. Эта функция закрывает индексный файл и файл с данными и освобождает память, которая была выделена под внутренние буферы.

При добавлении в базу новой записи необходимо указать ключ записи и данные, связанные с этим ключом. Так, если база данных хранит данные о сотрудниках, в качестве ключа может использоваться идентификатор сотрудника, а в качестве данных — имя сотрудника, его домашний адрес, номер телефона, дата приема на работу и т. п. Наша реализация требует, чтобы ключ для каждой из записей имел уникальное значение. (Это означает, что мы не сможем, например, создать две записи с одинаковыми идентификаторами.)

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data, int flag);
```

Возвращает 0 в случае успеха, ненулевое значение —
в случае ошибки (см. ниже)

Аргументы `key` и `data` — это строки, завершающиеся нулевым символом. Единственное ограничение, связанное с этими строками, состоит в том, что они не могут содержать нулевые символы в середине, зато могут содержать, например, символы перевода строки.

Аргумент `flag` может принимать значения `DB_INSERT` (при добавлении новой записи), `DB_REPLACE` (при изменении существующей записи) или `DB_STORE` (при добавлении новой или изменении существующей записи, в зависимости от наличия записи в базе). Эти три константы определены в заголовочном файле `apue_db.h`.

Если указан флаг `DB_INSERT` или `DB_STORE` и запись не существует в базе, будет добавлена новая запись. Если указан флаг `DB_REPLACE` или `DB_STORE` и запись уже существует в базе, существующая запись будет замещена новой. Если указан флаг `DB_REPLACE`, а искомой записи в базе не окажется, функция вернет значение `-1` и код ошибки `ENOENT` в переменной `errno`, при этом новая запись добавлена не будет. Если указан флаг `DB_INSERT` и запись уже существует в базе, добавление записи в базу не производится. В этом случае возвращается значение `1`, чтобы можно было отличить его от обычного завершения с ошибкой `(-1)`.

Извлечь запись из базы данных можно, указав ее ключ.

```
#include "apue_db.h"

char *db_fetch(DBHANDLE db, const char *key);
```

Возвращает указатель на данные в случае успеха,
NULL — если запись не была найдена

Если запись была найдена, функция возвращает указатель на данные, которые были сохранены с ключом *key*. Мы можем также удалить запись из базы данных, указав ее ключ.

```
#include "apue_db.h"

int db_delete(DBHANDLE db, const char *key);
```

Возвращает 0 в случае успеха, `-1` — если запись не была найдена

Кроме извлечения отдельных записей по заданным ключам, можно выполнить обход всей базы данных, извлекая записи по очереди. Для этого нужно сначала вызвать функцию `db_rewind`, чтобы переместиться на первую запись, и затем в цикле вызывать `db_nextrec`, читая записи одну за другой.

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);
```

Возвращает указатель на данные в случае успеха,
NULL — по достижении конца файла

Если в аргументе *key* передается непустой указатель, функция `db_nextrec` будет возвращать по этому адресу значение ключа очередной записи.

Порядок следования записей, возвращаемых `db_nextrec`, заранее не определен. Единственное, что можно гарантировать, — каждая запись будет возвращена всего один раз. Так, если в базе хранятся три записи с ключами А, В и С, нельзя заранее

предсказать, в каком порядке они будут возвращены функцией `db_nextrec`. Она может вернуть сначала В, потом А, а потом С или в каком-либо другом (совершенно случайном) порядке. Фактический порядок следования записей зависит от реализации базы данных.

Эти семь функций составляют интерфейс библиотеки базы данных. А теперь перейдем к описанию фактической реализации, выбранной нами.

20.4. Обзор реализации

Обычно библиотеки, реализующие доступ к базе данных, для хранения данных используют два файла: индексный файл и файл с данными. Индексный файл содержит значения индексов (ключей) и указатели на соответствующие им записи в файле с данными. Для организации хранения индексов используется множество методик, которые ускоряют поиск конкретного ключа; хеширование и сбалансированные двоичные деревья являются наиболее популярными. Мы выбрали методику на основе хеш-таблицы фиксированного размера с объединением в цепочки записей, имеющих одинаковые значения хешей. При описании функции `db_open` мы уже упоминали, что она создает два файла: один с расширением `.idx`, а другой с расширением `.dat`.

Мы будем хранить индексы и ключи в виде строк, завершающихся нулевым символом; они не допускают возможности хранения произвольных двоичных данных. Некоторые системы управления базами данных хранят числовые данные в двоичном формате (например, 1, 2 или 4 байта для хранения целых чисел), чтобы сэкономить место. Это приводит к усложнению функций и требует дополнительных усилий для обеспечения переносимости файлов базы данных между разными платформами. Например, если в сети имеются две системы, которые используют разные форматы представления целых чисел в двоичном формате, придется предусмотреть обработку ситуации, когда необходим доступ к базе данных из обеих систем. (Сегодня нет ничего необычного в том, что посредством сети файлы совместно используются системами с различной архитектурой.) Хранение всех записей, и ключей и данных, в виде строк символов упрощает реализацию. Такой подход ведет к увеличению занимаемого дискового пространства, но это не большая плата за переносимость.

Функция `db_store` допускает хранение только одной записи для каждого ключа. Некоторые системы управления базами данных позволяют хранить несколько записей с одинаковыми ключами и предоставляют возможность получить доступ ко всем записям, ассоциированным с заданным ключом. Кроме того, в нашем распоряжении будет всего один индексный файл — это означает, что каждой записи с данными может быть поставлен в соответствие только один ключ (мы не предусматриваем поддержку вторичных ключей). Базы данных, которые позволяют поставить в соответствие одной записи несколько ключей, очень часто создают по одному индексному файлу для каждого ключа. Каждый раз при удалении или добавлении записи все индексные файлы должны соответствующим образом обновляться. (Например, для файла с данными о сотрудниках мы могли бы опреде-

лить несколько индексов: один — идентификатор сотрудника, а другой — номер карточки социального обеспечения. Если в качестве индекса использовать имена сотрудников, это может породить определенные проблемы, так как имена могут не быть уникальными.)

На рис. 20.1 показана общая схема строения базы данных.

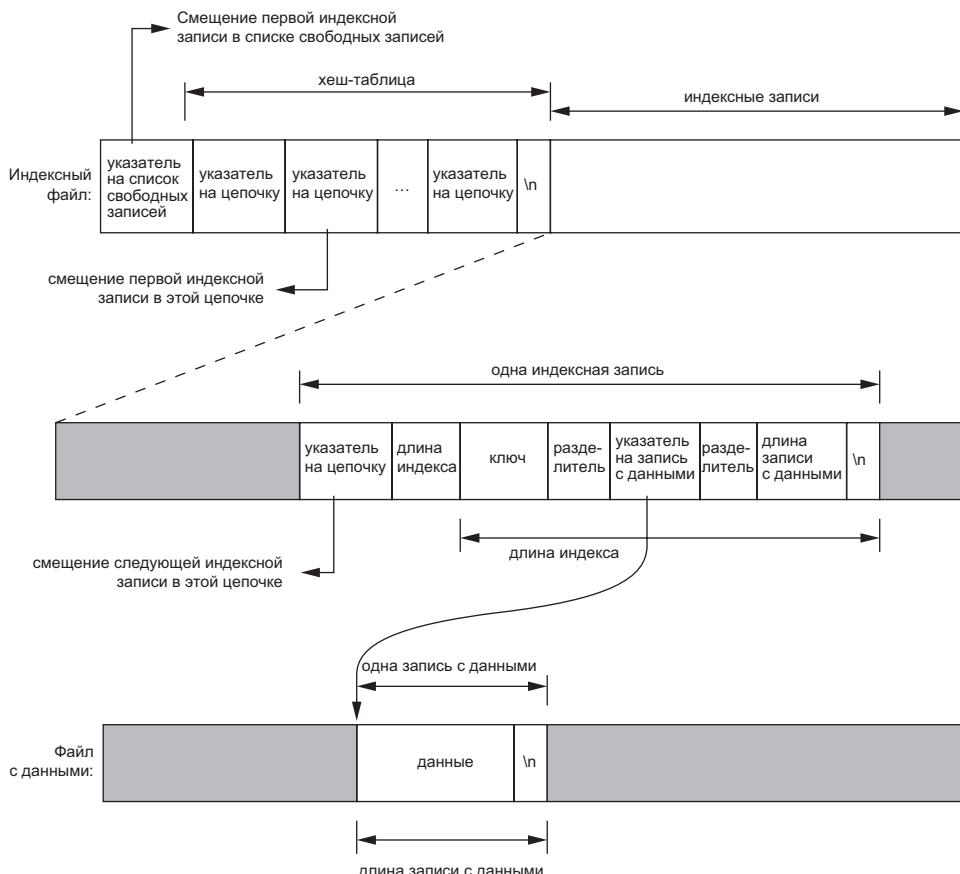


Рис. 20.1. Схема строения индексного файла и файла с данными

Индексный файл состоит из трех частей: указателя на список свободных записей, таблицы хешей и индексных записей. На рис. 20.1 все поля, которые названы указателями, представляют смещение от начала файла и хранятся в виде чисел в формате ASCII.

Чтобы отыскать в базе данных запись по заданному ключу, функция `db_fetch` рассчитывает значение хеша ключа, по которому отыскивается требуемая цепочка в таблице хешей. (Поле *указатель на цепочку* может содержать 0; это говорит о том, что цепочка пуста.) Затем осуществляется переход к найденной цепочке, которая представляет связанный список из всех индексных записей, имеющих то

же значение хеша. Когда функция обнаруживает 0 в поле *указатель на цепочку*, это означает, что достигнут конец цепочки.

Рассмотрим фактическое содержимое файлов базы данных. Программа в листинге 20.1 создает новую базу данных и записывает в нее три записи. Поскольку все поля в базе данных хранятся в виде символов ASCII, содержимое файлов можно просматривать, используя любые стандартные средства UNIX:

```
$ ls -l db4.*  
-rw-r--r-- 1 sar 28 Oct 19 21:33 db4.dat  
-rw-r--r-- 1 sar 72 Oct 19 21:33 db4.idx  
$ cat db4.idx  
 0 53 35 0  
 0 10Alpha:0:6  
 0 10beta:6:14  
 17 11gamma:20:8  
$ cat db4.dat  
data1  
Data for beta  
record3
```

Чтобы уменьшить объем примера, мы ограничили размеры полей указателей четырьмя символами ASCII, а количество цепочек в таблице хешей — тремя. Так как размер каждого указателя, являющегося смещением от начала файла, ограничен четырьмя символами, общий размер индексного файла и файла с данными не может превышать 10 000 байт. В разделе 20.9, измеряя производительность базы данных, мы установим размер каждого указателя равным шести символам (что позволит увеличить размер файлов до 1 000 000 байт), а количество цепочек в таблице хешей больше 100.

Первая строка в индексном файле

```
0 53 35 0
```

содержит указатель на первую запись в списке свободных записей (0 — список пуст) и три указателя на цепочки для каждого из хешей: 53, 35 и 0. Следующая строка

```
0 10Alpha:0:6
```

демонстрирует формат каждой индексной записи. Первое поле (0) — это четырехсимвольный указатель на следующую запись в цепочке. Данная запись является последней в цепочке. Следующее поле (10) — это длина индексной записи в 4-символьном формате. Каждая индексная запись читается в два приема: первая операция чтения возвращает два поля фиксированной длины (*указатель на следующую запись и длина индексной записи*), а вторая — остальную часть записи (переменной длины). Оставшиеся три поля в индексной записи — *ключ, указатель на запись с данными и длина записи с данными* — отделяются друг от друга символом-разделителем (в данном случае двоеточием). Символ-разделитель необходим, потому что каждое поле имеет переменную длину. Вследствие этого символ-разделитель не может входить в состав ключа. Завершает индексную запись символ перевода строки. Строго говоря, символ перевода строки не требуется, поскольку поле *длина индексной записи* содержит длину записи. Но мы

вставляем символ перевода строки для отделения одной индексной записи от другой, благодаря чему можем просматривать содержимое индексного файла стандартными утилитами, такими как `cat` или `more`. Поле `ключ` содержит значение, которое мы задали при добавлении записи в базу данных. Поля `указатель на запись с данными` (0) и `длина записи с данными` (6) относятся к файлу с данными. Эта информация говорит о том, что запись с данными расположена в самом начале файла и имеет длину 6 байт.

Листинг 20.1. Создает базу данных и записывает в нее три записи

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
                      FILE_MODE)) == NULL)
        err_sys("ошибка вызова функции db_open");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении первой записи");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении второй записи ");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("ошибка функции db_store при добавлении третьей записи ");

    db_close(db);
    exit(0);
}
```

(Как и в случае с индексами, мы автоматически добавляем символ перевода строки в конец каждой записи с данными, чтобы файл данных можно было просматривать с помощью стандартных утилит. Этот заключительный символ перевода строки не будет возвращаться функцией `db_fetch` вызывающему процессу.)

Если мы пройдемся по трем цепочкам хешей в данном примере, то увидим, что первая запись в первой цепочке имеет смещение 53 (`gamma`). Следующая запись в этой цепочке начинается со смещения 17 (`alpha`) и является последней записью в цепочке. Первая запись во второй цепочке начинается со смещения 35 (`beta`) и является последней записью в цепочке. Третья цепочка пустая.

Обратите внимание, что порядок ключей в индексном файле и порядок соответствующих им записей в файле с данными совпадают с порядком вызовов функции `db_store` из листинга 20.1. Так как при вызове функции `db_open` был указан флаг `O_TRUNC`, размеры индексного файла и файла с данными будут усечены и база данных заново инициализирована. В такой ситуации `db_store` просто добавляет новые индексные записи и записи с данными в конец соответствующего файла. Позже мы увидим, как `db_store` может повторно использовать участки в файлах, соответствующие удаленным записям.

Выбор алгоритма поиска на основе хеш-таблицы фиксированного размера — простое компромиссное решение. Этот алгоритм дает высокую скорость поиска, если размеры цепочек невелики. Нам необходима высокая скорость поиска, но мы не хотим усложнять структуры данных, используя алгоритмы поиска на основе двоичных деревьев или динамического хеширования. Динамическое хеширование дает возможность отыскать любую запись всего за два обращения к диску (подробности см. в [Litwin, 1980] или [Fagin et al., 1979]). Двоичные деревья дают возможность обхода базы данных в (отсортированном) порядке следования ключей (что невозможно сделать в функции `db_nextrec`, используя таблицу хешей).

20.5. Централизация или децентрализация?

Учитывая возможность обращения к одной базе данных сразу нескольких процессов, мы можем реализовать функции двумя способами.

1. Централизованный. Доступ к базе данных осуществляется посредством выделенного процесса диспетчера — единственного процесса, который напрямую обращается к базе данных. Взаимодействие с центральным процессом происходит с применением механизмов IPC.
2. Децентрализованный. Каждая функция доступа к базе данных должна сначала применить необходимые средства управления одновременным доступом (захват блокировки) и затем производить операции ввода/вывода.

Системы управления базами данных могут быть построены по любой из этих схем. При использовании эффективных алгоритмов наложения блокировок децентрализованные реализации, как правило, дают более высокую производительность, так как в них не задействованы механизмы IPC. На рис. 20.2 показана схема реализации базы данных на основе централизованного подхода.

Мы намеренно поместили в ядро блок, отображающий механизмы межпроцессных взаимодействий, потому что в большинстве случаев передача сообщений в UNIX производится именно таким образом. (Механизм разделяемой памяти, описанный в разделе 15.9, позволяет избежать копирования данных в пространство ядра.) Как видите, при использовании централизованной схемы запись читается центральным процессом и затем передается запрашивающему процессу через механизм IPC. Это основной недостаток централизованной схемы. Обратите внимание: фактический доступ к файлам базы данных осуществляется только центральный процесс.

Но централизованный подход имеет и преимущества — он позволяет точнее настраивать порядок взаимодействий с клиентскими процессами. Например, при использовании централизованной схемы процессам могут назначаться разные приоритеты. Они могут учитываться при планировании операций ввода/вывода центральным процессом. При использовании децентрализованной схемы сделать это намного сложнее. В этом случае все обычно зависит от того, как ядро планирует дисковые операции ввода/вывода: если, например, три процесса

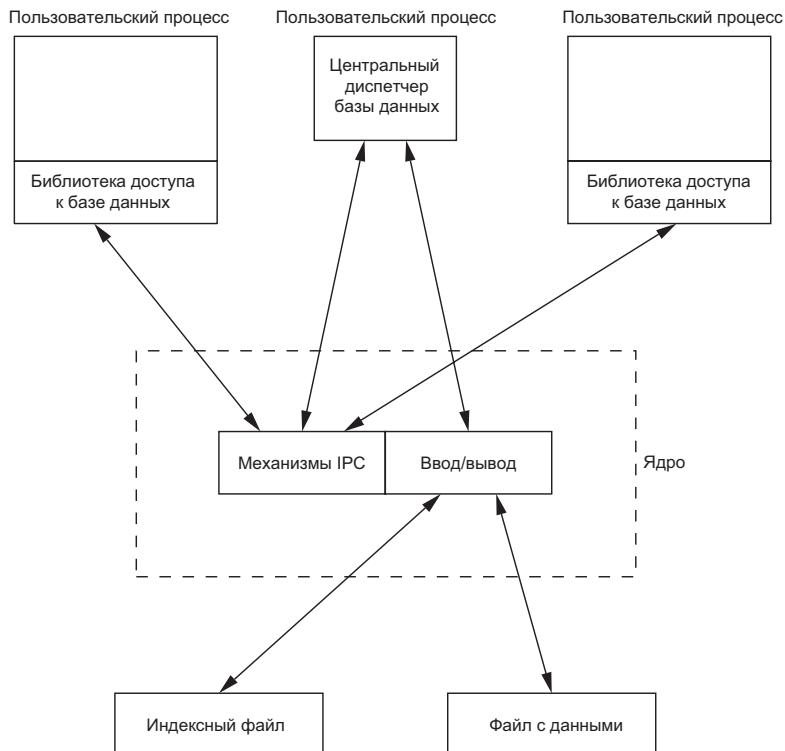


Рис. 20.2. Централизованный доступ к базе данных

ожидают снятия блокировки, какой из них первым сможет установить свою блокировку?

Еще одно преимущество централизованного подхода заключается в том, что восстановление после ошибок производится проще, чем при использовании децентрализованной схемы. При централизованном подходе вся информация о состоянии базы данных находится в одном месте, поэтому если процесс базы данных завершится аварийно, мы легко найдем информацию о незавершенных транзакциях, которая необходима, чтобы привести базу данных в непротиворечивое состояние.

На рис. 20.3 показана схема реализации базы данных на основе децентрализованного подхода. Именно эту схему мы реализуем в данной главе.

Пользовательские процессы, вызывающие функции из библиотеки базы данных для выполнения операций ввода/вывода, рассматриваются как сотрудничающие процессы, так как они используют механизм блокировки записей в файле для обеспечения одновременного доступа.

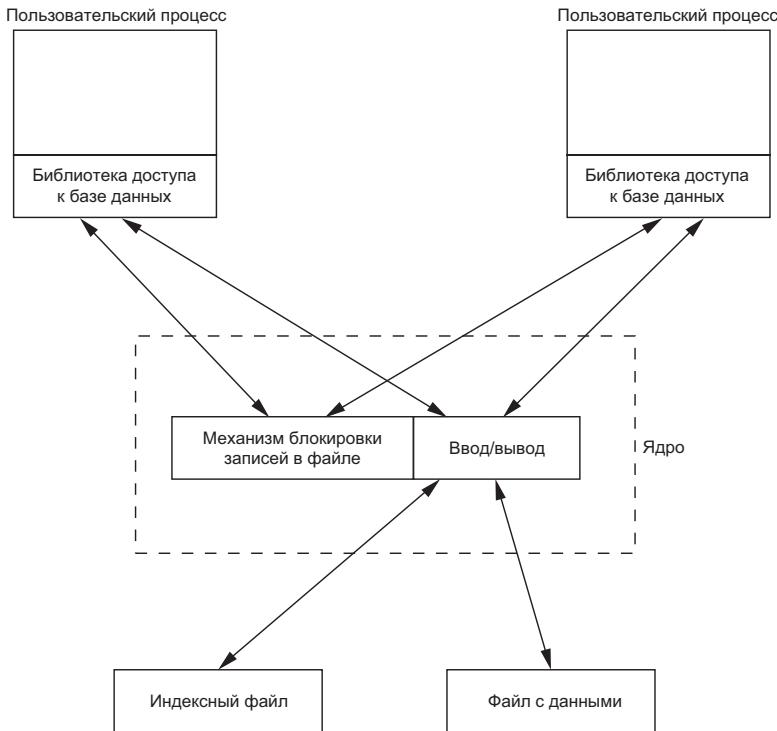


Рис. 20.3. Децентрализованный доступ к базе данных

20.6. Одновременный доступ

Мы специально выбрали реализацию базы данных на основе двух файлов (индексный файл и файл с данными), потому что это наиболее общий подход. Он требует наложения блокировок на оба файла. Но способов организовать такое наложение блокировок — множество.

Крупноблоччная блокировка

Самый простой вариант — использовать в качестве блокировки один из файлов базы данных и требовать, чтобы вызывающий процесс получил эту блокировку перед выполнением операций с базой данных. Мы назвали этот способ *крупноблоччной блокировкой* (coarse-grained locking). Например, можно сказать, что процесс, получивший блокировку для чтения нулевого байта индексного файла, обладает правом на чтение всей базы данных. Процесс, получивший блокировку

для записи в нулевой байт индексного файла, обладает правом на запись для всей базы данных. Мы можем использовать обычную семантику блокировок диапазонов байтов в UNIX, чтобы разрешить одновременное чтение данных сразу нескольким процессам, но запись — только одному процессу (табл. 14.2). При таком подходе функции `db_fetch` и `db_nextrec` должны приобретать блокировку для чтения, а функции `db_delete`, `db_store` и `db_open` — блокировку для записи. (Блокировка для записи в функции `db_open` нужна потому, что при создании индексного файла в него должен быть записан список пустых записей и хеш-таблица.)

Но крупноблочная блокировка ограничивает возможность одновременного доступа. Если один процесс добавляет запись в какую-либо цепочку хеш-таблицы, другой должен иметь возможность читать записи из другой цепочки.

Мелкоблочная блокировка

Чтобы добиться более высокой производительности, мы будем использовать блокировку, которую назвали *мелкоблочной блокировкой* (fine-grained locking). Прежде всего, читающий или пишущий процесс должен получить блокировку для чтения или для записи на цепочку, в которой находится заданная запись. Мы допускаем возможность одновременного чтения цепочки несколькими процессами, но запись в цепочку может выполнять только один процесс. Кроме того, процесс, которому требуется получить доступ для записи к списку свободных записей (функции `db_delete` и `db_store`), должен получить блокировку на список свободных записей. И наконец, всякий раз, когда производится добавление новой записи в конец индексного файла или в конец файла с данными, функция `db_store` должна получить блокировку для записи в этот участок файла.

Мы полагаем, что мелкоблочная блокировка даст выигрыш в производительности по сравнению с крупноблочной. В разделе 20.9 будут приведены результаты измерения производительности. В разделе 20.8 мы продемонстрируем исходный код нашей библиотеки на основе мелкоблочной блокировки и обсудим подробности реализации блокировок. (Крупноблочная блокировка, по сути, является упрощенным ее вариантом.)

В исходном коде библиотеки вместо функций стандартной библиотеки ввода/вывода мы будем использовать функции `read`, `write`, `readv` и `writev`. Блокировку диапазона байтов можно использовать совместно со стандартной библиотекой ввода/вывода, но при этом необходимо учитывать особенности буферизации. Нам на-верняка не нужно, чтобы `fgets` возвращала данные, прочитанные в стандартный буфер ввода/вывода 10 минут назад, если 5 минут назад запись была обновлена другим процессом.

Наше обсуждение проблемы одновременного доступа основывается на упрощенных требованиях к библиотеке базы данных. Коммерческие системы зачастую предъявляют дополнительные требования. Подробнее вопросы организации одновременного доступа обсуждаются в главе 16 в книге [Date, 2004].

20.7. Сборка библиотеки

Библиотека базы данных состоит из двух файлов: общедоступного заголовочного файла и файла с исходным кодом на С. Собрать статическую версию библиотеки можно командами:

```
gcc -I../include -Wall -c db.c  
ar rsv libapue_db.a db.o
```

Приложение, которое будет связано с библиотекой `libapue_db.a`, также должно быть связано с библиотекой `libapue.a`, поскольку мы использовали некоторые функции из нее в библиотеке базы данных.

С другой стороны, если потребуется динамическая версия библиотеки базы данных, ее можно собрать следующими командами:

```
gcc -I../include -Wall -fPIC -c db.c  
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \  
-L../lib -lapue -lc db.o
```

Полученный файл библиотеки `libapue_db.so.1` должен быть помещен в каталог, где динамический загрузчик/редактор связей сможет отыскать его. Как вариант, можно поместить файл библиотеки в произвольный каталог и изменить содержимое переменной окружения `LD_LIBRARY_PATH` так, чтобы она включала в себя путь к этой библиотеке.

Команды компиляции и сборки динамических библиотек могут различаться на разных платформах. Здесь мы привели команды для Linux, оснащенной компилятором GNU C compiler.

20.8. Исходный код

Демонстрацию исходного кода библиотеки мы начнем с заголовочного файла `apue_db.h`. Этот файл подключается библиотекой и всеми приложениями, которые к ней обращаются.

В этом обсуждении мы отступим от правил, которым следовали в предыдущих разделах. Во-первых, поскольку объем исходного кода значительно больше, чем обычно, мы будем нумеровать строки. Это позволит привязать обсуждение к конкретным участкам исходного кода. Во-вторых, мы будем помещать описание сразу же вслед за фрагментами кода, к которым оно относится.

Такой стиль был использован Джоном Лайонсом в его книге, описывающей исходный код UNIX Version 6 [Lions, 1977, 1996]. Это упрощает исследование больших объемов кода.

Обратите внимание: мы не стали нумеровать пустые строки. Хотя это не соответствует поведению таких утилит, как `rg(1)`, но нам нечего сказать о пустых строках.

```

1 #ifndef _APUE_DB_H
2 #define _APUE_DB_H

3 typedef void * DBHANDLE;

4 DBHANDLE db_open(const char *, int, ...);
5 void db_close(DBHANDLE);
6 char *db_fetch(DBHANDLE, const char *);
7 int db_store(DBHANDLE, const char *, const char *, int);
8 int db_delete(DBHANDLE, const char *);
9 void db_rewind(DBHANDLE);
10 char *db_nextrec(DBHANDLE, char *);

11 /*
12  * Флаги для функции db_store().
13 */
14 #define DB_INSERT 1 /* вставить новую запись */
15 #define DB_REPLACE 2 /* заменить существующую запись */
16 #define DB_STORE 3 /* заменить или вставить */

17 /*
18  * Ограничения реализации.
19 */
20 #define IDXLEN_MIN 6 /* ключ, разделитель, смещение, разделитель, длина, \n */
21 #define IDXLEN_MAX 1024 /* выбрано произвольно */
22 #define DATLEN_MIN 2 /* байт данных, перевод строки */
23 #define DATLEN_MAX 1024 /* выбрано произвольно */

24 #endif /* _APUE_DB_H */

```

[1–3] Использование символа `_APUE_DB_H` гарантирует, что содержимое данного заголовочного файла будет подключено только один раз. Тип `DBHANDLE` представляет активную ссылку на базу данных и используется для скрытия внутренних особенностей реализации базы данных от приложений. Сравните это с использованием структуры `FILE` при работе со стандартной библиотекой ввода/вывода.

[4–10] Здесь объявляются прототипы общедоступных библиотечных функций. Поскольку этот файл подключается из приложений, которые желают использовать библиотеку в своей работе, мы не должны объявлять здесь частные функции библиотеки.

[11–24] Здесь объявлены флаги, которые могут передаваться функции `db_store`. Вслед за флагами объявляются фундаментальные ограничения реализации. Значения этих ограничений можно изменить, если необходима поддержка баз данных большего объема.

Минимальная длина индексной записи определяется константой `IDXLEN_MIN`. Значение этой константы складывается из 1 байта ключа, 1 байта символа-разделителя, 1 байта смещения, еще 1 байта символа-разделителя, 1 байта длины и завершающего символа перевода строки. (Формат индексной записи был показан на рис. 20.1.) Обычно индексные записи будут превышать `IDXLEN_MIN` байт, но в данном случае мы определяем минимально возможный размер.

Следующий файл — db.c — содержит исходный код библиотеки на языке С. Для простоты мы включили все функции в один файл. Такая организация имеет свои преимущества, поскольку мы можем скрыть частные функции, объявив их со спецификатором `static`.

```

1 #include "apue.h"
2 #include "apue_db.h"
3 #include <fcntl.h> /* флаги для функций open и db_open */
4 #include <stdarg.h>
5 #include <errno.h>
6 #include <sys/uio.h> /* struct iovec */

7 /*
8  * Внутренние константы, имеющие отношение к индексному файлу.
9  * Они используются при создании записей в индексном
10 * файле и в файле с данными.
11 */
12 #define IDXLEN_SZ 4 /* размер поля длины индексной записи */
13 #define SEP      ':' /* символ-разделитель полей в индексной записи */
14 #define SPACE    ' ' /* символ пробела */
15 #define NEWLINE  '\n' /* символ перевода строки */

16 /*
17 * Следующие определения необходимы для работы с цепочками
18 * в хеш-таблице и в списке свободных записей в индексном файле.
19 */
20 #define PTR_SZ      7 /* размер поля-указателя в цепочке */
21 #define PTR_MAX     999999 /* максимальное смещение */
22 #define NHASH_DEF   137 /* размер хеш-таблицы по умолчанию */
23 #define FREE_OFF    0 /* начало списка свободных записей */
24 #define HASH_OFF    PTR_SZ /* начало хеш-таблицы в индексном файле */

25 typedef unsigned long DBHASH; /* значения хешей */
26 typedef unsigned long COUNT; /* беззнаковый счетчик */

```

[1–6] Здесь мы подключили apue.h, потому что библиотека базы данных использует некоторые функции из нашей частной библиотеки. В свою очередь, apue.h подключает ряд стандартных заголовочных файлов, среди которых <stdio.h> и <unistd.h>. Заголовочный файл <stdarg.h> необходим потому, что в нем находятся определения функций для работы со списками аргументов переменной длины, которые используются в функции db_open.

[7–26] Размер поля, хранящего длину индексной записи, определяется константой IDXLEN_SZ. Далее следуют определения некоторых символов, таких как двоеточие и перевод строки, которые будут использоваться в качестве символов-разделителей. Символ пробела используется для «затирания» удаленных записей.

Некоторые значения, определенные как константы, можно оформить в виде переменных — при небольшом усложнении реализации. Например, размер хеш-таблицы мы ограничили 137 записями. Наверное, лучше было бы позволить вызывающему процессу задавать это значение в виде аргумента функции db_open, основываясь на предполагаемом размере базы данных. Это значение затем можно было сохранять в начале индексного файла.

```

27 /*
28 * Внутреннее представление базы данных в библиотеке.
29 */
30 typedef struct {
31     int      idxfd; /* дескриптор индексного файла */
32     int      datfd; /* дескриптор файла с данными */
33     char    *idxbuf; /* адрес буфера для индексной записи */
34     char    *datbuf; /* адрес буфера для записи с данными */
35     char    *name;   /* имя базы данных, под которым она была открыта */
36     off_t   idxoff; /* смещение индексной записи в индексном файле, ключ */
37             /* начинается с позиции (idxoff + PTR_SZ + IDXLEN_SZ) */
38     size_t  idxlen; /* длина индексной записи */
39             /* кроме IDXLEN_SZ байт в начале записи */
40             /* включая символ перевода строки в конце записи */
41     off_t   datoff; /* смещение записи с данными в файле данных */
42     size_t  datlen; /* длина записи с данными */
43             /* включая символ перевода строки в конце записи */
44     off_t   ptrval; /* содержимое указателя на цепочку в индексной записи */
45     off_t   ptroff; /* смещение указателя, содержащего адрес этой записи */
46     off_t   chainoff; /* смещение цепочки для этой индексной записи */
47     off_t   hashoff; /* смещение хеш-таблицы в индексном файле */
48     DBHASH nhash;   /* текущий размер хеш-таблицы */
49     COUNT   cnt_delok; /* счетчик удачных операций удаления */
50     COUNT   cnt_delerr; /* счетчик ошибочных операций удаления */
51     COUNT   cnt_fetchok; /* счетчик удачных операций извлечения данных */
52     COUNT   cnt_fetcherr; /* счетчик ошибочных операций извлечения данных */
53     COUNT   cnt_nextrc; /* nextrec */
54     COUNT   cnt_stor1; /* store: DB_INSERT, нет пустых записей, добавить */
55     COUNT   cnt_stor2; /* store: DB_INSERT, есть пустые записи,
56                         /* использовать их */
56     COUNT   cnt_stor3; /* store: DB_REPLACE, другая длина, добавить */
57     COUNT   cnt_stor4; /* store: DB_REPLACE, та же длина, перезаписать */
58     COUNT   cnt_storerr; /* счетчик ошибок добавления записи */
59 } DB;

```

[27–48] В структуре DB будет храниться вся информация о каждой из открытых баз данных. Значение DBHANDLE, которое возвращается функцией db_open и используется всеми остальными функциями, в действительности представляет собой указатель на одну из этих структур, но это обстоятельство скрыто от вызывающего процесса.

Поскольку все указатели и размеры в базе данных хранятся в виде строк символов ASCII, мы будем преобразовывать их в числовые значения и сохранять в структуре. Кроме того, в структуре будет сохраняться размер хеш-таблицы, даже несмотря на то, что она имеет фиксированный размер. Сделано это на тот случай, если мы задумаем модернизировать библиотеку и разрешить вызывающему процессу определять размер хеш-таблицы на этапе создания базы данных (упражнение 20.7).

[49–59] Последние десять полей структуры DB — это счетчики операций, завершившихся успехом или неудачей. Если понадобится проанализировать производительность базы данных, мы сможем написать функцию, которая будет возвращать эти статистические характеристики, но пока мы только позаботимся о существовании самих счетчиков и их заполнении.

```

60 /*
61 * Внутренние функции.
62 */
63 static DB      *_db_alloc (int);
64 static void    _db_dodelete(DB *);
65 static int     _db_find_and_lock(DB *, const char *, int);
66 static int     _db_findfree(DB *, int, int);
67 static void    _db_free(DB *);
68 static DBHASH  _db_hash(DB *, const char *);
69 static char   * _db_readdat(DB *);
70 static off_t   _db_readidx(DB *, off_t);
71 static off_t   _db_readptr(DB *, off_t);
72 static void    _db_writedat(DB *, const char *, off_t, int);
73 static void    _db_writeidx(DB *, const char *, off_t, int, off_t);
74 static void    _db_writeptr(DB *, off_t, off_t);

75 /*
76 * Открывает или создает базу данных. Аргументы аналогичны функции open(2).
77 */
78 DBHANDLE
79 db_open(const char *pathname, int oflag, ...)
80 {
81     DB          *db;
82     int          len, mode;
83     size_t       i;
84     char        asciiipr[PTR_SZ + 1],
85                 hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                 /* +2 для символа перевода строки и нулевого символа */
87     struct stat  statbuff;

88 /*
89 * Разместить в памяти структуру DB и все необходимые буферы.
90 */
91     len = strlen(pathname);
92     if ((db = _db_alloc(len)) == NULL)
93         err_dump("db_open: ошибка_размещения структуры DB");

```

[60–74] Мы выбрали следующий порядок именования функций: имена всех общедоступных функций начинаются с префикса `db_`, а всех частных функций — с префикса `_db_`. Прототипы общедоступных функций объявлены в заголовочном файле библиотеки `arie_db.h`. Все частные функции объявлены со спецификатором `static`, благодаря чему они доступны только функциям, размещенным в том же файле (с реализацией библиотеки).

[75–93] Функция `db_open` принимает те же аргументы, что и функция `open(2)`. Если вызывающий процесс предполагает создание новых файлов базы данных, в третьем аргументе он должен определить права доступа к создаваемым файлам. Функция `db_open` открывает индексный файл и файл с данными и при необходимости инициализирует содержимое индексного файла. Начинается эта функция с вызова функции `_db_alloc`, которая размещает в памяти и инициализирует структуру `DB`.

```

94     db->nhash    = NHASH_DEF; /* размер таблицы хешей */
95     db->hashoff = HASH_OFF; /* начало хеш-таблицы в индексном файле */
96     strcpy(db->name, pathname);
97     strcat(db->name, ".idx");

98     if (oflag & O_CREAT) {
99         va_list ap;

100        va_start(ap, oflag);
101        mode = va_arg(ap, int);
102        va_end(ap);

103        /*
104         * Создать индексный файл и файл с данными.
105         */
106        db->idxfd = open(db->name, oflag, mode);
107        strcpy(db->name + len, ".dat");
108        db->datfd = open(db->name, oflag, mode);
109    } else {
110        /*
111         * Открыть индексный файл и файл с данными.
112         */
113        db->idxfd = open(db->name, oflag);
114        strcpy(db->name + len, ".dat");
115        db->datfd = open(db->name, oflag);
116    }

117    if (db->idxfd < 0 || db->datfd < 0) {
118        _db_free(db);
119        return(NULL);
120    }

```

[94–97] Продолжение инициализации структуры DB. Имя базы данных, которое передает вызывающий процесс, используется как префикс для имен файлов базы данных. Чтобы получить имя индексного файла, к полученному префикску добавляется расширение .idx.

[98–108] Если вызывающий процесс желает создать новые файлы базы данных, тогда, чтобы получить значение третьего аргумента, используются функции для работы со списками аргументов переменной длины из заголовочного файла `<stdarg.h>`. После этого вызовом `open` создаются и открываются файлы базы данных. Обратите внимание, что имя файла с данными начинается с того же префикса, что и имя индексного файла, но, в отличие от последнего, имеет расширение .dat.

[109–116] Если вызывающий процесс не указал флаг `O_CREAT`, открываются существующие файлы базы данных. В этом случае мы просто вызываем `open` с двумя аргументами.

[117–120] Если в процессе открытия файлов возникла ошибка, вызывается функция `_db_free`, чтобы освободить память, занимаемую структурой DB, и вызывающему процессу возвращается значение `NULL`. Если так случилось, что один файл был благополучно открыт, а второй нет, функция `_db_free` позаботится о закрытии открытого дескриптора файла, в чём мы вскоре убедимся.

```

121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123         * Если создана новая база данных, ее нужно инициализировать.
124         * Блокировка для записи всего файла обеспечит атомарность
125         * операции получения его характеристик и инициализации.
126         */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: ошибка вызова функции writew_lock");
129
130         if (fstat(db->idxfd, &statbuff) < 0)
131             err_sys("db_open: ошибка вызова функции fstat");
132
133         if (statbuff.st_size == 0) {
134             /*
135             * Создать список из (NHASH_DEF + 1) указателей на цепочки с
136             * нулевыми значениями. В данном случае +1 - это место для
137             * указателя на список свободных записей перед таблицей.
138             */
139             sprintf(asciiptr, "%*d", PTR_SZ, 0);

```

[121–130] При создании базы данных мы сталкиваемся с необходимостью наложения блокировки. Рассмотрим два процесса, которые одновременно пытаются создать базу данных с одним и тем же именем. Пусть первый процесс вызвал функцию `fstat` и был приостановлен ядром после того, как `fstat` вернула управление. Второй процесс также вызвал `db_open`, обнаружил, что индексный файл имеет нулевую длину, и инициализировал цепочки хешей и список свободных записей. Затем второй процесс записал одну запись в базу. В этот момент ядро приостановило второй процесс и передало управление первому процессу, который продолжил выполнение функции `db_open` сразу после вызова `fstat`. Первый процесс обнаруживает, что индексный файл имеет нулевой размер (поскольку вызов `fstat` произошел еще до того, как второй процесс инициализировал индексный файл), и повторно инициализирует цепочки хешей и список свободных записей, стирая то, что записал в базу данных второй процесс. Чтобы предотвратить подобное развитие событий, необходимо применять блокировки. Здесь мы используем макросы `readw_lock`, `writew_lock` и `unlock` из раздела 14.3.

[131–137] Если индексный файл имеет нулевой размер, следовательно, он был только что создан и необходимо инициализировать цепочки хешей и список свободных записей. Обратите внимание, что для преобразования числа, представляющего значение указателя, в строку ASCII мы используем формат `%*d`. (Аналогичную строку формата мы будем использовать в функциях `_db_writeidx` и `_db_writeptr`). Этот формат сообщает функции `sprintf`, что она должна использовать аргумент `PTR_SZ` в качестве минимального размера ширины поля для вывода следующего аргумента, который в данном случае представлен числом 0 (мы инициализируем нулями все указатели, поскольку создается новая база данных). Это приводит к тому, что создается строка, содержащая по меньшей мере `PTR_SZ` символов (дополненная слева пробелами). В функциях `_db_writeidx` и `_db_writeptr` вместо нуля мы будем передавать фактические значения указателей и обязательно сравнивать эти значения с константой `PTR_MAX`, чтобы гарантировать, что каждый указатель, записываемый в базу данных, имеет размер точно `PTR_SZ` (7) символов.

```

138         hash[0] = 0;
139         for (i = 0; i < NHASH_DEF + 1; i++)
140             strcat(hash, asciiptr);
141             strcat(hash, "\n");
142             i = strlen(hash);
143             if (write(db->idxfd, hash, i) != i)
144                 err_dump("db_open: ошибка инициализации индексного файла");
145             }
146             if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147                 err_dump("db_open: ошибка вызова функции un_lock");
148             }
149         db_rewind(db);
150     return(db);
151 }

152 /*
153 * Размещает в памяти и инициализирует структуру DB и ее буферы.
154 */
155 static DB *
156 _db_alloc(int namelen)
157 {
158     DB      *db;

159     /*
160     * Функция calloc выделяет память и забивает ее нулями.
161     */
162     if ((db = calloc(1, sizeof(DB))) == NULL)
163         err_dump("_db_alloc: ошибка размещения структуры DB");
164     db->idxfd = db->datfd = -1; /* дескрипторы */

165     /*
166     * Выделить место для имени.
167     * +5 для ".idx" или ".dat" и нулевого байта в конце.
168     */
169     if ((db->name = malloc(namelen + 5)) == NULL)
170         err_dump("_db_alloc: ошибка выделения памяти для имени");

```

[138–151] Продолжается инициализация вновь созданной базы данных. Здесь производится построение таблицы хешей и запись ее в индексный файл. После этого снимается блокировка с индексного файла, производится переход к началу базы данных и вызывающему процессу возвращается указатель на структуру DB как своего рода дескриптор, который будет использоваться при вызове всех остальных функций библиотеки.

[152–164] Для размещения в памяти структуры DB, индексного буфера и буфера с данными функция `db_open` вызывает `_db_alloc`. Для выделения памяти под структуру DB мы используем функцию `calloc`, которая очищает выделенную память, забивая ее нулями. При этом в качестве побочного эффекта мы получаем дескрипторы файлов базы данных со значениями 0, поэтому нам необходимо записать в них число –1, чтобы указать, что дескрипторы еще не открыты.

[165–170] Далее выделяется пространство для хранения имени файла базы данных. Этот буфер будет использоваться для конструирования имен обоих файлов путем изменения расширения, как мы это видели в функции `db_open`.

```
171     /*
172      * Выделить память для индексного буфера и буфера данных.
173      * +2 для символа перевода строки и нулевого символа в конце.
174     */
175    if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176        err_dump("_db_alloc: ошибка распределения индексного буфера");
177    if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178        err_dump("_db_alloc: ошибка распределения буфера данных");
179    return(db);
180 }

181 /*
182  * Закрывает доступ к базе данных.
183 */
184 void
185 db_close(DBHANDLE h)
186 {
187     _db_free((DB *)h); /* закрывает дескрипторы, освобождает память */
188 }

189 /*
190  * Освобождает память, занимаемую структурой DB и буферами.
191  * А также закрывает дескрипторы файлов, которые могут быть открыты.
192 */
193 static void
194 _db_free(DB *db)
195 {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171–180] Выделяем память для буферов с информацией из индексного файла и файла с данными. Размеры буферов определены в заголовочном файле `apue_db.h`. При дальнейшем усовершенствовании библиотеки мы сможем увеличить эти размеры, если понадобится. При этом мы должны будем отслеживать их текущие размеры и вызывать функцию `realloc`, когда возникает необходимость в буферах большего размера. В завершение возвращаем указатель на структуру `DB`, которую только что распределили.

[181–188] Функция `db_close` — это функция-обертка, которая приводит дескриптор базы данных к типу `DB*` и передает его функции `_db_free`, чтобы освободить все занимаемые ресурсы.

[189–199] Функция `_db_free` вызывается из `db_open`, если в процессе открытия базы данных возникли ошибки, а также из `db_close`, когда приложение прекращает работу с базой данных. Если дескриптор индексного файла открыт, мы закрываем его. То же происходит с дескриптором файла данных. (Мы уже говорили ранее, что функция `_db_alloc` инициализирует дескрипторы файлов значениями `-1`. Если мы не сможем открыть какой-либо из файлов базы данных, соответствующий ему дескриптор будет иметь значение `-1` и мы не будем даже пытаться закрыть его.)

```

200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207 }

208 /*
209  * Извлекает одну запись. Возвращает указатель на строку с данными.
210 */
211 char *
212 db_fetch(DBHANDLE h, const char *key)
213 {
214     DB      *db = h;
215     char    *ptr;

216     if (_db_find_and_lock(db, key, 0) < 0) {
217         ptr = NULL; /* ошибка, запись не найдена */
218         db->cnt_fetcherr++;
219     } else {
220         ptr = _db_readdat(db); /* вернуть указатель на строку с данными */
221         db->cnt_fetchok++;
222     }

223     /*
224      * Снять блокировку с цепочки, установленную в _db_find_and_lock.
225     */
226     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227         err_dump("db_fetch: ошибка вызова функции un_lock");
228     return(ptr);
229 }

```

[200–207] Далее освобождаем память, занимаемую буферами, распределенными динамически. Мы можем без опаски передать функции `free` пустой указатель, поэтому нет необходимости выполнять дополнительные проверки значений каждого из указателей, но мы делаем такие проверки, потому что считаем, что лучше освобождать только те объекты, которые действительно размещены (не все функции освобождения динамической памяти так дружелюбны, как `free`). В заключение мы освобождаем память, занимаемую структурой DB.

[208–218] Функция `db_fetch` используется для извлечения записи по заданному ключу. Прежде всего мы пытаемся с помощью функции `_db_find_and_lock` найти требуемую запись. Если запись не найдена, мы записываем NULL в возвращаемое значение и увеличиваем счетчик неудачных обращений. Поскольку `_db_find_and_lock` устанавливает блокировку на индексный файл, мы не можем вернуть управление, пока не снимем ее.

[219–229] Если запись найдена, вызывается функция `_db_readdat`, которая читает данные и увеличивает счетчик удачных обращений. Перед возвратом управления мы снимаем блокировку с индексного файла вызовом функции `un_lock`. После этого возвращаем указатель на найденную запись (или NULL, если запись не найдена).

```
230 /*
231 * Ищет заданную запись. Вызывается из db_delete, db_fetch
232 * и db_store. Устанавливает блокировку на цепочку из хеш-таблицы.
233 */
234 static int
235 _db_find_and_lock(DB *db, const char *key, int writelock)
236 {
237     off_t      offset, nextoffset;

238     /*
239     * Рассчитать значение хеша для данного ключа и найти
240     * смещение соответствующей цепочки в хеш-таблице.
241     * С этого места начинается поиск. Прежде всего мы должны
242     * рассчитать смещение в хеш-таблице для данного ключа.
243     */
244     db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245     db->ptroff = db->chainoff;

246     /*
247     * Здесь устанавливается блокировка. Вызывающая функция должна снять ее.
248     * Внимание: блокировка устанавливается только на первый байт.
249     */
250     if (writelock) {
251         if (writeln_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252             err_dump("_db_find_and_lock: ошибка вызова writeln_lock");
253     } else {
254         if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255             err_dump("_db_find_and_lock: ошибка вызова readw_lock");
256     }

257     /*
258     * Получить смещение первой записи в данной цепочке от начала
259     * индексного файла (может быть 0).
260     */
261     offset = _db_readptr(db, db->ptroff);
```

[230–237] Функция `_db_find_and_lock` используется библиотекой для поиска записи по заданному ключу. В аргументе `writelock` передается ненулевое значение, если на время поиска необходимо установить блокировку для записи. Чтобы на время поиска установить блокировку для чтения, в аргументе `writelock` передается 0.

[238–256] В функции `_db_find_and_lock` производится подготовка к обходу цепочки. Ключ преобразуется в значение хеша, которое используется для вычисления смещения цепочки от начала файла (`chainoff`). Прежде чем приступить к поиску по цепочке, мы ждем, пока установится блокировка. Обратите внимание: блокировка устанавливается только на первый байт цепочки. Это позволяет нескольким процессам одновременно производить поиск по разным цепочкам.

[257–261] Чтобы получить первый указатель из цепочки, мы вызываем функцию `_db_readptr`. Если она возвращает 0, значит, цепочка пуста.

```

262     while (offset != 0) {
263         nextoffset = _db_readidx(db, offset);
264         if (strcmp(db->idxbuf, key) == 0)
265             break; /* найдено совпадение */
266         db->ptroff = offset; /* смещение данной записи */
267         offset = nextoffset; /* переход к следующей записи */
268     }
269     /*
270      * offset == 0 означает ошибку (запись не найдена).
271      */
272     return(offset == 0 ? -1 : 0);
273 }

274 /*
275  * Вычисляет значение хеша по ключу.
276  */
277 static DBHASH
278 _db_hash(DB *db, const char *key)
279 {
280     DBHASH hval = 0;
281     char c;
282     int i;

283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i; /* произведение ASCII-кода символа и его индекса */
285     return(hval % db->nhash);
286 }

```

[262–268] В цикле `while` производится обход всех индексных записей в цепочке и выполняется сравнение ключей. Чтение записей выполняется функцией `_db_readidx`. Она заполняет буфер `idxbuf` строкой ключа из текущей записи. Если функция `_db_readidx` возвращает 0, мы достигли конца цепочки.

[269–273] Если после выхода из цикла в переменной `offset` содержится 0, это значит, что мы добрались до конца цепочки, но искомую запись не нашли, поэтому возвращается значение `-1`. Иначе совпадение найдено (выполнение цикла `while` прервано оператором `break`), и возвращается признак успеха — значение `0`. В этом случае поле `ptroff` хранит адрес предыдущей индексной записи, `dataoff` — адрес записи с данными, а `datalen` — размер записи с данными. Так как при обходе цепочки мы сохраняем адрес предыдущей индексной записи, которая ссылается на текущую, мы будем использовать это обстоятельство при удалении записи, поскольку в этом случае надо изменить указатель в предыдущей записи, чтобы удалить текущую.

[274–286] Функция `_db_hash` вычисляет значение хеша по заданному ключу. Она находит сумму произведений ASCII-кодов символов на их индексы (начиная с 1) и делит результат на количество записей в хеш-таблице. Согласно [Knuth, 1998], элементарные хеш-функции обычно дают более равномерные характеристики распределения.

```

287 /*
288  * Читает поле указателя на цепочку из индексного файла:
289  * указатель на список свободных записей, на цепочку из хеш-таблицы
290  * или на индексную запись в цепочке.
291 */
292 static off_t
293 _db_readptr(DB *db, off_t offset)
294 {
295     char asciiptr[PTR_SZ + 1];
296
297     if (lseek(db->idxfd, offset, SEEK_SET) == 1)
298         err_dump("_db_readptr: ошибка перемещения на поле с указателем");
299     if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
300         err_dump("_db_readptr: ошибка чтения поля с указателем");
301     asciiptr[PTR_SZ] = 0; /* завершающий нулевой символ */
302     return(atol(asciiptr));
303 }
304
305  * Читает следующую индексную запись, начиная с указанного смещения
306  * в индексном файле. Индексная запись считывается в буфер db->idxbuf,
307  * а символы-разделители замещаются нулевыми байтами. Если все в порядке,
308  * в db->datoff и db->datlen записываются смещение и длина
309  * соответствующей записи из файла с данными.
310 */
311 static off_t
312 _db_readidx(DB *db, off_t offset)
313 {
314     ssize_t      i;
315     char        *ptr1, *ptr2;
316     char        asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
317     struct iovec  iov[2];

```

[287–302] Функция `_db_readptr` читает один из трех возможных указателей: (а) указатель на первую запись из списка свободных индексных записей, (б) указатель в хеш-таблице, ссылающийся на первую запись в цепочке или (в) указатель, который хранится в начале каждой индексной записи (неважно, является ли эта запись частью цепочки или частью списка свободных записей). Перед возвратом значение указателя преобразуется из ASCII-представления в длинное целое. Функция `_db_readptr` не устанавливает никаких блокировок — это должно выполняться в вызывающей функции.

[303–316] Функция `_db_readidx` используется для чтения индексной записи с заданным смещением из индексного файла. В случае успеха функция возвращает смещение очередной записи в списке и заполняет некоторые поля структуры DB: `idxoff` — смещение текущей записи в индексном файле, `ptrval` — смещение следующей записи в списке, `idxlen` — длина текущей индексной записи, `idxbuf` — сама индексная запись, `dataoff` — смещение записи в файле с данными и `datalen` — длина записи с данными.

```

317     /*
318      * Позиция в файле и смещение записи. db_nextrec вызывает
319      * эту функцию с offset==0, что означает чтение из текущей позиции.
320      * Мы все равно должны вызвать lseek, чтобы прочитать запись.
321      */
322     if ((db->idxoff = lseek(db->idxfd, offset,
323                               offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readididx: ошибка вызова функции lseek");

325     /*
326      * Прочитать длину записи и указатель на следующую запись
327      * в начале текущей индексной записи. Это позволит нам
328      * прочитать оставшуюся часть индексной записи.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len = PTR_SZ;
332     iov[1].iov_base = ascilen;
333     iov[1].iov_len = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1); /* признак конца файла для db_nextrec */
337         err_dump("_db_readididx: ошибка readv при чтении индексной записи");
338     }

339     /*
340      * Это возвращаемое значение, всегда >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;          /* завершающий нулевой символ */
343     db->ptrval = atol(asciiptr); /* смещение следующей записи в цепочке */

344     ascilen[IDXLEN_SZ] = 0;        /* завершающий нулевой символ */
345     if ((db->idxlen = atoi(ascilen)) < IDXLEN_MIN ||
346         db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readididx: неверная длина записи");

```

[317–324] Начинаем с установки позиции в индексном файле, полученной от вызывающей функции. Смещение записывается в структуру DB, поэтому даже если вызывающая функция предполагает чтение из текущей позиции в файле (передавая 0 в аргументе `offset`), мы все равно должны вызвать `lseek`, чтобы определить эту позицию. Поскольку ни одна индексная запись не хранится со смещением 0, мы можем определить для этого значения специальный смысл — «прочитать запись из текущей позиции».

[325–338] С помощью функции `readv` из начала текущей записи производится чтение двух полей фиксированной длины: указателя на следующую запись в цепочке и размера текущей индексной записи.

[339–347] Смещение следующей записи преобразуется в целое число и запоминается в его поле `ptrval` (оно будет использовано как возвращаемое значение этой функции). Затем выполняется аналогичное преобразование длины текущей записи в целое число, которое сохраняется в поле `idxlen`.

```

348     /*
349      * Теперь будет прочитана сама запись. Мы прочитаем ее в индексный
350      * буфер, который был распределен при открытии базы данных.
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: ошибка чтения индексной записи");
354     if (db->idxbuf[db->idxlen1] != NEWLINE) /* проверка целостности */
355         err_dump("_db_readidx: отсутствует символ перевода строки");
356     db->idxbuf[db->idxlen-1] = 0; /* заменить NL нулевым символом */

357     /*
358      * Найти символы-разделители в индексной записи.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: отсутствует первый разделитель");
362     *ptr1++ = 0; /* заменить SEP нулевым символом */

363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: отсутствует второй разделитель");
365     *ptr2++ = 0; /* заменить SEP нулевым символом */

366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: слишком много символов-разделителей");

368     /*
369      * Получить смещение и длину записи с данными.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: смещение записи с данными < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: неверная длина записи с данными");
375     return(db->ptrval); /* вернуть позицию следующей индексной записи */
376 }

```

[348–356] Читаем индексную запись в поле `idxbuf` структуры `DB`. Запись должна заканчиваться символом перевода строки, который мы заменяем нулевым символом. Если индексный файл поврежден, мы завершаем работу процесса с созданием файла `coge`, вызвав функцию `err_dump`.

[357–367] Делим индексную запись на три поля: ключ, позиция соответствующей записи с данными и длина записи с данными. Функция `strchr` отыскивает первое вхождение данного символа в заданной строке. В данной ситуации нас интересуют символы-разделители (константа `SEP`, которая определена как символ двоеточия).

[368–376] Позиция записи с данными и ее длина преобразуются в числовое представление и запоминаются в структуре `DB`. Затем мы возвращаем позицию следующей индексной записи в цепочке. Обратите внимание: мы не читаем запись с данными. Это будет сделано в вызывающей функции — например, в `db_fetch`. Мы не читаем записи с данными, пока функция `_db_find_and_lock` не прочитает индексную запись, совпадающую с искомой.

```

377 /*
378 * Читает текущую запись с данными в буфер.
379 * Возвращает указатель на буфер со строкой, завершающейся нулевым символом.
380 */
381 static char *
382 _db_readdat(DB *db)
383 {
384     if (lseek(db->datfd, db->dataoff, SEEK_SET) == -1)
385         err_dump("_db_readdat: ошибка вызова функции lseek");
386     if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387         err_dump("_db_readdat: ошибка вызова функции read");
388     if (db->datbuf[db->datlen-1] != NEWLINE) /* проверка целостности */
389         err_dump("_db_readdat: отсутствует символ перевода строки");
390     db->datbuf[db->datlen-1] = 0; /* заменить NL нулевым символом */
391     return(db->datbuf); /* вернуть указатель на запись с данными */
392 }

393 /*
394 * Удаляет заданную запись.
395 */
396 int
397 db_delete(DBHANDLE h, const char *key)
398 {
399     DB      *db = h;
400     int      rc = 0; /* предполагается, что запись будет найдена */
401
402     if (_db_find_and_lock(db, key, 1) == 0) {
403         _db_dodelete(db);
404         db->cnt_delok++;
405     } else {
406         rc = -1; /* не найдена */
407         db->cnt_delerr++;
408     }
409     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
410         err_dump("db_delete: ошибка вызова функции un_lock");
411 }

```

[377–392] Функция `_db_readdat` заполняет поле `datbuf` структуры `DB` содержимым записи с данными, предполагая, что в поля `dataoff` и `datlen` предварительно были записаны корректные значения.

[393–411] Функция `db_delete` используется для удаления записи по заданному ключу. С помощью функции `_db_find_and_lock` выполняется поиск требуемой записи в базе данных и в случае успеха вызывает функцию `_db_dodelete`, которая выполняет все необходимые действия. Третий аргумент функции `_db_find_and_lock` определяет характер блокировки — для чтения или для записи. Здесь мы запрашиваем установку блокировки для записи, поскольку, вероятнее всего, нам потребуется внести изменения в список. Так как `_db_find_and_lock` возвращает управление с установленной блокировкой, необходимо снять ее независимо от того, была найдена запись или нет.

```
412 /*
413  * Удаляет текущую запись, заданную в структуре DB.
414  * Эта функция вызывается из db_delete и db_store после того,
415  * как запись будет найдена функцией _db_find_and_lock.
416 */
417 static void
418 _db_dodelete(DB *db)
419 {
420     int      i;
421     char    *ptr;
422     off_t   freeptr, saveptr;
423
424     /*
425      * Очистить индексный буфер и буфер с данными, забив их пробелами.
426      */
427     for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
428         *ptr++ = SPACE;
429     *ptr = 0; /* завершающий нулевой символ для _db_writedat */
430     ptr = db->idxbuf;
431     while (*ptr)
432         *ptr++ = SPACE;
433
434     /*
435      * Мы должны заблокировать список свободных записей.
436      */
437     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
438         err_dump("_db_dodelete: ошибка вызова функции writew_lock");
439
440     /*
441      * Записать очищенную запись с данными.
442      */
443     _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412–431] Функция `_db_dodelete` выполняет все необходимые действия по удалению записи из базы данных. (Эта функция также вызывается из `db_store`.) По сути, эта функция лишь обновляет два связанных списка: список свободных записей и цепочку из хеш-таблицы, в которой находился заданный ключ. При удалении записи ключ и запись с данными заполняются пробелами. Это обстоятельство будет использовано функцией `db_nextrec`, которую мы исследуем в конце этого раздела.

[432–440] Чтобы установить блокировку для записи на список свободных записей, вызывается функция `writew_lock`. Это предотвратит возможность взаимовлияния различных процессов при одновременном удалении записей из различных цепочек. Так как удаление записи сопряжено с изменением списка свободных записей, в каждый момент времени только один процесс должен делать это.

Заполненная пробелами запись с данными записывается функцией `_db_writedat`. Обратите внимание: в этом случае не нужно устанавливать блокировку на файл с данными. Так как `db_delete` установила блокировку для записи на цепочку из хеш-таблицы, мы точно знаем, что никакой другой процесс не сможет прочитать или изменить запись с данными.

```

441      /*
442       * Прочитать указатель на первую запись в списке свободных записей.
443       * На его место будет записан указатель на удаляемую запись.
444       * Это означает, что удаляемая запись вставляется в начало списка.
445       */
446     freeptr = _db_readptr(db, FREE_OFF);

447     /*
448      * Сохранить указатель на запись, следующую за удаляемой,
449      * прежде чем он будет затерт функцией _db_writeidx.
450      */
451     saveptr = db->ptrval;

452     /*
453      * Переписать индексную запись. В результате также будут переписаны
454      * значения длины индексной записи, позиции и длины записи с данными,
455      * ни одно из которых не было изменено, но так и должно быть.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

458     /*
459      * Записать новый указатель на начало списка свободных записей.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);

462     /*
463      * Изменить указатель, который указывает на удаляемую запись.
464      * Мы уже упоминали, что _db_find_and_lock записывает в db->ptroff
465      * адрес этого указателя. Мы запишем в этот указатель адрес записи,
466      * которая следует за удаляемой, то есть saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: ошибка вызова функции un_lock");
471 }

```

[441–461] Читаем указатель на первую запись в списке свободных записей и затем обновляем удаляемую индексную запись, чтобы она указывала на первую запись в списке свободных записей. (Если до этого список был пуст, указатель на следующую запись будет содержать 0.) Ключ у нас уже забит пробелами. Затем мы обновляем указатель на первую запись в списке свободных записей, чтобы он указывал на удаляемую запись. Это означает, что список свободных записей обслуживается по принципу «последний пришел, первый ушел», то есть удаляемая запись вставляется в начало списка (хотя удалять записи из этого списка мы будем по принципу «первого подходящего»).

Мы не предусматриваем отдельных списков свободных записей для каждого из файлов. Когда мы добавляем удаляемую индексную запись в список свободных записей, она по-прежнему ссылается на соответствующую удаленную ей запись с данными. Существуют более удобные способы удаления записей, но они требуют усложнения функции.

[462–471] Обновляем предыдущую запись в цепочке, чтобы она указывала на запись, следующую за удаляемой. В результате удаляемая запись исключается из цепочки. В заключение мы снимаем блокировку со списка свободных записей.

```

472 /*
473  * Сохраняет запись с данными. Вызывается из _db_dodelete (чтобы сохранить
474  * запись, заполненную пробелами) и из db_store.
475 */
476 static void
477 _db_writedat(DB *db, const char *data, off_t offset, int whence)
478 {
479     struct iovec iov[2];
480     static char newline = NEWLINE;
481
482     /*
483      * Если мы добавляем запись в конец файла, необходимо предварительно
484      * установить блокировку, чтобы выполнить lseek и write атомарно.
485      * Если перезаписывается существующая запись, блокировка не нужна.
486      */
487     if (whence == SEEK_END)/* добавить в конец, заблокировать весь файл */
488         if (writelock(db->datfd, 0, SEEK_SET, 0) < 0)
489             err_dump("_db_writedat: ошибка вызова функции writelock");
490
491     if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
492         err_dump("_db_writedat: ошибка вызова функции lseek");
493     db->datlen = strlen(data) + 1; /* в datlen включен символ NL */
494
495     iov[0].iov_base = (char *) data;
496     iov[0].iov_len = db->datlen - 1;
497     iov[1].iov_base = &newline;
498     iov[1].iov_len = 1;
499     if (writev(db->datfd, &iov[0], 2) != db->datlen)
500         err_dump("_db_writedat: ошибка вывода записи с данными");
501 }

```

[472–491] Мы вызываем `_db_writedat`, чтобы вывести в файл запись с данными. Когда удаляется запись, мы с помощью `_db_writedat` перезаписываем запись, заполненную пробелами. При вызове `_db_writedat` не требуется устанавливать блокировку на файл с данными, поскольку `db_delete` уже установила блокировку для записи на цепочку хешей, в которой находится эта запись. То есть никакой другой процесс не сможет ни прочитать, ни перезаписать эту запись. Далее в этом разделе, в обсуждении функции `db_store`, мы столкнемся с ситуацией, когда `_db_writedat` производит добавление записи в конец файла и должна установить блокировку на весь файл.

Перемещаемся в позицию, куда необходимо выполнить запись. Объем записываемых данных равен размеру записи плюс 1 байт для завершающего нулевого символа.

[492–501] Заполняем поля структуры `iovec` и вызываем `writev`, чтобы записать данные и символ перевода строки. Мы не можем полагаться на то, что в буфере, полученном от вызывающей функции, достаточно места, чтобы добавить символ перевода строки, поэтому берем его из отдельного буфера. После вывода записи в файл мы снимаем блокировку, которую установили ранее.

```

502 /*
503  * Сохраняет индексную запись. Перед этой функцией вызывается _db_writedat,
504  * которая устанавливает значения полей datoff и datlen в структуре DB,
505  * необходимые для создания индексной записи.
506 */
507 static void
508 _db_writeidx(DB *db, const char *key,
509                 off_t offset, int whence, off_t ptrval)
510 {
511     struct iovec iov[2];
512     char      asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
513     int       len;
514
515     if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
516         err_quit("_db_writeidx: неверный указатель: %d", ptrval);
517     sprintf(db->idxbuf, "%s%c%lld%c%ld\n", key, SEP,
518             (long long)db->datoff, SEP, (long)db->datlen);
519     len = strlen(db->idxbuf);
520     if (len < IDXLEN_MIN || len > IDXLEN_MAX)
521         err_dump("_db_writeidx: неверная длина");
522     sprintf(asciiptrlen, "%*lld%*d", PTR_SZ, (long long)ptrval,
523             IDXLEN_SZ, len);
524
525     /*
526      * Если запись добавляется в конец файла, необходимо предварительно
527      * установить блокировку, чтобы выполнить lseek и write атомарно.
528      * Если перезаписывается существующая запись, блокировка не нужна.
529     */
530     if (whence == SEEK_END) /* добавление в конец файла */
531         if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
532                         SEEK_SET, 0) < 0)
533             err_dump("_db_writeidx: ошибка вызова функции writew_lock");

```

[502–522] Функция `_db_writeidx` вызывается, чтобы вывести в файл индексную запись. После проверки указателя на следующую запись в цепочке мы создаем индексную запись и сохраняем ее вторую половину в буфере `idxbuf`. Нам потребуется размер этой части, чтобы создать первую половину индексной записи, которую мы сохраняем в переменной `asciiptrlen`.

Обратите внимание: чтобы гарантировать соответствие аргументов спецификациям форматов, в вызове функции `sprintf` выполняется приведение их типов. Это обусловлено тем, что типы `off_t` и `size_t` могут иметь разные размеры на разных платформах. Даже 32-разрядные системы могут предоставлять 64-разрядные значения смещения в файле, поэтому мы не можем делать какие-либо предположения о размерах типов `off_t` и `size_t`.

[523–531] Как и `_db_writedat`, эта функция устанавливает блокировку, только если новая индексная запись добавляется в конец индексного файла. Когда эта функция вызывается из `_db_dodelete`, мы перезаписываем существующую индексную запись. В этой ситуации вызывающая функция устанавливает блокировку для записи на цепочку из хеш-таблицы, и поэтому установка дополнительной блокировки не требуется.

```

532     /*
533      * Позиция в индексном файле и смещение записи.
534      */
535     if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
536         err_dump("_db_writeidx: ошибка вызова функции lseek");
537
537     iov[0].iov_base = asciiptrlen;
538     iov[0].iov_len = PTR_SZ + IDXLEN_SZ;
539     iov[1].iov_base = db->idxbuf;
540     iov[1].iov_len = len;
541     if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
542         err_dump("_db_writeidx: ошибка вывода в файл индексной записи");
543
543     if (whence == SEEK_END)
544         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
545             SEEK_SET, 0) < 0)
546             err_dump("_db_writeidx: ошибка вызова функции un_lock");
547 }
548 */
549 * Записывает значение указателя в индексный файл:
550 * в список свободных записей, хеш-таблицу или индексную запись.
551 */
552 static void
553 _db_writeptr(DB *db, off_t offset, off_t ptrval)
554 {
555     char    asciiptr[PTR_SZ + 1];
556
556     if (ptrval < 0 || ptrval > PTR_MAX)
557         err_quit("_db_writeptr: неверный указатель: %d", ptrval);
558     sprintf(asciiptr, "%lld", PTR_SZ, (long long)ptrval);
559
559     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
560         err_dump("_db_writeptr: ошибка перемещения на поле с указателем");
561     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
562         err_dump("_db_writeptr: ошибка записи в поле с указателем");
563 }

```

[532–547] Перемещаемся в позицию, куда должна быть записана индексная запись, и сохраняем это смещение в поле `idxoff` структуры `DB`. Поскольку индексная запись собрана в двух отдельных буферах, для ее сохранения в индексном файле используется функция `writev`. Если производилось добавление новой записи в конец файла, мы снимаем блокировку, которую установили перед изменением текущей позиции файла. Это позволяет производить операции изменения текущей позиции файла и записи атомарно для процессов, работающих параллельно и добавляющих новые записи в ту же базу данных.

[548–563] Функция `_db_writeptr` используется для записи в индексный файл указателя на очередную запись. Этот указатель проверяется на превышение допустимых пределов и преобразуется в строку символов ASCII. Мы переходим в заданную позицию в индексном файле и записываем указатель.

```

564 /*
565  * Сохраняет запись в базе данных. Возвращает 0 в случае успеха; 1, если
566  * запись существует и установлен флаг DB_INSERT; -1 в случае ошибки.
567 */
568 int
569 db_store(DBHANDLE h, const char *key, const char *data, int flag)
570 {
571     DB      *db = h;
572     int      rc, keylen, datlen;
573     off_t    ptrval;
574
575     if (flag != DB_INSERT && flag != DB_REPLACE &&
576         flag != DB_STORE) {
577         errno = EINVAL;
578         return(-1);
579     }
580     keylen = strlen(key);
581     datlen = strlen(data) + 1; /* +1 для символа перевода строки */
582     if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
583         err_dump("db_store: неверная длина записи");
584
585     /*
586      * _db_find_and_lock вычисляет, в какую хеш-таблицу должна
587      * быть добавлена новая запись (db->chainoff), независимо от того,
588      * существует она или нет. Следующий вызов _db_writeptr изменит
589      * запись в хеш-таблице, записав в нее указатель на новую запись.
590      * Новая запись вставляется в начало цепочки.
591     */
592     if (_db_find_and_lock(db, key, 1) < 0) { /* запись не найдена */
593         if (flag == DB_REPLACE) {
594             rc = -1;
595             db->cnt_storerr++;
596             errno = ENOENT; /* ошибка, запись не найдена */
597             goto doreturn;
598         }

```

[564–582] Функция `db_store` используется для добавления новых записей в базу данных. Сначала она проверяет значения флагов, переданных ей. Затем длину записи. Если размер записи выходит за допустимые пределы, создается файл `core` и работа процесса завершается. Такое поведение допустимо для библиотеки-примера, но если вы собираетесь создать библиотеку для использования в действующих приложениях, необходимо вместо завершения процесса возвращать признак ошибки, чтобы позволить приложению исправить ее.

[583–596] Мы вызываем `_db_find_and_lock`, чтобы убедиться в существовании записи. Ситуации, когда запись не найдена и установлен флаг `DB_INSERT` или `DB_STORE` либо когда запись существует и установлен флаг `DB_REPLACE` или `DB_STORE`, не считаются ошибочными. Если выполняется замещение существующей записи, это означает, что ключи записей идентичны, но сами данные могут отличаться. Обратите внимание, что последний аргумент функции `_db_find_and_lock` указывает, что на цепочку в хеш-таблице должна быть установлена блокировка для записи, поскольку она, скорее всего, будет подвергнута изменениям.

```

597      /*
598      * _db_find_and_lock уже заблокировала цепочку в хеш-таблице;
599      * прочитать указатель на первую индексную запись в цепочке.
600      */
601      ptrval = _db_readptr(db, db->chainoff);

602      if (_db_findfree(db, keylen, datlen) < 0) {
603          /*
604          * Не найдена пустая запись достаточного размера. Добавить
605          * новые записи в конец индексного файла и файла с данными.
606          */
607          _db_writedat(db, data, 0, SEEK_END);
608          _db_writeidx(db, key, 0, SEEK_END, ptrval);

609          /*
610          * Значение db->idxoff было установлено в _db_writeidx.
611          * Новая запись добавляется в начало цепочки хеш-таблицы.
612          */
613          _db_writeptr(db, db->chainoff, db->idxoff);
614          db->cnt_stor1++;
615      } else {
616          /*
617          * Использовать повторно пустую запись. _db_findfree удалит ее
618          * из списка свободных записей и установит значения db->datoff
619          * и db->idxoff. Запись добавляется в начало списка.
620          */
621          _db_writedat(db, data, db->datoff, SEEK_SET);
622          _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
623          _db_writeptr(db, db->chainoff, db->idxoff);
624          db->cnt_stor2++;
625      }

```

[597–601] После вызова `_db_find_and_lock` возможны четыре сценария дальнейшего развития событий. В первых двух, когда запись не найдена, необходимо добавить новую запись. Мы читаем указатель на первую запись в цепочке хеш-таблицы.

[602–614] Случай 1: с помощью функции `_db_findfree` мы пытаемся отыскать в списке свободных записей ранее удаленную запись с тем же размером ключа и объемом данных. Если таковая не найдена, добавляем новые записи в конец индексного файла и файла с данными. Для записи данных вызывается функция `_db_writedat`, для записи индекса — функция `_db_writeidx`, а для вставки новой индексной записи в начало цепочки хеш-таблицы — функция `_db_writeptr`. Затем мы увеличиваем счетчик (`cnt_stor1`) ситуаций, развивающихся по этому сценарию, что позволит нам в дальнейшем проанализировать поведение базы данных.

[615–625] Случай 2: функция `_db_findfree` нашла пустую запись требуемого размера и исключила ее из списка свободных записей (вскоре мы рассмотрим реализацию функции `_db_findfree`). Мы записываем индексную запись и запись с данными и добавляем адрес записи в начало цепочки хеш-таблицы, как и в первом случае. Затем увеличиваем счетчик (`cnt_stor2`) ситуаций, развивающихся по данному сценарию.

```

626     } else { /* запись найдена */
627         if (flag == DB_INSERT) {
628             rc = 1; /* ошибка, запись уже имеется в базе данных */
629             db->cnt_storerr++;
630             goto doreturn;
631         }
632         /*
633          * Производится замена существующей записи. Мы знаем, что
634          * новый ключ равен существующему, но нам нужно проверить
635          * равенство размеров записей с данными.
636          */
637         if (datlen != db->datlen) {
638             _db_dodelete(db); /* удалить существующую запись */
639             /*
640              * Перечитать указатель из хеш-таблицы
641              * (он мог измениться в процессе удаления).
642              */
643             ptrval = _db_readptr(db, db->chainoff);
644             /*
645              * Добавить новые записи в конец файлов.
646              */
647             _db_writedat(db, data, 0, SEEK_END);
648             _db_writeidx(db, key, 0, SEEK_END, ptrval);
649             /*
650              * Вставить указатель на запись в начало цепочки.
651              */
652             _db_writeptr(db, db->chainoff, db->idxoff);
653             db->cnt_stor3++;
654         } else {

```

[626–631] Теперь мы перешли к двум возможным ситуациям, когда запись с тем же самым ключом уже существует в базе данных. Если вызывающий процесс не указал, что запись должна быть замещена, мы записываем в возвращаемое значение код, который свидетельствует о том, что запись уже существует, увеличиваем счетчик ошибок операций записи и переходим в конец функции, где реализован алгоритм выхода.

[632–654] Случай 3: существующая запись должна быть замещена, но длина записи с данными отличается от длины существующей записи с данными. Мы вызываем функцию `_do_delete`, которая удалит существующую запись. Как вы помните, при этом она вставит удаленную запись в начало списка свободных записей. Затем мы добавляем новые записи в конец индексного файла и в конец файла с данными с помощью функций `_db_writeidx` и `_db_writedat`. (Существуют и другие способы обработки этой ситуации. Можно, например, попытаться отыскать свободную запись подходящего размера.) Новая запись добавляется в начало цепочки хеш-таблицы вызовом функции `_db_writeptr`. В счетчик `cnt_stor3` структуры `DB` записывается количество ситуаций, развивающихся по этому сценарию.

```

655      /*
656       * Размеры данных совпадают, просто заменить запись.
657       */
658       _db_writedat(db, data, db->datoff, SEEK_SET);
659       db->cnt_stor4++;
660   }
661 }
662 rc = 0; /* OK */

663 doreturn: /* снять блокировку, установленную в _db_find_and_lock */
664     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
665         err_dump("db_store: ошибка вызова функции un_lock");
666     return(rc);
667 }

668 /*
669  * Пытается отыскать свободную индексную запись с данными
670  * нужного размера. Эта функция вызывается только из db_store.
671 */
672 static int
673 _db_findfree(DB *db, int keylen, int datlen)
674 {
675     int      rc;
676     off_t   offset, nextoffset, saveoffset;

677     /*
678      * Заблокировать указатель на список свободных записей.
679      */
680     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
681         err_dump("_db_findfree: ошибка вызова функции writew_lock");

682     /*
683      * Прочитать указатель на первую запись в списке.
684      */
685     saveoffset = FREE_OFF;
686     offset = _db_readptr(db, saveoffset);

```

[655–661] Случай 4: существующая запись должна быть замещена, и размер новой записи с данными совпадает с размером существующей записи с данными. Это самый простой случай — нужно лишь записать новые данные в файл и увеличить счетчик (`cnt_stor4`) аналогичных ситуаций.

[662–667] Если все в порядке, мы записываем в возвращаемое значение признак успешного завершения и переходим к выполнению алгоритма выхода. Здесь мы снимаем с цепочки в хеш-таблице блокировку, установленную функцией `_db_find_and_lock`, и возвращаем управление вызывающему процессу.

[668–686] Функция `_db_findfree` пытается найти свободную индексную запись и связанную с ней запись с данными заданных размеров. Чтобы избежать взаимовлияния с другими процессами, необходимо установить блокировку для записи на список свободных записей. После установки блокировки мы читаем адрес первой записи в списке.

```

687     while (offset != 0) {
688         nextoffset = _db_readidx(db, offset);
689         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
690             break; /* совпадение найдено */
691         saveoffset = offset;
692         offset = nextoffset;
693     }
694
695     if (offset == 0) {
696         rc = -1; /* совпадений не найдено */
697     } else {
698         /*
699          * Найдена запись требуемого размера. Индексная запись
700          * была прочитана ранее в _db_readidx, которая установила значение
701          * db->ptrval. Кроме того, saveoffset указывает на запись
702          * в списке свободных записей, соответствующую найденной записи.
703          * Мы записываем в нее значение db->ptrval, исключая тем самым
704          * найденную запись из списка свободных записей.
705          */
706         _db_writeptr(db, saveoffset, db->ptrval);
707         rc = 0;
708
709         /*
710          * Обратите внимание: _db_readidx записывает значения в db->idxoff
711          * и в db->datoff. Это обстоятельство используется вызывающей
712          * функцией db_store для вывода новых записей в файлы.
713          */
714
715         /*
716         * Снять блокировку со списка свободных записей.
717         */
718         if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
719             err_dump("_db_findfree: ошибка вызова функции un_lock");
720     }

```

[687–693] В цикле `while` производится обход списка свободных записей в поисках записи с соответствующими размерами ключа и данных. В этой простой реализации мы повторно используем удаленные записи, только если размеры ключа и данных совпадают с размерами ключа и данных вставляемой записи. Существуют более эффективные алгоритмы использования свободного пространства, но они требуют усложнения реализации.

[694–712] Если запись с требуемыми размерами ключа и данных не была найдена, мы записываем в возвращаемое значение код, который свидетельствует о неудаче. В противном случае записываем в указатель предыдущей записи адрес записи, следующей за найденной. Таким способом мы исключаем найденную запись из списка свободных записей.

[713–719] По окончании операций со списком свободных записей снимаем блокировку и возвращаем код завершения операции вызывающей функции.

```
720 /*
721  * Переход к первой записи для функции db_nextrec.
722  * Автоматически вызывается из db_open.
723  * Должна вызываться перед первым обращением к db_nextrec.
724 */
725 void
726 db_rewind(DBHANDLE h)
727 {
728     DB      *db = h;
729     off_t   offset;

730     offset = (db->nhash + 1) * PTR_SZ; /* +1 для списка свободных записей */

731     /*
732      * Просто устанавливаем текущую позицию в файле для данного
733      * процесса на первую индексную запись — блокировка не требуется.
734      * +1, чтобы перешагнуть символ перевода строки в конце хеш-таблицы.
735      */
736     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
737         err_dump("db_rewind: ошибка вызова функции lseek");
738 }

739 /*
740  * Возвращает следующую запись.
741  * Мы просто двигаемся по индексному файлу, игнорируя удаленные записи.
742  * Перед первым обращением к этой функции должна быть вызвана
743  * функция db_rewind.
744 */
745 char *
746 db_nextrec(DBHANDLE h, char *key)
747 {
748     DB      *db = h;
749     char    c;
750     char    *ptr;
```

[720–738] Функция `db_rewind` используется для перехода к «началу» базы данных — она устанавливает текущую позицию в индексном файле на начало первой записи (которая находится сразу же за хеш-таблицей). (Вспомните структуру индексного файла на рис. 20.1.)

[739–750] Функция `db_nextrec` возвращает следующую запись из базы данных. Вызывающему процессу возвращается указатель на буфер с данными. Если в аргументе `key` передается непустой указатель, по заданному адресу будет возвращен ключ, который соответствует записи с данными. Вся ответственность за выделение буфера достаточного размера для хранения ключа возлагается на вызывающий процесс. Буфер с размером `IDXLEN_MAX` сможет вместить в себя любой ключ.

Записи возвращаются в порядке, в котором они были записаны в базу данных. То есть записи не сортируются по ключу. Кроме того, поскольку мы не принимаем во внимание цепочки из хеш-таблицы, в процессе обхода базы данных могут обнаружиться удаленные записи, но они не должны возвращаться вызывающему процессу.

```

751     /*
752      * На список свободных записей устанавливается блокировка для чтения,
753      * чтобы в процессе чтения нельзя было удалить запись.
754      */
755     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
756         err_dump("db_nextrec: ошибка вызова функции readw_lock");

757     do {
758         /*
759          * Прочитать очередную запись.
760          */
761         if (_db_readididx(db, 0) < 0) {
762             ptr = NULL; /* конец индексного файла */
763             goto doreturn;
764         }

765         /*
766          * Проверить, не заполнен ли ключ пробелами (пустая запись).
767          */
768         ptr = db->idxbuf;
769         while ((c = *ptr++) != 0 && c == SPACE)
770             ; /* перейти к первому символу, отличному от пробела */
771         } while (c == 0); /* повторять, пока не встретится непустой ключ */

772         if (key != NULL)
773             strcpy(key, db->idxbuf); /* вернуть ключ */
774         ptr = _db_readdat(db); /* вернуть указатель на буфер */
775         db->cnt_nextrec++;

776     doreturn:
777         if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
778             err_dump("db_nextrec: ошибка вызова функции un_lock");
779         return(ptr);
780     }

```

[751–756] Прежде всего, нам необходимо установить блокировку для чтения, чтобы никакой другой процесс не смог удалить запись в процессе ее чтения.

[757–771] Для чтения очередной записи вызывается `_db_readididx`. Мы передаем ей 0 в качестве смещения, чтобы указать, что чтение должно производиться с текущей позиции. Поскольку мы последовательно читаем все записи из индексного файла, мы можем обнаружить удаленные записи. Но так как должны возвращаться только нормальные записи, необходимо пропускать записи, ключи которых заполнены пробелами (функция `_db_dodelete` очищает строку ключа, заполняя ее пробелами).

[772–780] Встретив нормальный ключ, мы копируем его в буфер вызывающего процесса, если он был предоставлен. Затем мы читаем запись с данными и записываем возвращаемое значение указатель на внутренний буфер, содержащий эту запись. После этого увеличиваем счетчик обращений к функции `db_nextrec`, снимаем блокировку списка свободных записей и возвращаем указатель на буфер с данными.

Как правило, функции `db_rewind` и `db_nextrec` используются в цикле, например

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
    /* обработка полученной записи */
}
```

Как мы уже предупреждали ранее, записи возвращаются не по порядку — они не сортируются по значению ключа. Если в процессе извлечения записей в цикле с помощью функции `db_nextrec` база данных будет изменяться, записи, возвращаемые `db_nextrec`, будут представлять собой просто срезы изменяющейся базы данных в некоторый момент времени. Функция `db_nextrec` всегда возвращает запись, которая была действительна на момент вызова функции, то есть она не возвращает записи, которые были удалены. Но вполне возможно, что запись будет удалена сразу после возврата из функции `db_nextrec`. Аналогично, если пустая запись была заполнена уже после того, как `db_nextrec` перешагнула через нее, мы не сможем увидеть новую запись, пока не вернемся к началу базы данных и не повторим цикл обхода. Если важно получить точный «замороженный» срез базы данных с помощью `db_nextrec`, в системе не должно быть процессов, которые могли бы вставить новые или удалить существующие записи во время получения среза.

Взгляните, как `db_nextrec` использует механизм блокировок. Мы не учитываем цепочки в хеш-таблице и не можем определить, какой цепочке принадлежит та или иная запись. То есть вполне возможна ситуация, когда индексная запись будет находиться в процессе удаления, в то время как `db_nextrec` читает ее. Чтобы предотвратить это, `db_nextrec` устанавливает блокировку для чтения на список свободных записей, благодаря чему исключается возможность взаимовлияния с функциями `_db_dodelete` и `_db_findfree`.

Прежде чем завершить исследование файла `db.c`, мы должны описать принцип действия блокировки, которая устанавливается при добавлении новых записей в конец файла. В случаях 1 и 3 функция `db_store` вызывает `_db_writeidx` и `_db_writedat`, передавая им в третьем аргументе значение 0, а в четвертом — `SEEK_END`. Этот четвертый аргумент служит признаком добавления новой записи в конец файла. Функция `_db_writeidx` устанавливает блокировку для записи от конца цепочки хеш-таблицы до конца файла. Такой прием не повлияет на другие читающие или пишущие в базу данных процессы (так как они будут устанавливать блокировку на цепочку хеш-таблицы) и при этом не даст возможности другим процессам в то же самое время добавлять записи в конец файла. Функция `_db_writedat` устанавливает блокировку для записи на весь файл с данными. Это также не повлияет на другие читающие или пишущие в базу процессы (так как они даже не будут пытаться установить блокировку на файл с данными) и в то же время не даст возможности другим процессам добавлять записи в конец файла (упражнение 20.3).

20.9. Производительность

Чтобы протестировать библиотеку базы данных и получить некоторые временные характеристики производительности, была написана тестовая программа. Эта программа принимает два аргумента командной строки: количество создаваемых дочерних процессов и количество записей (*nrec*), которые каждый процесс должен записать в базу данных. Программа создает пустую базу данных (вызовом функции `db_open`), порождает заданное число дочерних процессов и ожидает их завершения. Каждый дочерний процесс выполняет следующие действия.

1. Записывает *nrec* записей в базу данных.
2. Читает *nrec* записей по заданному ключу.
3. Выполняет следующий цикл: $nrec \times 5$ раз.
 - 1) читает случайную запись;
 - 2) через каждые 37 циклов удаляет случайную запись;
 - 3) через каждые 11 циклов вставляет новую запись и читает ее обратно;
 - 4) через каждые 17 циклов замещает случайную запись новой записью. Новая запись имеет либо тот же размер строки с данными, либо больший — через раз.
4. Удаляет все созданные им записи. Каждый раз при удалении записи выполняется поиск десяти случайных записей.

Количество операций с базой данных сохраняется в счетчиках `cnt_xxx` структуры `DB`. Количество операций, выполняемых каждым из процессов, различно, поскольку для выборки записей используется генератор случайных чисел, инициализированный идентификатором дочернего процесса. Типичные значения счетчиков операций, производимых каждым дочерним процессом при *nrec*, равном 500, приводятся в табл. 20.2.

Количество операций по извлечению записей примерно в десять раз превышает количество операций по удалению или добавлению новых записей, что типично для большинства приложений баз данных.

Каждый дочерний процесс выполняет все операции (извлечение, удаление и сохранение) только с теми записями, которые были записаны самим дочерним процессом. В процессе тестирования активно использовались средства управления одновременным доступом, поскольку все дочерние процессы работали с одной и той же базой данных (хотя и с разными записями). Общее количество записей в базе данных возрастает пропорционально количеству дочерних процессов. (Один дочерний процесс изначально записывает в базу данных *nrec* записей, два дочерних процесса — $nrec \times 2$ записей и т. д.)

Чтобы получить и сравнить временные характеристики при использовании крупноблочной и мелкоблочной блокировок, а также выполнить сравнение трех типов блокировок (отсутствие блокировок, рекомендательные блокировки, принудительные блокировки), мы запускали три версии программы. Первая версия (исходный код которой приведен в разделе 20.8) использует мелкоблочную блокировку. Вторая версия программы использует крупноблочную блокировку, как это

Таблица 20.2. Типичные значения счетчиков операций, выполняемых каждым из процессов при $nrec = 500$

Операция	Вызовов fcntl (на одну операцию)		Количество операций ($nrec = 2000$)
	Крупно- блочная блокировка	Мелкоблоч- ная блокиро- вка	
<code>db_store</code> , <code>DB_INSERT</code> , подходящая пустая запись не найдена, добавление в конец файла	2	8	2920
<code>db_store</code> , <code>DB_INSERT</code> , используется пустая запись	2	4	468
<code>db_store</code> , <code>DB_REPLACE</code> , новая запись имеет другой размер, добавление в конец файла	2	8	405
<code>db_store</code> , <code>DB_REPLACE</code> , новая запись имеет тот же размер, добавление в конец файла	2	2	416
<code>db_store</code> , запись не найдена	2	2	71
<code>db_fetch</code> , запись найдена	2	2	32 873
<code>db_fetch</code> , запись не найдена	2	2	2966
<code>db_delete</code> , запись найдена	2	4	3388
<code>db_delete</code> , запись не найдена	2	2	422

описано в разделе 20.6. Из третьей версии были удалены все функции установки блокировок, что дало возможность определить накладные расходы на использование механизма блокировок. Первая и вторая версия программы (мелкоблочные блокировки и крупноблочные блокировки) могли использовать как рекомендательные, так и принудительные блокировки, для этого достаточно было изменить права доступа к файлам базы данных. (Во всех отчетах, приводимых в данном разделе, при использовании принудительных блокировок измерения производились только для версии с мелкоблочными блокировками.)

Все испытания, описываемые в этом разделе, проводились на компьютере с процессором Intel Core-i5, работающим под управлением Linux 3.2.0. Этот процессор имеет четыре ядра, обеспечивая возможность параллельного выполнения до четырех процессов.

Результаты для единственного процесса

В табл. 20.3 приводятся результаты хронометража для случая с одним процессом и значением $nrec$, равным 2000, 6000 и 12 000.

Результаты измерений приводятся в табл. 20.3 в секундах. Во всех случаях сумма пользовательского и системного времени выполнения примерно равна общему времени. Это говорит о том, что в основном использовалась производительность центрального процессора, а не дисковой подсистемы.

Таблица 20.3. Один процесс, различные значения nrec, различные типы блокировок

nrec	Нет блокировок			Рекомендательные блокировки						Принудительные блокировки		
				Крупноблочные			Мелкоблочные			Мелкоблочные		
	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время
2000	0,10	0,22	0,33	0,17	0,33	0,51	0,13	0,38	0,51	0,14	0,43	0,58
6000	0,59	1,32	1,91	0,88	2,13	3,03	0,90	2,14	3,05	0,99	2,52	3,53
12 000	4,37	9,58	13,97	5,38	12,60	18,01	5,34	12,63	18,01	5,53	15,03	20,60

В шести колонках, которые соответствуют рекомендательным блокировкам, значения времени практически одинаковы в каждой из строк. Это говорит о том, что в случае единственного процесса отсутствуют различия между крупноблочными и мелкоблочными блокировками, кроме количества вызовов `fcntl`.

Сравнение времени работы при использовании рекомендательных блокировок со временем работы версии, в которой блокировки вообще не использовались, показывает, что использование механизма блокировок добавляет от 32 до 73% к системному времени работы. Даже несмотря на то что механизм блокировок фактически не использовался (поскольку работал только один процесс), обращения к системному вызову `fcntl` заняли определенное время. Обратите внимание, что пользовательское время работы для всех четырех случаев практически одинаково. Это объясняется тем, что код, работающий в пространстве пользователя, практически не изменялся (за исключением нескольких вызовов функции `fcntl`).

И последнее замечание к результатам в табл. 20.3: использование принудительных блокировок увеличило системное время работы на 13–19% по сравнению с результатами, полученными при использовании рекомендательных блокировок. Так как количество наложений блокировок для версий с мелкоблочными принудительными и мелкоблочными рекомендательными блокировками одно и то же, можно утверждать, что дополнительные накладные расходы связаны с операциями чтения и записи.

В заключительном тесте была предпринята попытка запустить несколько дочерних процессов для версии, которая не использует механизм блокировок. Как и следовало ожидать, в результате мы получали нерегулярные ошибки. Как правило, процессы не могли найти записи, которые были добавлены в базу данных, что приводило к аварийному завершению. Каждый раз при запуске программы мы получали разные типы ошибок. Это пример классического состояния гонки за ресурсами: множество процессов обновляют один и тот же файл, не используя никаких блокировок.

Результаты для нескольких процессов

Следующие ниже результаты демонстрируют главным образом различия между крупноблочными и мелкоблочными блокировками. Как уже говорилось, интуитивно мы ожидали, что мелкоблочные блокировки обеспечат дополнительную производительность, так как в этом случае блокируются небольшие участки базы данных. В табл. 20.4 приводятся результаты для $nrec = 2000$ и количества дочерних процессов от 1 до 16.

Все результаты приводятся в секундах и представляют суммарное время для всех дочерних и родительского процессов. Полученные результаты позволяют сделать ряд выводов.

Первое, на что следует обратить внимание: при наличии нескольких процессов, выполняющихся параллельно, сумма пользовательского и системного времени превышает общее время. На первый взгляд это выглядит странно, но в этом нет ничего необычного, если учесть, что процессор имеет несколько ядер. Дело в том, что время выполнения всех процессов складывается; процессорное время, которое приводится в таблице, является суммой времени работы всех ядер, используемых программой. Так как у нас выполняется сразу несколько процессов (по одному на ядро), процессорное время может превышать общее время.

Восьмая колонка, отмеченная как «Δ Общее время», представляет различия в секундах между значениями общего времени при использовании рекомендательных крупноблочных и мелкоблочных блокировок. Это значение демонстрирует прирост производительности, который достигается при переходе от крупноблочных к мелкоблочным блокировкам. В системе, на которой проводились испытания, прирост производительности практически отсутствует, пока количество процессов не превышает одного. Но с увеличением количества одновременно работающих процессов прирост производительности за счет использования мелкоблочных блокировок становится более заметным (около 30%).

Мы предполагали, что при переходе от крупноблочных к мелкоблочным блокировкам общее время выполнения в случае нескольких процессов будет уменьшаться. Но системное время выполнения при использовании мелкоблочных блокировок должно увеличиться независимо от количества одновременно работающих процессов, потому что при использовании мелкоблочных блокировок требуется больше обращений к функции `fcntl`, чем при использовании крупноблочных блокировок. Если подсчитать количество вызовов `fcntl` из табл. 20.2, получится, что в случае крупноблочных блокировок требуется в среднем 87 858 вызовов `fcntl`, а в случае мелкоблочных блокировок — 115 520 вызовов. Мы ожидали, что в случае мелкоблочных блокировок увеличение количества вызовов `fcntl` на 31% приведет к увеличению системного времени выполнения. Таким образом, уменьшение системного времени выполнения при использовании мелкоблочных блокировок для двух процессов и относительное небольшое увеличение, когда количество одновременно работающих процессов больше двух, выглядит несколько загадочным.

Таблица 20.4. Сравнение различных типов блокировок для $n_{\text{тес}} = 2000$

Коли-чество процессов	Рекомендательные блокировки				Принудительные блокировки					
	Крупноблочные		Мелкоблочные		Δ Общее время		Мелкоблочные			
Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Пользовательское время	Системное время	Общее время	Δ Системное время	
1	0,14	0,35	0,50	0,14	0,35	0,50	0	0,15	0,42	0,58
2	0,60	1,43	1,88	0,54	1,36	1,10	71	0,65	2,01	1,59
3	0,97	2,67	3,48	1,37	3,73	2,20	45	1,62	5,67	3,28
4	2,38	6,17	5,59	2,83	8,15	4,07	37	3,29	12,35	6,31
5	3,72	10,17	8,37	4,28	11,86	6,09	37	4,96	18,47	9,49
6	5,02	14,52	11,52	6,04	17,46	8,89	30	6,66	26,38	13,22
7	7,00	20,16	15,84	8,06	23,23	11,88	33	9,12	36,13	18,09
8	9,12	26,20	20,31	10,50	30,50	15,48	31	11,81	47,20	23,49
9	11,60	33,91	25,64	13,40	37,80	19,29	33	14,54	60,23	29,66
10	14,28	42,24	31,35	16,39	47,01	23,74	32	17,84	74,05	36,27
11	17,37	51,12	37,50	19,71	56,59	28,57	31	21,57	90,14	44,10
12	20,70	60,48	44,24	23,47	66,10	33,34	33	25,57	108,94	53,11
13	25,13	70,67	51,96	27,70	77,76	39,21	33	29,71	133,31	63,07
14	28,40	82,23	59,88	32,34	91,45	46,22	30	34,22	155,80	73,86
15	32,23	94,26	68,30	36,32	102,97	51,82	32	39,05	180,66	84,14
16	37,24	107,87	78,67	42,17	118,20	59,72	32	44,11	208,28	96,82

Такое поведение имеет два объяснения. Во-первых, как видно из табл. 20.3, когда конкуренция за блокировки отсутствует, разница между версиями с крупноблочными и мелкоблочными блокировками практически отсутствует. То есть дополнительные вызовы функции `fcntl` практически не оказывают влияния на производительность тестовой программы. Во-вторых, при использовании крупноблочных блокировок мы устанавливаем блокировки на более длительные периоды времени, что увеличивает вероятность простояивания других процессов в ожидании снятия блокировки. При использовании мелкоблочных блокировок они устанавливаются на менее продолжительные периоды времени, поэтому вероятность простояивания на блокировке уменьшается. Если мы проанализируем поведение системы, то увидим, что при использовании крупноблочных блокировок процессы блокируются чаще. Например, для случая с четырьмя процессами крупноблочные блокировки запираются в пять раз чаще, чем мелкоблочные. Из-за этого при использовании крупноблочных блокировок процессам приходится чаще приостанавливаться и возобновлять работу, что уменьшает разницу между двумя подходами к организации блокировок.

В последней колонке, которая обозначена как «Δ Системное время», приводится процент увеличения системного времени работы при переходе от рекомендательных мелкоблочных блокировок к принудительным мелкоблочным блокировкам. Эти значения показывают, что использование принудительных блокировок значительно увеличивает системное время (от 20 до 76%) с ростом конкуренции.

Поскольку код, выполняющийся в пространстве пользователя, практически идентичен для всех версий (если не учитывать некоторое увеличение количества обращений к функции `fcntl` при использовании мелкоблочных блокировок как в рекомендательном, так и в принудительном варианте), мы ожидали, что пользовательское время работы в каждой строке будет примерно одинаковым.

Когда мы в первый раз провели тестирование, пользовательское время для версии с крупноблочными блокировками у нас получилось почти в два раза больше, чем для версии с мелкоблочными блокировками, когда одновременно выполнялось несколько конкурирующих процессов. Поскольку обе версии базы данных почти ничем не отличались, кроме количества вызовов `fcntl`, такой результат нам показался бессмысленным. Однако, проведя исследование, мы обнаружили следующее: из-за более высокой конкуренции в версии с крупноблочными блокировками процессы были вынуждены простоявать дольше, и операционная система решила снизить частоту процессора для экономии электроэнергии. При использовании мелкоблочных блокировок процессы действовали более активно, поэтому система увеличила частоту процессора. Это обстоятельство отрицательно сказалось на тестах с крупноблочными блокировками. После этого мы отключили автоматическое регулирование частоты процессора и вновь провели тестирование. В результате разница в пользовательском времени выполнения заметно уменьшилась.

Значения из первой строки в табл. 20.4 совпадают со значениями из табл. 20.3 для $nrec = 2000$. Это вполне соответствует нашим ожиданиям.

На рис. 20.4 данные из табл. 20.4 для рекомендательных мелкоблочных блокировок представлены в виде графика. Мы построили график зависимости общего

времени выполнения от количества процессов (1–16), а также графики, отображающие зависимость отношения пользовательского и системного времени выполнения к количеству процессов.

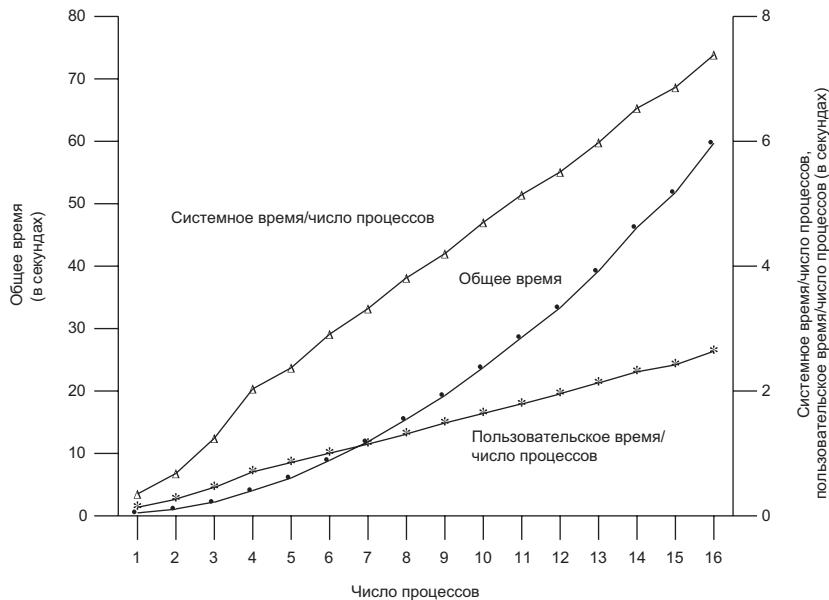


Рис. 20.4. Значения из табл. 20.4 для рекомендательных мелкоблочных блокировок

Обратите внимание: оба графика, которые соответствуют отношениям времени выполнения к количеству процессов, практически линейны, в то время как график общего времени — нелинейный. Вероятно, причина кроется в том, что при увеличении количества процессов операционной системе требуется больше времени для переключения между ними. Накладные расходы, связанные с работой самой операционной системы, должны были проявиться в виде увеличения общего времени, но не должны сказываться на процессорном времени, затраченном каждым из процессов.

Причина роста пользовательского времени выполнения при увеличении количества процессов связана с увеличением количества записей в базе данных. Каждая цепочка в хеш-таблице становится длиннее, вследствие чего функции `_db_find_and_lock` приходится выполнять больший объем работы при поиске записей.

20.10. Подведение итогов

В этой главе мы детально разобрали архитектуру и реализацию библиотеки базы данных. Для наглядности мы старались сохранить небольшой размер и простоту библиотеки, но при этом она поддерживает механизм блокировок, который позволяет нескольким процессам одновременно работать с базой данных.

Мы также проанализировали производительность этой библиотеки при одновременной работе различного количества процессов в случаях отсутствия блокировок, рекомендательных блокировок (крупноблочных и мелкоблочных) и принудительных блокировок. Мы увидели, что для случая с единственным процессом использование рекомендательных блокировок увеличивает общее время на 29 и 59% по сравнению с версией библиотеки, в которой механизм блокировок не используется, а применение принудительных блокировок увеличивает общее время еще на 15% по сравнению с версией, использующей рекомендательные блокировки.

Упражнения

- 20.1 Блокировка в функции `_db_dodelete` выполнена в несколько консервативном стиле. Мы, например, могли бы получить дополнительный прирост производительности при одновременной работе нескольких процессов, если бы устанавливали блокировку для записи на список свободных записей только тогда, когда это действительно необходимо, — то есть мы могли бы вставить вызов функции `writelock` между вызовами `_db_writedat` и `_db_readptr`. Что произойдет, если сделать это?
- 20.2 Представьте, что `db_nextrec` не устанавливает блокировку для чтения на список свободных записей и что запись, которая была прочитана, одновременно была удалена другим процессом. Опишите, как `db_nextrec` могла бы вернуть корректный ключ и запись с данными, заполненную пробелами (следовательно, неправильную). (Подсказка: загляните в функцию `_db_dodelete`.)
- 20.3 В конце раздела 20.8 мы описали принцип действия блокировок, устанавливаемых в `_db_writeidx` и `_db_writedat`. Мы утверждали, что эти блокировки не оказывают влияния на другие читающие или пишущие процессы, за исключением вызовов функции `db_store`. Будет ли истинным это утверждение при использовании принудительных блокировок?
- 20.4 Как бы вы интегрировали функцию `fsync` в эту библиотеку базы данных?
- 20.5 В функции `db_store` мы сначала записываем данные, а потом индекс. Что произойдет, если запись будет производиться в обратном порядке?
- 20.6 Создайте новую базу данных и добавьте в нее несколько записей. Напишите программу, которая просматривала бы все записи с помощью `db_nextrec` и вызывала `_db_hash`, чтобы вычислить хеш каждой записи. Программа должна выводить гистограмму, отражающую количество записей в каждой из цепочек хеш-таблицы. Ответьте на вопрос: насколько равномерное распределение дает хеш-функция, реализованная в `_db_hash`?
- 20.7 Измените библиотеку базы данных так, чтобы количество цепочек в хеш-таблице можно было указать в момент создания базы данных.
- 20.8 Сравните производительность библиотеки базы данных в случаях, когда файлы базы данных находятся: (а) в локальной файловой системе

и (б) в удаленной файловой системе, доступ к которой организован средствами NFS. Будет ли механизм блокировок работать во втором случае?

- 20.9** База данных повторно использует место, занимавшееся удаленными записями, только если размеры буферов ключа и данных новой записи точно совпадают с размерами тех же буферов удаленной записи. Измените библиотеку базы данных так, чтобы при создании новой записи можно было использовать место, занимавшееся удаленными записями, размеры которых превышают запрошенные. Как бы вы изменили формат хранения записей в базе данных для поддержки этой возможности?
- 20.10** После выполнения упражнения 20.9 напишите программу преобразования базы данных из одного формата в другой.

21

Взаимодействие с сетевым принтером

21.1. Введение

В этой главе мы разработаем программу, которая будет взаимодействовать с сетевым принтером. Подобные принтеры могут быть связаны сразу с несколькими компьютерами посредством Ethernet и зачастую поддерживают, наряду с простыми текстовыми файлами, печать файлов в формате PostScript. Для взаимодействия с такими принтерами приложения обычно используют протокол IPP (Internet Printing Protocol — протокол печати через Интернет), хотя некоторые принтеры поддерживают альтернативные протоколы.

Мы опишем две программы: демон спулера (диспетчер очереди) печати, который передает задания печати принтеру, и утилиту, с помощью которой задания для печати передаются демону спулера. Поскольку спулер печати выполняет массу разнообразных действий (взаимодействие с клиентом, взаимодействие с принтером, чтение файлов, сканирование каталогов и пр.), это позволит нам использовать функции, которые были описаны в предыдущих главах. Например, для упрощения архитектуры демона мы будем использовать потоки выполнения (главы 11 и 12), а для взаимодействия между спулером печати и программой, которая передает ему печатаемый файл, и между спулером печати и сетевым принтером — сокеты (глава 16).

21.2. Протокол печати через Интернет

Протокол печати через Интернет определяет правила построения сетевых систем печати. Благодаря наличию сервера IPP, встроенного в сетевую плату, принтер может обслуживать запросы от множества компьютерных систем. Однако совсем необязательно, чтобы эти компьютерные системы физически находились в той же самой сети, что и принтер. Протокол IPP работает поверх стандартных протоколов Интернета (IP), благодаря чему любой компьютер сможет создать TCP/IP-соединение с принтером и передать ему задание для печати.

Протокол IPP определяется целой серией документов (RFC, Requests For Comments — запросы на комментарии), доступных по адресу <http://www.ietf.org/rfc.html>. Предложенные проекты стандартов были разработаны рабочей группой Printer Working Group, относящейся к организации IEEE. Эти проекты доступны

по адресу <http://www.pwg.org/ipp>. Основные документы перечислены в табл. 21.1, хотя существуют и другие документы, определяющие административные процедуры, атрибуты заданий и т. п.

Таблица 21.1. Основные документы RFC, определяющие протокол IPP

RFC	Заголовок
2567	Design Goals for an Internet Printing Protocol – Цели разработки протокола печати через Интернет
2568	Rationale for Structure of the Model and Protocol for the Internet Printing Protocol – Обоснование структурной модели протокола IPP
2911	Internet Printing Protocol/1.1:Model and Semantics – Протокол IPP/1.1:Модель и семантика
2910	Internet Printing Protocol/1.1:Encoding and Transport – Протокол IPP/1.1:Кодировка и передача данных
3196	Internet Printing Protocol/1.1:Implementator's Guide – Протокол IPP/1.1:Руководство разработчика
Предварительный стандарт 5100.12-2011	Internet Printing Protocol Version 2.0, Second Edition – Протокол IPP/2.0, второе издание

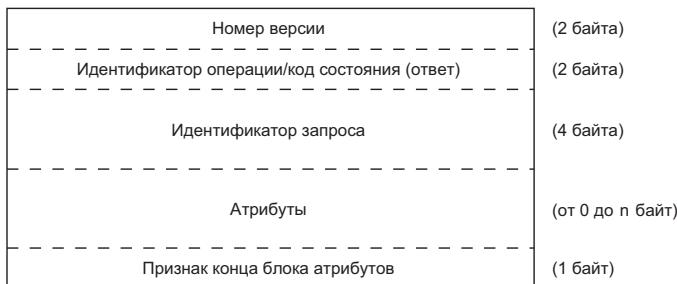
Проект стандарта 5100.12-2011 определяет все особенности, которые должны поддерживаться реализациями, чтобы соответствовать разным версиям стандарта IPP. Существует множество предложенных расширений протокола IPP (особенности, определяемые другими документами, имеющими отношение к IPP). Эти особенности разделены на группы по уровням соответствия; каждый уровень – это отдельная версия протокола. Для совместимости каждый более высокий уровень требует от реализаций соответствия требованиям более низких версий стандарта. При разработке примера в этой главе мы будем опираться на стандарт IPP версии 1.1.

Протокол IPP реализован поверх протокола HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста, раздел 21.3). В свою очередь, протокол HTTP реализован поверх TCP/IP. Структура сообщения протокола IPP показана на рис. 21.1.

Протокол IPP построен по принципу «запрос/ответ». Клиент передает сообщение-запрос серверу, а сервер возвращает сообщение-ответ. В заголовке IPP имеется поле, определяющее запрашиваемую операцию. Возможные операции включают запуск задания печати, отмену задания печати, получение характеристик задания, получение характеристик принтера, приостановку и перезапуск принтера, приостановку задания печати, возобновление приостановленного задания печати.

На рис. 21.2 показана структура заголовка сообщения IPP. Первые 2 байта – это номер версии IPP. Для протокола версии 1.1 в каждом байте хранится число 1. Следующие 2 байта в случае запроса содержат значение, определяющее запрашиваемую операцию. В случае ответа эти 2 байта содержат код статуса.

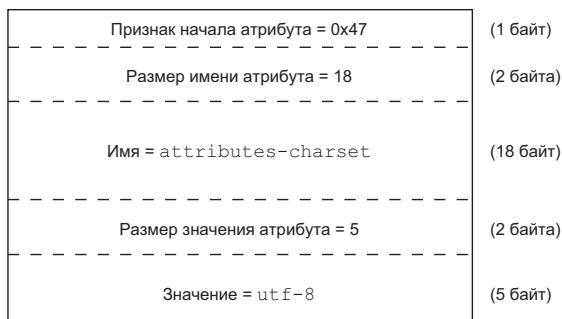
Заголовок Ethernet	Заголовок IP	Заголовок TCP	Заголовок HTTP	Заголовок IPP	Данные для печати
--------------------	--------------	---------------	----------------	---------------	-------------------

Рис. 21.1. Структура сообщения протокола IPP**Рис. 21.2.** Структура заголовка IPP

Следующие 4 байта содержат целочисленный идентификатор запроса, который позволяет сопоставлять запросы и ответы. Далее следуют необязательные атрибуты, завершающиеся признаком конца блока атрибутов. Сразу за блоком атрибутов располагаются данные, которые могут быть связаны с запросом.

Целые числа в заголовке сохраняются со знаком в двоичном формате с прямым (сетевым, big-endian) порядком байтов. Атрибуты хранятся в виде групп. Каждая группа начинается с 1-байтного признака, идентифицирующего группу, за которым следуют 2 байта длины имени атрибута, имя атрибута, 2 байта длины значения атрибута и само значение. Значения атрибутов могут быть представлены в виде строк, целых чисел в двоичном формате или более сложных структур, таких как структуры представления даты и времени.

На рис. 21.3 показано, как в заголовке IPP будет представлен атрибут **attributes-charset** со значением **utf-8**.

**Рис. 21.3.** Пример представления атрибута в заголовке IPP

В зависимости от запрашиваемой операции некоторые атрибуты могут быть обязательными, другие — необязательными. Например, в табл. 21.2 приводятся некоторые атрибуты, сопровождающие запрос на печать задания.

Таблица 21.2. Атрибуты запроса на печать задания

Атрибут	Статус	Описание
<code>attributes charset</code>	Обязательный	Кодировка символов, используемая такими атрибутами, как <code>type</code> или <code>name</code>
<code>attributes natural language</code>	Обязательный	Естественный язык, используемый такими атрибутами, как <code>type</code> или <code>name</code>
<code>printer uri</code>	Обязательный	Универсальный идентификатор ресурса (URI) принтера
<code>requesting user name</code>	Необязательный	Имя пользователя, отправившего задание печати (если поддерживается, используется для аутентификации пользователя)
<code>job name</code>	Необязательный	Имя задания, используемое для идентификации различных заданий
<code>ipp attribute fidelity</code>	Необязательный	Когда имеет истинное значение, принтер должен отвергнуть задание, если получены не все атрибуты, иначе — принтер должен сделать все возможное, чтобы напечатать задание
<code>document name</code>	Необязательный	Название документа (может потребоваться, например, для печати колонтипов)
<code>document format</code>	Необязательный	Формат документа (обычный текст, PostScript и пр.)
<code>document natural language</code>	Необязательный	Естественный язык документа
<code>compression</code>	Необязательный	Алгоритм сжатия документа
<code>job k octets</code>	Необязательный	Размер документа в блоках по 1024 октета
<code>job impressions</code>	Необязательный	Количество отпечатков (фоновых изображений, встраиваемых в страницу), переданных вместе с заданием
<code>job media sheets</code>	Необязательный	Количество листов в задании

Заголовок IPP содержит как текстовые, так и двоичные данные. Имена атрибутов сохраняются в текстовом виде, а их размеры — в виде целых чисел в двоичном представлении. Это усложняет процесс сборки и анализа заголовка, поскольку необходимо постоянно помнить о сетевом порядке байтов и о том, может ли про-

цессор размещать целые числа с произвольного адреса. Было бы лучше, если бы заголовок был разработан так, чтобы все данные в нем хранились только в текстовом представлении. Это упростило бы обработку, хотя и за счет некоторого увеличения размера сообщений.

21.3. Протокол передачи гипертекста

Версия 1.1 протокола HTTP определяется в RFC 2616. Протокол HTTP также работает по принципу «запрос/ответ». Сообщение-запрос содержит начальную строку, за которой следуют строки заголовка, пустая строка и необязательное тело запроса. В нашем случае тело запроса содержит заголовок IPP и данные.

Заголовки HTTP передаются в формате ASCII, где каждая строка завершается символами возврата каретки (\r) и перевода строки (\n). Начальная строка содержит метод выполнения запроса, универсальный адрес ресурса (Uniform Resource Locator, URL), который описывает сервер и протокол, и строку, определяющую версию протокола HTTP. Протокол IPP поддерживает только один метод HTTP для передачи данных серверу — метод POST.

Строки заголовка определяют атрибуты, такие как формат и размер тела запроса. Каждая строка заголовка содержит имя атрибута, далее следуют двоеточие, необязательный пробел и значение атрибута. Завершается строка символами возврата каретки и перевода строки. Например, чтобы указать, что тело содержит сообщение IPP, нужно включить в заголовок строку

`Content-Type: application/ipp`

Ниже приводится пример заголовка HTTP-запроса на печать, отправляемого принтеру Xerox Phaser 8560 автора:

```
POST /ipp HTTP/1.1^M
Content-Length: 21931^M
Content-Type: application/ipp^M
Host: phaser8560:631^M
^M
```

Строка `Content-Length` определяет размер в байтах блока данных внутри HTTP-сообщения. Размер HTTP-заголовка в это число не входит, но входит размер заголовка IPP. Стока `Host` определяет имя хоста и номер порта сервера, которому отправляется сообщение.

Символы ^M в конце каждой строки — это символы возврата каретки, предшествующие символам перевода строки. Перевод строки не отображается как печатный символ. Обратите внимание, что последняя строка заголовка пустая — она содержит только символы возврата каретки и перевода строки.

Начальная строка сообщения-ответа HTTP содержит версию протокола, за которой следуют код статуса и сообщение. Завершается начальная строка символами возврата каретки и перевода строки. Остальная часть сообщения-ответа имеет тот же формат, что и сообщение-запрос: строки заголовка, за которыми следуют пустая строка и необязательное тело сообщения.

В ответ на запрос принтер может вернуть следующее сообщение:

```
HTTP/1.1 200 OK^M
Content-Type: application/ipp^M
Cache-Control: no-cache, no-store, must-revalidate^M
Expires: THU, 26 OCT 1995 00:00:00 GMT^M
Content-Length: 215^M
Server: Allegro-Software-RomPager/4.34^M
^M
```

С точки зрения реализации спулера печати основной интерес для нас представляется только первая строка: она сообщает об успехе или неудаче выполнения запроса посредством числового кода и короткой строки. Оставшаяся часть сообщения содержит дополнительную информацию, управляющую кэшированием ответа на узлах, которые могут находиться между клиентом и сервером, и сведения о версии программного обеспечения, выполняющегося на сервере.

21.4. Очередь печати

Программы, которые мы разработаем в этой главе, представляют собой основу простого спулера (диспетчера очереди) печати. С помощью специальной команды пользователь посыпает файл спулеру принтера, спулер сохраняет его на диск, ставит запрос в очередь и в конечном счете отправляет файл принтеру.

Любая версия UNIX предоставляет по меньшей мере одну систему печати. Так, FreeBSD распространяется вместе с системой LPD (Line Printer Daemon — демон последовательной печати) (см. `lpd(8)` и главу 13 [Stevens, 1990]). Linux и Mac OS X включают систему печати CUPS (Common UNIX Printing System — универсальная система печати в UNIX) (см. `cupsd(8)`). Solaris распространяется со стандартным для System V спулером печати (см. `lp(1)` и `lpsched(1M)`). В данной главе основной интерес представляют не сами эти системы печати, а порядок взаимодействия с сетевым принтером. Нам необходимо разработать свою систему печати, которая будет способна организовать доступ нескольких пользователей к единственному ресурсу (принтеру).

Мы создадим простую утилиту, которая будет читать файл и передавать его демону спулера печати. Утилита будет иметь один параметр — для печати файлов обычного текстового формата (по умолчанию предполагается, что файл имеет формат PostScript). Мы назвали эту утилиту `print`.

Демон спулера печати `printd` будет иметь многопоточную архитектуру, чтобы распределить между потоками работу, которая должна быть выполнена демоном.

- Один поток ожидает поступления через сокет новых запросов от клиентов, запустивших утилиту `print`.
- Для обслуживания каждого клиента порождается отдельный поток, который копирует файл в область очереди печати.
- Один поток взаимодействует с принтером, передавая ему задания из очереди.
- Один поток обслуживает сигналы.

На рис. 21.4 показано, как все эти компоненты связаны друг с другом.

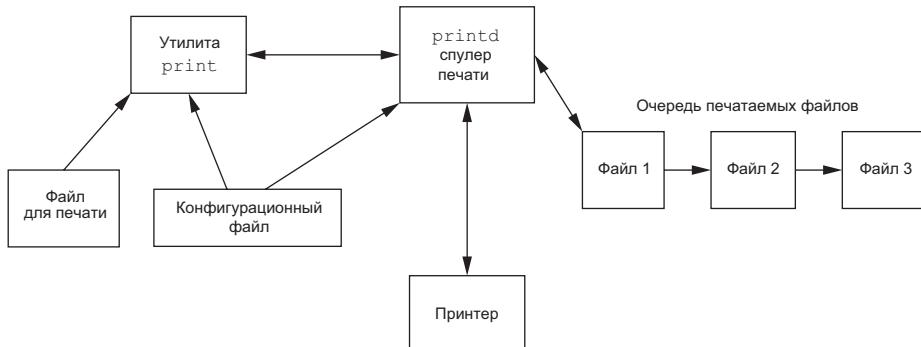


Рис. 21.4. Компоненты системы печати

Конфигурационный файл системы печати называется `/etc/printer.conf`. Он определяет имя сервера, где запущен демон спулера печати, и сетевое имя принтера. Демон спулера идентифицируется строкой, начинающейся с ключевого слова `printserver`, за которым следуют пробел и сетевое имя сервера. Принтер идентифицируется строкой, начинающейся с ключевого слова `printer`, за которым следуют пробел и сетевое имя принтера.

Типичный конфигурационный файл содержит следующие строки:

```
printserver fujin
printer phaser8560
```

где `fujin` — сетевое имя сервера, где запущен демон спулера печати, а `phaser8560` — сетевое имя принтера. Мы будем предполагать, что эти имена перечислены в файле `/etc/hosts` или зарегистрированы в используемой службе имен, чтобы мы могли преобразовывать имена в сетевые адреса.

Команду `print` можно вызвать на том же компьютере, где работает демон спулера печати, или выполнить ее на любом другом компьютере в той же сети. В последнем случае потребуется настроить в файле `/etc/printer.conf` только параметр `printserver`, потому что имя принтера необходимо только демону печати.

Безопасность

Программы, которые работают с привилегиями суперпользователя, потенциально открывают лазейку для нападения. Сами по себе такие программы обычно не более уязвимы, чем любые другие, но в случае обнаружения уязвимостей они могут позволить атакующему получить неограниченный доступ к системе.

Демон печати, который рассматривается в этой главе, запускается с привилегиями суперпользователя, чтобы назначить сокету привилегированный номер порта TCP. Чтобы сделать демон менее уязвимым, мы можем:

- Спроектировать демон в соответствии с принципами минимизации привилегий (раздел 8.11). После назначения сокету привилегированного номера

порта можно заменить идентификаторы пользователя и группы демона на какие-либо другие, отличные от `root` (например, `lp`). Все файлы и каталоги, используемые для хранения заданий, поставленных в очередь печати, должны принадлежать этому непривилегированному пользователю. Благодаря этому обнаружение уязвимости даст атакующему доступ только к подсистеме печати. Это тоже неприятно, но гораздо менее серьезно, чем если бы атакующий получил неограниченный доступ ко всей системе.

- Проверить исходный код демона на наличие всех известных потенциальных уязвимостей, таких как переполнение буфера.
- Журнилировать случаи неожиданного или подозрительного поведения, чтобы впоследствии администратор мог обнаружить их и изучить.

21.5. Исходный код

Исходный код, рассматриваемый в этой главе, содержится в пяти файлах, за исключением некоторых библиотечных функций, которые мы использовали в предыдущих главах:

ipp.h Заголовочный файл с определениями IPP.

print.h Заголовочный файл с константами общего назначения, определениями структур данных и объявлениями служебных процедур.

util.c Служебные процедуры, используемые обеими программами.

print.c Исходные тексты утилиты `print`, используемой для печати файлов.

printd.c Исходные тексты демона спулера печати.

Мы будем исследовать эти файлы именно в таком порядке.

Начнем с заголовочного файла `ipp.h`.

```
1 #ifndef _IPP_H
2 #define _IPP_H

3 /*
4 * Определения протокола IPP, касающиеся взаимодействия между
5 * планировщиком и принтером. Основаны на RFC2911 и RFC2910.
6 */

7 /*
8 * Классы кодов состояния.
9 */
10 #define STATCLASS_OK(x) ((x) >= 0x0000 && (x) <= 0x00ff)
11 #define STATCLASS_INFO(x) ((x) >= 0x0100 && (x) <= 0x01ff)
12 #define STATCLASS_REDIR(x) ((x) >= 0x0200 && (x) <= 0x02ff)
13 #define STATCLASS_CLIERR(x) ((x) >= 0x0400 && (x) <= 0x04ff)
14 #define STATCLASS_SRVERR(x) ((x) >= 0x0500 && (x) <= 0x05ff)

15 /*
16 * Коды состояния.
17 */
18 #define STAT_OK 0x0000 /* успех */
19 #define STAT_OK_ATTRIGN 0x0001 /* OK; некоторые атрибуты игнорируются */
20 #define STAT_OK_ATTRCON 0x0002 /* OK; конфликты между некоторыми атрибутами */

21 #define STAT_CLI_BADREQ 0x0400 /* неверный запрос клиента */
22 #define STAT_CLI_FORBID 0x0401 /* запрещенный запрос */
23 #define STAT_CLI_NOAUTH 0x0402 /* требуется аутентификация */
24 #define STAT_CLI_NOPERM 0x0403 /* клиент не авторизован */
25 #define STAT_CLI_NOTPOS 0x0404 /* невозможно выполнить запрос */
26 #define STAT_CLI_TIMOUT 0x0405 /* истекло время ожидания клиента */
27 #define STAT_CLI_NOTFND 0x0406 /* не найден объект по данному URI */
28 #define STAT_CLI_OBJGONE 0x0407 /* объект больше не доступен */
29 #define STAT_CLI_TOOBIG 0x0408 /* запрошенный объект слишком велик */
30 #define STAT_CLI_TOOLNG 0x0409 /* слишком большое значение атрибута */
31 #define STAT_CLI_BADFMT 0x040a /* неподдерживаемый формат документа */
32 #define STAT_CLI_NOTSUP 0x040b /* неподдерживаемые атрибуты */
33 #define STAT_CLI_NOSCHM 0x040c /* неподдерживаемая схема URI */
34 #define STAT_CLI_NOCHAR 0x040d /* неподдерживаемый набор символов */
35 #define STAT_CLI_ATTRCON 0x040e /* конфликтующие атрибуты */
36 #define STAT_CLI_NOCOMP 0x040f /* сжатие не поддерживается */
37 #define STAT_CLI_COMPERR 0x0410 /* данные не могут быть разжаты */
38 #define STAT_CLI_FMTERR 0x0411 /* ошибка в формате документа */
39 #define STAT_CLI_ACCERR 0x0412 /* ошибка доступа к данным */
```

[1–14] Начинается заголовочный файл со стандартного определения `#ifndef`, чтобы предотвратить возникновение ошибок, связанных с повторным подключением одного и того же заголовочного файла. Далее следуют определения классов кодов статуса IPP (раздел 13 RFC 2911).

[15–39] Мы определяем конкретные коды состояния на основе RFC 2911. Они не будут использоваться в нашей программе, но понадобятся в упражнении 21.1.

```

40 #define STAT_SRV_INTERN 0x0500 /* неожиданная внутренняя ошибка */
41 #define STAT_SRV_NOTSUP 0x0501 /* операция не поддерживается */
42 #define STAT_SRV_UNAVAIL 0x0502 /* услуга недоступна */
43 #define STAT_SRV_BADVER 0x0503 /* неподдерживаемая версия */
44 #define STAT_SRV_DEVERR 0x0504 /* ошибка устройства */
45 #define STAT_SRV_TMPERR 0x0505 /* временная ошибка */
46 #define STAT_SRV_REJECT 0x0506 /* сервер не принял задание */
47 #define STAT_SRV_TOOBUSY 0x0507 /* сервер занят */
48 #define STAT_SRV_CANCEL 0x0508 /* задание было отменено */
49 #define STAT_SRV_NOMULTI 0x0509 /* задания из нескольких документов */
/* не поддерживается */
50 /*
51 * Идентификаторы операций.
52 */
53 #define OP_PRINT_JOB 0x02
54 #define OP_PRINT_URI 0x03
55 #define OP_VALIDATE_JOB 0x04
56 #define OP_CREATE_JOB 0x05
57 #define OP_SEND_DOC 0x06
58 #define OP_SEND_URI 0x07
59 #define OP_CANCEL_JOB 0x08
60 #define OP_GET_JOB_ATTR 0x09
61 #define OP_GET_JOBS 0x0a
62 #define OP_GET_PRINTER_ATTR 0x0b
63 #define OP_HOLD_JOB 0x0c
64 #define OP_RELEASE_JOB 0x0d
65 #define OP_RESTART_JOB 0x0e
66 #define OP_PAUSE_PRINTER 0x10
67 #define OP_RESUME_PRINTER 0x11
68 #define OP_PURGE_JOBS 0x12

69 /*
70 * Признаки атрибутов.
71 */
72 #define TAG_OPERATION_ATTR 0x01 /* признак атрибутов операции */
73 #define TAG_JOB_ATTR 0x02 /* признак атрибутов задания */
74 #define TAG_END_OF_ATTR 0x03 /* признак конца списка атрибутов */
75 #define TAG_PRINTER_ATTR 0x04 /* признак атрибутов принтера */
76 #define TAG_UNSUPP_ATTR 0x05 /* признак неподдерживаемых атрибутов */

```

[40–49] Продолжение определений кодов состояния. Коды в диапазоне от `0x500` до `0x5ff` являются кодами ошибок сервера. Описание всех кодов вы найдете в разделах с 13.1.1 по 13.1.5 RFC 2911.

[50–68] Далее мы определяем идентификаторы различных операций. Каждой выполняемой задаче, определяемой протоколом IPP, соответствует свой идентификатор (раздел 4.4.15 RFC 2911). В нашем примере мы будем использовать только операцию `OP_PRINT_JOB`.

[69–76] Признаки атрибутов, разделяющих группы атрибутов в сообщениях протокола IPP. Значения признаков определены в разделе 3.5.1 «RFC 2910».

```
77 /*  
78 * Значения признаков.  
79 */  
80 #define TAG_UNSUPPORTED 0x10 /* неподдерживаемое значение */  
81 #define TAG_UNKNOWN 0x12 /* неизвестное значение */  
82 #define TAG_NONE 0x13 /* нет значения */  
83 #define TAG_INTEGER 0x21 /* целое */  
84 #define TAG_BOOLEAN 0x22 /* булево */  
85 #define TAG_ENUM 0x23 /* перечисление */  
86 #define TAG_OCTSTR 0x30 /* строка октетов */  
87 #define TAG_DATETIME 0x31 /* дата и время */  
88 #define TAG_RESOLUTION 0x32 /* разрешающая способность */  
89 #define TAG_INTRANGE 0x33 /* диапазон целых чисел */  
90 #define TAG_TEXTWLANG 0x35 /* текст с признаком языка */  
91 #define TAG_NAMEWLANG 0x36 /* имя с признаком языка */  
92 #define TAG_TEXTWOLANG 0x41 /* текст */  
93 #define TAG_NAMEWOLANG 0x42 /* имя */  
94 #define TAG_KEYWORD 0x44 /* ключевое слово */  
95 #define TAG_URI 0x45 /* URI */  
96 #define TAG_URISHEME 0x46 /* схема URI */  
97 #define TAG_CHARSET 0x47 /* кодировка символов */  
98 #define TAG_NATULANG 0x48 /* естественный язык */  
99 #define TAG_MIMETYPE 0x49 /* тип MIME */  
  
100 struct ipp_hdr {  
101 int8_t major_version; /* всегда 1 */  
102 int8_t minor_version; /* всегда 1 */  
103 union {  
104 int16_t op; /* идентификатор операции */  
105 int16_t st; /* статус */  
106 } u;  
107 int32_t request_id; /* идентификатор запроса */  
108 char attr_group[1]; /* начало группы optionalных атрибутов */  
109 /* далее могут следовать дополнительные данные */  
110 };  
  
111 #define operation u.op  
112 #define status u.st  
  
113 #endif /* _IPP_H */
```

[77–99] Значения признаков определяют формат отдельных атрибутов и параметров. Они определены в разделе 3.5.2 RFC 2910.

[100–113] Определение структуры заголовка IPP. Сообщения-запросы и сообщения-ответы имеют одинаковую структуру заголовка, за исключением идентификатора операции, который в сообщении-ответе замещается кодом состояния.

В конце заголовочного файла находится закрывающий `#endif`, который соответствует директиве `#ifndef`, расположенной в начале заголовочного файла.

Далее следует заголовочный файл `print.h`.

```
1 #ifndef _PRINT_H
2 #define _PRINT_H

3 /*
4 * Заголовочный файл сервера печати.
5 */
6 #include <sys/socket.h>
7 #include <arpa/inet.h>
8 #include <netdb.h>
9 #include <errno.h>

10 #define CONFIG_FILE "/etc/printer.conf"
11 #define SPOOLDIR "/var/spool/printer"
12 #define JOBFILE "jobno"
13 #define DATADIR "data"
14 #define REQDIR "reqs"

15 #if defined(BSD)
16 #define LPNAME "daemon"
17 #elif defined(MACOS)
18 #define LPNAME "_lp"
19 #else
20 #define LPNAME "lp"
21 #endif
```

[1–9] Подключаются все заголовочные файлы, которые могут потребоваться приложению. Приложения могут просто подключать файл `print.h`, что облегчает отслеживание всех зависимостей заголовочных файлов.

[10–14] Определяются файлы и каталоги, используемые в данной реализации. Настройки с именами хостов демона очереди печати и сетевого принтера хранятся в файле `/etc/printer.conf`. Копии печатаемых файлов сохраняются в каталоге `/var/spool/printer/data`, управляющая информация по каждому из запросов — в каталоге `/var/spool/printer/reqs`. Файл, в котором хранится номер следующего задания печати, — `/var/spool/printer/jobno`.

Каталоги создаваться администратором, а их владельцем должен быть пользователь, с привилегиями которого будет выполняться демон печати. Демон не будет пытаться создавать эти каталоги самостоятельно, потому что для создания подкаталогов в каталоге `/var/spool` могут потребоваться привилегии суперпользователя. При создании демона мы будем следовать принципу минимизации привилегий, чтобы снизить риск создания брешей в системе безопасности.

[15–21] Далее следуют определения учетной записи, с привилегиями которой будет выполнятся демон печати. В Linux и Solaris — это `lp`. В Mac OS X — `_lp`. Однако в FreeBSD отсутствует отдельная учетная запись для подсистемы печати, поэтому мы будем использовать учетную запись, зарезервированную для системных демонов.

```
22 #define FILENMSZ 64
23 #define FILEPERM (S_IRUSR|S_IWUSR)

24 #define USERNM_MAX 64
25 #define JOBNM_MAX 256
26 #define MSGLEN_MAX 512

27 #ifndef HOST_NAME_MAX
28 #define HOST_NAME_MAX 256
29 #endif

30 #define IPP_PORT 631
31 #define QLEN 10

32 #define IBUFSZ 512 /* размер буфера для заголовка IPP */
33 #define HBUFSZ 512 /* размер буфера для заголовка HTTP */
34 #define IOBUFSZ 8192 /* размер буфера для данных */

35 #ifndef ETIME
36 #define ETIME ETIMEDOUT
37 #endif

38 extern int getaddrlist(const char *, const char *,
39 struct addrinfo **);
40 extern char *get_printserver(void);
41 extern struct addrinfo *get_printaddr(void);
42 extern ssize_t tread(int, void *, size_t, unsigned int);
43 extern ssize_t treadn(int, void *, size_t, unsigned int);
44 extern int connect_retry(int, int, int, const struct sockaddr *,
45 socklen_t);
46 extern int initserver(int, const struct sockaddr *, socklen_t,
47 int);
```

[22–34] Далее следуют определения констант и пределов. При создании копий файлов, переданных для печати, им присваиваются права доступа FILEPERM. Права доступа к копиям файлов ограничены, потому что мы хотим предотвратить доступ других пользователей к этим файлам, пока они ожидают вывода на принтер. Константа HOST_NAME_MAX определяет максимально допустимую длину имени хоста на случай, если нам не удастся определить это ограничение с помощью функции `sysconf`.

Порт 631 используется протоколом IPP по умолчанию. Константа QLEN определяет значение аргумента `backlog` функции `listen` (раздел 16.4).

[35–37] Некоторые платформы не определяют код ошибки `ETIME`, поэтому мы сами определяем эту константу для использования на таких платформах. Этот код ошибки будет возвращаться по истечении тайм-аута чтения (сервер не должен блокироваться «навечно» в операции чтения из сокета).

[38–47] Далее определяются все общедоступные функции, которые содержатся в файле `util.c` (он следует чуть ниже). Обратите внимание, что функции `connect_retry` из листинга 16.2 и `initserver` из листинга 16.9 не включены в файл `util.c`.

```
48 /*
49 * Структура, описывающая запрос утилиты print.
50 */
51 struct printreq {
52     uint32_t size; /* размер в байтах */
53     uint32_t flags; /* см. ниже */
54     char usernm[USERNM_MAX]; /* имя пользователя */
55     char jobnm[JOBNM_MAX]; /* имя задания */
56 };

57 /*
58 * Флаги запроса.
59 */
60 #define PR_TEXT 0x01 /* интерпретировать файл как обычный текст */

61 /*
62 * Ответ демона на запрос утилиты print.
63 */
64 struct printresp {
65     uint32_t retcode; /* 0=успех, !0=код ошибки */
66     uint32_t jobid; /* идентификатор задания */
67     char msg[MSGLEN_MAX]; /* сообщение об ошибке */
68 };

69 #endif /* _PRINT_H */
```

[48–69] Структуры `printreq` и `printresp` определяют протокол взаимодействия между утилитой `print` и демоном печати. Утилита `print` отправляет структуру `printreq`, в которой определены имя пользователя, имя задания и размер файла. Демон отвечает структурой `printresp`, содержащей возвращаемый код, идентификатор задания и текст сообщения об ошибке в случае невозможности выполнить запрос.

Характеристика `PR_TEXT` задания печати указывает, что содержимое печатаемого файла должно интерпретироваться как обычный текст (а не как содержимое в формате PostScript). Мы решили определять флаги в виде битовой маски. На текущий момент определен только один флаг, но в будущем нам может потребоваться расширить протокол обмена дополнительными характеристиками. Например, можно было бы добавить флаг двусторонней печати. Мы можем добавить еще 31 дополнительный флаг, не изменяя размер структуры. Изменение размера структуры подразумевает появление проблем совместимости между клиентом и сервером, для устранения которых требуется одновременное их обновление. Альтернативное решение заключается в том, чтобы добавить поле с номером версии, тем самым обеспечив возможность изменения структуры между версиями.

Обратите внимание, что все целочисленные поля в структурах протокола имеют типы, описывающие точный размер. Это поможет избежать конфликтов при обмене данными между клиентом и сервером, где длинные целые числа имеют разное число разрядов.

Далее следует файл `util.c`, содержащий служебные функции.

```
1 #include "apue.h"
2 #include "print.h"
3 #include <ctype.h>
4 #include <sys/select.h>

5 #define MAXCFGLINE 512
6 #define MAXKWLEN 16
7 #define MAXFMTLEN 16

8 /*
9 * Получает перечень адресов для заданного хоста и службы и возвращает
10 * его в ailistpp. Возвращает 0 в случае успеха или ненулевое значение
11 * в случае ошибки (код ошибки). Обратите внимание: код ошибки
12 * не записывается в переменную errno.
13 *
14 * БЛОКИРОВКИ: отсутствуют.
15 */
16 int
17 getaddrlist(const char *host, const char *service,
18 struct addrinfo **ailistpp)
19 {
20 int err;
21 struct addrinfo hint;

22 hint.ai_flags = AI_CANONNAME;
23 hint.ai_family = AF_INET;
24 hint.ai_socktype = SOCK_STREAM;
25 hint.ai_protocol = 0;
26 hint.ai_addrlen = 0;
27 hint.ai_canonname = NULL;
28 hint.ai_addr = NULL;
29 hint.ai_next = NULL;
30 err = getaddrinfo(host, service, &hint, ailistpp);
31 return(err);
32 }
```

[1–7] Прежде всего мы устанавливаем пределы, необходимые для работы функций из этого файла. Константа `MAXCFGLINE` определяет максимальный размер строки конфигурационного файла, `MAXKWLEN` — максимальный размер ключевого слова в конфигурационном файле, `MAXFMTLEN` — максимальный размер строки формата, которая передается функции `sscanf`.

[8–32] Первая функция в файле — `getaddrlist`. Это обертка вокруг `getaddrinfo` (раздел 16.3.3). Мы написали ее потому, что всегда будем вызывать `getaddrinfo` с одними и теми же значениями полей структуры `hint`. Обратите внимание: в этой функции не требуется использовать мьютексы. Комментарий **БЛОКИРОВКИ** в начале каждой функции описывает используемые блокировки. В нем перечисляются предположения, касающиеся блокировок (если таковые имеются), и блокировки, которые должны быть установлены или сняты функцией, а также блокировки, которые должны быть установлены перед ее вызовом.

```

33 /*
34 * Ищет заданное ключевое слово в конфигурационном файле
35 * и возвращает строку, соответствующую этому ключевому слову.
36 *
37 * БЛОКИРОВКИ: отсутствуют.
38 */
39 static char *
40 scan_configfile(char *keyword)
41 {
42 int n, match;
43 FILE *fp;
44 char keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45 char line[MAXCFGLINE];
46 static char valbuf[MAXCFGLINE];

47 if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48 log_sys("невозможно открыть %s", CONFIG_FILE);
49 sprintf(pattern, "%%%ds %%%ds", MAXKWLEN1, MAXCFGLINE1);
50 match = 0;
51 while (fgets(line, MAXCFGLINE, fp) != NULL) {
52 n = sscanf(line, pattern, keybuf, valbuf);
53 if (n == 2 && strcmp(keyword, keybuf) == 0) {
54 match = 1;
55 break;
56 }
57 }
58 fclose(fp);
59 if (match != 0)
60 return(valbuf);
61 else
62 return(NULL);
63 }

```

[33–46] Функция `scan_configfile` отыскивает в конфигурационном файле заданное ключевое слово.

[47–63] Мы открываем конфигурационный файл для чтения и строим строку формата, которая соответствует шаблону поиска. Нотация `%%%ds` создает спецификатор формата, который ограничивает размер строки, благодаря чему можно не опасаться ошибки переполнения буфера, размещаемого на стеке. Мы читаем строки из файла по одной и выделяем из них две подстроки, разделенные пробелами. Если они найдены, сравниваем первую подстроку с заданным ключевым словом. В случае совпадения или достижения конца файла цикл завершается и мы закрываем файл. Если найдено совпадение с заданным ключевым словом, возвращается указатель на буфер, содержащий вторую подстроку, расположенную после ключевого слова, иначе возвращается `NULL`.

Возвращаемая подстрока сохраняется в статическом буфере (`valbuf`), который может перезаписываться при успешных вызовах функции. Поэтому функцию `scan_configfile` нельзя использовать в многопоточных приложениях, если не позаботиться, чтобы ее вызов из нескольких потоков одновременно был невозможен.

```
64 /*
65 * Возвращает имя хоста демона печати или NULL в случае ошибки
66 *
67 * БЛОКИРОВКИ: отсутствуют.
68 */
69 char *
70 get_printserver(void)
71 {
72 return(scan_configfile("printserver"));
73 }

74 /*
75 * Возвращает адрес сетевого принтера или NULL в случае ошибки.
76 *
77 * БЛОКИРОВКИ: отсутствуют.
78 */
79 struct addrinfo *
80 get_printaddr(void)
81 {
82 int err;
83 char *p;
84 struct addrinfo *ailist;

85 if ((p = scan_configfile("printer")) != NULL) {
86 if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87 log_msg("нет сведений об адресе %s", p);
88 return(NULL);
89 }
90 return(ailist);
91 }
92 log_msg("не задан адрес принтера");
93 return(NULL);
94 }
```

[64–73] Функция `get_printserver` — это просто функция-обертка, которая вызывает `scan_configfile`, чтобы отыскать имя системы, в которой работает демон печати.

[74–94] Функция `get_printaddr` возвращает адрес сетевого принтера. Она похожа на предыдущую функцию, но, в отличие от нее, не просто извлекает имя принтера из конфигурационного файла, а использует его для получения сетевого адреса принтера.

Обе функции, `get_printserver` и `get_printaddr`, вызывают `scan_configfile`, которая, не сумев открыть конфигурационный файл, вызывает `log_sys`, чтобы вывести сообщение об ошибке, и завершается. Хотя функция `get_printserver` предназначена для использования утилитой `print`, а `get_printaddr` — демоном печати, вызов `log_sys` в обоих случаях можно считать вполне нормальным, поскольку мы можем простым изменением глобальной переменной заставить функции журналирования выводить сообщения не в файл журнала, а на стандартное устройство вывода сообщений об ошибках.

```
95 /*
96 * Ограниченнная по времени операция чтения – тайм-аут задается в секундах
97 * (5-й аргумент функции select, который определяет предельное время
98 * ожидания данных). Возвращает количество прочитанных байтов или
99 * -1 (ошибка).
100 * БЛОКИРОВКИ: отсутствуют.
101 */
102 ssize_t
103 tread(int fd, void *buf, size_t nbytes, unsigned int timeout)
104 {
105     int nfds;
106     fd_set readfds;
107     struct timeval tv;

108     tv.tv_sec = timeout;
109     tv.tv_usec = 0;
110     FD_ZERO(&readfds);
111     FD_SET(fd, &readfds);
112     nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113     if (nfds <= 0) {
114         if (nfds == 0)
115             errno = ETIME;
116         return(-1);
117     }
118     return(read(fd, buf, nbytes));
119 }
```

[95–107] Функция `tread` читает заданное количество байтов, но блокирует вызывающий процесс не более чем на `timeout` секунд. Эта функция удобна для чтения данных из сокета или неименованного канала. Если в течение тайм-аута данные так и не поступили, возвращается значение `-1` и код ошибки `ETIME` в переменной `errno`. Если данные стали доступны, возвращается до `nbytes` байт данных, но `read` может вернуть меньше байтов, чем запрошено, если не все данные пришли вовремя.

Мы будем использовать функцию `tread` для предотвращения атак типа «отказ в обслуживании» (denial-of-service – DOS) на демона печати. Злоумышленник мог бы непрерывно пытаться подключиться к демону, не передавая ему никаких данных, что лишило бы остальных пользователей возможности передать демону свои задания печати. Установив предел времени ожидания, мы исключаем возможность возникновения таких ситуаций. Сложность состоит в том, чтобы правильно подобрать значение этого предела, которое должно быть достаточно большим, чтобы предотвратить возможность преждевременной потери запросов при высокой нагрузке на систему, когда для выполнения задач требуется больше времени. Однако, выбрав слишком большое значение тайм-аута, мы рискуем подвергнуться атакам типа «отказ в обслуживании», позволяя демону захватить слишком много ресурсов для обслуживания ожидающих обработки запросов.

[108–119] Мы ожидаем, когда заданный дескриптор станет доступен для чтения, используя функцию `select`. Если время тайм-аута истечет раньше, чем появятся доступные для чтения данные, функция `select` вернет значение `0`, в этом случае в переменную `errno` запишется значение `ETIME`. По истечении тайм-аута или в случае ошибки функции `select` возвращается значение `-1`. Иначе возвращаются данные, которые удалось прочитать.

```
120 /*
121 * Ограниченнная по времени операция чтения – тайм-аут задается в секундах
122 * на каждый вызов read функция пытается прочитать nbytes байт.
123 * Возвращает количество прочитанных байтов или -1 в случае ошибки.
124 *
125 * БЛОКИРОВКИ: отсутствуют.
126 */
127 ssize_t
128 treadn(int fd, void *buf, size_t nbytes, unsigned int timeout)
129 {
130     size_t nleft;
131     ssize_t nread;

132     nleft = nbytes;
133     while (nleft > 0) {
134         if ((nread = tread(fd, buf, nleft, timeout)) < 0) {
135             if (nleft == nbytes)
136                 return(-1); /* ошибка, вернуть -1 */
137             else
138                 break; /* ошибка, вернуть то, что удалось прочитать */
139         } else if (nread == 0) {
140             break; /* конец файла */
141         }
142         nleft -= nread;
143         buf += nread;
144     }
145     return(nbytes - nleft); /* вернуть значение >= 0 */
146 }
```

[120–146] Мы реализовали еще одну версию функции `tread`, которую назвали `treadn`. Эта функция пытается прочитать точно запрошенное количество байтов. Она напоминает функцию `readn`, описанную в разделе 14.7, но имеет дополнительный аргумент, в котором задается время тайм-аута.

Чтобы прочитать заданное количество байтов, может понадобиться несколько раз вызвать функцию `read`. Сложность заключается в использовании единого времени тайм-аута для всех вызовов `read`. Мы не хотели использовать таймер, поскольку обслуживать сигналы в многопоточных приложениях достаточно сложно. Кроме того, мы не можем полагаться на то, что система обновит содержимое структуры `timeval` при выходе из функции `select`, чтобы показать время, оставшееся до истечения тайм-аута, так как многие платформы не поддерживают эту возможность (раздел 14.4.1). Поэтому мы пошли на компромисс и определили значение тайм-аута для каждого отдельного вызова `read`. Вместо ограничения общего времени ожидания мы ограничили время ожидания в каждой итерации цикла. Максимально возможное время ожидания ограничено значением $nbytes \times timeout$ секунд (в худшем случае мы будем получать не более 1 байта за раз).

Переменная `nleft` используется для хранения количества байтов, которое осталось прочитать. Если функция `tread` терпит неудачу, но на предыдущих итерациях удалось прочитать некоторый объем данных, мы прерываем цикл `while` и возвращаем то, что удалось прочитать, иначе возвращается `-1`.

Далее следуют исходные тексты утилиты `print`, которая используется для передачи задания печати. Файл с исходным кодом на С называется `print.c`.

```
1 /*
2 * Утилита печати документов. Открывает файл и отправляет демону печати.
3 * Использование:
4 * print [-t] filename
5 */
6 #include "apue.h"
7 #include "print.h"
8 #include <fcntl.h>
9 #include <pwd.h>

10 /*
11 * Необходимо для функций журналирования.
12 */
13 int log_to_stderr = 1;
14 void submit_file(int, int, const char *, size_t, int);

15 int
16 main(int argc, char *argv[])
17 {
18 int fd, sockfd, err, text, c;
19 struct stat sbuf;
20 char *host;
21 struct addrinfo *ailist, *aip;

22 err = 0;
23 text = 0;
24 while ((c = getopt(argc, argv, "t")) != 1) {
25 switch (c) {
26 case 't':
27 text = 1;
28 break;
29 case '?':
30 err = 1;
31 break;
32 }
33 }
```

[1–14] Мы определяем целочисленную переменную `log_to_stderr`, чтобы иметь возможность использовать в нашей библиотеке функции журналирования. Если переменная имеет ненулевое значение, сообщения об ошибках будут выводиться в стандартное устройство вывода сообщений об ошибках, а не в файл журнала. Хотя в файле `print.c` не используются функции журналирования, но при сборке выполняемого файла `print` мы связываем `print.o` и `util.o`, а `util.c` содержит функции как для сервера, так и для клиента.

[15–33] Поддерживается единственный параметр `-t`, с помощью которого мы указываем, что файл должен печататься как обычный текст (а не как PostScript, например). Для обработки параметров командной строки используется функция `getopt` (раздел 17.6).

```
34 if (err || (optind != argc - 1))
35 err_quit("Использование: print [-t] filename");
36 if ((fd = open(argv[optind], O_RDONLY)) < 0)
37 err_sys("print: невозможно открыть %s", argv[optind]);
38 if (fstat(fd, &sbuf) < 0)
39 err_sys("print: невозможно получить сведения о %s", argv[optind]);
40 if (!S_ISREG(sbuf.st_mode))
41 err_quit("print: %s должен быть обычным файлом\n", argv[optind]);

42 /*
43 * Получить имя хоста, который выступает в роли сервера печати.
44 */
45 if ((host = get_printserver()) == NULL)
46 err_quit("print: сервер печати не определен");
47 if ((err = getaddrlist(host, "print", &ailist)) != 0)
48 err_quit("print: ошибка getaddrinfo: %s", gai_strerror(err));

49 for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50 if ((sfd = connect_retry(AF_INET, SOCK_STREAM, 0,
51 aip->ai_addr, aip->ai_addrlen)) < 0) {
52 err = errno;
```

[34–41] Когда функция `getopt` заканчивает обработку списка аргументов, она записывает в переменную `optind` индекс первого обязательного аргумента. Если это значение будет отличаться от индекса последнего аргумента, следовательно, программа получила неверное количество аргументов (поддерживается только один обязательный аргумент). Обработка ошибок включает проверку возможности открытия файла, отправляемого на печать, и проверку, является ли он обычным файлом (то есть не каталогом или файлом какого-либо другого типа).

[42–48] Получаем имя хоста, где выполняется демон печати, вызовом функции `get_printserver` из `util.c` и затем преобразуем его в сетевой адрес вызовом функции `getaddrlist` (также из файла `util.c`).

Обратите внимание: мы определили имя службы как «`print`». При установке демона печати необходимо убедиться, что в `/etc/services` (или эквивалентной базе данных) имеется запись, соответствующая службе печати. При выборе номера порта для демона мы приняли правильное решение, взяв номер порта из привилегированного диапазона. Тем самым мы лишили потенциального злоумышленника возможности написать свою программу, имитирующую поведение демона печати, чтобы перехватывать копии файлов, отправляемых на печать. Это означает, что номер порта должен быть меньше 1024 (раздел 16.3.4) и демон должен запускаться с привилегиями суперпользователя, чтобы иметь возможность связать сокет с привилегированным номером порта.

[49–52] Мы пытаемся соединиться с демоном, используя поочередно адреса из списка, возвращаемого функцией `getaddrinfo`. Для передачи файла будет использоваться первый адрес, с которым нам удастся установить соединение.

```
53 } else {
54 submit_file(fd, sfd, argv[optind], sbuf.st_size, text);
55 exit(0);
56 }
57 }
58 err_exit(err, "print: невозможно соединиться с %s", host);
59 }

60 /*
61 * Отправить файл демону печати.
62 */
63 void
64 submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
65 int text)
66 {
67 int nr, nw, len;
68 struct passwd *pwd;
69 struct printreq req;
70 struct printresp res;
71 char buf[IOBUFSZ];

72 /*
73 * Сначала соберем заголовок.
74 */
75 if ((pwd = getpwuid(geteuid()) == NULL) {
76 strcpy(req.usernm, "unknown");
77 } else {
78 strncpy(req.usernm, pwd->pw_name, USERNM_MAX-1);
79 req.usernm[USERNM_MAX-1] = '\0';
80 }
```

[53–59] Если удалось установить соединение, мы отправляем файл демону печати с помощью функции `submit_file` и выходим с кодом 0, свидетельствующим об успехе. Если установить соединение не удалось, мы выводим сообщение об ошибке вызовом `err_exit` и выходим с кодом 1, чтобы сообщить об ошибке. (Исходный код функции `err_exit` и других функций вывода сообщений об ошибках можно найти в приложении B.)

[60–80] Функция `submit_file` отправляет запрос на печать демону и получает от него ответ. Для начала мы собираем заголовок запроса `printreq`. С помощью функции `getuid` мы определяем эффективный идентификатор пользователя, который затем передаем функции `getpwuid`, чтобы отыскать имя пользователя в файле паролей. Далее мы копируем полученное имя пользователя в заголовок запроса или, если идентифицировать пользователя не удалось, записываем в заголовок строку `unknown` (неизвестен). Для копирования имени пользователя в заголовок запроса используется функция `strncpy`, чтобы избежать ошибки переполнения буфера. Если имя оказывается длиннее размера буфера, функция `strncpy` не запишет в буфер завершающий нулевой символ, поэтому мы делаем это вручную.

```
81 req.size = htonl(nbytes);

82 if (text)
83 req.flags = htonl(PR_TEXT);
84 else
85 req.flags = 0;

86 if ((len = strlen(fname)) >= JOBNM_MAX) {
87 /*
88 * Усечь имя файла (с учетом 5 символов, отводимых под
89 * четыре символа префикса и завершающий нулевой символ).
90 */
91 strcpy(req.jobnm, "... ");
92 strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
93 } else {
94 strcpy(req.jobnm, fname);
95 }

96 /*
97 * Отправить заголовок серверу.
98 */
99 nw = writen(sockfd, &req, sizeof(struct printreq));
100 if (nw != sizeof(struct printreq)) {
101 if (nw < 0)
102 err_sys("невозможно передать запрос серверу");
103 else
104 err_quit("запрос серверу был передан не полностью (%d/%d)",
105 nw, sizeof(struct printreq));
106 }
```

[81–95] Далее мы записываем в заголовок размер отправляемого файла, попутно преобразуя его в значение с сетевым порядком байтов. То же самое делается с флагом `PR_TEXT`, если файл должен печататься как простой текст. Благодаря преобразованию целых чисел в сетевой порядок байтов, мы обеспечиваем безошибочное взаимодействие клиента и сервера, выполняющихся в разных системах, с аппаратными архитектурами, имеющими разный порядок байтов. (Порядок байтов обсуждался в разделе 16.3.1.)

Из имени печатаемого файла мы собираем имя задания. Если имя файла длиннее, чем может вместить сообщение, мы усекаем его, а первые четыре символа замещаем многоточием, чтобы показать, что имя файла целиком не уместилось в поле структуры.

[96–106] После этого мы отправляем заголовок запроса демону с помощью функции `writen`. (Исходный код функции `writen` приводится в листинге 14.9.) При необходимости функция `writen` многократно вызывает `write`, чтобы записать указанный объем данных. Если попытка записи не удалась или объем записанных данных оказался меньше указанного объема, выводится сообщение об ошибке и работа программы завершается.

```
107 /*
108 * Теперь отправить файл.
109 */
110 while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
111 nw = writen(sockfd, buf, nr);
112 if (nw != nr) {
113 if (nw < 0)
114 err_sys("невозможно отправить файл серверу");
115 else
116 err_quit("файл серверу был передан не полностью (%d/%d)",
117 nw, nr);
118 }
119 }

120 /*
121 * Прочитать ответ.
122 */
123 if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
124 sizeof(struct printresp))
125 err_sys("невозможно прочитать ответ сервера");
126 if (res.retcode != 0) {
127 printf("запрос отвергнут: %s\n", res.msg);
128 exit(1);
129 } else {
130 printf("идентификатор задания %ld\n", (long)ntohl(res.jobid));
131 }
132 }
```

[107–119] После передачи заголовка мы отправляем демону файл, который должен быть напечатан. Мы читаем файл блоками по `IOBUFSZ` байт и отправляем их демону с помощью функции `writen`. Как и в случае с передачей заголовка, если какая-либо операция записи завершилась неудачей или объем записанных данных оказался меньше требуемого, мы выводим сообщение об ошибке и завершаем работу программы.

[120–132] После отправки файла серверу печати мы читаем ответ сервера. Если запрос отвергнут, возвращаемый код (`retcode`) будет не равен нулю, поэтому мы выводим текстовое сообщение об ошибке, включенное в ответ. Если запрос был благополучно принят сервером, мы выводим идентификатор задания на печать, чтобы пользователь знал, как ссылаться на запрос. (Мы оставляем реализацию утилиты, с помощью которой можно отменить запрос на печать, в качестве самостоятельного упражнения; в этом случае для нужд идентификации задания, удаляемого из очереди печати, может использоваться его идентификатор. См. раздел 21.5.) Когда `submit_file` вернется в функцию `main`, мы завершаем выполнение с признаком успеха.

Обратите внимание: прием задания сервером еще не означает, что принтер сможет напечатать файл. Это лишь означает, что демон благополучно добавил задание в очередь печати.

На этом мы завершаем обзор утилиты `print`. Последний файл, который мы рассмотрим, содержит исходный код демона печати на языке C.

```
1 /*
2 * Демон сервера печати.
3 */
4 #include "apue.h"
5 #include <fcntl.h>
6 #include <dirent.h>
7 #include <ctype.h>
8 #include <pwd.h>
9 #include <pthread.h>
10 #include <strings.h>
11 #include <sys/select.h>
12 #include <sys/uio.h>

13 #include "print.h"
14 #include "ipp.h"

15 /*
16 * Ответы принтера по протоколу HTTP.
17 */
18 #define HTTP_INFO(x) ((x) >= 100 && (x) <= 199)
19 #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20 /*
21 * Описание заданий для печати.
22 */
23 struct job {
24 struct job *next; /* следующее задание в списке */
25 struct job *prev; /* предыдущее задание в списке */
26 int32_t jobid; /* идентификатор задания */
27 struct printreq req; /* копия запроса на печать */
28 };

29 /*
30 * Описание потока, обрабатывающего запрос от клиента.
31 */
32 struct worker_thread {
33 struct worker_thread *next; /* следующее описание в списке */
34 struct worker_thread *prev; /* предыдущее описание в списке */
35 pthread_t tid; /* идентификатор потока */
36 int sockfd; /* сокет */
37 };
```

[1–19] Демон печати подключает описанный ранее заголовочный файл протокола IPP, так как он взаимодействует с принтером по этому протоколу. Макросы `HTTP_INFO` и `HTTP_SUCCESS` описывают коды состояния запроса HTTP (мы уже говорили, что протокол IPP реализован поверх протокола HTTP). Коды состояния HTTP-запросов описываются в разделе 10 в RFC 2616.

[20–37] Структуры `job` и `worker_thread` используются демоном для отслеживания заданий печати и потоков, принявших запросы на печать соответственно.

```
38 /*
39 * Для журналирования.
40 */
41 int log_to_stderr = 0;

42 /*
43 * Переменные, имеющие отношение к принтеру.
44 */
45 struct addrinfo *printer;
46 char *printer_name;
47 pthread_mutex_t configlock = PTHREAD_MUTEX_INITIALIZER;
48 int reread;

49 /*
50 * Переменные, имеющие отношение к потокам.
51 */
52 struct worker_thread *workers;
53 pthread_mutex_t workerlock = PTHREAD_MUTEX_INITIALIZER;
54 sigset_t mask;

55 /*
56 * Переменные, имеющие отношение к заданиям.
57 */
58 struct job *jobhead, *jobtail;
59 int jobfd;
```

[38–41] Наша функция журналирования сообщений требует определения переменной `log_to_stderr`. В эту переменную должно быть записано значение 0, чтобы сообщения выводились в системный журнал, а не в стандартное устройство вывода сообщений об ошибках. В файле `print.c` мы определяли переменную `log_to_stderr` и записывали в нее значение 1, хотя функции журналирования не использовались в утилите `print`. Можно было бы избежать этого, разделив служебные функции на два отдельных файла — один для сервера и один для клиентских приложений.

[42–48] В переменной `printer` хранится сетевой адрес принтера. Сетевое имя принтера хранится в переменной `printer_name`. Мьютекс `configlock` защищает доступ к переменной `reread`, которая указывает демону, что он должен перечитать конфигурационный файл, — например, когда администратор изменил принтер или его сетевой адрес.

[49–54] Далее мы определяем переменные, имеющие отношение к потокам. Переменная `workers` хранит указатель на начало двусвязного списка потоков, принимающих файлы от клиентов. Доступ к этому списку осуществляется под защитой мьютекса `workerlock`. В переменной `mask` хранится маска сигналов, используемая потоками.

[55–59] В переменной `jobhead` хранится указатель на начало, а в переменной `jobtail` — на конец списка заданий, ожидающих обработки. Этот список также является двусвязным, но мы будем добавлять задания в конец списка, поэтому необходимо хранить указатель на конец списка. В случае с рабочими потоками порядок их расположения в списке не имеет значения, поэтому мы можем добавлять сведения о новых потоках в начало списка и указатель на конец списка не нужен. Переменная `jobfd` — это дескриптор файла заданий.

```
60 int32_t nextjob;
61 pthread_mutex_t joblock = PTHREAD_MUTEX_INITIALIZER;
62 pthread_cond_t jobwait = PTHREAD_COND_INITIALIZER;

63 /*
64 * Прототипы функций.
65 */
66 void init_request(void);
67 void init_printer(void);
68 void update_jobno(void);
69 int32_t get_newjobno(void);
70 void add_job(struct printreq *, int32_t);
71 void replace_job(struct job *);
72 void remove_job(struct job *);
73 void build_qonstart(void);
74 void *client_thread(void *);
75 void *printer_thread(void *);
76 void *signal_thread(void *);
77 ssize_t readmore(int, char **, int, int *);
78 int printer_status(int, struct job *);
79 void add_worker(pthread_t, int);
80 void kill_workers(void);
81 void client_cleanup(void *);

82 /*
83 * Главный поток сервера печати. Принимает запросы на соединение
84 * от клиентов и запускает дополнительные потоки для обработки запросов.
85 *
86 * БЛОКИРОВКИ: отсутствуют.
87 */
88 int
89 main(int argc, char *argv[])
90 {
91 pthread_t tid;
92 struct addrinfo *ailist, *aip;
93 int sockfd, err, i, n, maxfd;
94 char *host;
95 fd_set rendezvous, rset;
96 struct sigaction sa;
97 struct passwd *pwdp;
```

[60–62] Переменная `nextjob` — это идентификатор следующего задания, которое будет принято. Мыutex `joblock` служит для защиты доступа к связному списку заданий и к состоянию, представленному переменной состояния `jobwait`.

[63–81] Объявления прототипов функций, которые будут использоваться в этом файле. Разместив прототипы в начале файла, мы можем больше не задумываться, в каком порядке они будут определяться в файле.

[82–97] Функция `main` демона печати выполняет две задачи: инициализирует демон и принимает запросы на соединение от клиентов.

```
98 if (argc != 1)
99 err_quit("Использование: printd");
100 daemonize("printd");

101 sigemptyset(&sa.sa_mask);
102 sa.sa_flags = 0;
103 sa.sa_handler = SIG_IGN;
104 if (sigaction(SIGPIPE, &sa, NULL) < 0)
105 log_sys("ошибка вызова функции sigaction");
106 sigemptyset(&mask);
107 sigaddset(&mask, SIGHUP);
108 sigaddset(&mask, SIGTERM);
109 if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
110 log_sys("ошибка вызова функции pthread_sigmask");

111 n = sysconf(_SC_HOST_NAME_MAX);
112 if (n < 0) /* лучшее, что можно предположить */
113 n = HOST_NAME_MAX;
114 if ((host = malloc(n)) == NULL)
115 log_sys("ошибка вызова функции malloc");
116 if (gethostname(host, n) < 0)
117 log_sys("gethostname error");
118 if ((err = getaddrlist(host, "print", &ailist)) != 0) {
119 log_quit("ошибка вызова getaddrinfo: %s", gai_strerror(err));
120 exit(1);
121 }
```

[98–100] Демон не принимает аргументов командной строки, поэтому если значение `argc` не равно 1, мы вызываем `err_quit`, которая выводит сообщение об ошибке и завершает работу приложения. Далее вызывается функция `daemonize` из листинга 13.1, которая переводит процесс в режим демона. С этого момента мы уже не можем выводить сообщения на стандартное устройство вывода сообщений об ошибках — вместо этого мы должны выводить их в журнал.

[101–110] Мы будем игнорировать сигнал `SIGPIPE`. Запись будет выполняться в дескриптор сокета, и нам совершенно не нужно, чтобы ошибка записи приводила к генерации сигнала `SIGPIPE`, действие по умолчанию для которого заключается в завершении процесса. Далее мы включаем сигналы `SIGHUP` и `SIGTERM` в маску сигналов потока. Все потоки, которые будут запущены впоследствии, унаследуют эту маску сигналов. Мы будем использовать сигнал `SIGHUP`, чтобы сообщить демону о необходимости перечитать конфигурационный файл, а `SIGTERM` — чтобы сообщить ему, что он должен корректно завершить свою работу.

[111–117] Мы вызываем функцию `sysconf`, чтобы получить максимально возможный размер имени хоста. Если вызов `sysconf` завершается неудачей или значение указанного предела не определено, мы будем использовать константу `HOST_NAME_MAX` — это лучшее, что можно сделать в данной ситуации. На некоторых платформах эта константа может быть уже определена, но если это не так, мы будем использовать значение, указанное в заголовочном файле `print.h`. Затем мы выделяем память для хранения имени хоста и вызываем `gethostname`, чтобы получить его.

[118–121] Далее мы пытаемся определить сетевой адрес, который, как предполагается, должен использовать демон для предоставления услуг очереди печати.

```
122 FD_ZERO(&rendezvous);
123 maxfd = -1;
124 for (aip = aplist; aip != NULL; aip = aip->ai_next) {
125 if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
126 aip->ai_addrlen, QLEN)) >= 0) {
127 FD_SET(sockfd, &rendezvous);
128 if (sockfd > maxfd)
129 maxfd = sockfd;
130 }
131 }
132 if (maxfd == -1)
133 log_quit("служба отключена");

134 pwdp = getpwnam(LPNAME);
135 if (pwdp == NULL)
136 log_sys("отсутствует пользователь %s", LPNAME);
137 if (pwdp->pw_uid == 0)
138 log_quit("%s является привилегированным пользователем", LPNAME);
139 if (setgid(pwdp->pw_gid) < 0 || setuid(pwdp->pw_uid) < 0)
140 log_sys("ошибка смены идентификатора на пользователя %s", LPNAME);

141 init_request();
142 init_printer();
```

[122–131] Очищаем набор дескрипторов `rendezvous` для передачи функции `select` в ожидании поступления запросов на соединение. Инициализируем переменную максимального номера дескриптора значением `-1`, в результате первый же дескриптор, который удастся получить, наверняка будет больше `maxfd`. Для каждого из полученных сетевых адресов мы вызываем функцию `initserver` (листинг 16.9), которая размещает и инициализирует сокет. В случае успешного завершения `initserver` мы добавляем полученный от нее дескриптор в набор `rendezvous` и, если он больше максимального, заносим в переменную `maxfd` номер этого дескриптора.

[132–133] Если после просмотра списка структур `addrinfo` значение переменной `maxfd` осталось равным `-1`, мы не можем запустить службу печати, поэтому выводим сообщение в журнал и завершаем работу.

[134–140] Чтобы назначить сокету привилегированный номер порта, демон должен обладать привилегиями суперпользователя. Теперь, когда сокеты уже созданы, мы можем понизить привилегии демона, заменив существующие идентификаторы пользователя и группы на идентификаторы пользователя `LPNAME`. Мы должны следовать принципам минимизации привилегий, чтобы избежать появления потенциальных уязвимостей в демоне. Мы вызываем функцию `getpwnam`, чтобы найти в файле паролей запись, связанную с пользователем `LPNAME`. Если такой пользователь не будет найден или если он существует, но его идентификатор совпадает с идентификатором суперпользователя, мы выводим сообщение в журнал и завершаем работу. В противном случае мы изменяем реальный и эффективный идентификаторы вызовом функций `setgid` и `setuid`. Чтобы не подвергать систему опасности, мы предпочитаем вообще не предоставлять услуги, если невозможно понизить привилегии.

[141–142] Функция `init_request` вызывается, чтобы инициализировать прием запросов на выполнение заданий и убедиться, что запущена единственная копия демона, а затем вызывается функция `init_printer`, которая инициализирует информацию о принтере (вскоре мы рассмотрим обе функции).

```

143 err = pthread_create(&tid, NULL, printer_thread, NULL);
144 if (err == 0)
145 err = pthread_create(&tid, NULL, signal_thread, NULL);
146 if (err != 0)
147 log_exit(err, "невозможно запустить поток");
148 build_qonstart();

149 log_msg("демон инициализирован");

150 for (;;) {
151 rset = rendezvous;
152 if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
153 log_sys("ошибка вызова функции select");
154 for (i = 0; i <= maxfd; i++) {
155 if (FD_ISSET(i, &rset)) {
156 /*
157 * Принять и обработать запрос.
158 */
159 if ((sockfd = accept(i, NULL, NULL)) < 0)
160 log_ret("ошибка вызова функции accept");
161 pthread_create(&tid, NULL, client_thread,
162 (void *)((long)sockfd));
163 }
164 }
165 }
166 exit(1);
167 }

```

[143–149] Мы запускаем один поток для обработки сигналов и один поток для взаимодействия с принтером. (Разместив все операции с принтером в одном потоке, мы можем упростить алгоритмы блокировки структур данных, связанных с принтером.) Затем вызываем `build_qonstart`, чтобы отыскать в каталоге `/var/spool/printer` подкаталоги, соответствующие заданиям, ожидающим обработки. Для каждого найденного задания создается структура, которая позволит потоку взаимодействия с принтером узнать, какие файлы отправить принтеру. На этом мы заканчиваем инициализацию демона и выводим в журнал сообщение, которое говорит, что инициализация прошла успешно.

[150–167] Копируем набор дескрипторов `rendezvous` (строку `fd_set`) в переменную `rset` и вызываем `select`, чтобы дождаться момента, когда один из дескрипторов станет доступен для чтения. Мы используем копию `rendezvous`, потому что `select` модифицирует переданную ей структуру `fd_set` и оставляет в ней только дескрипторы, соответствующие наступившему событию. Поскольку сокеты инициализированы для использования сервером, доступность дескрипторов для чтения означает, что поступил запрос на соединение. После возврата из функции `select` мы проверяем, какие дескрипторы из набора `rset` доступны для чтения. Если такие дескрипторы найдены, они передаются функции `accept`, чтобы принять соединение. Если функция `accept` терпит неудачу, мы выводим в журнал сообщение и продолжаем проверку набора в поисках дескрипторов, доступных для чтения. Иначе запускается поток, который займется обслуживанием соединения с клиентом. Функция `main` входит в бесконечный цикл, принимая запросы и передавая их для обработки другим потокам, — она никогда не должна дойти до выполнения строки с вызовом функции `exit`.

```
168 /*
169 * Инициализирует файл с идентификатором задания. Устанавливает блокировку
170 * для записи, чтобы предотвратить запуск других копий демона.
171 */
172 * БЛОКИРОВКИ: отсутствуют, кроме блокировки на файл задания.
173 */
174 void
175 init_request(void)
176 {
177 int n;
178 char name[FILENMSZ];

179 sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
180 jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
181 if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
182 log_quit("демон уже запущен");

183 /*
184 * Повторно использовать буфер с именем файла для счетчика заданий.
185 */
186 if ((n = read(jobfd, name, FILENMSZ)) < 0)
187 log_sys("невозможно прочитать содержимое файла задания");
188 if (n == 0)
189 nextjob = 1;
190 else
191 nextjob = atol(name);
192 }
```

[168–182] Функция `init_request` выполняет два действия: устанавливает блокировку на файл задания `/var/spool/printer/jobno` и читает его содержимое, чтобы определить номер для следующего задания. Мы устанавливаем блокировку для записи на весь файл, чтобы показать, что демон уже запущен. Если кто-либо попытается запустить еще одну копию демона печати, эта дополнительная копия не сможет установить блокировку на файл и завершит работу. (Этот прием мы использовали в листинге 13.2, а макрос `write_lock` был описан в разделе 14.3.)

[183–192] Файл задания содержит номер следующего задания в виде строки ASCII. Если файл только что создан и пока не содержит данных, мы записываем в переменную `nextjob` число 1. Иначе с помощью функции `atol` выполняется преобразование строки в целое число, которое будет использовано в качестве номера следующего задания. Мы оставляем дескриптор `jobfd` открытым, чтобы можно было обновлять номер в файле по мере поступления новых заданий. Мы не можем закрыть этот файл, потому что тогда блокировка, которую мы только что установили, будет снята автоматически.

В системах, где длинные целые содержат 64 разряда, для представления наибольшего длинного целого числа в виде строки потребуется буфер длиной не менее 21 байта. Для этих целей вполне подходит буфер с именем файла, потому что константа `FILENMSZ` определена в заголовочном файле `print.h` со значением 64.

```

193 /*
194 * Инициализирует информацию о принтере из конфигурационного файла.
195 *
196 * БЛОКИРОВКИ: отсутствуют.
197 */
198 void
199 init_printer(void)
200 {
201     printer = get_printaddr();
202     if (printer == NULL)
203         exit(1); /* сообщение уже выведено в журнал */
204     printer_name = printer->ai_canonname;
205     if (printer_name == NULL)
206         printer_name = "printer";
207     log_msg("имя принтера: %s", printer_name);
208 }

209 /*
210 * Обновляет идентификатор следующего задания в файле.
211 * Не обрабатывает случаи переполнения.
212 *
213 * БЛОКИРОВКИ: отсутствуют.
214 */
215 void
216 update_jobno(void)
217 {
218     char buf[32];

219     if (lseek(jobfd, 0, SEEK_SET) == -1)
220         log_sys("невозможно перейти в начало файла задания");
221     sprintf(buf, "%d", nextjob);
222     if (write(jobfd, buf, strlen(buf)) < 0)
223         log_sys("невозможно обновить файл с номером задания");
224 }

```

[193–208] Функция `init_printer` инициализирует имя принтера и его сетевой адрес. Адрес принтера мы получаем вызовом `get_printaddr` (из `util.c`). Если вызов этой функции завершится неудачей, мы выводим сообщение в журнал и завершаем работу. В случае неудачи функция `get_printaddr` выводит в журнал собственное сообщение. Однако если ей удалось определить адрес принтера, мы принимаем имя принтера из поля `ai_canonname` структуры `addrinfo`. Если это поле пустое, мы записываем имя принтера по умолчанию — `printer`. Обратите внимание: имя используемого принтера выводится в журнал, чтобы помочь администраторам в диагностике возможных проблем с системой печати.

[209–224] Функция `update_jobno` записывает номер следующего задания в файл `/var/spool/printer/jobno`. Прежде всего, мы устанавливаем текущую позицию записи в начало файла. Затем преобразуем целочисленный номер задания в строку и записываем ее в файл. Если операция записи не увенчалась успехом, мы выводим сообщение об ошибке в журнал и завершаем работу. Номера заданий увеличиваются монотонно; обработку ситуации переполнения мы оставляем в качестве самостоятельного упражнения (см. упражнение 21.9).

```
225 /*
226 * Получает номер следующего задания.
227 *
228 * БЛОКИРОВКИ: запирает и отпирает joblock.
229 */
230 int32_t
231 get_newjobno(void)
232 {
233     int32_t jobid;

234     pthread_mutex_lock(&joblock);
235     jobid = nextjob++;
236     if (nextjob <= 0)
237         nextjob = 1;
238     pthread_mutex_unlock(&joblock);
239     return(jobid);
240 }

241 /*
242 * Добавляет в очередь новое задание. После этого посыпает
243 * потоку принтера сигнал, что появилось новое задание.
244 *
245 * БЛОКИРОВКИ: запирает и отпирает joblock.
246 */
247 void
248 add_job(struct printreq *reqp, int32_t jobid)
249 {
250     struct job *jp;

251     if ((jp = malloc(sizeof(struct job))) == NULL)
252         log_sys("ошибка вызова функции malloc");
253     memcpy(&jp->req, reqp, sizeof(struct printreq));
```

[225–240] Функция `get_newjobno` возвращает номер следующего задания. Сначала мы запираем мьютекс `joblock`. Увеличиваем на 1 значение переменной `nextjob` и обрабатываем ситуацию ее переполнения. Затем отпираем мьютекс и возвращаем значение, которое имела переменная `nextjob` до увеличения. Функция `get_newjobno` может вызываться из нескольких потоков одновременно, поэтому мы должны упорядочить доступ к номеру следующего задания, чтобы каждый поток получил свой, уникальный номер задания. (На рис. 11.4 мы уже показывали, что может произойти, если не организовать поочередный доступ к данным из нескольких потоков.)

[241–253] Функция `add_job` используется для добавления нового запроса в конец очереди заданий печати. Функция начинается с выделения памяти под структуру `job`. Если память не может быть выделена, мы выводим сообщение об ошибке в журнал и завершаем работу. К этому моменту запрос на печать уже сохранен на диске, поэтому он будет принят демоном после перезапуска. После выделения памяти для структуры мы копируем структуру запроса `printreq`, полученную от клиента, в структуру `job`. В файле `print.h` мы видели, что структура `job` состоит из пары указателей, идентификатора задания и копии структуры `printreq`, полученной от клиентской утилиты `print`.

```
254 jp->jobid = jobid;
255 jp->next = NULL;
256 pthread_mutex_lock(&joblock);
257 jp->prev = jobtail;
258 if (jobtail == NULL)
259 jobhead = jp;
260 else
261 jobtail->next = jp;
262 jobtail = jp;
263 pthread_mutex_unlock(&joblock);
264 pthread_cond_signal(&jobwait);
265 }

266 /*
267 * Вставляет задание в начало списка.
268 *
269 * БЛОКИРОВКИ: запирает и отпирает joblock.
270 */
271 void
272 replace_job(struct job *jp)
273 {
274 pthread_mutex_lock(&joblock);
275 jp->prev = NULL;
276 jp->next = jobhead;
277 if (jobhead == NULL)
278 jobtail = jp;
279 else
280 jobhead->prev = jp;
281 jobhead = jp;
282 pthread_mutex_unlock(&joblock);
283 }
```

[254–265] Сохраняем идентификатор задания и запираем мьютекс, чтобы гарантировать исключительность доступа к списку заданий печати. Добавление новой структуры производится в конец списка. Мы записываем в указатель на предыдущий элемент новой структуры адрес последнего задания в списке. Если список пуст, адрес новой структуры записывается в `jobhead`, иначе — в указатель на следующий элемент в последней структуре в списке. Затем адрес новой структуры записывается в `jobtail`. В заключение мы отпираем мьютекс и посылаем сигнал потоку, обслуживающему принтер, чтобы известить его о получении нового задания.

[266–283] Функция `replace_job` вставляет задание в начало списка. Мы запираем мьютекс `joblock`, записываем `NULL` в указатель на предыдущий элемент списка в добавляемой структуре, а в указатель на следующий элемент — адрес начала списка. Если список пуст, адрес добавляемой структуры записывается в `jobtail`, иначе адрес добавляемой структуры записывается в указатель на предыдущий элемент в первой структуре в списке. После этого адрес добавляемой структуры записывается в `jobhead`. В заключение мы отпираем мьютекс `joblock`.

```
284 /*
285 * Удаляет задание из очереди.
286 *
287 * БЛОКИРОВКИ: вызывающая функция должна запереть joblock.
288 */
289 void
290 remove_job(struct job *target)
291 {
292 if (target->next != NULL)
293 target->next->prev = target->prev;
294 else
295 jobtail = target->prev;
296 if (target->prev != NULL)
297 target->prev->next = target->next;
298 else
299 jobhead = target->next;
300 }

301 /*
302 * Вызывается при запуске и проверяет наличие заданий в каталоге очереди.
303 *
304 * БЛОКИРОВКИ: отсутствуют.
305 */
306 void
307 build_qonstart(void)
308 {
309 int fd, err, nr;
310 int32_t jobid;
311 DIR *dirp;
312 struct dirent *entp;
313 struct printreq req;
314 char dname[FILENMSZ], fname[FILENMSZ];

315 sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
316 if ((dirp = opendir(dname)) == NULL)
317 return;
```

[284–300] Функция `remove_job` удаляет из очереди задание, указатель на которое передается в функцию. Вызывающая функция должна запереть мьютекс `joblock` перед вызовом `remove_job`. Если указатель на следующий элемент не пустой, в указатель на предыдущий элемент следующего элемента списка мы записываем адрес элемента, предшествующего удаляемому. Иначе задание является последним в очереди, поэтому адрес предыдущего элемента списка записывается в переменную `jobtail`. Если указатель на предыдущий элемент не пустой, в указатель на следующий элемент предыдущего элемента списка мы записываем адрес элемента, следующего за удаляемым. Иначе задание является первым в списке, поэтому адрес следующего элемента записывается в переменную `jobhead`.

[301–317] При запуске демон вызывает функцию `build_qonstart`, которая собирает в памяти список заданий из файлов, сохраняемых в каталоге `/var/spool/printer/reqs`. Если открыть этот каталог не получается, значит, задания печати отсутствуют, поэтому мы просто возвращаем управление.

```
318 while ((entp = readdir(dirp)) != NULL) {  
319 /*  
320 * Пропустить каталоги "." и ".."  
321 */  
322 if (strcmp(entp->d_name, ".") == 0 ||  
323 strcmp(entp->d_name, "..") == 0)  
324 continue;  
  
325 /*  
326 * Прочитать структуру запроса.  
327 */  
328 sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);  
329 if ((fd = open(fname, O_RDONLY)) < 0)  
330 continue;  
331 nr = read(fd, &req, sizeof(struct printreq));  
332 if (nr != sizeof(struct printreq)) {  
333 if (nr < 0)  
334 err = errno;  
335 else  
336 err = EIO;  
337 close(fd);  
338 log_msg("build_qonstart: невозможно прочитать %s: %s",  
339 fname, strerror(err));  
340 unlink(fname);  
341 sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,  
342 entp->d_name);  
343 unlink(fname);  
344 continue;  
345 }  
346 jobid = atol(entp->d_name);  
347 log_msg("в очередь добавлено задание %d", jobid);  
348 add_job(&req, jobid);  
349 }  
350 closedir(dirp);  
351 }
```

[318–324] Читаем все записи из каталога, по одной за раз. При этом пропускаем каталоги «.» и «..».

[325–345] Для каждой записи собирается полный путь к файлу, который затем открывается для чтения. Если открыть файл не удалось, мы просто пропускаем его. Иначе читаем содержимое структуры `printreq` из файла. Если прочитать структуру целиком не удается, мы закрываем файл, выводим в журнал сообщение об ошибке и удаляем файл. После этого собираем полное имя соответствующего файла с данными и также удаляем его.

[346–351] Если нам удалось прочитать структуру `printreq`, мы преобразуем имя файла в идентификатор задания (имя файла представляет идентификатор задания), выводим сообщение в журнал и добавляем запрос в очередь заданий печати. Когда все записи будут прочитаны и функция `readdir` вернет `NULL`, мы закрываем каталог и возвращаем управление.

```
352 /*
353 * Принимает задание печати от клиента.
354 *
355 * БЛОКИРОВКИ: отсутствуют.
356 */
357 void *
358 client_thread(void *arg)
359 {
360 int n, sockfd, nr, nw, first;
361 int32_t jobid;
362 pthread_t tid;
363 struct printreq req;
364 struct printresp res;
365 char name[FILENMSZ];
366 char buf[IOBUFSZ];

367 tid = pthread_self();
368 pthread_cleanup_push(client_cleanup, (void *)((long)tid));
369 sockfd = (long)arg;
370 add_worker(tid, sockfd);

371 /*
372 * Прочитать заголовок запроса.
373 */
374 if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
375 sizeof(struct printreq)) {
376 res.jobid = 0;
377 if (n < 0)
378 res.retcode = htonl(errno);
379 else
380 res.retcode = htonl(EIO);
381 strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
382 writen(sockfd, &res, sizeof(struct printresp));
383 pthread_exit((void *)1);
384 }
```

[352–370] Функция `client_thread` — это функция запуска потока, порождаемого функцией `main` при получении запроса на соединение от клиента. Задача этого потока заключается в том, чтобы принять от клиента файл для печати. Для обработки каждого запроса, полученного от клиента, запускается отдельный поток.

Прежде всего, необходимо установить обработчик завершения потока (обработчики завершения обсуждались в разделе 11.5). В качестве обработчика завершения потока используется функция `client_cleanup`, которую мы рассмотрим несколько позже. Она получает единственный аргумент — идентификатор потока. Затем вызывается `add_worker`, чтобы создать структуру `worker_thread` и добавить ее в список активных клиентских потоков.

[371–384] На этом инициализация потока завершается и мы переходим к чтению заголовка запроса клиента. Если от клиента было получено меньше данных, чем ожидается, или в процессе чтения возникла ошибка, мы посыпаем клиенту сообщение-ответ, в котором указываем причину ошибки, и вызовом `pthread_exit` завершаем работу потока.

```

385 req.size = ntohs(req.size);
386 req.flags = ntohs(req.flags);

387 /*
388 * Создать файл данных.
389 */
390 jobid = get_newjobno();
391 sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
392 fd = creat(name, FILEPERM);
393 if (fd < 0) {
394     res.jobid = 0;
395     res.retcode = htonl(errno);
396     log_msg("client_thread: невозможно создать %s: %s", name,
397             strerror(res.retcode));
398     strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
399     writen(sockfd, &res, sizeof(struct printresp));
400     pthread_exit((void *)1);
401 }

402 /*
403 * Прочитать файл и сохранить его в каталоге очереди печати.
404 * Попытаться определить тип файла: PostScript или
405 * простой текстовый файл.
406 */
407 first = 1;
408 while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
409     if (first) {
410         first = 0;
411         if (strncmp(buf, "%!PS", 4) != 0)
412             req.flags |= PR_TEXT;
413 }

```

[385–401] Преобразуем целочисленные поля заголовка в значения с порядком байтов, соответствующим аппаратной архитектуре, и вызываем `get_newjobno`, чтобы зарезервировать очередной идентификатор для данного запроса. Мы создаем файл с данными для печати под именем `/var/spool/printer/data/jobid`, где `jobid` — идентификатор задания для данного запроса. При создании файла мы даем такие права доступа, которые не позволяют всем остальным прочитать файл (константа `FILEPERM` определена как `S_IRUSR|S_IWUSR` в заголовочном файле `print.h`). Если создать файл не удалось, мы выводим в журнал сообщение об ошибке, отправляем код ошибки клиенту и завершаем работу потока вызовом функции `pthread_exit`.

[402–413] Принимаем от клиента содержимое файла, которое записываем в свою копию файла. Но прежде чем записать что-либо, в первой итерации цикла проверяем, имеет ли принимаемый файл формат PostScript. Если содержимое файла не начинается с последовательности `%!PS`, мы предполагаем, что файл содержит обычный текст, и устанавливаем флаг `PR_TEXT` в заголовке запроса. (Мы уже говорили, что клиент тоже может установить этот флаг, запустив утилиту `print` с ключом `-t`.) Файлы формата PostScript не обязательно должны начинаться с данной последовательности, но руководство по форматированию документов [Adobe Systems, 1999] настоятельно это рекомендует.

```
414 nw = write(fd, buf, nr);
415 if (nw != nr) {
416     res.jobid = 0;
417     if (nw < 0)
418         res.retcode = htonl(errno);
419     else
420         res.retcode = htonl(EIO);
421     log_msg("client_thread: невозможно записать в %s: %s", name,
422             strerror(res.retcode));
423     close(fd);
424     strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
425     writen(sockfd, &res, sizeof(struct printresp));
426     unlink(name);
427     pthread_exit((void *)1);
428 }
429 }
430 close(fd);

431 /*
432 * Создать управляющий файл. Затем записать информацию
433 * о запросе на печать в управляющий
434 * файл.
435 */
436 sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jobid);
437 fd = creat(name, FILEPERM);
438 if (fd < 0) {
439     res.jobid = 0;
440     res.retcode = htonl(errno);
441     log_msg("client_thread: невозможно создать %s: %s", name,
442             strerror(res.retcode));
443     strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
444     writen(sockfd, &res, sizeof(struct printresp));
445     sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
446     unlink(name);
447     pthread_exit((void *)1);
448 }
```

[414–430] Записываем в файл данные, полученные от клиента. Если операция записи терпит неудачу, выводим в журнал сообщение, закрываем дескриптор файла данных, отправляем сообщение об ошибке клиенту, удаляем файл с данными и завершаем работу потока вызовом функции `pthread_exit`. По окончании приема данных для печати закрываем дескриптор файла с данными.

[431–448] Затем создаем файл `/var/spool/printer/reqs/jobid` для хранения содержимого запроса на печать. Если создать файл не удалось, выводим в журнал сообщение, отправляем сообщение об ошибке клиенту, удаляем файл с данными и завершаем работу потока.

```

449 nw = write(fd, &req, sizeof(struct printreq));
450 if (nw != sizeof(struct printreq)) {
451     res.jobid = 0;
452     if (nw < 0)
453         res.retcode = htonl(errno);
454     else
455         res.retcode = htonl(EIO);
456     log_msg("client_thread: невозможно записать в %s: %s", name,
457             strerror(res.retcode));
458     close(fd);
459     strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
460     writen(sockfd, &res, sizeof(struct printresp));
461     unlink(name);
462     sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
463     unlink(name);
464     pthread_exit((void *)1);
465 }
466 close(fd);

467 /*
468 * Отправить ответ клиенту.
469 */
470 res.retcode = 0;
471 res.jobid = htonl(jobid);
472 sprintf(res.msg, "запрос с идентификатором %d", jobid);
473 writen(sockfd, &res, sizeof(struct printresp));

474 /*
475 * Оповестить поток обслуживания принтера и корректно выйти.
476 */
477 log_msg("в очередь добавлено задание %d", jobid);
478 add_job(&req, jobid);
479 pthread_cleanup_pop(1);
480 return((void *)0);
481 }

```

[449–465] Записываем структуру `printreq` в управляющий файл. В случае ошибки выводим в журнал сообщение, закрываем дескриптор управляющего файла, отправляем клиенту сообщение об ошибке, удаляем файл с данными и управляющий файл и завершаем работу потока.

[466–473] Закрываем дескриптор управляющего файла и отправляем клиенту сообщение об успешной постановке задания в очередь вместе с идентификатором задания (`retcode = 0`).

[474–481] Вызываем `add_job`, чтобы добавить принятое задание в очередь, и `pthread_cleanup_pop`, чтобы освободить занятые ресурсы. Работа потока завершается оператором `return`.

Обратите внимание: перед завершением потока необходимо закрыть все дескрипторы файлов, которые более не понадобятся. В отличие от процедуры завершения процесса, при завершении потока дескрипторы файлов не закрываются автоматически, если в процессе остается еще хотя бы один поток. Поэтому если мы не будем закрывать дескрипторы, то рано или поздно столкнемся с нехваткой ресурсов.

```
482 /*
483 * Добавляет новый поток в список рабочих потоков.
484 *
485 * БЛОКИРОВКИ: запирает и отпирает workerlock.
486 */
487 void
488 add_worker(pthread_t tid, int sockfd)
489 {
490     struct worker_thread *wtp;

491     if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
492         log_ret("add_worker: ошибка вызова функции malloc");
493         pthread_exit((void *)1);
494     }
495     wtp->tid = tid;
496     wtp->sockfd = sockfd;
497     pthread_mutex_lock(&workerlock);
498     wtp->prev = NULL;
499     wtp->next = workers;
500     if (workers != NULL)
501         workers->prev = wtp;
502     workers = wtp;
503
504     pthread_mutex_unlock(&workerlock);
505 }

506 /*
507 * Завершает все рабочие потоки.
508 *
509 * БЛОКИРОВКИ: запирает и отпирает workerlock.
510 */
511 void
512 kill_workers(void)
513 {
514     struct worker_thread *wtp;

515     pthread_mutex_lock(&workerlock);
516     for (wtp = workers; wtp != NULL; wtp = wtp->next)
517         pthread_cancel(wtp->tid);
518     pthread_mutex_unlock(&workerlock);
519 }
```

[482–505] Функция `add_worker` добавляет новую структуру `worker_thread` в список активных потоков. Мы выделяем память для структуры, инициализируем ее, запираем мьютекс `workerlock`, добавляем структуру в начало списка и отпираем мьютекс.

[506–519] Функция `kill_workers` обходит список рабочих потоков и пытается завершить их один за другим. На время обхода списка мы запираем мьютекс `workerlock`. Как отмечалось выше, функция `pthread_cancel` просто посылает запрос на завершение, а фактическое завершение произойдет, лишь когда поток достигнет ближайшей точки выхода.

```
520 /*
521 * Процедура выхода для рабочего потока.
522 *
523 * БЛОКИРОВКИ: запирает и отпирает workerlock.
524 */
525 void
526 client_cleanup(void *arg)
527 {
528 struct worker_thread *wtp;
529 pthread_t tid;

530 tid = (pthread_t)((long)arg);
531 pthread_mutex_lock(&workerlock);
532 for (wtp = workers; wtp != NULL; wtp = wtp->next) {
533 if (wtp->tid == tid) {
534 if (wtp->next != NULL)
535 wtp->next->prev = wtp->prev;
536 if (wtp->prev != NULL)
537 wtp->prev->next = wtp->next;
538 else
539 workers = wtp->next;
540 break;
541 }
542 }
543 pthread_mutex_unlock(&workerlock);
544 if (wtp != NULL) {
545 close(wtp->sockfd);
546 free(wtp);
547 }
548 }
```

[520–542] Функция `client_cleanup` — это обработчик выхода для рабочих потоков, которые занимаются взаимодействием с клиентами. Она вызывается, когда поток вызывает функцию `pthread_exit` или `pthread_cleanup_pop` с ненулевым аргументом, а также в ответ на запрос о принудительном завершении. В качестве аргумента ей передается идентификатор потока, завершающего работу.

Мы запираем мьютекс `workerlock` и обходим список в поисках потока с заданным идентификатором. Если соответствующий поток найден, удаляем структуру из списка и прекращаем поиск.

[543–548] Отпираем мьютекс `workerlock`, закрываем дескриптор сокета, использовавшийся для взаимодействия с клиентом, и освобождаем память, занимаемую структурой `worker_thread`.

Поскольку мы пытаемся запереть мьютекс `workerlock`, если поток достигает точки выхода раньше, чем функция `kill_workers` успеет обойти весь список, нам придется ждать, пока `kill_workers` не отопрет мьютекс, и лишь после этого мы сможем продолжить работу.

```
549 /*
550 * Обслуживание сигналов.
551 *
552 * БЛОКИРОВКИ: запирает и отпирает configlock.
553 */
554 void *
555 signal_thread(void *arg)
556 {
557 int err, signo;

558 for (;;) {
559 err = sigwait(&mask, &signo);
560 if (err != 0)
561 log_quit("ошибка вызова функции sigwait: %s", strerror(err));
562 switch (signo) {
563 case SIGHUP:
564 /*
565 * Запланировать чтение конфигурационного файла.
566 */
567 pthread_mutex_lock(&configlock);
568 reread = 1;
569 pthread_mutex_unlock(&configlock);
570 break;
571 case SIGTERM:
572 kill_workers();
573 log_msg("завершение по сигналу %s", strsignal(signo));
574 exit(0);
575 default:
576 kill_workers();
577 log_quit("принят неожиданный сигнал %d", signo);
578 }
579 }
580 }
```

[549–562] Функция `signal_thread` — это функция запуска потока, ответственного за обработку сигналов. В маску сигналов, которую мы инициализировали в функции `main`, включены сигналы `SIGHUP` и `SIGTERM`. Здесь мы вызываем функцию `sigwait`, ожидая доставки одного из этих сигналов. Если она возвращает признак ошибки, выводим в журнал сообщение об ошибке и завершаем работу.

[563–570] Если принят сигнал `SIGHUP`, запираем мьютекс `configlock`, записываем 1 в переменную `reread` и отпираем мьютекс. Таким способом мы сообщаем демону о необходимости перечитать содержимое конфигурационного файла в ближайшей итерации главного цикла.

[571–574] Если принят сигнал `SIGTERM`, завершаем все рабочие потоки вызовом функции `kill_workers`, выводим в журнал сообщение и вызываем функцию `exit`, которая завершает работу процесса.

[575–580] Если принят сигнал, которого мы не ожидали, все рабочие потоки завершаются и вызывается функция `log_quit`, которая выводит в журнал сообщение и завершает процесс.

```
581 /*
582 * Добавляет атрибут в заголовок IPP.
583 *
584 * БЛОКИРОВКИ: отсутствуют.
585 */
586 char *
587 add_option(char *cp, int tag, char *optname, char *optval)
588 {
589     int n;
590     union {
591         int16_t s;
592         char c[2];
593     } u;
594
595     *cp++ = tag;
596     n = strlen(optname);
597     u.s = htons(n);
598     *cp++ = u.c[0];
599     *cp++ = u.c[1];
600     strcpy(cp, optname);
601     cp += n;
602     n = strlen(optval);
603     u.s = htons(n);
604     *cp++ = u.c[0];
605     *cp++ = u.c[1];
606     strcpy(cp, optval);
607 }
```

[581–593] Функция `add_option` добавляет атрибут в заголовок IPP, который будет передан принтеру. На рис. 21.3 было показано, что атрибут состоит из 1 байта признака, описывающего тип атрибута, за которым следуют 2-байтное целое в двоичном формате, представляющее длину имени атрибута, имя атрибута, размер значения атрибута и, наконец, само значение.

Протокол IPP не предполагает какого-либо выравнивания целых чисел в двоичном представлении, имеющихся в заголовке. Некоторые аппаратные архитектуры, такие как SPARC, не могут хранить целые числа, начиная с произвольного адреса. Это означает, что мы не можем сохранить целое число в заголовке простым приведением типа адреса в заголовке, куда должно быть записано число, к типу указателя на `int16_t`. Вместо этого мы должны скопировать число как строку, байт за байтом. По этой причине мы определили объединение (`union`), содержащее 16-разрядное целое и 2 байта.

[594–607] Мы сохраняем признак атрибута в заголовке и преобразуем значение длины имени атрибута в значение с сетевым порядком байтов. Далее побайтно копируем в заголовок длину имени и само имя атрибута. Тот же процесс повторяется для значения атрибута и возвращается адрес в заголовке, с которого должен начинаться следующий раздел заголовка.

```
608 /*
609 * Единственный поток, который занимается взаимодействием с принтером.
610 *
611 * БЛОКИРОВКИ: запирает и отпирает joblock и configlock.
612 */
613 void *
614 printer_thread(void *arg)
615 {
616     struct job *jp;
617     int hlen, ilen, sockfd, fd, nr, nw, extra;
618     char *icp, *hcp, *p;
619     struct ipp_hdr *hp;
620     struct stat sbuf;
621     struct iovec iov[2];
622     char name[FILENMSZ];
623     char hbuf[HBUFSZ];
624     char ibuf[IBUFSZ];
625     char buf[IOBUFSZ];
626     char str[64];
627     struct timespec ts = { 60, 0 }; /* 1 минута */

628     for (;;) {
629         /*
630         * Получить задание печати.
631         */
632         pthread_mutex_lock(&joblock);
633         while (jobhead == NULL) {
634             log_msg("printer_thread: ожидание...");
635             pthread_cond_wait(&jobwait, &joblock);
636         }
637         remove_job(jp = jobhead);
638         log_msg("printer_thread: принято задание %d", jp->jobid);
639         pthread_mutex_unlock(&joblock);
640         update_jobno();
```

[608–627] `printer_thread` — функция запуска потока, который взаимодействует с сетевым принтером. Переменные `icp` и `ibuf` используются для сборки заголовка IPP, а переменные `hcp` и `hbuf` — для сборки заголовка HTTP. Заголовки должны собираться в отдельных буферах. Заголовок HTTP включает поле длины в формате ASCII, но пока не будет собран заголовок IPP, значение этого поля неизвестно. Мы используем единственный вызов `writev` для записи обоих заголовков сразу.

[628–640] Поток взаимодействия с принтером входит в бесконечный цикл и ожидает заданий для передачи принтеру. Доступ к списку заданий осуществляется под защитой мьютекса `joblock`. Если очередь заданий пуста, вызываем функцию `pthread_cond_wait`, чтобы дождаться хотя бы одного задания. Когда задание появится в очереди, мы удаляем его из списка вызовом функции `remove_job`. В этот момент список все еще находится под защитой мьютекса, поэтому мы отпираем его и вызываем `update_jobno`, чтобы записать номер следующего задания в файл `/var/spool/printer/jobno`.

```
641 /*
642 * Проверить наличие изменений в конфигурационном файле.
643 */
644 pthread_mutex_lock(&configlock);
645 if (reread) {
646 freeaddrinfo(printer);
647 printer = NULL;
648 printer_name = NULL;
649 reread = 0;
650 pthread_mutex_unlock(&configlock);
651 init_printer();
652 } else {
653 pthread_mutex_unlock(&configlock);
654 }

655 /*
656 * Отправить задание принтеру.
657 */
658 sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jp->jobid);
659 if ((fd = open(name, O_RDONLY)) < 0) {
660 log_msg("задание %d отменено – невозможно открыть %s: %s",
661 jp->jobid, name, strerror(errno));
662 free(jp);
663 continue;
664 }
665 if (fstat(fd, &sbuf) < 0) {
666 log_msg("задание %d отменено – ошибка fstat для %s: %s",
667 jp->jobid, name, strerror(errno));
668 free(jp);
669 close(fd);
670 continue;
671 }
```

[641–654] Теперь, когда имеется задание печати, проверяем наличие изменений в конфигурационном файле. Для этого запираем мьютекс `configlock` и проверяем значение переменной `reread`. Если она имеет ненулевое значение, освобождаем память, занимаемую списком `addrinfo` принтера, очищаем указатели, отпираем мьютекс и вызываем `init_printer`, чтобы провести повторную инициализацию информации о принтере. Так как после инициализации информации о принтере в функции `main` это единственное место в программе, где данная информация может быть изменена, нам не требуется никакая-либо дополнительная синхронизация для доступа к переменной `reread`, кроме использования мьютекса `configlock`.

Обратите внимание: хотя в этой функции мы запираем и отпираем различные мьютексы, мы нигде не удерживаем их запертыми одновременно, поэтому нам не придется ломать голову над иерархией блокировок (раздел 11.6.2).

[655–671] Если невозможно открыть файл данных, в журнал выводится сообщение, освобождается память, занимаемая структурой `job`, и управление передается в начало цикла. После открытия файла мы вызываем функцию `fstat`, чтобы узнать размер файла. Если это не удается, мы выводим в журнал сообщение, освобождаем память, занимаемую структурой `job`, закрываем дескриптор файла и переходим в начало цикла.

```
672 if ((sockfd = connect_retry(AF_INET, SOCK_STREAM, 0,
673 printer->ai_addr, printer->ai_addrlen)) < 0) {
674 log_msg("задание %d отложено - невозможно соединиться "
675 "с принтером: %s", jp->jobid, strerror(errno));
676 goto defer;
677 }

678 /*
679 * Собрать заголовок IPP.
680 */
681 icp = ibuf;
682 hp = (struct ipp_hdr *)icp;
683 hp->major_version = 1;
684 hp->minor_version = 1;
685 hp->operation = htons(OP_PRINT_JOB);
686 hp->request_id = htonl(jp->jobid);
687 icp += offsetof(struct ipp_hdr, attr_group);
688 *icp++ = TAG_OPERATION_ATTR;
689 icp = add_option(icp, TAG_CHARSET, "attributes-charset",
690 "utf-8");
691 icp = add_option(icp, TAG_NATULANG,
692 "attributes-natural-language", "en-us");
693 sprintf(str, "http://%s/ipp", printer_name);
694 icp = add_option(icp, TAG_URI, "printer-uri", str);
695 icp = add_option(icp, TAG_NAMEWOLANG,
696 "requesting-user-name", jp->req.usernm);
697 icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
698 jp->req.jobnm);
```

[672–677] Открываем потоковый сокет для взаимодействия с принтером. Если вызов функции `connect_retry` терпит неудачу, выполняется переход к метке `defer`, где освобождаются занятые ресурсы, и после задержки попытка повторяется.

[677–698] Затем начинается сборка заголовка IPP. В качестве операции назначается операция запроса на печать задания. С помощью функций `htons` и `htonl` мы преобразуем 2-байтный идентификатор операции и 4-байтный идентификатор задания из значений с аппаратным порядком байтов в значения с сетевым порядком байтов. После начальной части заголовка вставляем признак начала блока атрибутов операции и добавляем атрибуты вызовом `add_option`. В табл. 21.2 перечислены обязательные и необязательные атрибуты запроса на печать. Первые три являются обязательными. В качестве кодировки символов мы определяем UTF-8, которая должна поддерживаться принтером. Естественный язык мы задаем как `en-us`, что соответствует американскому английскому. Еще один обязательный атрибут — универсальный идентификатор ресурса принтера (Uniform Resource Identifier, URI). Мы определяем его как `http://printer_name/ipp`.

Атрибут `requesting-user-name` рекомендуется, но не является обязательным. Атрибут `job-name` также является необязательным. Мы уже говорили, что утилита `print` в качестве имени задания передает имя файла, который должен быть напечатан, что помогает пользователям разобраться в большом количестве заданий, ожидающих обработки.

```

699 if (jp->req.flags & PR_TEXT) {
700 p = "text/plain";
701 extra = 1;
702 } else {
703 p = "application/postscript";
704 extra = 0;
705 }
706 icp = add_option(icp, TAG_MIMETYPE, "document-format", p);
707 *icp++ = TAG_END_OF_ATTR;
708 ilen = icp - ibuf;

709 /*
710 * Собрать заголовок HTTP.
711 */
712 hcp = hbuf;
713 sprintf(hcp, "POST /ipp HTTP/1.1\r\n");
714 hcp += strlen(hcp);
715 sprintf(hcp, "Content-Length: %ld\r\n",
716 (long)sbuf.st_size + ilen + extra);
717 hcp += strlen(hcp);
718 strcpy(hcp, "Content-Type: application/ipp\r\n");
719 hcp += strlen(hcp);
720 sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
721 hcp += strlen(hcp);
722 *hcp++ = '\r';
723 *hcp++ = '\n';
724 hlen = hcp - hbuf;

```

[699–708] Последний атрибут, который мы добавляем, — это `document-format`. Если опустить его, принтер будет полагать, что формат документа соответствует формату принтера по умолчанию. Для принтеров PostScript это, скорее всего, формат PostScript, но некоторые принтеры могут определять формат автоматически и выбирать между PostScript и простым текстом или между PostScript и PCL (Printer Command Language — язык команд принтера компании Hewlett-Packard). Если установлен флаг `PR_TEXT`, формат документа определяется как `text/plain`, иначе — как `application/postscript`. Затем мы вставляем признак конца блока атрибутов и подсчитываем размер получившегося заголовка IPP.

Целочисленная переменная `extra` определяет количество дополнительных символов, которые может потребоваться передать принтеру. Как будет показано ниже, при печати простого текста необходимо передать дополнительный символ, чтобы повысить надежность печати. Значение переменной `extra` понадобится нам, когда мы будем подсчитывать длину содержимого.

[709–724] Теперь, когда размер заголовка IPP известен, можно приступить к сборке заголовка HTTP. Мы устанавливаем значение атрибута `Content-Length` равным сумме размеров заголовка IPP, печатаемого файла и количества дополнительных символов. В атрибут `ContentType` записывается значение `application/ipp`. Конец заголовка HTTP отмечается символами возврата каретки и перевода строки. В заключение подсчитывается размер заголовка HTTP.

```
725 /*
726 * Передать сначала заголовки, потом файл.
727 */
728 iov[0].iov_base = hbuf;
729 iov[0].iov_len = hlen;
730 iov[1].iov_base = ibuf;
731 iov[1].iov_len = ilen;
732 if (writev(sockfd, iov, 2) != hlen + ilen) {
733 log_ret("невозможно передать данные принтеру");
734 goto defer;
735 }

736 if (jp->req.flags & PR_TEXT) {
737 /*
738 * Хак: позволить напечатать PostScript как простой текст.
739 */
740 if (write(sockfd, "\b", 1) != 1) {
741 log_ret("невозможно передать данные принтеру");
742 goto defer;
743 }
744 }

745 while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
746 if ((nw = writen(sockfd, buf, nr)) != nr) {
747 if (nw < 0)
748 log_ret("невозможно передать данные принтеру");
749 else
750 log_msg("данные переданы частично (%d/%d)", nw, nr);
751 goto defer;
752 }
753 }
```

[725–735] В первый элемент массива `iovecs` записывается заголовок HTTP, а во второй — заголовок IPP. Затем с помощью функции `writev` оба заголовка отправляются принтеру. Если операция записи терпит неудачу, мы выводим в журнал сообщение и переходим на метку `defer`, где производится освобождение занятых ресурсов и выполняется задержка перед повторной попыткой.

[736–744] Даже если мы указали, что файл содержит простой текст, принтер Phaser 8560 все равно попытается сам определить формат документа. Чтобы предотвратить автоматическое опознание файла, когда требуется напечатать файл PostScript как простой текст, мы посылаем первый символ, который является символом забоя (`backspace`). Этот символ не печатается и мешает принтеру автоматически определить формат файла. Этот прием позволит печатать файлы PostScript в простом текстовом виде.

[745–753] Затем мы отправляем принтеру файл с данными порциями по `IOBUFSZ` байт. Функция `write` может послать меньше байтов, чем запрошено, если буфер сокета окажется заполнен, поэтому, чтобы обойти эту ситуацию, мы используем функцию `writen`. Подобная ситуация с заголовками маловероятна из-за их небольшого размера, поэтому при их отправке мы не заботимся о возможности неполной передачи данных.

```
754 if (nr < 0) {
755 log_ret("невозможно прочитать из %s", name);
756 goto defer;
757 }

758 /*
759 * Прочитать ответ принтера.
760 */
761 if (printer_status(sockfd, jp)) {
762 unlink(name);
763 sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jp->jobid);
764 unlink(name);
765 free(jp);
766 jp = NULL;
767 }
768 defer:
769 close(fd);
770 if (sockfd >= 0)
771 close(sockfd);
772 if (jp != NULL) {
773 replace_job(jp);
774 nanosleep(&ts, NULL);
775 }
776 }
777 }

778 /*
779 * Читает данные из принтера – возможно, увеличивая приемный буфер.
780 * Возвращает смещение конца данных в буфере или -1 в случае ошибки.
781 *
782 * БЛОКИРОВКИ: отсутствуют.
783 */
784 ssize_t
785 readmore(int sockfd, char **bpp, int off, int *bszp)
```

[754–757] По достижении конца файла функция `read` вернет 0. Однако если `read` терпит неудачу, мы записываем в журнал сообщение и переходим на метку `defer`.

[758–767] После передачи файла принтеру вызываем функцию `printer_status`, которая принимает ответ принтера на наш запрос. Если функция `printer_status` завершается успехом, она возвращает положительное значение, после чего мы удаляем файл с данными и управляющий файл. Затем освобождаем память, занимаемую структурой `job`, записывая `NULL` в указатель на нее и переходим к метке `defer`.

[768–777] На метке `defer` мы закрываем дескриптор файла с данными. Если дескриптор сокета открыт, закрываем его. В случае ошибки указатель `jp` будет ссылаться на структуру `job` задания, которое мы пытались напечатать, поэтому помещаем задание обратно в начало очереди заданий и выполняем задержку на 1 минуту. В случае успеха указатель `jp` будет содержать значение `NULL`, поэтому мы просто возвращаемся к началу цикла, чтобы обработать следующее задание печати.

[778–785] Функция `readmore` используется, чтобы прочитать часть сообщения, отправленного принтером.

```
786 {
787     ssize_t nr;
788     char *bp = *bpp;
789     int bsz = *bszp;
790
791     if (off >= bsz) {
792         bsz += IOBUFSZ;
793         if ((bp = realloc(*bpp, bsz)) == NULL)
794             log_sys("readmore: невозможно увеличить размер буфера");
795         *bszp = bsz;
796         *bpp = bp;
797     }
798     if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
799         return(off+nr);
800     else
801         return(-1);
802 }
803 */
804 * Читает и анализирует ответ принтера. Возвращает 1, если ответ
805 * свидетельствует об успехе, 0 - в противном случае.
806 *
807 * БЛОКИРОВКИ: отсутствуют.
808 */
809 int
810 printer_status(int sfd, struct job *jp)
811 {
812     int i, success, code, len, found, bufsz, datsz;
813     int32_t jobid;
814     ssize_t nr;
815     char *bp, *cp, *statcode, *reason, *contentlen;
816     struct ipp_hdr h;
817
818     /*
819      * Прочитать заголовок HTTP и следующий за ним заголовок IPP.
820      * Для их получения может понадобиться несколько попыток чтения.
821      * Для определения объема читаемых данных используется Content-Length.
822     */
```

[786–801] Если текущая позиция чтения находится в конце буфера, мы увеличиваем его размер и возвращаем адрес начала нового буфера и его размер через аргументы `bpp` и `bszp` соответственно. В любом случае мы пытаемся прочитать столько данных, сколько поместится в буфер, дописывая новые данные после данных, уже находящихся в буфере. Мы возвращаем новое значение смещения конца данных в буфере. Если операция чтения потерпела неудачу или истекло время тайм-аута, возвращается значение `-1`.

[802–820] Функция `printer_status` читает ответ принтера на наш запрос. Нам неизвестно заранее, как ответит принтер: он может разделить ответ на несколько сообщений, отправить его в виде одного сообщения или использовать промежуточные сообщения HTTP `100 Continue`. Необходимо обработать все возможные ситуации.

```
821 success = 0;
822 bufsz = IOBUFSZ;
823 if ((bp = malloc(IOBUFSZ)) == NULL)
824 log_sys("printer_status: невозможно разместить буфер чтения");

825 while ((nr = tread(sfd, bp, bufsz, 5)) > 0) {
826 /*
827 * Отыскать код статуса. Ответ начинается со строки "HTTP/x.y",
828 * поэтому нужно пропустить 8 символов.
829 */
830 cp = bp + 8;
831 datsz = nr;
832 while (isspace((int)*cp))
833 cp++;
834 statcode = cp;
835 while (isdigit((int)*cp))
836 cp++;
837 if (cp == statcode) { /* неверный формат, записать его в журнал */
838 log_msg(bp);
839 } else {
840 *cp++ = '\0';
841 reason = cp;
842 while (*cp != '\r' && *cp != '\n')
843 cp++;
844 *cp = '\0';
845 code = atoi(statcode);
846 if (HTTP_INFO(code))
847 continue;
848 if (!HTTP_SUCCESS(code)) { /* возможная ошибка: записать ее */
849 bp[datsz] = '\0';
850 log_msg("ошибка: %s", reason);
851 break;
852 }
```

[821–838] Размещаем буфер в динамической памяти и читаем данные из принтера, предполагая, что операция чтения займет не более 5 секунд. Пропускаем строку HTTP/1.1 в начале сообщения и все последующие пробельные символы. Далее должен располагаться числовой код статуса. Если это не так, выводим в журнал содержимое сообщения.

[839–844] Если в ответе обнаружен числовой код статуса, необходимо первый нецифровой символ заменить нулевым символом (это должен быть один из пробельных символов). Далее должна следовать строка с описанием причины (текст сообщения). Находим завершающие символы возврата каретки и перевода строки и также вставляем в конец строки завершающий нулевой символ.

[845–852] Код преобразуется в число с помощью функции `atoi`. Если это всего лишь информационное сообщение, мы игнорируем его и переходим к началу цикла, чтобы продолжить чтение. Мы ожидаем получить либо сообщение об успехе операции, либо сообщение об ошибке. Если получено сообщение об ошибке, мы выводим его в журнал и прерываем работу цикла.

```
853 /*
854 * Заголовок HTTP в порядке, но нам нужно проверить статус
855 * IPP. Для начала найдем атрибут Content-Length.
856 */
857 i = cp - bp;
858 for (;;) {
859 while (*cp != 'C' && *cp != 'c' && i < datsz) {
860 cp++;
861 i++;
862 }
863 if (i >= datsz) { /* продолжить чтение заголовка */
864 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
865 goto out;
866 } else {
867 cp = &bp[i];
868 datsz += nr;
869 }
870 }

871 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
872 cp += 15;
873 while (isspace((int)*cp))
874 cp++;
875 contentlen = cp;
876 while (isdigit((int)*cp))
877 cp++;
878 *cp++ = '\0';
879 i = cp - bp;
880 len = atoi(contentlen);
881 break;
882 } else {
883 cp++;
884 i++;
885 }
886 }
```

[853–870] Если получено сообщение об успехе, необходимо проверить статус в заголовке IPP. Мы ищем в тексте сообщения строку `Content-Length`, которая может начинаться с символа С или с. Так как заголовки HTTP нечувствительны к регистру, приходится искать символы верхнего и нижнего регистров. При выходе за пределы буфера вызываем функцию `readmore`, которая вызывает функцию `realloc`, чтобы увеличить размер буфера. Это может привести к изменению адреса самого буфера, поэтому необходимо установить указатель `cp` так, чтобы он указывал на нужное место в буфере.

[871–886] Для сравнения без учета регистра символов мы используем функцию `strncasecmp`. После обнаружения строки с именем атрибута `Content-Length` нужно получить его значение. Преобразуем строку из цифровых символов в целое число, прерываем цикл `for`. Если операция сравнения потерпела неудачу, продолжаем поиск в буфере байт за байтом. Если достигнут конец буфера, но строка `Content-Length` не найдена, продолжаем работу цикла, повторяя чтение данных из принтера и продолжаем поиск.

```

887 if (i >= datsz) { /* продолжить чтение заголовка */
888 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
889 goto out;
890 } else {
891 cp = &bp[i];
892 datsz += nr;
893 }
894 }

895 found = 0;
896 while (!found) { /* поиск конца заголовка HTTP */
897 while (i < datsz - 2) {
898 if (*cp == '\n' && *(cp + 1) == '\r' &&
899 *(cp + 2) == '\n') {
900 found = 1;
901 cp += 3;
902 i += 3;
903 break;
904 }
905 cp++;
906 i++;
907 }
908 if (i >= datsz) { /* продолжить чтение заголовка */
909 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
910 goto out;
911 } else {
912 cp = &bp[i];
913 datsz += nr;
914 }
915 }
916 }
917 if (datsz - i < len) { /* продолжить чтение заголовка */
918 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
919 goto out;
920 } else {
921 cp = &bp[i];
922 datsz += nr;

```

[887–916] Теперь нам известна длина сообщения (по значению атрибута `Content-Length`). При исчерпании содержимого буфера мы продолжаем чтение данных от принтера. Далее мы начинаем поиск конца заголовка HTTP (пустой строки). Обнаружив такую строку, мы устанавливаем флаг `found` и пропускаем ее. Всякий раз, вызывая `readmore`, мы сохраняем в `cp` указатель на ту же позицию в самом буфере, на которой находились до вызова, на случай, если адрес буфера изменится в результате его расширения.

[917–922] Когда конец заголовка HTTP будет найден, мы вычисляем количество байтов, занимаемых заголовком HTTP. Если объем прочитанных данных, за вычетом размера заголовка HTTP, не совпадает с размером сообщения IPP (этот размер был получен из атрибута `Content-Length`), следовательно, нужно продолжить чтение.

```
923 }
924 }

925 memcpy(&h, cp, sizeof(struct ipp_hdr));
926 i = ntohs(h.status);
927 jobid = ntohl(h.request_id);

928 if (jobid != jp->jobid) {
929 /*
930 * Другие задания. Игнорировать их.
931 */
932 log_msg("задание %d, код статуса %d", jobid, i);
933 break;
934 }

935 if (STATCLASS_OK(i))
936 success = 1;
937 break;
938 }
939 }
940 out:
941 free(bp);
942 if (nr < 0) {
943 log_msg("задание %d: ошибка чтения ответа от принтера: %s",
944 jobid, strerror(errno));
945 }
946 return(success);
947 }
```

[923–927] Мы извлекаем код состояния и идентификатор задания из заголовка IPP. Оба значения хранятся в виде целых чисел с сетевым порядком байтов, поэтому их нужно преобразовать в значения с аппаратным порядком байтов вызовом функций `ntohs` и `ntohl`.

[928–939] Если идентификатор задания не совпадает с ожидаемым, выводим в журнал сообщение и прерываем выполнение внешнего цикла `while`. Если статус IPP говорит об успехе, мы сохраняем возвращаемое значение и прерываем работу цикла.

[940–947] Перед возвратом мы освобождаем буфер, где хранилось сообщение-ответ. Если наш запрос успешно выполнен, возвращается значение 1, если обнаружена ошибка — 0.

На этом мы заканчиваем рассмотрение этого достаточно объемного примера. Программы из этой главы были протестированы с сетевым PostScript-принтером Xerox Phaser 8560. К сожалению, этот принтер не позволяет отключить функцию автоматического распознавания документов, даже когда принудительно указан тип документа `text/plain`. Из-за этого нам пришлось использовать грубый прием, чтобы обмануть функцию принтера автоматического определения формата документа и обеспечить возможность печати документов в простом текстовом виде. Однако документы в формате PostScript можно также печатать в простом текстовом виде, используя некоторые другие утилиты, такие как `a2ps(1)`, инкапсулирующие программы PostScript перед печатью.

21.6. Подведение итогов

В этой главе были подробно рассмотрены две законченные программы: демон спулема печати, который посылает задание печати сетевому принтеру, и утилита, которая может использоваться для передачи задания печати демону. Благодаря этому мы смогли увидеть, как в реальной программе могут использоваться функциональные возможности, описанные в предыдущих главах: потоки выполнения, мультиплексирование ввода/вывода, операции файлового ввода/вывода, сокеты и сигналы.

Упражнения

- 21.1** Переведите значения кодов ошибок IPP, перечисленные в `ipp.h`, в текстовые сообщения. Затем измените демон печати так, чтобы в конце функции `printer_status` в журнал выводилось текстовое сообщение об ошибке, когда заголовок IPP свидетельствует о ее наличии.
- 21.2** Добавьте в утилиту `print` и демон `printd` поддержку двусторонней печати. Добавьте также возможность изменения ориентации бумаги.
- 21.3** Измените демон печати так, чтобы при запуске он запрашивал у принтера перечень поддерживаемых функциональных возможностей. Это необходимо, чтобы демон не указывал атрибуты, которые не поддерживаются принтером.
- 21.4** Напишите утилиту, с помощью которой можно было бы получать информацию о состоянии заданий, стоящих в очереди.
- 21.5** Напишите утилиту, с помощью которой можно было бы отменить печать задания, стоящего в очереди. Идентификатор отменяемого задания должен передаваться в виде аргумента командной строки. Как можно помешать пользователям отменять чужие задания?
- 21.6** Добавьте в демон печати возможность одновременной работы с несколькими принтерами. Предусмотрите возможность переброски задания печати с одного принтера на другой.
- 21.7** Объясните, почему не требуется принудительно перечитывать конфигурационный файл в демоне печати, когда поток обработки сигналов принимает сигнал `SIGHUP` и присваивает переменной `reread` значение 1.
- 21.8** В функции `printer_status` мы определяем длину заголовка IPP по атрибуту `Content-Length`. Этот прием не будет работать с принтерами, отправляющими сообщения фрагментами (`chunked transfer encoding`). Изучите RFC 2616, где описывается формат фрагментированной передачи, и затем добавьте в функцию `printer_status` поддержку ответов такого рода.
- 21.9** Когда в функции `get_newjobno` в результате увеличения номера следующего задания `nextjob` происходит переполнение, этой переменной присваивается значение 1 (загляните в функцию `get_newjobno`, чтобы увидеть, как это происходит). В результате функция `update_jobno` может записать меньшее число поверх большего. Это может привести к тому, что при запуске демон прочитает неправильное число. Как проще всего решить эту проблему?

У. Ричард Стивенс, Стивен А. Раго

UNIX. Профессиональное программирование

3-е издание

Перевел с английского А. Киселев

Заведующая редакцией

Ю. Сергиенко

Ведущий редактор

Н. Римицан

Художественный редактор

С. Заматевская

Корректоры

С. Беляева, Н. Викторова

Верстка

Н. Лукьянова

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,

улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.

Дата изготовления: 12.2017. 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 04.12.17. Формат 70x100/16. Бумага офсетная. Усл. п. л. 76,110. 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

A Прототипы функций

Это приложение содержит прототипы функций, определяемых стандартами ISO C и POSIX, а также функций UNIX, описанных в этой книге. Часто необходимо узнать, какие аргументы принимает та или иная функция («В каком аргументе функции `fgets` передается указатель на структуру `FILE?`») или что она возвращает («Что возвращает функция `sprintf` — указатель или счетчик?»). В описаниях прототипов указаны заголовочные файлы, которые нужно подключить для получения определений всех специальных констант и прототипов функций ISO C, что поможет в диагностике ошибок времени компиляции.

Для каждой функции справа от первого заголовочного файла указывается номер страницы в книге, где приводился прототип этой функции. Там же следует искать дополнительную информацию о ней.

Некоторые функции поддерживаются не всеми платформами, описанными в этой книге. Кроме того, некоторые платформы поддерживают флаги функций, не поддерживаемые другими платформами. Обычно мы будем перечислять платформы, которые поддерживают ту или иную функциональность. Однако в отдельных случаях будут перечислены платформы, на которых поддержка отсутствует.

void	abort (void);	<stdlib.h>	c. 434
Эта функция никогда не возвращает управление			
int	accept (int <i>sockfd</i> , struct <i>sockaddr</i> *restrict <i>addr</i> , socklen_t *restrict <i>len</i>);	<sys/socket.h>	c. 697
Возвращает дескриптор файла (сокета) в случае успеха, -1 — в случае ошибки			
int	access (const char * <i>path</i> , int <i>mode</i>);	<unistd.h>	c. 147
<i>mode</i> : R_OK, W_OK, X_OK, F_OK			
Возвращает 0 в случае успеха, -1 — в случае ошибки			

int	aio_cancel(int <i>fd</i> , struct aiocb * <i>aiocb</i>); <aio.h>	c. 594
	Возвращает AIO_ALLDONE, AIO_CANCELED или AIO_NOTCANCELED в случае успеха, -1 — в случае ошибки	
int	aio_error(const struct aiocb * <i>aiocb</i>); <aio.h>	c. 598
	Возвращает 0 в случае успеха, EINPROGRESS — если выполнение операции продолжается, код ошибки — если операция потерпела неудачу, -1 — в случае ошибки	
int	aio_fsync(int <i>op</i> , struct aiocb * <i>aiocb</i>); <aio.h>	c. 592
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	aio_read(struct aiocb * <i>aiocb</i>); <aio.h>	c. 592
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
ssize_t	aio_return(const struct aiocb * <i>aiocb</i>); <aio.h>	c. 593
	Возвращает результат асинхронной операции в случае успеха, -1 — в случае ошибки	
int	aio_suspend(const struct aiocb *const <i>list</i> [], int <i>nent</i> , const struct timespec * <i>timeout</i>); <aio.h>	c. 593
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	aio_write(struct aiocb * <i>aiocb</i>); <aio.h>	c. 592
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
unsigned int	alarm(unsigned int <i>seconds</i>); <unistd.h>	c. 406
	Возвращает 0 или число секунд, оставшихся до истечения установленного интервала времени	
int	atexit(void (* <i>func</i>)(void)); <stdlib.h>	c. 255
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки	

int	bind(int sockfd, const struct sockaddr *addr, socklen_t len);	
	<sys/socket.h>	c. 692
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	*calloc(size_t nobj, size_t size);	
	<stdlib.h>	c. 262
	Возвращает непустой указатель в случае успеха, NULL — в случае ошибки	
speed_t	cfgetispeed(const struct termios *termptr);	
	<termios.h>	c. 785
	Возвращает значение скорости в бодах	
speed_t	cfgetospeed(const struct termios *termptr);	
	<termios.h>	c. 785
	Возвращает значение скорости в бодах	
int	cfsetispeed(struct termios *termptr, speed_t speed);	
	<termios.h>	c. 785
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	cfsetospeed(struct termios *termptr, speed_t speed);	
	<termios.h>	c. 785
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	chdir(const char *path);	
	<unistd.h>	c. 184
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	chmod(const char *path, mode_t mode);	
	<sys/stat.h>	c. 151
	mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	chown(const char *path, uid_t owner, gid_t group);	
	<unistd.h>	c. 155
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	clearerr(FILE *fp);	
	<stdio.h>	c. 201

int	clock_getres (clockid_t <i>clock_id</i> , struct timespec * <i>tsp</i>);	c. 244
	<sys/time.h>	
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC,	
	CLOCK_PROCESS_CPUTIME_ID,	
	CLOCK_THREAD_CPUTIME_ID	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	clock_gettime (clockid_t <i>clock_id</i> , struct timespec * <i>tsp</i>);	c. 243
	<sys/time.h>	
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC,	
	CLOCK_PROCESS_CPUTIME_ID,	
	CLOCK_THREAD_CPUTIME_ID	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	clock_nanosleep (clockid_t <i>clock_id</i> , int <i>flags</i> ,	c. 445
	const struct timespec * <i>reqtp</i> ,	
	struct timespec * <i>remtp</i>);	
	<time.h>	
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC,	
	CLOCK_PROCESS_CPUTIME_ID,	
	CLOCK_THREAD_CPUTIME_ID	
	<i>flags</i> : TIMER_ABSTIME	
	Возвращает 0, если установленное время истекло,	
	или код ошибки — в случае неудачи	
int	clock_settime (clockid_t <i>clock_id</i> , const struct timespec * <i>tsp</i>);	c. 244
	<sys/time.h>	
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC,	
	CLOCK_PROCESS_CPUTIME_ID,	
	CLOCK_THREAD_CPUTIME_ID	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	close (int <i>fd</i>);	c. 109
	<unistd.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	closedir (DIR * <i>dp</i>);	c. 179
	<dirent.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	closeelog (void);	c. 545
	<syslog.h>	

<pre>unsigned char *CMMSG_DATA(struct cmsghdr *cp); <sys/socket.h></pre>	c. 735
<p>Возвращает указатель на данные, связанные со структурой <code>cmsghdr</code></p>	
<pre>struct cmsghdr *CMMSG_FIRSTHDR(struct msghdr *mp); <sys/socket.h></pre>	c. 735
<p>Возвращает указатель на первую структуру <code>cmsghdr</code>, связанную со структурой <code>msghdr</code>, или <code>NULL</code> — если такой не существует</p>	
<pre>unsigned int CMMSG_LEN(unsigned int nbytes); <sys/socket.h></pre>	c. 735
<p>Возвращает объем памяти, который необходимо выделить для хранения объекта размером <code>nbytes</code></p>	
<pre>struct cmsghdr *CMMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp); <sys/socket.h></pre>	c. 735
<p>Возвращает указатель на следующую структуру <code>cmsghdr</code>, связанную со структурой <code>msghdr</code>, которую представляет текущая структура <code>cmsghdr</code>, или <code>NULL</code> — если таковой не существует</p>	
<pre>int connect(int sockfd, const struct sockaddr *addr, socklen_t len); <sys/socket.h></pre>	c. 694
<p>Возвращает 0 в случае успеха, <code>-1</code> — в случае ошибки</p>	
<pre>int creat(const char *path, mode_t mode); <fcntl.h></pre>	c. 108
<p><code>mode</code>: <code>S_IS[UG]ID</code>, <code>S_ISVTX</code>, <code>S_I[RWX](USR GRP OTH)</code></p>	
<p>Возвращает дескриптор файла, открытый только для записи, в случае успеха, <code>-1</code> — в случае ошибки</p>	
<pre>char *ctermid(char *ptr); <stdio.h></pre>	c. 787
<p>Возвращает указатель на имя управляющего терминала в случае успеха, указатель на пустую строку — в случае ошибки</p>	

int	dprintf (int <i>fd</i> , const char *restrict <i>format</i> , ...); <stdio.h>	c. 210
	Возвращает число выведенных символов в случае успеха, отрицательное значение — в случае ошибки	
int	dup (int <i>fd</i>); <unistd.h>	c. 123
	Возвращает новый дескриптор файла в случае успеха, -1 — в случае ошибки	
int	dup2 (int <i>fd</i> , int <i>fd2</i>); <unistd.h>	c. 123
	Возвращает новый дескриптор файла в случае успеха, -1 — в случае ошибки	
void	endgrent (void); <grp.h>	c. 236
void	endhostent (void); <netdb.h>	c. 685
void	endnetent (void); <netdb.h>	c. 686
void	endprotoent (void); <netdb.h>	c. 687
void	endpwent (void); <pwd.h>	c. 232
void	endservent (void); <netdb.h>	c. 687
void	endspent (void); <shadow.h>	c. 235
	Платформы: Linux 3.2.0, Solaris 10	
int	execl (const char * <i>path</i> , const char * <i>arg0</i> , ... /* (char *) 0 */); <unistd.h>	c. 309
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	

int	<code>execle(const char *path, const char *arg0, ... /* (char *) 0, char *const envp[] */);</code>	c. 309
	<unistd.h>	
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	
int	<code>execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);</code>	c. 309
	<unistd.h>	
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	
int	<code>execv(const char *path, char *const argv[]);</code>	c. 309
	<unistd.h>	
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	
int	<code>execve(const char *path, char *const argv[], char *const envp[]);</code>	c. 309
	<unistd.h>	
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	
int	<code>execvp(const char *filename, char *const argv[]);</code>	c. 309
	<unistd.h>	
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	
void	<code>_Exit(int status);</code>	c. 253
	<stdlib.h>	
	Эта функция никогда не возвращает управление	
void	<code>_exit(int status);</code>	c. 253
	<unistd.h>	
	Эта функция никогда не возвращает управление	
void	<code>exit(int status);</code>	c. 253
	<stdlib.h>	
	Эта функция никогда не возвращает управление	
int	<code>faccessat(int fd, const char *path, int mode, int flag);</code>	c. 147
	<unistd.h>	
	<i>mode</i> : R_OK, W_OK, X_OK, F_OK	
	<i>flag</i> : AT_EACCESS	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	fchdir(int fd); <unistd.h>	c. 184
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fchmod(int fd, mode_t mode); <sys/stat.h>	c. 151
	<i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fchmodat(int fd, const char *path, mode_t mode, int flag); <sys/stat.h>	c. 151
	<i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)	
	<i>flag</i> : AT_SYMLINK_NOFOLLOW	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fchown(int fd, uid_t owner, gid_t group); <unistd.h>	c. 155
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fchownat(int fd, const char *path, uid_t owner, gid_t group, int flag); <unistd.h>	c. 155
	<i>flag</i> : AT_SYMLINK_NOFOLLOW	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fclose(FILE *fp); <stdio.h>	c. 200
	Возвращает 0 в случае успеха, EOF — в случае ошибки	
int	fcntl(int fd, int cmd, ... /* int arg */); <fcntl.h>	c. 126
	<i>cmd</i> : F_DUPFD, F_DUPFD_CLOEXEC, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETOWN, F_SETOWN, F_GETLK, F_SETLK, F_SETLKW	
	Возвращаемое значение зависит от аргумента <i>cmd</i> в случае успеха, -1 — в случае ошибки	
int	fdatsync(int fd); <unistd.h>	c. 125
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
	Платформы: Linux 3.2.0, Solaris 10	

void	FD_CLR(int fd, fd_set *fdset); <sys/select.h>	c. 582
int	FD_ISSET(int fd, fd_set *fdset); <sys/select.h>	c. 582
	Возвращает ненулевое значение, если <i>fd</i> имеется в наборе, 0 — в противном случае	
FILE	*fdopen(int fd, const char *type); <stdio.h> <i>type</i> : "r", "w", "a", "r+", "w+", "a+"	c. 197
	Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки	
DIR	*fdopendir(int fd); <dirent.h>	c. 179
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
void	FD_SET(int fd, fd_set *fdset); <sys/select.h>	c. 582
void	FD_ZERO(fd_set *fdset); <sys/select.h>	c. 582
int	feof(FILE *fp); <stdio.h>	c. 201
	Возвращает ненулевое значение (истина), если достигнут конец файла, 0 (ложь) — в противном случае	
	int ferror(FILE *fp); <stdio.h>	c. 201
	Возвращает ненулевое значение (истина), если при работе с потоком возникла ошибка, 0 (ложь) — в противном случае	
int	fexecve(int fd, char *const argv[], char *const envp[]); <unistd.h>	c. 309
	Возвращает -1 в случае ошибки, в случае успеха не возвращает управление	

<code>int fflush(FILE *fp);</code>	<code><stdio.h></code>	c. 197
	Возвращает 0 в случае успеха, EOF — в случае ошибки	
<code>int fgetc(FILE *fp);</code>	<code><stdio.h></code>	c. 201
	Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки	
<code>int fgetpos(FILE *restrict fp, fpos_t *restrict pos);</code>	<code><stdio.h></code>	c. 210
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки	
<code>char *fgets(char *restrict buf, int n, FILE *restrict fp);</code>	<code><stdio.h></code>	c. 203
	Возвращает указатель на <i>buf</i> в случае успеха, NULL — по достижении конца файла или в случае ошибки	
<code>int fileno(FILE *fp);</code>	<code><stdio.h></code>	c. 216
	Возвращает дескриптор файла, ассоциированный с потоком в случае успеха, -1 — в случае ошибки	
<code>void flockfile(FILE *fp);</code>	<code><stdio.h></code>	c. 515
<code>FILE *fmemopen(void *restrict buf, size_t size,</code>	<code><stdio.h></code>	c. 224
<code>const char *restrict type);</code>		
<code>type: "r", "w", "a", "r+", "w+", "a+"</code>		
	Возвращает указатель на поток в случае успеха, NULL — в случае ошибки	
<code>FILE *fopen(const char *restrict path, const char *restrict type);</code>	<code><stdio.h></code>	c. 197
<code>type: "r", "w", "a", "r+", "w+", "a+"</code>		
	Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки	

pid_t	fork(void);	c. 286
	<unistd.h>	
	Возвращает 0 в дочернем процессе, идентификатор дочернего процесса — в родительском процессе, -1 — в случае ошибки	
long	fpathconf(int fd, int name);	c. 82
	<unistd.h>	
	name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,	
	_PC_FILESIZEBITS, _PC_LINK_MAX,	
	_PC_MAX_CANON, _PC_MAX_INPUT,	
	_PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX,	
	_PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX,	
	_PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION,	
	_PC_2_SYMLINKS, _PC_VDISABLE	
	Возвращает соответствующее значение в случае успеха, -1 — в случае ошибки	
int	fprintf(FILE *restrict fp, const char *restrict format, ...);	c. 210
	<stdio.h>	
	Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки	
int	fputc(int c, FILE *fp);	c. 202
	<stdio.h>	
	Возвращает символ <i>c</i> в случае успеха, EOF — в случае ошибки	
int	fputs(const char *restrict str, FILE *restrict fp);	c. 204
	<stdio.h>	
	Возвращает неотрицательное значение в случае успеха, EOF — в случае ошибки	
size_t	fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);	c. 207
	<stdio.h>	
	Возвращает количество прочитанных блоков	
void	free(void *ptr);	c. 262
	<stdlib.h>	
void	freeaddrinfo(struct addrinfo *ai);	c. 688
	<sys/socket.h>	
	<netdb.h>	

FILE	*freopen(const char *restrict path, const char *restrict type, FILE *restrict fp);	c. 197
	<stdio.h>	
	<i>type</i> : "r", "w", "a", "r+", "w+", "a+"	
	Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки	
int	fscanf(FILE *restrict fp, const char *restrict format, ...);	c. 214
	<stdio.h>	
	Возвращает количество введенных элементов, EOF — по достижении конца файла или в случае ошибки перед выполнением преобразования	
int	fseek(FILE *fp, long offset, int whence);	c. 209
	<stdio.h>	
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fseeko(FILE *fp, off_t offset, int whence);	c. 209
	<stdio.h>	
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fsetpos(FILE *fp, const fpos_t *pos);	c. 210
	<stdio.h>	
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки	
int	fstat(int fd, struct stat *buf);	c. 137
	<sys/stat.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fstatat(int fd, const char *restrict path, struct stat *restrict buf, int flag);	c. 137
	<sys/stat.h>	
	<i>flag</i> : AT_SYMLINK_NOFOLLOW	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	fsync(int fd);	c. 125
	<unistd.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

long	f.tell(FILE *fp);	<stdio.h>	c. 209
Возвращает значение текущей позиции в файле в случае успеха, -1L — в случае ошибки			
off_t	f.tello(FILE *fp);	<stdio.h>	c. 209
Возвращает значение текущей позиции в файле в случае успеха, (off_t)-1 — в случае ошибки			
key_t	f.tok(const char *path, int id);	<sys/ipc.h>	c. 639
Возвращает значение ключа в случае успеха, (key_t)-1 — в случае ошибки			
int	f.truncate(int fd, off_t Length);	<unistd.h>	c. 158
Возвращает 0 в случае успеха, -1 — в случае ошибки			
int	f.trylockfile(FILE *fp);	<stdio.h>	c. 515
Возвращает 0 в случае успеха, ненулевое значение — если блокировка не может быть установлена			
void	f.unlockfile(FILE *fp);	<stdio.h>	c. 515
int	f.utimens(int fd, const struct timespec times[2]);	<sys/stat.h>	c. 174
Возвращает 0 в случае успеха, -1 — в случае ошибки			
int	f.wide(FILE *fp, int mode);	<stdio.h> <wchar.h>	c. 193
Возвращает положительное значение, если поток ориентирован на работу с многобайтными (wide) символами, отрицательное — с однобайтными сим- волами и 0 — если поток не имеет ориентации			
size_t	f.write(const void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);	<stdio.h>	c. 207
Возвращает количество записанных блоков			

const char	*gai_strerror(int error);	c. 689
Возвращает указатель на строку с описанием ошибки		
int	getaddrinfo(const char *restrict host, const char *restrict service, const struct addrinfo *restrict hint, struct addrinfo **restrict res);	c. 688
Возвращает 0 в случае успеха, ненулевой код ошибки — в случае неудачи		
int	getc(FILE *fp);	c. 201
Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки		
int	getchar(void);	c. 201
Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки		
int	getchar_unlocked(void);	c. 516
Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки		
int	getc_unlocked(FILE *fp);	c. 516
Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки		
char	*getcwd(char *buf, size_t size);	c. 185
Возвращает указатель на <i>buf</i> в случае успеха, NULL — в случае ошибки		
gid_t	getegid(void);	c. 285
Возвращает эффективный идентификатор группы вызывающего процесса		

char	*getenv(const char *name);	c. 266
<stdlib.h>		
	Возвращает указатель на значение переменной окружения с именем <i>name</i> , NULL — если переменная не найдена	
uid_t	geteuid(void);	c. 285
<unistd.h>		
	Возвращает эффективный идентификатор пользователя вызывающего процесса	
gid_t	getgid(void);	c. 285
<unistd.h>		
	Возвращает реальный идентификатор группы вызывающего процесса	
struct group	*getgrent(void);	c. 236
<grp.h>		
	Возвращает указатель в случае успеха, NULL — по достижении конца файла или в случае ошибки	
struct group	*getgrgid(gid_t gid);	c. 235
<grp.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
struct group	*getgrnam(const char *name);	c. 235
<grp.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
int	getgroups(int gidsetsizе, gid_t groupList[]);	c. 237
<unistd.h>		
	Возвращает количество идентификаторов дополнительных групп в случае успеха, -1 — в случае ошибки	
struct hostent	*gethostent(void);	c. 685
<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки	

int	gethostname (char * <i>name</i> , int <i>namelen</i>);		c. 242
	<unistd.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки		
char	*getlogin (void);		c. 337
	<unistd.h>		
	Возвращает указатель на строку с именем пользователя в случае успеха, NULL — в случае ошибки		
int	getnameinfo (const struct sockaddr *restrict <i>addr</i> , socklen_t <i>alen</i> , char *restrict <i>host</i> , socklen_t <i>hostlen</i> , char *restrict <i>service</i> , socklen_t <i>servlen</i> , unsigned int <i>flags</i>);		c. 689
	<sys/socket.h>		
	<netdb.h>		
	<i>flags</i> : NI_DGRAM, NI_NAMEREQD, NI_NOFQDN, NI_NUMERICHOST, NI_NUMERICSCOPE, NI_NUMERICSERV		
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки		
struct netent	*getnetbyaddr (uint32_t <i>net</i> , int <i>type</i>);		c. 686
	<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки		
struct netent	*getnetbyname (const char * <i>name</i>);		c. 686
	<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки		
struct netent	*getnetent (void);		c. 686
	<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки		
int	getopt (int <i>argc</i> , const * const <i>argv</i> [], const char * <i>options</i>);		c. 752
	<fcntl.h>		
	extern int <i>optind</i> , <i>opterr</i> , <i>optopt</i> ;		
	extern char * <i>optarg</i> ;		
	Возвращает символ следующей опции или -1 — если все параметры обработаны		

int	getpeername (int <i>sockfd</i> , struct sockaddr *restrict <i>addr</i> , socklen_t *restrict <i>alenp</i>);	c. 693
<sys/socket.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
pid_t	getpgid (pid_t <i>pid</i>);	c. 356
<unistd.h>		
	Возвращает идентификатор группы процессов в случае успеха, -1 — в случае ошибки	
pid_t	getpgrp (void);	c. 356
<unistd.h>		
	Возвращает идентификатор группы процессов вызывающего процесса	
pid_t	getpid (void);	c. 285
<unistd.h>		
	Возвращает идентификатор процесса вызывающего процесса	
pid_t	getppid (void);	c. 285
<unistd.h>		
	Возвращает идентификатор родительского процесса	
int	getpriority (int <i>which</i> , id_t <i>who</i>);	c. 339
<sys/resource.h>		
	<i>which</i> : PRIO_PROCESS, PRIO_PGRP, PRIO_USER	
	Возвращает значение коэффициента уступчивости между -NZERO и NZERO-1 в случае успеха, -1 — в случае ошибки	
struct protoent *	getprotobyname (const char * <i>name</i>);	c. 687
<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
struct protoent *	getprotobynumber (int <i>proto</i>);	c. 687
<netdb.h>		
	Возвращает указатель в случае успеха, NULL — в случае ошибки	

<code>struct protoent *getprotoent(void);</code>	<code><netdb.h></code>	c. 687
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
<code>struct passwd *getpwent(void);</code>	<code><pwd.h></code>	c. 232
	Возвращает указатель в случае успеха, NULL — в случае ошибки или по достижении конца файла	
<code>struct passwd *getpwnam(const char *name);</code>	<code><pwd.h></code>	c. 232
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
<code>struct passwd *getpwuid(uid_t uid);</code>	<code><pwd.h></code>	c. 232
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
<code>int getrlimit(int resource, struct rlimit *rlptr);</code>	<code><sys/resource.h></code>	c. 277
	<i>resource:</i> RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE, RLIMIT_NOFILE, RLIMIT_STACK, RLIMIT_AS (FreeBSD 8.0, Linux 3.2.0, Solaris 10), RLIMIT_MEMLOCK (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_MSGQUEUE (Linux 3.2.0), RLIMIT_NICE (Linux 3.2.0), RLIMIT_NPROC (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_NPTS (FreeBSD 8.0), RLIMIT_RSS (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_SBSIZE (FreeBSD 8.0), RLIMIT_SIGPENDING (Linux 3.2.0), RLIMIT_SWAP (FreeBSD 8.0), RLIMIT_VMEM (Solaris 10)	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

char	*gets(char *buf);	c. 203
	<i><stdio.h></i>	
	Возвращает указатель на <i>buf</i> в случае успеха, NULL — по достижении конца файла или в случае ошибки	
struct		
servent	*getservbyname(const char *name, const char *proto);	c. 687
	<i><netdb.h></i>	
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
struct		
servent	*getservbyport(int port, const char *proto);	c. 687
	<i><netdb.h></i>	
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
struct		
servent	*getservent(void);	c. 687
	<i><netdb.h></i>	
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
pid_t	getsid(pid_t pid);	c. 359
	<i><unistd.h></i>	
	Возвращает идентификатор группы процессов лидера сеанса в случае успеха, -1 — в случае ошибки	
int	getsockname(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict alenp);	c. 693
	<i><sys/socket.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	getsockopt(int sockfd, int level, int option, void *restrict val, socklen_t *restrict lenp);	c. 714
	<i><sys/socket.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
struct		
spwd	*getspent(void);	c. 235
	<i><shadow.h></i>	
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
	Платформы: Linux 3.2.0, Solaris 10	

<code>struct spwd *getspnam(const char *name);</code>	<code><shadow.h></code>	c. 235
	Возвращает указатель в случае успеха, NULL — в случае ошибки	
	Платформы: Linux 3.2.0, Solaris 10	
<code>int gettimeofday(struct timeval *restrict tp, void *restrict tzp);</code>	<code><sys/time.h></code>	c. 244
	Всегда возвращает 0	
<code>uid_t getuid(void);</code>	<code><unistd.h></code>	c. 285
	Возвращает реальный идентификатор пользователя вызывающего процесса	
<code>struct tm *gmtime(const time_t *calptr);</code>	<code><time.h></code>	c. 246
	Возвращает указатель на структуру с временем, разложенным на составляющие, NULL — в случае ошибки	
<code>int grantpt(int fd);</code>	<code><stdlib.h></code>	c. 815
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
<code>uint32_t htonl(uint32_t hostint32);</code>	<code><arpa/inet.h></code>	c. 682
	Возвращает 32-разрядное целое с сетевым порядком байтов	
<code>uint16_t htons(uint16_t hostint16);</code>	<code><arpa/inet.h></code>	c. 682
	Возвращает 16-разрядное целое с сетевым порядком байтов	
<code>const char *inet_ntop(int domain, const void *restrict addr, char *restrict str, socklen_t size);</code>	<code><arpa/inet.h></code>	c. 684
	Возвращает указатель на строку с адресом в случае успеха, NULL — в случае ошибки	

int	inet_ntop (int <i>domain</i> , const char *restrict <i>str</i> , void *restrict <i>addr</i>); <i><arpa/inet.h></i>	c. 684
	Возвращает 1 в случае успеха, 0 — в случае неверного формата, -1 — в случае ошибки	
int	initgroups (const char * <i>username</i> , gid_t <i>basegid</i>); <i><grp.h></i> /* Linux и Solaris */ <i><unistd.h></i> /* FreeBSD и Mac OS X */	c. 237
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	ioctl (int <i>fd</i> , int <i>request</i> , ...); <i><unistd.h></i> /* System V */ <i><sys/ioctl.h></i> /* BSD и Linux */	c. 132
	Возвращает -1 в случае ошибки, любое другое значение — в случае успеха	
int	isatty (int <i>fd</i>); <i><unistd.h></i>	c. 788
	Возвращает 1 (истина), если это терминальное устройство, 0 (ложь) — в противном случае	
int	kill (pid_t <i>pid</i> , int <i>signo</i>); <i><signal.h></i>	c. 405
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	lchown (const char * <i>path</i> , uid_t <i>owner</i> , gid_t <i>group</i>); <i><unistd.h></i>	c. 155
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	link (const char * <i>existingpath</i> , const char * <i>newpath</i>); <i><unistd.h></i>	c. 163
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	linkat (int <i>efd</i> , const char * <i>existingpath</i> , int <i>nfd</i> , const char * <i>newpath</i> , int <i>flag</i>); <i><unistd.h></i> <i>flag</i> : AT_SYMLINK_NOFOLLOW	c. 163
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	lio_listio (int <i>mode</i> , struct aiocb *restrict const <i>list</i> [restrict], int <i>nent</i> , struct sigevent *restrict <i>sigeve</i>); <i><aiocb.h></i>	c. 594
	<i>mode</i> : LIO_NOWAIT, LIO_WAIT	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	listen (int <i>sockfd</i> , int <i>backLog</i>); <i><sys/socket.h></i>	c. 696
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
struct tm	*localtime (const time_t * <i>calptr</i>); <i><time.h></i>	c. 246
	Возвращает указатель на структуру с временем, раз- ложенным на составляющие, NULL — в случае ошибки	
void	longjmp (jmp_buf <i>env</i> , int <i>val</i>); <i><setjmp.h></i>	c. 272
	Эта функция никогда не возвращает управление	
off_t	lseek (int <i>fd</i> , off_t <i>offset</i> , int <i>whence</i>); <i><unistd.h></i>	c. 109
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END	
	Возвращает новую текущую позицию в файле в слу- чае успеха, -1 — в случае ошибки	
int	lstat (const char *restrict <i>path</i> , struct stat *restrict <i>buf</i>); <i><sys/stat.h></i>	c. 137
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	*malloc (size_t <i>size</i>); <i><stdlib.h></i>	c. 262
	Возвращает непустой указатель в случае успеха, NULL — в случае ошибки	
int	mkdir (const char * <i>path</i> , mode_t <i>mode</i>); <i><sys/stat.h></i>	c. 177
	<i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	mkdirat(int fd, const char *path, mode_t mode);	c. 177
	<i><sys/stat.h></i>	
	<i>mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)</i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
char	*mkdtemp(char *template);	c. 221
	<i><stdlib.h></i>	
	Возвращает указатель на имя каталога в случае успеха, NULL — в случае ошибки	
int	mkfifo(const char *path, mode_t mode);	c. 634
	<i><sys/stat.h></i>	
	<i>mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)</i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	mkfifoat(int fd, const char *path, mode_t mode);	c. 634
	<i><sys/stat.h></i>	
	<i>mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH)</i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	mkstemp(char *template);	c. 221
	<i><stdlib.h></i>	
	Возвращает дескриптор файла в случае успеха, -1 — в случае ошибки	
time_t	mktime(struct tm *tmptr);	c. 246
	<i><time.h></i>	
	Возвращает календарное время в случае успеха, -1 — в случае ошибки	
void	*mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);	c. 605
	<i><sys/mman.h></i>	
	<i>prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE</i>	
	<i>flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE</i>	
	Возвращает начальный адрес отображенной области в случае успеха, MAP_FAILED — в случае ошибки	
int	mprotect(void *addr, size_t len, int prot);	c. 608
	<i><sys/mman.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	<code>msgctl(int msqid, int cmd, struct msqid_ds *buf);</code>	c. 645
	<code><sys/msg.h></code>	
	<code>cmd: IPC_STAT, IPC_SET, IPC_RMID</code>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	<code>msgget(key_t key, int flag);</code>	c. 644
	<code><sys/msg.h></code>	
	<code>flag: IPC_CREAT, IPC_EXCL</code>	
	Возвращает идентификатор очереди сообщений в случае успеха, -1 — в случае ошибки	
ssize_t	<code>msgrecv(int msqid, void *ptr, size_t nbytes, long type, int flag);</code>	c. 647
	<code><sys/msg.h></code>	
	<code>flag: IPC_NOWAIT, MSG_NOERROR</code>	
	Возвращает размер блока данных сообщения в случае успеха, -1 — в случае ошибки	
int	<code>msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);</code>	c. 645
	<code><sys/msg.h></code>	
	<code>flag: IPC_NOWAIT</code>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	<code>msync(void *addr, size_t len, int flags);</code>	c. 608
	<code><sys/mman.h></code>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	<code>munmap(void *addr, size_t len);</code>	c. 609
	<code><sys/mman.h></code>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	<code>nanosleep(const struct timespec *reqtp,</code> <code> struct timespec *remtp);</code>	c. 444
	<code><time.h></code>	
	Возвращает 0, если установленное время истекло, -1 — в случае неудачи	
int	<code>nice(int incr);</code>	c. 339
	<code><unistd.h></code>	
	Возвращает новое значение коэффициента уступчивости минус NZERO в случае успеха, -1 — в случае ошибки	

<code>uint32_t ntohs(uint32_t netint32);</code>	<code><arpa/inet.h></code>	c. 682
Возвращает 32-разрядное целое с аппаратным порядком байтов		
<code>uint16_t ntohs(uint16_t netint16);</code>	<code><arpa/inet.h></code>	c. 682
Возвращает 16-разрядное целое с аппаратным порядком байтов		
<code>int open(const char *path, int oflag, ... /* mode_t mode */);</code>	<code><fcntl.h></code>	c. 104
<i>oflag</i> : O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, O_SEARCH; O_APPEND, O_CLOEXEC, O_CREAT, O_DIRECTORY, O_DSYNC, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT <i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH) Возвращает дескриптор файла в случае успеха, -1 — в случае ошибки Платформы: флаг O_FSYNC — для FreeBSD 8.0 и Mac OS X 10.6.8		
<code>int openat(int fd, const char *path, int oflag, ... /* mode_t mode */);</code>	<code><fcntl.h></code>	c. 104
<i>oflag</i> : O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, O_SEARCH; O_APPEND, O_CLOEXEC, O_CREAT, O_DIRECTORY, O_DSYNC, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT <i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH) Возвращает дескриптор файла в случае успеха, -1 — в случае ошибки Платформы: флаг O_FSYNC — для FreeBSD 8.0 и Mac OS X 10.6.8		
<code>DIR *opendir(const char *path);</code>	<code><direct.h></code>	c. 179
Возвращает указатель на структуру DIR в случае успеха, NULL — в случае ошибки		

void	openlog (const char * <i>ident</i> , int <i>option</i> , int <i>facility</i>);	c. 545
	<i><syslog.h></i>	
	<i>option</i> : LOG_CONS, LOG_NDELAY, LOG_NOWAIT,	
	LOG_ODELAY, LOG_PERROR, LOG_PID	
	<i>facility</i> : LOG_AUTH, LOG_AUTHPRIV, LOG_CRON,	
	LOG_DAEMON, LOG_FTP, LOG_KERN,	
	LOG_LOCAL[0-7], LOG_LPR, LOG_MAIL,	
	LOG_NEWS, LOG_SYSLOG, LOG_USER, LOG_UUCP	
FILE	*open_memstream (char ** <i>bufp</i> , size_t * <i>sizep</i>);	c. 226
	<i><stdio.h></i>	
	Возвращает указатель на поток ввода/вывода в случае успеха, NULL — в случае ошибки	
FILE	*open_wmemstream (wchar_t ** <i>bufp</i> , size_t * <i>sizep</i>);	c. 226
	<i><wchar.h></i>	
	Возвращает указатель на поток ввода/вывода в случае успеха, NULL — в случае ошибки	
long	pathconf (const char * <i>path</i> , int <i>name</i>);	c. 82
	<i><unistd.h></i>	
	<i>name</i> : _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,	
	_PC_FILESIZEBITS, _PC_LINK_MAX,	
	_PC_MAX_CANON, _PC_MAX_INPUT,	
	_PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX,	
	_PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX,	
	_PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION,	
	_PC_2_SYMLINKS, _PC_VDISABLE	
	Возвращает соответствующее значение в случае успеха, -1 — в случае ошибки	
int	pause (void);	c. 407
	<i><unistd.h></i>	
	В случае ошибки возвращает -1 и код ошибки EINTR в переменной errno	
int	pclose (FILE * <i>fp</i>);	c. 623
	<i><stdio.h></i>	
	Возвращает код завершения команды cmdstring функции popen, -1 — в случае ошибки	

void	perror (const char *msg);		c. 49
	<stdio.h>		
int	pipe (int fd[2]);		c. 616
	<unistd.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки		
int	poll (struct pollfd fdarray[], nfds_t nfds, int timeout);		c. 585
	<poll.h>		
	Возвращает количество дескрипторов, готовых к выполнению операции, 0 — в случае истечения времени тайм-аута, -1 — в случае ошибки		
FILE	*popen (const char *cmdstring, const char *type);		c. 623
	<stdio.h>		
	type: "r", "w"		
	Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки		
int	posix_openpt (int oflag);		c. 815
	<stdlib.h>		
	<fcntl.h>		
	oflag: O_RDWR, O_NOCTTY		
	Возвращает дескриптор следующего доступного ведущего PTY в случае успеха, -1 — в случае ошибки		
ssize_t	pread (int fd, void *buf, size_t nbytes, off_t offset);		c. 122
	<unistd.h>		
	Возвращает количество прочитанных байтов, 0 по достижении конца файла, -1 — в случае ошибки		
int	printf (const char *restrict format, ...);		c. 210
	<stdio.h>		
	Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки		
int	pselect (int maxfdp1, fd_set *restrict readfds,		c. 584
	fd_set *restrict writefds, fd_set *restrict exceptfds,		
	const struct timespec *restrict tsprt,		
	const sigset_t *restrict sigmask);		
	<sys/select.h>		
	Возвращает количество дескрипторов, готовых к выполнению операции, 0 — в случае истечения тайм-аута, -1 — в случае ошибки		

void	psiginfo (const siginfo_t *info, const char *msg); <signal.h>	c. 450
void	psignal (int signo, const char *msg); <signal.h> <siginfo.h> /* в Solaris */	c. 450
int	pthread_atfork (void (*prepare)(void), void (*parent)(void), void (*child)(void); <pthread.h>)	c. 531
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_attr_destroy (pthread_attr_t *attr); <pthread.h>	c. 498
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate); <pthread.h>	c. 499
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_attr_getguardsize (const pthread_attr_t *restrict attr, size_t *restrict guardsize); <pthread.h>	c. 501
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_attr_getstack (const pthread_attr_t *restrict attr, void **restrict stackaddr, size_t *restrict stacksize); <pthread.h>	c. 500
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_attr_getstacksize (const pthread_attr_t *restrict attr, size_t *restrict stacksize); <pthread.h>	c. 501
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	<code>pthread_condattr_getclock(const pthread_condattr_t *restrict attr, clockid_t *restrict clock_id);</code>	c. 512
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_condattr_getpshared(const pthread_condattr_t *restrict attr, int *restrict pshared);</code>	c. 512
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_condattr_init(pthread_condattr_t *attr);</code>	c. 512
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_condattr_setclock(pthread_condattr_t *attr, clockid_t clock_id);</code>	c. 512
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);</code>	c. 512
	<i>pshared:</i> PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_cond_broadcast(pthread_cond_t *cond);</code>	c. 487
	Возвращает 0 в случае успеха, код ошибки в случае неудачи	
int	<code>pthread_cond_destroy(pthread_cond_t *cond);</code>	c. 486
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	<code>pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t *restrict attr);</code>	c. 486
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_cond_signal(pthread_cond_t *cond);</code>	c. 487
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict timeout);</code>	c. 486
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);</code>	c. 486
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void), void *restrict arg);</code>	c. 457
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_detach(pthread_t tid);</code>	c. 468
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_equal(pthread_t tid1, pthread_t tid2);</code>	c. 456
	Возвращает ненулевое значение, если идентификаторы равны, 0 — в противном случае	
void	<code>pthread_exit(void *rval_ptr);</code>	c. 460
	<pthread.h>	

void	* pthread_getspecific (pthread_key_t key);	c. 522
<pthread.h>		
	Возвращает адрес области памяти с локальными данными потока или NULL, если ключ <i>key</i> не был ассоциирован с локальными данными	
int	pthread_join (pthread_t thread, void **rval_ptr);	c. 461
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_key_create (pthread_key_t *keyp, void (*destructor)(void *));	c. 519
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_key_delete (pthread_key_t key);	c. 520
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_kill (pthread_t thread, int signo);	c. 528
<signal.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_mutexattr_destroy (pthread_mutexattr_t *attr);	c. 502
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_mutexattr_getpshared (const pthread_mutexattr_t *restrict attr, int *restrict pshared);	c. 503
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_mutexattr_getrobust (const pthread_mutexattr_t *restrict attr, int *restrict robust);	c. 503
<pthread.h>		
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	pthread_mutexattr_gettype (const pthread_mutexattr_t *restrict <i>attr</i> , int *restrict <i>type</i>);	
	<pthread.h>	c. 505
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutexattr_init (pthread_mutexattr_t * <i>attr</i>);	
	<pthread.h>	c. 502
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutexattr_setpshared (pthread_mutexattr_t * <i>attr</i> , int <i>pshared</i>);	
	<pthread.h>	c. 503
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutexattr_setrobust (pthread_mutexattr_t * <i>attr</i> , int <i>robust</i>);	
	<pthread.h>	c. 503
	<i>robust</i> : PTHREAD_MUTEX_ROBUST, PTHREAD_MUTEX_STALLED	
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutexattr_settype (pthread_mutexattr_t * <i>attr</i> , int <i>type</i>);	
	<pthread.h>	c. 505
	<i>type</i> : PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE, PTHREAD_MUTEX_DEFAULT	
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutex_consistent (pthread_mutex_t * <i>mutex</i>);	
	<pthread.h>	c. 504
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	
int	pthread_mutex_destroy (pthread_mutex_t * <i>mutex</i>);	
	<pthread.h>	c. 472
	Возвращает 0 в случае успеха, код ошибки — в слу- чае неудачи	

int	<code>pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);</code>	c. 472
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_mutex_lock(pthread_mutex_t *mutex);</code>	c. 472
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict tspr);</code>	c. 479
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_mutex_trylock(pthread_mutex_t *mutex);</code>	c. 472
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_mutex_unlock(pthread_mutex_t *mutex);</code>	c. 472
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_once(pthread_once_t *initflag, void (*initfn)(void);</code>	c. 521
	<code> <pthread.h></code>	
	<code> pthread_once_t initflag = PTHREAD_ONCE_INIT;</code>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);</code>	c. 511
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr, int *restrict pshared);</code>	c. 511
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	<code>pthread_rwlockattr_init(pthread_rwlockattr_t *attr);</code>	c. 511
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,</code>	c. 511
	<i>int pshared);</i>	
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_destroy(pthread_rwlock_t *rwLock);</code>	c. 481
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_init(pthread_rwlock_t *restrict rwLock,</code>	c. 481
	<i>const pthread_rwlockattr_t *restrict attr);</i>	
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_rdlock(pthread_rwlock_t *rwLock);</code>	c. 482
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwLock,</code>	c. 485
	<i>const struct timespec *restrict tsprtr);</i>	
	<i><pthread.h></i>	
	<i><time.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwLock,</code>	c. 485
	<i>const struct timespec *restrict tsprtr);</i>	
	<i><pthread.h></i>	
	<i><time.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	<code>pthread_rwlock_tryrdlock(pthread_rwlock_t *rwLock);</code>	c. 482
	<i><pthread.h></i>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	pthread_rwlock_trywrlock (pthread_rwlock_t *rwLock);	
	<pthread.h>	c. 482
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_rwlock_unlock (pthread_rwlock_t *rwLock);	
	<pthread.h>	c. 482
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_rwlock_wrlock (pthread_rwlock_t *rwLock);	
	<pthread.h>	c. 482
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
pthread_t	pthread_self (void);	
	<pthread.h>	c. 456
	Возвращает идентификатор вызывающего потока	
int	pthread_setcancelstate (int state, int *oldstate);	
	<pthread.h>	c. 523
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_setcanceltype (int type, int *oldtype);	
	<pthread.h>	c. 526
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_setspecific (pthread_key_t key, const void *value);	
	<pthread.h>	c. 522
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_sigmask (int how, const sigset_t *restrict set, sigset_t *restrict oset);	
	<signal.h>	c. 527
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_spin_destroy (pthread_spinlock_t *lock);	
	<pthread.h>	c. 490
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	

int	pthread_spin_init(pthread_spinlock_t *Lock, int pshared);	c. 490
	<pthread.h>	
	<i>pshared</i> : PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_spin_lock(pthread_spinlock_t *Lock);	c. 490
	<pthread.h>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_spin_trylock(pthread_spinlock_t *Lock);	c. 490
	<pthread.h>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
int	pthread_spin_unlock(pthread_spinlock_t *Lock);	c. 490
	<pthread.h>	
	Возвращает 0 в случае успеха, код ошибки — в случае неудачи	
void	pthread_testcancel(void);	c. 526
	<pthread.h>	
char	*ptsname(int fd);	c. 816
	<stdlib.h>	
	Возвращает указатель на имя ведомого PTY в случае успеха, NULL — в случае ошибки	
int	putc(int c, FILE *fp);	c. 202
	<stdio.h>	
	Возвращает символ <i>c</i> в случае успеха, EOF — в случае ошибки	
int	putchar(int c);	c. 202
	<stdio.h>	
	Возвращает символ <i>c</i> в случае успеха, EOF — в случае ошибки	
int	putchar_unlocked(int c);	c. 516
	<stdio.h>	
	Возвращает символ <i>c</i> в случае успеха, EOF — в случае ошибки	

int	putc_unlocked (int <i>c</i> , FILE * <i>fp</i>);		
	<stdio.h>		c. 516
	Возвращает символ <i>c</i> в случае успеха, EOF — в случае ошибки		
int	putenv (char * <i>str</i>);		
	<stdlib.h>		c. 268
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки		
int	puts (const char * <i>str</i>);		
	<stdio.h>		c. 204
	Возвращает неотрицательное значение в случае успеха, EOF — в случае ошибки		
ssize_t	pwrite (int <i>fd</i> , const void * <i>buf</i> , size_t <i>nbytes</i> , off_t <i>offset</i>);		
	<unistd.h>		c. 122
	Возвращает количество записанных байтов в случае успеха, -1 — в случае ошибки		
int	raise (int <i>signo</i>);		
	<signal.h>		c. 405
	Возвращает 0 в случае успеха, -1 — в случае ошибки		
ssize_t	read (int <i>fd</i> , void * <i>buf</i> , size_t <i>nbytes</i>);		
	<unistd.h>		c. 113
	Возвращает количество прочитанных байтов в случае успеха, 0 — по достижении конца файла, -1 — в случае ошибки		
struct dirent	*readdir (DIR * <i>dp</i>);		
	<dirent.h>		c. 179
	Возвращает указатель в случае успеха, NULL — по достижении конца каталога или в случае ошибки		
ssize_t	readlink (const char *restrict <i>path</i> , char *restrict <i>buf</i> , size_t <i>bufsize</i>);		
	<unistd.h>		c. 171
	Возвращает количество прочитанных байтов в случае успеха, -1 — в случае ошибки		

```
ssize_t    readlinkat(int fd, const char *restrict path,
                      char *restrict buf, size_t bufsize);
                      <unistd.h>
```

c. 171

Возвращает количество прочитанных байтов в случае успеха, -1 — в случае ошибки

```
ssize_t    readv(int fd, const struct iovec *iov, int iovcnt);  
          <sys/uio.h>
```

c. 600

Возвращает количество прочитанных байтов в случае успеха, 0 — по достижении конца файла, -1 — в случае ошибки

```
void *realloc(void *ptr, size_t newsize);  
          <stdlib.h>
```

c. 262

Возвращает непустой указатель в случае успеха,
NULL — в случае ошибки.

```
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
           <sys/socket.h>
```

c. 701

flags: MSG_PEEK MSG_OOB MSG_WAITALL

MSG_CMSG_CLOEXEC (Linux 3.2.0)

MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
Solaris 10)

MSG_ERROUEUE (Linux 3.2.0)

MSG_TRUNC (Linux 3.2.0)

Возвращает длину сообщения в байтах, 0 – если нет доступных сообщений и на другом конце соединения была запрещена операция записи, -1 – в случае ошибки

```
ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr,
                 socklen_t *restrict addrlen);
```

c. 702

flags: MSG_PEEK, MSG_OOB, MSG_WAITALL,

MSG_CMSG_CLOEXEC (Linux 3.2.0),

`MSG_DONTWAIT` (FreeBSD 8.0, Linux 3.2.0,
Solaris 10),

MSG_ERRQUEUE (Linux 3.2.0),

MSG_TRUNC (Linux 3.2.0)

Возвращает длину сообщения в байтах, 0 – если нет доступных сообщений и на другом конце соединения запрещена операция записи, -1 – в случае ошибки

void	seekdir (DIR *dp, long loc); <dirent.h>	c. 179
int	select (int <i>maxfdp1</i> , fd_set *restrict <i>readfds</i> , fd_set *restrict <i>writefd</i> s, fd_set *restrict <i>exceptfd</i> s, struct timeval *restrict <i>tvp</i> tr); <sys/select.h>	c. 580
	Возвращает количество дескрипторов, готовых к выполнению операции, 0 – по истечении тайм- аута, –1 – в случае ошибки	
int	sem_close (sem_t *sem); <semaphore.h>	c. 665
	Возвращает 0 в случае успеха, –1 – в случае ошибки	
int	semctl (int semid, int semnum, int cmd, ... /* union semun arg */); <sys/sem.h> cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT, GETVAL, SETVAL, GETALL, SETALL	c. 651
	Возвращаемое значение зависит от типа команды, –1 – в случае ошибки	
int	sem_destroy (sem_t *sem); <semaphore.h>	c. 668
	Возвращает 0 в случае успеха, –1 – в случае ошибки	
int	semget (key_t key, int nsems, int flag); <sys/sem.h> flag: IPC_CREAT, IPC_EXCL	c. 651
	Возвращает идентификатор семафора в случае успе- ха, –1 – в случае ошибки	
int	sem_getvalue (sem_t *restrict sem, int *restrict valp); <semaphore.h>	c. 668
	Возвращает 0 в случае успеха, –1 – в случае ошибки	
int	sem_init (sem_t *sem, int pshared, unsigned int value); <semaphore.h>	c. 667
	Возвращает 0 в случае успеха, –1 – в случае ошибки	
int	semop (int semid, struct sembuf semoparray[], size_t nops); <sys/sem.h>	c. 652
	Возвращает 0 в случае успеха, –1 – в случае ошибки	

sem_t	*sem_open(const char *name, int oflag, ... /* mode_t mode, unsigned int value */); <semaphore.h>	c. 664
	flag: IPC_CREAT, IPC_EXCL Возвращает указатель на семафор в случае успеха, SEM_FAILED — в случае ошибки	
int	sem_post(sem_t *sem); <semaphore.h>	c. 667
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sem_timedwait(sem_t *restrict sem, const struct timespec *restrict tsprtr); <semaphore.h> <time.h>	c. 667
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sem_trywait(sem_t *sem); <semaphore.h>	c. 666
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sem_unlink(const char *name); <semaphore.h>	c. 666
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sem_wait(sem_t *sem); <semaphore.h>	c. 666
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
ssize_t	send(int sockfd, const void *buf, size_t nbytes, int flags); <sys/socket.h>	c. 698
	flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL, MSG_CONFIRM (Linux 3.2.0), MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10), MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10), MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8), MSG_MORE (Linux 3.2.0) Возвращает количество переданных байтов в случае успеха, -1 — в случае ошибки	

<code>ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);</code>	c. 700
<code><sys/socket.h></code>	
<code>flags:</code> <code>MSG_EOR, MSG_OOB, MSG_NOSIGNAL,</code>	
<code>MSG_CONFIRM (Linux 3.2.0),</code>	
<code>MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,</code>	
<code>Mac OS X 10.6.8, Solaris 10),</code>	
<code>MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,</code>	
<code>Mac OS X 10.6.8, Solaris 10),</code>	
<code>MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),</code>	
<code>MSG_MORE (Linux 3.2.0)</code>	
Возвращает количество переданных байтов в случае успеха, <code>-1</code> — в случае ошибки	
<code>ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,</code>	c. 699
<code>const struct sockaddr *destaddr, socklen_t destlen);</code>	
<code><sys/socket.h></code>	
<code>flags:</code> <code>MSG_EOR, MSG_OOB, MSG_NOSIGNAL,</code>	
<code>MSG_CONFIRM (Linux 3.2.0),</code>	
<code>MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,</code>	
<code>Mac OS X 10.6.8, Solaris 10),</code>	
<code>MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,</code>	
<code>Mac OS X 10.6.8, Solaris 10),</code>	
<code>MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),</code>	
<code>MSG_MORE (Linux 3.2.0)</code>	
Возвращает количество переданных байтов в случае успеха, <code>-1</code> — в случае ошибки	
<code>void setbuf(FILE *restrict fp, char *restrict buf);</code>	c. 196
<code><stdio.h></code>	
<code>int setegid(gid_t gid);</code>	c. 319
<code><unistd.h></code>	
Возвращает <code>0</code> в случае успеха, <code>-1</code> — в случае ошибки	
<code>int setenv(const char *name, const char *value, int rewrite);</code>	c. 268
<code><stdlib.h></code>	
Возвращает <code>0</code> в случае успеха, <code>-1</code> — в случае ошибки	
<code>int seteuid(uid_t uid);</code>	c. 319
<code><unistd.h></code>	
Возвращает <code>0</code> в случае успеха, <code>-1</code> — в случае ошибки	

int	setgid(gid_t <i>gid</i>);	c. 316
	<i><unistd.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	setgrent(void);	c. 236
	<i><grp.h></i>	
int	setgroups(int <i>ngroups</i>, const gid_t <i>grouplist</i>[]);	c. 237
	<i><grp.h></i> /* в Linux */	
	<i><unistd.h></i> /* в FreeBSD, Mac OS X и Solaris */	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	sethostent(int <i>stayopen</i>);	c. 685
	<i><netdb.h></i>	
int	setjmp(jmp_buf <i>env</i>);	c. 272
	<i><setjmp.h></i>	
	Возвращает 0, если вызывается непосредственно, ненулевое значение — если возврат произошел вследствие вызова longjmp	
int	setlogmask(int <i>maskpri</i>);	c. 545
	<i><syslog.h></i>	
	Возвращает предыдущее значение маски приоритета журналируемых сообщений	
void	setnetent(int <i>stayopen</i>);	c. 686
	<i><netdb.h></i>	
int	setpgid(pid_t <i>pid</i>, pid_t <i>pgid</i>);	c. 357
	<i><unistd.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	setpriority(int <i>which</i>, id_t <i>who</i>, int <i>value</i>);	c. 340
	<i><sys/resource.h></i>	
	<i>which</i> : PRIO_PROCESS, PRIO_PGRP, PRIO_USER	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	setprotoent(int <i>stayopen</i>);	c. 687
	<i><netdb.h></i>	
void	setpwent(void);	c. 232
	<i><pwd.h></i>	

int	setregid(gid_t rgid, gid_t egid); <unistd.h>	c. 318
Возвращает 0 в случае успеха, -1 — в случае ошибки		
int	setreuid(uid_t ruid, uid_t euid); <unistd.h>	c. 318
Возвращает 0 в случае успеха, -1 — в случае ошибки		
int	setrlimit(int resource, const struct rlimit *rlptr); <sys/resource.h>	c. 277
resource: RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE, RLIMIT_NOFILE, RLIMIT_STACK, RLIMIT_AS (FreeBSD 8.0, Linux 3.2.0, Solaris 10), RLIMIT_MEMLOCK (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_MSGQUEUE (Linux 3.2.0), RLIMIT_NICE (Linux 3.2.0), RLIMIT_NPROC (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_NPTS (FreeBSD 8.0), RLIMIT_RSS (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8), RLIMIT_SBSIZE (FreeBSD 8.0), RLIMIT_SIGPENDING (Linux 3.2.0), RLIMIT_SWAP (FreeBSD 8.0), RLIMIT_VMEM (Solaris 10)		
Возвращает 0 в случае успеха, -1 — в случае ошибки		
void	setservent(int stayopen); <netdb.h>	c. 687
pid_t	setsid(void); <unistd.h>	c. 358
Возвращает идентификатор группы процессов в случае успеха, -1 — в случае ошибки		
int	setsockopt(int sockfd, int Level, int option, const void *val, socklen_t len); <sys/socket.h>	c. 713
Возвращает 0 в случае успеха, -1 — в случае ошибки		
void	setspent(void); <shadow.h>	c. 235
Платформы: Linux 3.2.0, Solaris 10		

int	setuid(uid_t uid);	c. 316
	<unistd.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);	c. 196
	<stdio.h>	
	<i>mode</i> : _IOFBF, _IOLBF, _IONBF	
	Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки	
void	*shmat(int shmid, const void *addr, int flag);	c. 658
	<sys/shm.h>	
	<i>flag</i> : SHM_RND, SHM_RDONLY	
	Возвращает указатель на сегмент разделяемой памяти в случае успеха, -1 — в случае ошибки	
int	shmctl(int shmid, int cmd, struct shmid_ds *buf);	c. 658
	<sys/shm.h>	
	<i>cmd</i> : IPC_STAT, IPC_SET, IPC_RMID,	
	SHM_LOCK (Linux 3.2.0, Solaris 10),	
	SHM_UNLOCK (Linux 3.2.0, Solaris 10)	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	shmctl(void *addr);	c. 659
	<sys/shm.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	shmget(key_t key, size_t size, int flag);	c. 657
	<sys/shm.h>	
	<i>flag</i> : IPC_CREAT, IPC_EXCL	
	Возвращает неотрицательный идентификатор сегмента разделяемой памяти в случае успеха, -1 — в случае ошибки	
int	shutdown(int sockfd, int how);	c. 680
	<sys/socket.h>	
	<i>how</i> : SHUT_RD, SHUT_WR, SHUT_RDWR	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sig2str(int signo, char *str);	c. 451
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
	Платформы: Solaris 10	

int	sigaction(int signo, const struct sigaction *restrict act, struct sigaction *restrict oact);	c. 418
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigaddset(sigset_t *set, int signo);	c. 412
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigdelset(sigset_t *set, int signo);	c. 412
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigemptyset(sigset_t *set);	c. 412
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigfillset(sigset_t *set);	c. 412
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigismember(const sigset_t *set, int signo);	c. 412
	<signal.h>	
	Возвращает 1, если утверждение истинно, 0 — если ложно, -1 — в случае ошибки	
void	siglongjmp(sigjmp_buf env, int val);	c. 425
	<setjmp.h>	
	Эта функция никогда не возвращает управление	
void	(*signal(int signo, void (*func)(int)))(int);	c. 389
	<signal.h>	
	Возвращает предыдущую диспозицию сигнала в случае успеха, SIG_ERR — в случае ошибки	
int	sigpending(sigset_t *set);	c. 416
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);	c. 414
	<signal.h>	
	how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	sigqueue(pid_t pid, int signo, const union sigval value)	c. 446
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sigsetjmp(sigjmp_buf env, int savemask);	c. 425
	<setjmp.h>	
	Возвращает 0, если вызывается непосредственно, ненулевое значение — если возврат произошел вследствие вызова siglongjmp	
int	sigsuspend(const sigset_t *sigmask);	c. 428
	<signal.h>	
	Возвращает -1, с кодом ошибки EINTR в переменной errno	
int	sigwait(const sigset_t *restrict set, int *restrict signop);	c. 528
	<signal.h>	
	Возвращает 0 в случае успеха и код ошибки — в случае неудачи	
unsigned int	sleep(unsigned int seconds);	c. 442
	<unistd.h>	
	Возвращает 0 или количество секунд, оставшихся до окончания приостановки	
int	snprintf(char *restrict buf, size_t n, const char *restrict format, ...);	c. 210
	<stdio.h>	
	Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования	
int	sockatmark(int sockfd);	c. 716
	<sys/socket.h>	
	Возвращает 1, если достигнут маркер, 0 — если нет, -1 — в случае ошибки	
int	socket(int domain, int type, int protocol);	c. 677
	<sys/socket.h>	
	<i>type</i> : SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,	
	Возвращает дескриптор файла (сокета) в случае успеха, -1 — в случае ошибки	

int	socketpair(int domain, int type, int protocol, int sockfd[2]);	c. 719
	<sys/socket.h>	
	<i>type</i> : SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	sprintf(char *restrict buf, const char *restrict format, ...);	c. 210
	<stdio.h>	
	Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования	
int	sscanf(const char *restrict buf, const char *restrict format, ...);	c. 214
	<stdio.h>	
	Возвращает количество введенных элементов, EOF — по достижении конца файла или в случае ошибки перед выполнением преобразования	
int	stat(const char *restrict path, struct stat *restrict buf);	c. 137
	<sys/stat.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	str2sig(const char *str, int *signop);	c. 451
	<signal.h>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
	Платформы: Solaris 10	
char	*strerror(int errnum);	c. 49
	<string.h>	
	Возвращает указатель на строку сообщения	
size_t	strftime(char *restrict buf, size_t maxsize,	c. 246
	const char *restrict format,	
	const struct tm *restrict tmprtr);	
	<time.h>	
	Возвращает количество символов, сохраненных в массиве, если достаточно места, 0 — в противном случае	
size_t	strftime_l(char *restrict buf, size_t maxsize,	c. 246
	const char *restrict format,	
	const struct tm *restrict tmprtr, locale_t locale);	
	<time.h>	
	Возвращает количество символов, сохраненных в массиве, если достаточно места, 0 — в противном случае	

char	* strptime (const char *restrict <i>buf</i> , const char *restrict <i>format</i> , struct tm *restrict <i>tmptr</i>); <time.h>	c. 249
	Возвращает указатель на символ, находящийся за последним проанализированным символом, NULL — в противном случае	
char	* strsignal (int <i>signo</i>); <string.h>	c. 450
	Возвращает указатель на строку с описанием сигнала	
int	symlink (const char * <i>actualpath</i> , const char * <i>sympath</i>); <unistd.h>	c. 170
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	symlinkat (const char * <i>actualpath</i> , int <i>fd</i> , const char * <i>sympath</i>); <unistd.h>	c. 170
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
void	sync (void); <unistd.h>	c. 125
long	sysconf (int <i>name</i>); <unistd.h> <i>name</i> : _SC_ARG_MAX, _SC_ASYNCHRONOUS_IO, _SC_ATEXIT_MAX, _SC_BARRIERS, _SC_CHILD_MAX, _SC_CLK_TCK, _SC_CLOCK_SELECTION, _SC_COLL_WEIGHTS_MAX, _SC_DELAYTIMER_MAX, _SC_HOST_NAME_MAX, _SC_IOV_MAX, _SC_JOB_CONTROL, _SC_LINE_MAX, _SC_LOGIN_NAME_MAX, _SC_MAPPED_FILED, _SC_MEMORY_PROTECTION, _SC_NGROUPS_MAX, _SC_OPEN_MAX, _SC_PAGESIZE, _SC_PAGE_SIZE, _SC_READER_WRITER_LOCKS, _SC_REALTIME_SIGNALS, _SC_RE_DUP_MAX, _SC_RTSIG_MAX, _SC_SAVED_IDS, _SC_SEMAPHORES, _SC_SEM_NSEMS_MAX, _SC_SEM_VALUE_MAX, _SC_SHELL, _SC_SIGQUEUE_MAX, _SC_SPIN_LOCKS, _SC_STREAM_MAX, _SC_SYMLOOP_MAX, _SC_THREAD_SAFE_FUNCTIONS,	c. 82

```
_SC_THREADS, _SC_TIMER_MAX,
_SC_TIMERS, _SC_TTY_NAME_MAX,
_SC_TZNAME_MAX, _SC_VERSION,
_SC_XOPEN_CRYPT, _SC_XOPEN_REALTIME,
_SC_XOPEN_REALTIME_THREADS, _SC_XOPEN_SHM,
_SC_XOPEN_VERSION
```

Возвращает соответствующее значение в случае успеха, -1 — в случае ошибки

```
void syslog(int priority, const char *format, ...);
<syslog.h>
```

с. 545

```
int system(const char *cmdstring);
<stdlib.h>
```

с. 326

Возвращает код завершения командной оболочки

```
int tcdrain(int fd);
<termios.h>
```

с. 786

Возвращает 0 в случае успеха, -1 — в случае ошибки

```
int tcflow(int fd, int action);
<termios.h>
```

с. 786

action: TCOFF, TCOON, TCIOFF, TCION

Возвращает 0 в случае успеха, -1 — в случае ошибки

```
int tcflush(int fd, int queue);
<termios.h>
```

с. 786

queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH

Возвращает 0 в случае успеха, -1 — в случае ошибки

```
int tcgetattr(int fd, struct termios *termpptr);
<termios.h>
```

с. 776

Возвращает 0 в случае успеха, -1 — в случае ошибки

```
pid_t tcgetpgrp(int fd);
<unistd.h>
```

с. 361

Возвращает идентификатор группы процессов переднего плана в случае успеха, -1 — в случае ошибки

```
pid_t tcgetsid(int fd);
<termios.h>
```

с. 362

Возвращает идентификатор группы процессов лидера сеанса в случае успеха, -1 — в случае ошибки

int	tcsendbreak(int fd, int duration);	c. 786
<termios.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	tcsetattr(int fd, int opt, const struct termios *termptr);	c. 776
<termios.h>		
	<i>opt:</i> TCSANOW, TCSADRAIN, TCSAFLUSH	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	tcsetpgrp(int fd, pid_t pgrpid);	c. 361
<unistd.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
long	telldir(DIR *dp);	c. 179
<dirent.h>		
	Возвращает значение текущей позиции в каталоге, ассоциированное с <i>dp</i>	
time_t	time(time_t *calptr);	c. 243
<time.h>		
	Возвращает значение текущего времени в случае успеха, -1 — в случае ошибки	
clock_t	times(struct tms *buf);	c. 342
<sys/times.h>		
	Возвращает значение общего времени выполнения процесса в тактах в случае успеха, -1 — в случае ошибки	
FILE	*tmpfile(void);	c. 222
<stdio.h>		
	Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки	
char	*tmpnam(char *ptr);	c. 222
<stdio.h>		
	Возвращает указатель на строку с уникальным именем файла	
int	truncate(const char *path, off_t length);	c. 158
<unistd.h>		
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

char	*ttynname(int fd);	c. 788
	<i><unistd.h></i>	
	Возвращает указатель на строку с именем специального файла устройства терминала, NULL — в случае ошибки	
mode_t	umask(mode_t cmask);	c. 149
	<i><sys/stat.h></i>	
	Возвращает предыдущее значение маски режима создания файлов	
int	uname(struct utsname *name);	c. 241
	<i><sys/utsname.h></i>	
	Возвращает неотрицательное значение в случае успеха, -1 — в случае ошибки	
int	ungetc(int c, FILE *fp);	c. 202
	<i><stdio.h></i>	
	Возвращает c в случае успеха, EOF — в случае ошибки	
int	unlink(const char *path);	c. 164
	<i><unistd.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	unlinkat(int fd, const char *path, int flag);	c. 164
	<i><unistd.h></i>	
	<i>flag: AT_REMOVEDIR</i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	unlockpt(int fd);	c. 815
	<i><stdlib.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	unsetenv(const char *name);	c. 268
	<i><stdlib.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	utimensat(int fd, const char *path, <i>const struct timespec times[2], int flag);</i>	c. 174
	<i><sys/stat.h></i>	
	<i>flag: AT_SYMLINK_NOFOLLOW</i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	

int	<code>utimes(const char *path, const struct timeval times[2]);</code>	c. 175
	<i><sys/time.h></i>	
	Возвращает 0 в случае успеха, -1 — в случае ошибки	
int	<code>vdprintf(int fd, const char *restrict format, va_list arg);</code>	c. 213
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки	
int	<code>vfprintf(FILE *restrict fp, const char *restrict format, va_list arg);</code>	c. 213
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки	
int	<code>vfscanf(FILE *restrict fp, const char *restrict format, va_list arg);</code>	c. 216
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество введенных элементов, EOF — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования	
int	<code>vprintf(const char *restrict format, va_list arg);</code>	c. 213
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки	
int	<code>vscanf(const char *restrict format, va_list arg);</code>	c. 216
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество введенных элементов, EOF — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования	
int	<code>vsnprintf(char *restrict buf, size_t n,</code>	c. 213
	<code>const char *restrict format, va_list arg);</code>	
	<i><stdarg.h></i>	
	<i><stdio.h></i>	
	Возвращает количество символов, сохраненных в массиве, если буфер имеет достаточный размер, отрицательное значение — в случае ошибки преобразования	

<pre>int vsprintf(char *restrict buf, const char *restrict format, va_list arg); <stdarg.h> <stdio.h></pre>	c. 213
<p>Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования</p>	
<pre>int vsscanf(const char *restrict buf, const char *restrict format, va_list arg); <stdarg.h> <stdio.h></pre>	c. 216
<p>Возвращает количество введенных элементов, EOF — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования</p>	
<pre>void vsyslog(int priority, const char *format, va_list arg); <syslog.h> <stdarg.h></pre>	c. 549
<p>Платформы: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10</p>	
<pre>pid_t wait(int *statloc); <sys/wait.h></pre>	c. 297
<p>Возвращает идентификатор процесса в случае успеха, 0 или -1 — в случае ошибки</p>	
<pre>int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options); <sys/wait.h></pre>	c. 303
<p><i>idtype</i>: P_PID, P_PGID, P_ALL <i>options</i>: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED Возвращает 0 в случае успеха, -1 — в случае ошибки Платформы: Linux 3.2.0, Solaris 10</p>	
<pre>pid_t waitpid(pid_t pid, int *statloc, int options); <sys/wait.h></pre>	c. 297
<p><i>options</i>: WCONTINUED, WNOHANG, WUNTRACED Возвращает идентификатор процесса в случае успеха, 0 или -1 — в случае ошибки</p>	

```
pid_t    wait3(int *statloc, int options, struct rusage *rusage);          c. 304
        <sys/types.h>
        <sys/wait.h>
        <sys/time.h>
        <sys/resource.h>
options: WNOHANG, WUNTRACED
Возвращает идентификатор процесса в случае успеха, 0 или -1 — в случае ошибки
Платформы: FreeBSD 8.0, Linux 3.2.0,
Mac OS X 10.6.8, Solaris 10

pid_t    wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);      c. 304
        <sys/types.h>
        <sys/wait.h>
        <sys/time.h>
        <sys/resource.h>
options: WNOHANG, WUNTRACED
Возвращает идентификатор процесса в случае успеха, 0 или -1 — в случае ошибки
Платформы: FreeBSD 8.0, Linux 3.2.0,
Mac OS X 10.6.8, Solaris 10

ssize_t   write(int fd, const void *buf, size_t nbytes);                         c. 115
        <unistd.h>
Возвращает количество записанных байтов в случае успеха, -1 — в случае ошибки

ssize_t   writev(int fd, const struct iovec *iov, int iovcnt);                  c. 600
        <sys/uio.h>
Возвращает количество записанных байтов в случае успеха, -1 — в случае ошибки
```

B

Различные исходные тексты

B.1. Наш заголовочный файл

Большинство программ в книге подключают заголовочный файл `apue.h`, содержимое которого приводится в листинге B.1. В нем определяются значения констант (таких, как `MAXLINE`) и прототипы наших собственных функций.

Большинство программ должны также подключать следующие заголовочные файлы: `<stdio.h>`, `<stdlib.h>` (где определен прототип функции `exit`) и `<unistd.h>` (содержащий прототипы всех стандартных функций UNIX). Поэтому наш заголовочный файл автоматически подключает эти системные заголовочные файлы, а также файл `<string.h>`. Это позволило также сократить размер листингов в книге.

Листинг B.1. Наш заголовочный файл `apue.h`

```
/*
 * Наш заголовочный файл, который подключается перед любыми
 * стандартными системными заголовочными файлами
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS)          /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif

#include <sys/types.h>    /* некоторые системы требуют этого заголовка */
#include <sys/stat.h>
#include <sys/termios.h> /* структура winsize */

#if defined(MACOS) || !defined(TIOCGWINSZ)
```

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

#include <stdio.h>    /* для удобства */
#include <stdlib.h>   /* для удобства */
#include <stddef.h>   /* макрос offsetof */
#include <string.h>    /* для удобства */
#include <unistd.h>   /* для удобства */
#include <signal.h>    /* константа SIG_ERR */

#define MAXLINE 4096 /* максимальная длина строки */

/*
 * Права доступа по умолчанию к создаваемым файлам.
 */
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Права доступа по умолчанию к создаваемым каталогам.
 */
#define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void Sigfunc(int); /* обработчики сигналов */

#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(a,b) ((a) > (b) ? (a) : (b))

/*
 * Прототипы наших собственных функций.
 */
char    *path_alloc(size_t *);           /* листинг 2.3 */
long    open_max(void);                /* листинг 2.4 */

int     set_cloexec(int);              /* листинг 13.5 */
void    clr_fl(int, int);
void    set_fl(int, int);              /* листинг 3.5 */

void    pr_exit(int);                 /* листинг 8.5 */

void    pr_mask(const char *);         /* листинг 10.10 */
Sigfunc *signal_intr(int, Sigfunc *);  /* листинг 10.12 */

void    daemonize(const char *);       /* листинг 13.1 */

void    sleep_us(unsigned int);        /* упражнение 14.5 */
ssize_t  readn(int, void *, size_t);   /* листинг 14.9 */
ssize_t  writen(int, const void *, size_t); /* листинг 14.9 */

int     fd_pipe(int *);               /* листинг 17.1 */
int     recv_fd(int, ssize_t (*func)(int,
                                      const void *, size_t)); /* листинг 17.10 */
int     send_fd(int, int);            /* листинг 17.9 */
int     send_err(int, int,
```

```

        const char *); /* листинг 17.8 */
int     serv_listen(const char *); /* листинг 17.5 */
int     serv_accept(int, uid_t *); /* листинг 17.6 */
int     cli_conn(const char *); /* листинг 17.7 */
int     buf_args(char *, int (*func)(int,
        char **)); /* листинг 17.19 */

int     tty_cbreak(int); /* листинг 18.10 */
int     tty_raw(int); /* листинг 18.10 */
int     tty_reset(int); /* листинг 18.10 */
void    tty_atexit(void); /* листинг 18.10 */
struct  termios *tty_termios(void); /* листинг 18.10 */

int     ptym_open(char *, int); /* листинг 19.1 */
int     ptys_open(char *); /* листинг 19.1 */
#endif TIOCGWINSZ
pid_t   pty_fork(int *, char *, int, const struct termios *,
                  const struct winsize *); /* листинг 19.2 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t); /* листинг 14.2 */
#define read_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t); /* листинг 14.3 */

#define is_read_lockable(fd, offset, whence, len) \
    (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
    (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_msg(const char *, ...); /* приложение В */
void    err_dump(const char *, ...) __attribute__((noreturn));
void    err_quit(const char *, ...) __attribute__((noreturn));
void    err_cont(int, const char *, ...);
void    err_exit(int, const char *, ...) __attribute__((noreturn));
void    err_ret(const char *, ...);
void    err_sys(const char *, ...) __attribute__((noreturn));

void    log_msg(const char *, ...); /* приложение В */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...) __attribute__((noreturn));
void    log_ret(const char *, ...);
void    log_sys(const char *, ...) __attribute__((noreturn));
void    log_exit(int, const char *, ...) __attribute__((noreturn));

```

```
void      TELL_WAIT(void); /* предок/потомок из раздела 8.9 */
void      TELL_PARENT(pid_t);
void      TELL_CHILD(pid_t);
void      WAIT_PARENT(void);
void      WAIT_CHILD(void);
#endif /* _APUE_H */
```

Наш заголовочный файл подключается первым, перед всеми обычными системными заголовочными файлами, потому что это позволяет нам дать определения, которые могут потребоваться другим заголовочным файлам, установить порядок подключения заголовочных файлов, а также переопределить некоторые значения, чтобы сгладить и скрыть различия между системами.

B.2. Стандартные процедуры обработки ошибок

В большинстве наших примеров используются два набора функций обработки ошибочных ситуаций. Один включает функции с именами, начинающимися с префикса `err_`, они выводят сообщения в стандартный поток вывода сообщений об ошибках. Другой включает функции с именами, начинающимися с префикса `log_`, они предназначены для использования в процессах-демонах (глава 13), которые, как правило, не имеют управляющего терминала.

Эти наборы функций позволяют обрабатывать ошибочные ситуации всего одной строчкой в программе, например:

```
if (условие ошибки)
    err_dump(формат в стиле printf с любым количеством аргументов);
вместо
if (условие ошибки) {
    char buf[200];

    sprintf(buf, формат в стиле printf с любым количеством
            аргументов);
    perror(buf);
    abort();
}
```

Наши функции обработки ошибок используют возможность передачи списка аргументов переменной длины, которая определяется стандартом ISO C. Дополнительные сведения вы найдете в разделе 7.3 [Kernighan and Ritchie, 1988]. Важно понимать, что функциональная возможность передачи списка аргументов переменной длины из стандарта ISO C отличается от функциональности `varargs`, которая предоставлялась ранними версиями системы (такими, как SVR3 и 4.3BSD). Имена макроопределений остались теми же, но аргументы некоторых из них изменились.

В табл. B.1 показаны различия между разными функциями обработки ошибок.

Таблица В.1. Наши стандартные функции обработки ошибок

Функция	Добавляет строку от strerror?	Аргументы для strerror	Завершает процесс?
err_dump	Да	errno	abort();
err_exit	Да	Явный параметр	exit(1);
err_msg	Нет		return;
err_quit	Нет		exit(1);
err_ret	Да	errno	return;
err_sys	Да	errno	exit(1);
log_msg	Нет		return;
log_quit	Нет		exit(2);
log_ret	Да	errno	return;
log_sys	Да	errno	exit(2);

В листинге В.2 приводятся функции обработки ошибок, которые выводят сообщения в стандартный поток вывода сообщений об ошибках.

Листинг В.2. Функции обработки ошибок, которые выводят сообщения в стандартное устройство вывода сообщений об ошибках

```
#include "apue.h"
#include <errno.h> /* определение переменной errno */
#include <stdarg.h> /* список аргументов переменной длины ISO C */

static void err_doit(int, int, const char *, va_list);

/*
 * Обрабатывает нефатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_ret(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_sys(const char *fmt, ...)
{
```

```
va_list      ap;

va_start(ap, fmt);
err_doit(1, errno, fmt, ap);
exit(1);
}

/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Код ошибки передается в аргументе.
 * Выводит сообщение и возвращает управление.
 */
void
err_cont(int error, const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Код ошибки передается в аргументе.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_exit(int error, const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение, создает файл core и завершает работу процесса.
 */
void
err_dump(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort(); /* записать дамп памяти в файл и завершить процесс */
    exit(1); /* этот вызов никогда не должен выполниться */
}
```

```

/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_msg(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_quit(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция определяет значение флага "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char      buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
                  strerror(error));
    strcat(buf, "\n");
    fflush(stdout); /* в случае, когда stdout и stderr - одно и то же устройство */
    fputs(buf, stderr);
    fflush(NULL);   /* сбрасывает все выходные потоки */
}

```

В листинге В.3 приводятся исходные тексты функций семейства `log_xxx`. Они требуют, чтобы в вызывающем процессе была определена глобальная переменная `log_to_stderr`. Эта переменная должна содержать ненулевое значение, если процесс выполняется не как демон. В этом случае сообщения выводятся в стандарт-

ный поток вывода сообщений об ошибках. Если `log_to_stderr` содержит 0, для вывода сообщений используется функция `syslog` (раздел 13.4).

Листинг B.3. Функции обработки ошибок для демонов

```
/*
 * Процедуры обработки ошибок для программ, которые могут работать как демоны.
 */
#include "apue.h"
#include <errno.h> /* определение переменной errno */
#include <stdarg.h> /* список аргументов переменной длины ISO C */
#include <syslog.h>

static void log_doit(int, int, int, const char *, va_list ap);

/*
 * Взывающем процессе должна быть определена и установлена эта переменная:
 * ненулевое значение - для интерактивных программ, нулевое - для демонов
 */
extern int log_to_stderr;

/*
 * Инициализировать syslog(), если процесс работает в режиме демона.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Обрабатывает нефатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение, соответствующее содержимому переменной errno,
 * и возвращает управление.
 */
void
log_ret(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    log_doit(1, errno, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_sys(const char *fmt, ...)
{
```

```
va_list      ap;

va_start(ap, fmt);
log_doit(1, errno, LOG_ERR, fmt, ap);
va_end(ap);
exit(2);
}

/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
log_msg(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_quit(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Номер ошибки передается в параметре.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_exit(int error, const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    log_doit(1, error, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}
```

```
/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция должна определить значения аргументов
 * "errnoflag" и "priority".
 */
static void
log_doit(int errnoflag, int error, int priority, const char *fmt,
          va_list ap)
{
    char      buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
                  strerror(error));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, "%s", buf);
    }
}
```

C Варианты решения некоторых упражнений

Глава 1

- 1.1 Для решения этого упражнения используем следующие два аргумента команды `ls(1)`: `-i` — для вывода номеров индексных узлов файлов и каталогов (более подробно об индексных узлах рассказывается в разделе 4.14), и `-d` — для вывода информации только о каталогах.
- В результате получим следующее:

```
$ ls -ldi /etc/. /etc/..      ключ -i заставляет выводить номера индексных
                               узлов
162561 drwxr-xr-x  66 root   4096 Feb  5 03:59 /etc/..
                2 drwxr-xr-x  19 root   4096 Jan 15 07:25 /etc/..
$ ls -ldi ./ ../..            оба каталога . и .. имеют один и тот же номер
                               inode - 2
                2 drwxr-xr-x  19 root   4096 Jan 15 07:25 ../
drwxr-xr-x  19 root   4096 Jan 15 07:25 ../
```

- 1.2 UNIX является многозадачной системой. Следовательно, между запусками нашей программы были запущены какие-то другие процессы.
- 1.3 Аргумент `msg` функции `perror` является указателем, поэтому `perror` может изменить содержимое строки, на которую указывает аргумент `msg`. Однако атрибут `const` говорит о том, что `perror` не изменяет строку, на которую ссылается указатель. С другой стороны, аргумент с кодом ошибки в функции `strerror` является целым числом, а так как он передается по значению, функция `strerror` не сможет изменить его, даже если захочет. (Если вы не совсем понимаете, как передаются и обрабатываются аргументы функций в языке C, обратитесь к разделу 5.2 [Kernighan and Ritchie, 1988].)
- 1.4 В 2038 году. Проблема может быть решена за счет увеличения размера типа `time_t` до 64 бит. Если это будет сделано для корректной работы всех приложений, использующих 32-разрядное представление, их необходимо будет

пересобрать. Но на самом деле проблема гораздо глубже. Некоторые файловые системы и носители, предназначенные для хранения резервных копий, используют 32-разрядное представление времени. Их также придется обновить, но при этом сохранить совместимость с устаревшим форматом.

- 1.5 Примерно 248 дней.

Глава 2

- 2.1 В FreeBSD используется следующий способ. Элементарные типы данных, которые могут быть объявлены в нескольких заголовочных файлах, определяются в файле `<machine/_types.h>`. Например:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int      __int32_t;
typedef unsigned int __uint32_t;
...
typedef __uint32_t __size_t;
...
#endif /* _MACHINE__TYPES_H_ */
```

В каждом заголовочном файле, где может определяться элементарный системный тип данных `size_t`, можно использовать такую последовательность:

```
#ifndef _SIZE_T_DECLARED
typedef __size_t size_t;
#define _SIZE_T_DECLARED
#endif
```

При таком подходе инструкция `typedef` для типа `size_t` будет выполнена всего один раз.

- 2.3 Если значение константы `OPEN_MAX` не определено или чрезвычайно велико (то есть равно `LONG_MAX`), для получения максимально возможного количества открытых файловых дескрипторов для процесса можно использовать функцию `getrlimit`. Учитывая, что предел для процесса может быть изменен, мы не можем повторно использовать значение, полученное в результате предыдущего вызова (так как он мог измениться). Решение приводится в листинге С.1.

Листинг С.1. Альтернативный способ определения максимально возможного количества файловых дескрипторов

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS 256
```

```

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
        openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("невозможно получить значение предела");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}

```

Глава 3

- 3.1** Все дисковые операции ввода/вывода выполняются с использованием буферов блоков, расположенных в пространстве ядра (которые также известны как буферный кэш ядра). Исключением являются операции ввода/вывода с неструктурированными дисковыми устройствами, которые мы не рассматривали. (Некоторые системы также поддерживают *непосредственные операции ввода/вывода* с дисковыми устройствами, чтобы дать приложениям возможность производить ввод/вывод в обход буферов в ядре, но мы не рассматривали такую возможность.) Работа буферного кэша описана в главе 3 [Bach, 1986]. Поскольку читаемые или записываемые данные буферизуются ядром, термин *небуферизованный ввод/вывод* скорее означает отсутствие автоматической буферизации в пользовательском процессе при использовании функций `read` и `write`. Каждая из этих функций обращается к единственному системному вызову.
- 3.3** Каждый вызов функции `open` создает новую запись в таблице файлов. Но так как обе операции открывают один и тот же файл, обе записи в таблице файлов будут указывать на одну и ту же запись в таблице виртуальных узлов. Вызов `dup` создаст еще одну ссылку на существующую запись в таблице файлов. Диаграмма, соответствующая данной ситуации, показана на рис. С.1. Функция `fcntl` с аргументами `F_SETFD` и `fd1` воздействует только на флаги дескриптора `fd1`. Но с аргументами `F_SETFL` и `fd1` она будет воздействовать на запись в таблице файлов и тем самым на оба дескриптора — `fd1` и `fd2`.
- 3.4** Для `fd` со значением 1 вызов `dup2(fd, 1)` вернет 1, оставив открытым дескриптор 1. (Вспомните обсуждение из раздела 3.12.) После выполнения трех вызовов `dup2` все три дескриптора будут ссылаться на одну и ту же запись в таблице файлов. Ни один из дескрипторов не будет закрыт. Однако для `fd` со значением 3, после третьего вызова `dup2`, на одну и ту же запись

в таблице файлов будут ссылаться уже четыре дескриптора. В этом случае нужно закрыть дескриптор с номером 3.

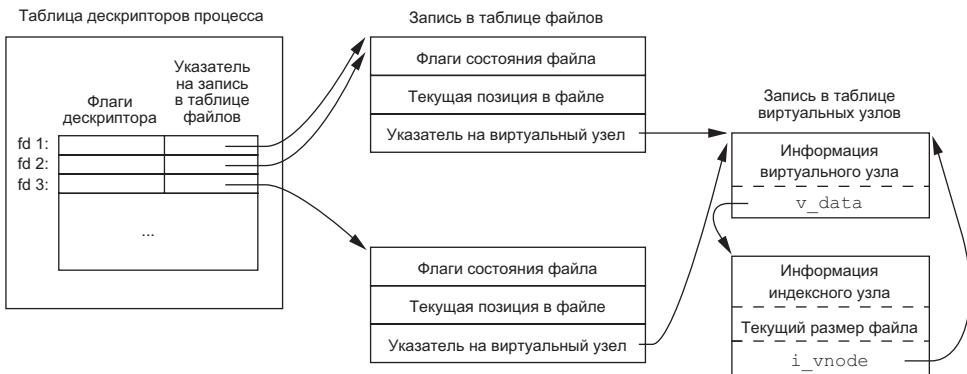


Рис. С.1. Результат работы функций `dup` и `open`

- 3.5** Поскольку командные оболочки обрабатывают аргументы командной строки слева направо, команда

```
./a.out > outfile 2>&1
```

сначала перенаправит стандартный вывод в файл `outfile`, а затем продублирует его на дескриптор с номером 2 (стандартный вывод сообщений об ошибках). В результате все, что будет выводиться в стандартный вывод и в стандартный вывод сообщений об ошибках, попадет в один и тот же файл. Дескрипторы 1 и 2 будут ссылаться на одну и ту же запись в таблице файлов. Однако команда

```
./a.out 2>&1 > outfile
```

сначала вызовет функцию `dup`, и в результате дескриптор с номером 2 будет ссылаться на терминал (предполагается, что команда запущена в интерактивном режиме). А затем стандартный вывод будет перенаправлен в файл `outfile`. В результате дескриптор с номером 1 будет ссылаться на запись в таблице файлов, которая соответствует файлу `outfile`, а дескриптор с номером 2 — на запись, которая соответствует терминалу.

- 3.6** Вы по-прежнему сможете использовать функцию `lseek` и читать данные из произвольного места в файле, но вызов функции `write` автоматически произведет переход в конец файла перед записью данных. В этом случае вы не сможете записать данные в произвольное место в файле.

Глава 4

- 4.1** Функция `stat` всегда пытается следовать по символьским ссылкам (табл. 4.9), поэтому программа никогда не выведет строку «символическая

ссылка». Для приведенного примера, где файл `/dev/cdrom` является символьической ссылкой на файл `/dev/sr0`, функция `stat` укажет, что файл `/dev/cdrom` является специальным файлом блочного устройства, а не символьической ссылкой. Если символьическая ссылка ссылается на несуществующий файл, функция `stat` вернет признак ошибки.

- 4.2** Все биты прав доступа окажутся сброшены:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----- 1 sar 29 Feb 5 14:06 temp.foo
```

- 4.3** Следующий пример показывает, что произойдет, если сбросить бит user-read:

```
$ date > foo
$ chmod u-r foo
$ ls -l foo
--w-r--r-- 1 sar 29 Feb 5 14:21 foo
$ cat foo
cat: foo: Permission denied
```

*сбросить бит user-read
проверить права доступа к файлу
и попытаться прочитать его*

- 4.4** Если попытаться с помощью функции `open` или `creat` создать файл, который уже существует, права доступа к файлу не изменятся. Мы можем убедиться в этом, запустив программу из листинга 4.3:

```
$ rm foo bar
$ date > foo
$ date > bar
$ chmod a-r foo bar
$ ls -l foo bar
--w----- 1 sar 29 Feb 5 14:25 bar
--w----- 1 sar 29 Feb 5 14:25 foo
$ ./a.out
$ ls -l foo bar
--w----- 1 sar 0 Feb 5 14:26 bar
--w----- 1 sar 0 Feb 5 14:26 foo
```

*удалить файлы, если они существуют
создать их и наполнить данными
сбросить биты права на чтение для всех
проверить права доступа
запустить программу из листинга 4.3
проверить права доступа и размеры файлов*

Обратите внимание, что права доступа не изменились, но файлы были усечены.

- 4.5** Размер каталога никогда не может быть равен 0, поскольку файлы каталогов содержат по крайней мере две записи — ссылки на каталоги `.` и `..`. Размер файла символьской ссылки определяется количеством символов в имени файла и пути к нему, а имя файла всегда содержит хотя бы один символ.

- 4.7** При создании файла `core` ядро по умолчанию использует определенные значения битов прав доступа. В данном примере это `rw-r--r--`. Это значение может модифицироваться, а может не модифицироваться значением `umask`. Командная оболочка также определяет значения битов прав доступа по умолчанию, которые устанавливаются для файлов, созданных в резуль-

тате перенаправления. В данном примере это `rw-rw-rw-`, а это значение всегда модифицируется текущим значением `umask`. В данном примере значением `umask` было число 02.

- 4.8 Мы не можем воспользоваться командой `du`, так как она требует указать имя файла, например

```
du tempfile
```

или имя каталога:

```
du .
```

Но после возврата из функции `unlink` запись в каталоге для файла `tempfile` исчезает. Команда `du .` не смогла бы показать, что содержимое файла `tempfile` по-прежнему продолжает занимать дисковое пространство. В этом примере мы должны использовать команду `df`, чтобы увидеть фактический объем свободного дискового пространства.

- 4.9 При удалении ссылки, которая не является последней, сам файл не удаляется. В этом случае обновляется время последнего изменения файла. Но если удаляется последняя ссылка на файл, обновление времени последнего изменения теряет всякий смысл, поскольку вся информация о файле (индексный узел) удаляется вместе с файлом.

- 4.10 Мы рекурсивно вызываем функцию `dopath` после открытия каталога функцией `opendir`. Предположим, что `opendir` использует единственный дескриптор — в этом случае каждый раз, спускаясь на один уровень вглубь иерархии дерева каталогов, мы используем другой дескриптор. (Если исходить из предположения, что дескрипторы не закрываются, пока не будет закончен обзор дерева каталогов и не будет вызвана функция `closedir`.) Это ограничивает глубину дерева каталогов, на которую мы можем погрузиться, максимальным количеством одновременно открытых дескрипторов. Обратите внимание: в расширениях XSI стандарта Single UNIX Specification определено, что функция `nftw` позволяет вызывающему процессу задать максимальное количество используемых дескрипторов, допуская закрытие и повторное использование дескрипторов.

- 4.12 Функция `chroot` используется в Интернете на серверах FTP для повышения безопасности. Пользователи, не имеющие учетных записей в системе (так называемые *анонимные пользователи FTP*), попадают в отдельный каталог, и этот каталог делается корневым с помощью функции `chroot`. Это предотвращает возможность доступа к файлам, расположенным за пределами нового корневого каталога.

Кроме того, функция `chroot` может использоваться для создания копии дерева каталогов на новом месте, чтобы затем изменять эту новую копию, не опасаясь внести изменения в оригиналную файловую систему. Это полезно, например, для тестирования результатов установки новых программных пакетов.

Только суперпользователь может вызвать функцию `chroot`, и после изменения корневого каталога процесс и все его потомки никогда не смогут вернуться к первоначальному корню файловой системы.

- 4.13** Прежде всего необходимо вызвать функцию `stat`, чтобы получить три значения времени для файла, затем вызвать `utimes`, чтобы изменить требуемое значение. Значение, которое не должно изменяться в результате вызова `utimes`, должно соответствовать значению, полученному от функции `stat`.
- 4.14** Команда `finger(1)` использует функцию `stat` для определения атрибутов времени почтового ящика. Время последнего изменения соответствует времени прибытия последнего электронного письма, а время последнего обращения — времени, когда в последний раз была прочитана почта.
- 4.15** Обе утилиты, `cpio` и `tar`, сохраняют в архиве только время последнего изменения (`st_mtime`). Время последнего обращения не сохраняется, поскольку его значение соответствует времени создания архива, так как для этого архиватор должен прочитать содержимое файла. Ключ `-a` команды `cpio` позволяет переустановить время последнего обращения для каждого прочитанного файла. В результате создание архива не влечет изменения времени последнего обращения. (Однако переустановка времени последнего обращения к файлу приводит к изменению времени последнего изменения статуса.) Время последнего изменения статуса не сохраняется в архиве, так как при извлечении файла из архива нет возможности запросить его значение, даже если оно было сохранено в архиве. (Функция `utimes` и родственные ей `futimens` и `utimensat` могут изменять только время последнего изменения файла и время последнего обращения к файлу.)

Когда архиватор `tar` извлекает файлы из архива, он по умолчанию восстанавливает время последнего изменения извлекаемых файлов. С помощью ключа `m` можно указать утилите `tar`, что она не должна восстанавливать время последнего изменения файла, тогда в качестве времени последнего изменения будет использоваться время извлечения из архива. При использовании архиватора `tar` время последнего обращения к файлу после его извлечения из архива в любом случае будет установлено равным времени извлечения.

Архиватор `cpio`, напротив, в качестве времени последнего изменения и времени последнего обращения устанавливает время извлечения из архива. По умолчанию он не пытается восстановить прежнее время последнего изменения файла, сохраненное в архиве. При использовании архиватора `cpio` для восстановления значений времени последнего обращения и времени последнего изменения, сохраненных в архиве, следует использовать ключ `-m`.

- 4.16** Ядро изначально не имеет ограничений на глубину вложенности каталогов. Но большинство команд завершаются ошибкой, если полные имена файлов или каталогов превышают длину `PATH_MAX`. Программа в листинге С.2 создает дерево каталогов, состоящее из 1000 уровней вложенности, на каждом уровне каталог имеет имя длиной 45 символов. Можно создать эту структуру на любой платформе, однако ни на одной из платформ нельзя получить

абсолютное полное имя каталога на тысячном уровне с помощью функции `getcwd`. В Mac OS X 10.6.8 мы никогда не сможем получить полное имя самого последнего каталога в таком длинном пути. В FreeBSD 8.0, Linux 3.2.0 и Solaris 10 программа сможет получить полное имя последнего каталога, но нам придется много раз вызвать функцию `realloc`, чтобы разместить буфер достаточно большого размера. Запуск этой программы в Linux 3.2.0 дал следующие результаты:

```
$ ./a.out
ошибка вызова функции getcwd, размер = 4096: Numerical result too large
ошибка вызова функции getcwd, размер = 4196: Numerical result too large
...
ошибка вызова функции getcwd, размер = 45896: Numerical result too large
ошибка вызова функции getcwd, размер = 45996: Numerical result too large
длина = 46004
здесь было выведено имя длиной 46 004 байт
```

Мы не сможем заархивировать это дерево каталогов с помощью архиватора `cpio`. Он выведет сообщение о слишком длинном имени файла. `cpio` не сможет заархивировать этот каталог ни на одной из четырех платформ. Напротив, `tar` сможет заархивировать этот каталог в FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8. Но в Linux 3.2.0 не получится извлечь это дерево каталогов из архива.

Листинг C.2. Создание дерева каталогов с глубокой вложенностью

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH      1000      /* глубина вложенности */
#define STARTDIR   "/tmp"
#define NAME        "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ      (10*8192)

int
main(void)
{
    int      i, size;
    size_t   size;
    char    *path;

    if (chdir(STARTDIR) < 0)
        err_sys("ошибка вызова функции chdir");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("ошибка вызова функции mkdir, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("ошибка вызова функции chdir, i = %d", i);
    }

    if (creat("afile", FILE_MODE) < 0)
```

```

        err_sys("ошибка вызова функции creat");

/*
 * Дерево каталогов с большой глубиной вложенности создано,
 * в каталоге создан файл. Теперь попробуем получить его полное имя.
 */
path = path_alloc(&size);
for ( ; ; ) {
    if (getcwd(path, size) != NULL) {
        break;
    } else {
        err_ret("ошибка вызова функции getcwd, размер = %ld", (long)
                size);
        size += 100;
        if (size > MAXSZ)
            err_quit("превышено наше ограничение");
        if ((path = realloc(path, size)) == NULL)
            err_sys("ошибка вызова функции realloc");
    }
}
printf("длина = %ld\n%s\n", (long)strlen(path), path);

exit(0);
}

```

- 4.17** Для каталога `/dev` все биты права на запись сброшены, что не позволит обычному пользователю удалять файлы из каталога. Это означает, что вызов функции `unlink` будет завершаться неудачей.

Глава 5

- 5.2** Функция `fgets` будет читать символы, пока не встретится символ перевода строки или пока буфер не заполнится (с учетом места, которое необходимо оставить для завершающего нулевого символа). Функция `fputs` будет выводить данные из буфера, пока не встретит завершающий нулевой символ — она не обращает внимания на символы перевода строки, которые могут находиться в буфере. То есть если значение `MAXLINE` будет слишком маленьким, обе функции по-прежнему будут работать, просто они будут вызываться намного чаще, чем при использовании буфера большого размера.

Если бы любая из этих функций удаляла или добавляла символ перевода строки (как это делают функции `gets` и `puts`), нам пришлось бы предусматривать размещение буферов достаточно большого объема, чтобы вместить самую длинную строку.

- 5.3** Вызов

```
printf("");
```

вернет 0, потому что не выводит ни одного символа.

- 5.4 Это достаточно распространенная ошибка. Возвращаемое значение функций `getc` и `getchar` имеет тип `int`, а не `char`. Зачастую константа `EOF` определена как `-1`, и поэтому, если в системе тип `char` имеет знак, этот код будет работать нормально. Но если в системе тип `char` не имеет знака, возвращаемое значение `EOF`, полученное от `getchar`, будет сохранено в переменной с беззнаковым типом `char` и перестанет быть равным `-1`, вследствие чего цикл никогда не закончится. На всех четырех платформах, описываемых в данной книге, тип `char` имеет знак, поэтому данный пример будет корректно работать на всех этих plataформах.
- 5.5 Вызывать функцию `fsync` после каждого вызова `fflush`. Аргумент функции `fsync` можно получить вызовом функции `fileno`. Вызов `fsync` без обращения к `fflush` может не дать ожидаемого результата, если данные все еще находятся во внутренних буферах приложения.
- 5.6 Когда программа работает в интерактивном режиме, стандартные потоки ввода и вывода буферизуются построчно. Когда вызывается функция `fgets`, содержимое потока стандартного вывода сбрасывается автоматически.
- 5.7 Реализация `fmemopen` для BSD-систем представлена в листинге С.3.

Листинг С.3. Реализация функции `fmemopen` для BSD-систем

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/*
 * Внутренняя структура для слежения за потоком ввода/вывода в памяти
 */
struct memstream
{
    char    *buf;      /* буфер в памяти */
    size_t   rsize;   /* фактический размер буфера */
    size_t   vsize;   /* виртуальный размер буфера */
    size_t   curpos;  /* текущая позиция в буфере */
    int     flags;   /* см. ниже */
};

/* флаги */
#define MS_READ      0x01 /* открыть для чтения */
#define MS_WRITE     0x02 /* открыть для записи */
#define MS_APPEND    0x04 /* открыть для добавления в конец */
#define MS_TRUNCATE  0x08 /* усечь при открытии */
#define MS_MYBUF     0x10 /* освободить буфер при закрытии */

#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif

static int    mstream_read(void *, char *, int);
static int    mstream_write(void *, const char *, int);
```

```

static fpos_t mstream_seek(void *, fpos_t, int);
static int    mstream_close(void *);
static int    type_to_flags(const char *__restrict type);
static off_t   find_end(char *buf, size_t len);

FILE *
fmemopen(void *__restrict buf, size_t size,
         const char *__restrict type)
{
    struct memstream *ms;
    FILE *fp;

    if (size == 0) {
        errno = EINVAL;
        return(NULL);
    }
    if ((ms = malloc(sizeof(struct memstream))) == NULL) {
        errno = ENOMEM;
        return(NULL);
    }
    if ((ms->flags = type_to_flags(type)) == 0) {
        errno = EINVAL;
        free(ms);
        return(NULL);
    }
    if (buf == NULL) {
        if ((ms->flags & (MS_READ|MS_WRITE)) !=
            (MS_READ|MS_WRITE)) {
            errno = EINVAL;
            free(ms);
            return(NULL);
        }
        if ((ms->buf = malloc(size)) == NULL) {
            errno = ENOMEM;
            free(ms);
            return(NULL);
        }
        ms->rsize = size;
        ms->flags |= MS_MYBUF;
        ms->curpos = 0;
    } else {
        ms->buf = buf;
        ms->rsize = size;
        if (ms->flags & MS_APPEND)
            ms->curpos = find_end(ms->buf, ms->rsize);
        else
            ms->curpos = 0;
    }
    if (ms->flags & MS_APPEND) { /* режим "a" */
        ms->vsize = ms->curpos;
    } else if (ms->flags & MS_TRUNCATE) { /* режим "w" */
        ms->vsize = 0;
    } else { /* режим "r" */

```

```
    ms->vsize = size;
}
fp = funopen(ms, mstream_read, mstream_write,
            mstream_seek, mstream_close);
if (fp == NULL) {
    if (ms->flags & MS_MYBUF)
        free(ms->buf);
    free(ms);
}
return(fp);
}

static int
type_to_flags(const char *__restrict type)
{
    const char *cp;
    int flags = 0;

    for (cp = type; *cp != 0; cp++) {
        switch (*cp) {
        case 'r':
            if (flags != 0)
                return(0); /* ошибка */
            flags |= MS_READ;
            break;

        case 'w':
            if (flags != 0)
                return(0); /* ошибка */
            flags |= MS_WRITE|MS_TRUNCATE;
            break;

        case 'a':
            if (flags != 0)
                return(0); /* ошибка */
            flags |= MS_APPEND;
            break;

        case '+':
            if (flags == 0)
                return(0); /* ошибка */
            flags |= MS_READ|MS_WRITE;
            break;

        case 'b':
            if (flags == 0)
                return(0); /* ошибка */
            break;

        default:
            return(0); /* ошибка */
        }
    }
}
```

```
        return(flags);
    }

    static off_t
    find_end(char *buf, size_t len)
    {
        off_t off = 0;

        while (off < len) {
            if (buf[off] == 0)
                break;
            off++;
        }
        return(off);
    }

    static int
    mstream_read(void *cookie, char *buf, int len)
    {
        int nr;
        struct memstream *ms = cookie;

        if (!(ms->flags & MS_READ)) {
            errno = EBADF;
            return(-1);
        }
        if (ms->curpos >= ms->vsize)
            return(0);

        /* прочитать можно только от текущей позиции до vsize */
        nr = MIN(len, ms->vsize - ms->curpos);
        memcpy(buf, ms->buf + ms->curpos, nr);
        ms->curpos += nr;
        return(nr);
    }

    static int
    mstream_write(void *cookie, const char *buf, int len)
    {
        int nw, off;
        struct memstream *ms = cookie;

        if (!(ms->flags & (MS_APPEND|MS_WRITE))) {
            errno = EBADF;
            return(-1);
        }
        if (ms->flags & MS_APPEND)
            off = ms->vsize;
        else
            off = ms->curpos;
        nw = MIN(len, ms->rsize - off);
        memcpy(ms->buf + off, buf, nw);
        ms->curpos = off + nw;
```

```
if (ms->curpos > ms->vsize) {
    ms->vsize = ms->curpos;
    if (((ms->flags & (MS_READ|MS_WRITE)) ==
        (MS_READ|MS_WRITE)) && (ms->vsize < ms->rsize))
        *(ms->buf + ms->vsize) = 0;
}
if ((ms->flags & (MS_WRITE|MS_APPEND)) &&
    !(ms->flags & MS_READ)) {
    if (ms->curpos < ms->rsize)
        *(ms->buf + ms->curpos) = 0;
    else
        *(ms->buf + ms->rsize - 1) = 0;
}
return(nw);
}

static fpos_t
mstream_seek(void *cookie, fpos_t pos, int whence)
{
    int off;
    struct memstream *ms = cookie;

    switch (whence) {
    case SEEK_SET:
        off = pos;
        break;
    case SEEK_END:
        off = ms->vsize + pos;
        break;
    case SEEK_CUR:
        off = ms->curpos + pos;
        break;
    }
    if (off < 0 || off > ms->vsize) {
        errno = EINVAL;
        return -1;
    }
    ms->curpos = off;
    return(off);
}

static int
mstream_close(void *cookie)
{
    struct memstream *ms = cookie;

    if (ms->flags & MS_MYBUF)
        free(ms->buf);
    free(ms);
    return(0);
}
```

Глава 6

- 6.1** Функции доступа к теневому файлу паролей в Linux и Solaris обсуждались в разделе 6.3. Мы не можем для сравнения с зашифрованным паролем использовать значение, возвращаемое в поле `pw_passwd` функциями, описанными в разделе 6.2, поскольку это поле не содержит зашифрованного пароля. Чтобы получить зашифрованный пароль, нужно отыскать требуемую учетную запись в теневом файле паролей и извлечь из нее зашифрованный пароль.

В FreeBSD и Mac OS X автоматически используется теневой файл паролей. В FreeBSD 8.0, в структуре `passwd`, возвращаемой функциями `getpwnam` и `getpwuid`, поле `pw_passwd` содержит зашифрованный пароль, но только если вызывающий процесс имеет эффективный идентификатор пользователя 0. В Mac OS X 10.6.8 зашифрованный пароль нельзя получить с помощью этих функций.

- 6.2** Программа из листинга С.4 выводит зашифрованный пароль в Linux 3.2.0 и Solaris 10. Если эту программу запустит обычный пользователь, вызов `getspnam` завершится неудачей с кодом ошибки `EACCES`.

Листинг С.4. Вывод зашифрованного пароля в ОС Linux и Solaris

```
#include "apue.h"
#include <shadow.h>

int
main(void) /* версия для Linux/Solaris */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("ошибка вызова функции getspnam");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL || 
          ptr->sp_pwdp[0] == 0 ? "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

В листинге С.5 приводится исходный текст программы, которая выведет зашифрованный пароль в FreeBSD 8.0, если запустить ее с привилегиями суперпользователя. В иных случаях в поле `pw_passwd` возвращается символ «звездочки». В Mac OS X 10.6.8 зашифрованный пароль выводится в любом случае, независимо от привилегий, с которыми запущена программа.

Листинг С.5. Вывод зашифрованного пароля в ОС FreeBSD и Mac OS X

```
#include "apue.h"
#include <pwd.h>

int
main(void) /* версия для FreeBSD/Mac OS X */
{
    struct passwd *ptr;
```

```

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("ошибка вызова функции getpwnam");
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL || 
        ptr->pw_passwd[0] == 0 ? "(null)" : ptr->pw_passwd);
    exit(0);
}

```

- 6.5** Программа из листинга С.6 выводит текущее время и дату в формате утилиты date.

Листинг С.6. Вывод текущего времени и даты в формате утилиты date

```

#include "apue.h"
#include <time.h>

int
main(void)
{
    time_t      calctime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ((calctime = time(NULL)) == -1)
        err_sys("ошибка вызова функции time");
    if ((tm = localtime(&calctime)) == NULL)
        err_sys("ошибка вызова функции localtime");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("ошибка вызова функции strftime");
    fputs(line, stdout);
    exit(0);
}

```

Запустив эту программу, мы получили следующее:

```

$ ./a.out                               часовой пояс автора по умолчанию US/Eastern
Wed Jul 25 22:58:32 EDT 2012
$ TZ=US/Mountain ./a.out                U.S. часовой пояс штата Монтана
Wed Jul 25 20:58:32 MDT 2012
$ TZ=Japan ./a.out                      Япония
Thu Jul 26 11:58:32 JST 2012

```

Глава 7

- 7.1** Похоже, что возвращаемое значение функции `printf` (количество выведенных символов) стало возвращаемым значением функции `main`. Чтобы проверить это предположение, измените длину выводимой строки и посмотрите, как изменится возвращаемое значение. Такое поведение наблюдается не во всех системах. Отметьте также, что если разрешить применение расширений ISO C в компиляторе `gcc`, возвращаемое значение всегда будет равно 0, как того требует стандарт.
- 7.2** Когда программа работает в интерактивном режиме, стандартный вывод обычно буферизуется построчно так, что фактический вывод происходит

только при выводе символа перевода строки. Но если стандартный поток вывода перенаправлен в файл, ему, скорее всего, будет назначен режим полной буферизации, и фактический вывод не будет производиться до освобождения ресурсов стандартной библиотеки ввода/вывода.

- 7.3 В большинстве версий UNIX это невозможно. Копии `argc` и `argv` не сохраняются в глобальных переменных, как, например, `environ`.
- 7.4 Это дает возможность аварийно завершить процесс при попытке обратиться к памяти по пустому указателю, что является достаточно распространенной ошибкой при программировании на языке C.
- 7.5 Вот эти определения:

```
typedef void    Exitfunc(void);
int atexit(Exitfunc *func);
```

- 7.6 Функция `calloc` инициализирует выделяемую память, обнуляя все биты. Стандарт ISO C не гарантирует, что в результате это даст числа с плавающей точкой, равные 0, или пустые указатели.
- 7.7 Куча и стек не размещаются в памяти, пока программа не будет запущена одной из функций семейства `exec` (описывается в разделе 8.10).
- 7.8 Выполняемый файл (`a.out`) содержит отладочную информацию, которая может оказаться полезной при анализе файла `core`. Удалить эту информацию можно командой `strip(1)`. Удаление отладочной информации из двух файлов `a.out` помогло уменьшить их размеры до 798 760 и 6200 байт.
- 7.9 Когда не используются разделяемые библиотеки, большую часть выполняемого файла занимает стандартная библиотека ввода/вывода.
- 7.10 Этот код содержит ошибку, поскольку пытается вернуть ссылку на переменную `val` с автоматическим классом размещения после того, как переменная перестала существовать. Автоматические переменные, объявленные после левой открывающей скобки, с которой начинается составной оператор, не видны за правой закрывающей скобкой.

Глава 8

- 8.1 Чтобы смоделировать ситуацию закрытия стандартного вывода при завершении дочернего процесса, добавьте следующую строку перед вызовом функции `exit` в дочернем процессе:

```
fclose(stdout);
```

Чтобы увидеть, как действует эта строка, замените вызов функции `printf` строками

```
i = printf("pid = %ld, glob = %d, var = %d\n",
           (long)getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

Вам также потребуется определить переменные `i` и `buf`.

Здесь предполагается, что стандартный поток `stdout` будет закрыт, когда дочерний процесс вызовет функцию `exit`, но дескриптор `STDOUT_FILENO` останется открытым. Некоторые версии стандартной библиотеки ввода/вывода при закрытии стандартного потока вывода закрывают и файловый дескриптор, в результате функция `write` также будет завершаться неудачей. В этом случае с помощью функции `dup` продублируйте стандартный вывод на какой-либо другой дескриптор и используйте его в функции `write`.

Рассмотрим программу в листинге C.7.

Листинг C.7. Некорректное использование функции `vfork`

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t    pid;

    if ((pid = vfork()) < 0)
        err_sys("ошибка вызова функции vfork");
    /*
     * Оба процесса, дочерний и родительский, выполняют возврат
     * в вызывающую функцию.
     */
}

static void
f2(void)
{
    char    buf[1000]; /* переменные с автоматическим классом размещения */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

К моменту вызова функции `vfork` указатель стека в родительском процессе будет содержать адрес кадра функции `f1`, которая вызвала `vfork`. Это показано на рис. С.2. После вызова `vfork` дочерний процесс первым получает управление и выполняет возврат из функции `f1`. После этого потомок вызы-

вает функцию `f2` и кадр стека этой функции накладывается на предыдущий кадр стека функции `f1`. Затем дочерний процесс забивает нулями 1000 байт автоматической переменной `buf`, размещенной на стеке. Далее дочерний процесс выполняет возврат из `f2` и вызывает `_exit`, но содержимое стека ниже кадра стека функции `main` уже изменилось. После этого родительский процесс возобновляет работу и производит возврат из функции `f1`. Адрес возврата из функции чаще всего хранится на стеке, но эта информация на верняка уже изменена дочерним процессом. Что может произойти с родительским процессом в данном примере, во многом зависит от различных особенностей реализации конкретной версии UNIX (где в стеке хранится адрес возврата из функции, какая информация на стеке будет уничтожена при изменении содержимого автоматической переменной и т. п.). Типичный результат — аварийное завершение родительского процесса с созданием файла `core`, но у вас результаты могут быть иными.



Рис. С.2. Содержимое стека в момент вызова функции `vfork`

- 8.4 В листинге 8.7 мы заставляли родительский процесс начинать вывод первым. Когда родительский процесс заканчивал вывод, свою строку начинал выводить дочерний процесс, но при этом мы разрешали родительскому процессу завершить работу, не дожидаясь завершения потомка. Что произойдет раньше, завершение работы родительского процесса или завершение вывода дочерним процессом, зависит от реализации алгоритма планирования процессов в ядре (еще одна разновидность гонки за ресурсами). Когда завершается родительский процесс, командная оболочка запускает следующую программу, и вывод этой программы смешивается с выводом дочернего процесса, запущенного предыдущей программой.

Мы можем предотвратить эту ситуацию, запретив родительскому процессу завершать работу раньше, чем дочерний процесс завершит вывод своей строки. Замените код, следующий за вызовом функции `fork`, следующим фрагментом:

```
else if (pid == 0) {
    WAIT_PARENT();          /* родительский процесс стартует первым */
    charatatime("от дочернего процесса\n");
    TELL_PARENT(getppid()); /* сообщить родителю о завершении вывода */
} else {
    charatatime("от родительского процесса\n");
    TELL_CHILD(pid);       /* сообщить потомку о завершении вывода */
    WAIT_CHILD();           /* подождать, пока потомок завершит вывод */
}
```

Мы не сможем наблюдать подобный эффект, если позволим дочернему процессу стартовать первым, поскольку командная оболочка не запустит следующую программу, пока не завершится родительский процесс.

- 8.5** Аргумент `argv[2]` будет иметь то же значение (`/home/sar/bin/testinterp`). Это объясняется тем, что работа функции `execvp` завершается вызовом `execve` с тем же значением аргумента *pathname*, что и при непосредственном обращении к функции `exec1` (рис. 8.2).
- 8.6** Программа из листинга С.8 создает процесс-зомби.

Листинг С.8. Создает процесс-зомби, состояние которого можно затем проверить с помощью `ps`

```
#include "apue.h"

#ifndef SOLARIS
#define PSCMD "ps -a -o pid,ppid,s,tty,comm"
#else
#define PSCMD "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0) /* потомок */
        exit(0);

    /* предок */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

Обычно команда `ps` обозначает процессы-зомби с помощью символа Z.

```
$ ./a.out
 PID  PPID S TT      COMMAND
2369  2208 S pts/2  -bash
7230  2369 S pts/2  ./a.out
7231  7230 Z pts/2  [a.out] <defunct>
7232  7230 S pts/2  sh -c ps -o pid,ppid,state,tty,command
7233  7232 R pts/2  ps -o pid,ppid,state,tty,command
```

Глава 9

- 9.1** Процесс `init` знает, когда пользователь производит выход из системы с терминала, потому что является родительским процессом по отношению к ко-

мандной оболочке входа и получает сигнал `SIGCHLD`, когда она завершает работу.

Однако в случае входа в систему через сетевое соединение процесс `init` никак не задействован. Записи в файлы `utmp` и `wtmp` о входе в систему и выходе из системы обычно записываются процессом, который обслуживает вход в систему и определяет момент выхода (в нашем случае — сервер `telnetd`).

Глава 10

- 10.1** Программа завершит работу, когда мы пошлем ей первый сигнал. Дело в том, что функция `pause` возвращает управление сразу, как только будет перехвачен какой-либо сигнал.
- 10.3** Схема состояния стека приводится на рис. С.3. Вызов функции `longjmp` из `sig_alarm` выполняет переход обратно в функцию `sleep2`, прерывая работу функции `sig_int`. В этой точке `sleep2` возвращает управление функции `main`.

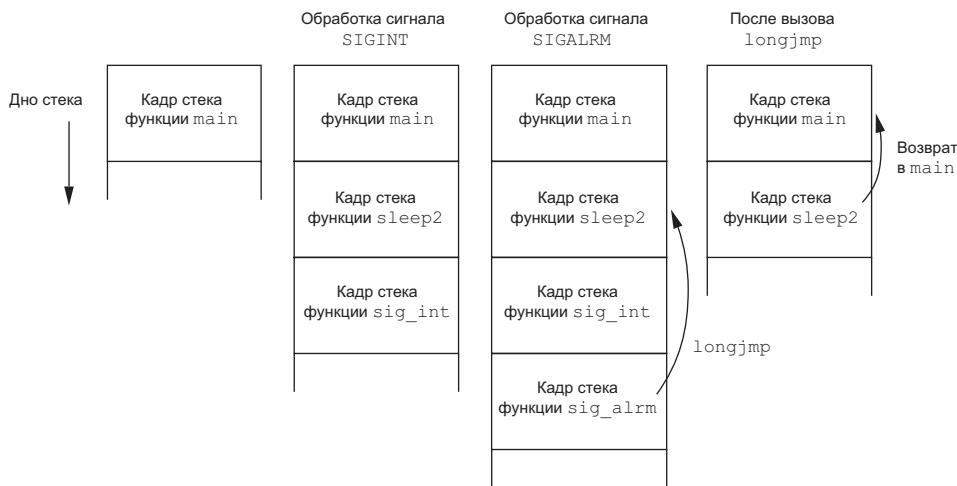


Рис. С.3. Состояние стека до и после вызова функции `longjmp`

- 10.4** Мы снова столкнулись с состоянием гонки за ресурсами, на этот раз между первым вызовом функции `alarm` и вызовом функции `setjmp`. Если ядро заблокирует процесс между этими двумя вызовами и истечет время тайм-аута, процесс получит сигнал и для его обработки вызовет обработчик сигнала, который, в свою очередь, вызовет функцию `longjmp`. Но так как `setjmp` еще не вызывалась, буфер `env_alarm` не будет заполнен корректными значениями. Поведение функции `longjmp` не определено, если буфер перехода не был инициализирован функцией `setjmp`.
- 10.5** За примерами обращайтесь к статье Дона Либеса (Don Libes) «Implementing Software Timers» (C Users Journal, vol. 8, no. 11, Nov 1990). Электронная ко-

ния этой статьи доступна по адресу <http://www.kohala.com/start/libes.timers.txt>.

- 10.7** Если просто вызвать функцию `_exit`, по коду завершения процесса не будет видно, что он завершился по сигналу `SIGABRT`.
- 10.8** Если сигнал был послан процессом, который принадлежит некоторому другому пользователю, этот процесс должен иметь сохраненный set-user-ID, равный идентификатору суперпользователя или идентификатору владельца процесса, принимающего сигнал, иначе функция `kill` не сможет послать сигнал. Поэтому реальный идентификатор несет больше информации для процесса, принимающего сигнал.
- 10.10** В одной из систем, используемых автором, значение количества секунд увеличивалось на 1 каждые 60–90 минут. Это отклонение обусловлено тем, что каждый вызов `sleep` планирует событие в будущем, но момент пробуждения процесса не совсем точно соответствует запланированному (из-за нагрузки на центральный процессор). Кроме того, некоторый объем времени требуется, чтобы возобновить работу процесса после приостановки и опять вызвать функцию `sleep`.

Такие программы, как `cron`, получают текущее время каждую минуту и в первый раз задают время приостановки таким, чтобы возобновить работу в начале следующей минуты (преобразуя текущее время в локальное и извлекая значение поля `tm_sec`). Каждую минуту они устанавливают величину очередного периода приостановки так, чтобы процесс возобновил работу в начале следующей минуты. Обычно это будут вызовы `sleep(60)` и изредка, для синхронизации с текущим временем, `sleep(59)`. Но иногда, когда выполнение запланированных команд занимает продолжительное время или при высокой нагрузке на систему, может быть выбрано меньшее значение аргумента функции `sleep`.

- 10.11** В Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 обработчик сигнала `SIGXFSZ` никогда не будет вызван. Но функция `write` вернет число 24, как только размер файла превысит 1024 байт. В FreeBSD 8.9 и Mac OS X 10.6.8, когда размер файла достигнет 1000 байт, обработчик сигнала будет вызван при следующей же попытке записать очередные 100 байт, а функция `write` вернет значение –1 с кодом ошибки `EFBIG` (File too big — файл слишком велик) в переменной `errno`.
- 10.12** Результат зависит от реализации стандартной библиотеки ввода/вывода: от того, как функция `fwrite` обрабатывает прерывание системного вызова `write`.

В Linux 3.2.0, например, когда функция `fwrite` используется для записи большого буфера, она вызывает `write` непосредственно, передавая ей то же количество байтов. Если в процессе выполнения системного вызова `write` поступит сигнал `SIGALRM`, приложение не получит его, пока `write` не завершит запись. По всей видимости ядро блокирует сигнал, если он поступает в процессе выполнения системного вызова `write`.

В Solaris 10, напротив, функция `fwrite` передает данные системному вызову `write` блоками по 8 Кбайт. Поэтому сигнал `SIGALRM` в этой системе может прервать выполнение функции `fwrite`. После возврата из обработчика сигнала управление будет передано во внутренний цикл функции `fwrite` и она продолжит запись данных блоками по 8 Кбайт.

Глава 11

- 11.1** Версия программы, которая выделяет область динамической памяти вместо использования автоматических переменных, приводится в листинге С.9.

Листинг С.9. Корректное использование возвращаемого значения потока

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    fputs(s, stdout);
    printf(" структура по адресу 0x%lx\n", (unsigned long)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo *fp;

    if ((fp = malloc (sizeof(struct foo))) == NULL)
        err_sys("невозможно выделить область динамической памяти");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("поток:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int err;
    pthread_t tid1;
    struct foo *fp;
```

```
err = pthread_create(&tid1, NULL, thr_fn1, NULL);
if (err != 0)
    err_exit(err, "невозможно создать поток 1");
err = pthread_join(tid1, (void *)&fp);
if (err != 0)
    err_exit(err, "невозможно присоединить поток 1");
printf("родительский процесс:\n", fp);
exit(0);
}
```

- 11.2** Чтобы изменить идентификатор потока для задания, ожидающего обработки, необходимо блокировку чтения/записи установить в режиме для записи, чтобы предотвратить возможность поиска по списку, пока не будет произведено изменение идентификатора. Проблема, связанная с текущим определением интерфейсов, заключается в том, что идентификатор задания может измениться между моментом, когда задание будет найдено функцией `job_find`, и моментом, когда задание будет исключено из списка функцией `job_remove`. Этую проблему можно решить добавлением счетчика ссылок и мьютекса в структуру `job`; тогда функция `job_find` должна будет увеличивать счетчик ссылок, а код, который производит изменение идентификатора, сможет пропускать те задания в списке, которые имеют ненулевой счетчик ссылок.
- 11.3** Во-первых, список защищен блокировкой чтения/записи, но переменная состояния должна быть под защитой мьютекса. Во-вторых, каждый поток должен ожидать появления задания для обработки на своей собственной переменной состояния, поэтому нам придется создать для каждого потока структуру данных, которая представляла бы это состояние. Как вариант, можно ввести переменную состояния и мьютекс в структуру `queue`, но это означало бы, что все рабочие потоки ожидали бы на одной и той же переменной состояния. При большом количестве рабочих потоков мы могли бы столкнуться с проблемой *громящего стада* (*thundering herd*), когда большое количество потоков возобновляют работу, но для них не находится заданий и в результате они впустую расходуют ресурсы процессора, ужесточая борьбу за обладание блокировкой.
- 11.4** Это зависит от обстоятельств. Вообще оба варианта могут работать вполне корректно, но каждый из них имеет свои недостатки. В первом случае ожидающие потоки будут запланированы на возобновление работы после вызова `pthread_cond_broadcast`. Если программа работает в многопроцессорной среде, некоторые запущенные потоки окажутся сразу же заблокированными, потому что мьютекс все еще заперт (не забывайте, что `pthread_cond_wait` возвращает управление с запертым мьютексом). Во втором случае работающий поток может успеть захватить мьютекс между действиями 3 и 4, среагировать на изменение состояния, сделав его недействительным, и освободить мьютекс. Затем, когда будет вызвана `pthread_cond_broadcast`, состояние больше не будет истинным, и поток отработает понапрасну. По этой причине поток всегда должен перепроверять истинность состояния, а не полагаться на то, что оно истинно, просто потому, что функция `pthread_cond_wait` вернула управление.

Глава 12

- 12.1 Эта проблема не связана с многопоточной архитектурой приложения, как может показаться на первый взгляд. Процедуры стандартной библиотеки ввода/вывода в действительности являются безопасными в контексте потоков. Когда мы называем функцию `fork`, каждый процесс получает отдельную копию структур данных стандартной библиотеки ввода/вывода. При запуске программы со стандартным выводом, присоединенным к терминалу, вывод будет буферизоваться построчно, поэтому каждый раз, когда мы выводим строку, стандартная библиотека ввода/вывода будет записывать ее в устройство терминала. Однако если перенаправить стандартный вывод в файл, библиотека выберет для него режим полной буферизации. Фактическая запись в файл будет произведена только при заполнении буфера или при закрытии потока. В этом примере к моменту вызова функции `fork` буфер уже содержит несколько еще не записанных в файл строк, поэтому, когда родительский и дочерний процессы наконецбросят свои копии буферов, первоначальное их содержимое будет записано в файл дважды.
- 12.3 Теоретически, заблокировав доставку всех сигналов при вызове обработчика сигнала, мы могли бы сделать функцию безопасной в контексте обработки асинхронных сигналов. Проблема в том, что мы не знаем, не разблокирует ли какая-либо функция, к которой мы обращаемся, какой-либо из заблокированных сигналов, сделав тем самым возможным повторное вхождение в обработчик другого сигнала.
- 12.4 В FreeBSD 8.0 программа завершилась аварийно с созданием файла `core`. С помощью отладчика `gdb` удалось определить, что программа застряла в бесконечном цикле инициализации. В процессе инициализации программа вызывала функции инициализации потоков, которые обращаются к функции `getenv`, чтобы получить значения переменных окружения `LIBPTHREAD_SPINLOOPS` и `LIBPTHREAD_YIELDLOOPS`. Однако наша потокобезопасная реализация `getenv` вызывает функции из библиотеки `pthread`, находясь в промежуточном, противоречивом состоянии. Кроме того, функции инициализации потоков пытаются вызвать `malloc`, которая, в свою очередь, вызывает `getenv`, чтобы получить значение переменной окружения `MALLOC_OPTIONS`.

Чтобы обойти эту проблему, можно по умолчанию полагать, что программа запускается как однопоточная, и сообщать нашей версии `getenv` о необходимости инициализации потока с помощью флага. При ложном значении флага наша версия `getenv` может действовать подобно нереентерабельной версии (и тем самым избежать вызовов функций из библиотеки `pthread` и `malloc`). Также можно было бы реализовать отдельную функцию инициализации, вызывающую `pthread_once`, чтобы не вызывать ее из `getenv`. При такой организации программа должна будет вызывать данную функцию инициализации перед вызовом `getenv`. Это решает проблему, потому что `gentenv` не будет вызвана, пока программа не завершит инициализацию. А после вызова функции инициализации наша версия `getenv` будет действовать потокобезопасным образом.

- 12.5 Функция `fork` по-прежнему необходима, если мы захотим запустить одну программу из другой (то есть вызывать `fork` перед вызовом `exec`).
- 12.6 В листинге С.10 приводится потокобезопасная реализация функции `sleep`, которая для организации задержки использует функцию `select`. Она безопасна в многопоточной среде потому, что не использует никаких незащищенных глобальных или статических данных и вызывает только безопасные функции.
- 12.7 Реализация переменной состояния, скорее всего, использует мьютекс для защиты ее внутренней структуры. Поскольку это уже относится к области реализации конкретных версий UNIX и скрыто от нас, какого-либо переносимого способа захватить или отпустить блокировку в момент ветвления процесса не существует. Поскольку мы не можем определить состояние внутренней блокировки в переменной состояния после вызова функции `fork`, использование переменных состояния в дочернем процессе будет небезопасным.

Листинг С.10. Реализация потокобезопасной функции sleep

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned nsec)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = nsec;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= nsec)
        return(0);
    return(nsec - slept);
}
```

Глава 13

- 13.1 Если процесс вызовет функцию `chroot`, он не сможет открыть устройство `/dev/log`. Решение заключается в том, чтобы вызвать функцию `openlog` с флагом `LOG_NDELAY` в аргументе `option` перед обращением к функции `chroot`. В результате демон откроет специальный файл устройства (сокет дейтаграмм из домена UNIX), что даст ему дескриптор, который останется

действительным даже после вызова `chroot`. С подобным алгоритмом можно столкнуться в таких демонах, как `ftpd` (демон службы передачи файлов по протоколу FTP), где функция `chroot` используется из соображений безопасности, но для регистрации ошибок в системном журнале используется `syslog`.

- 13.3** Решение приводится в листинге С.11. Результат зависит от платформы. Вспомните, что функция `daemonize` закрывает все дескрипторы файлов и снова открывает первые три на устройстве `/dev/null`. Это означает, что процесс не имеет управляющего терминала, в результате функция `getlogin` не сможет отыскать запись о процессе в файле `utmp`. Поэтому в Linux 3.2.0 и Solaris 10 можно обнаружить, что демоны не имеют имени пользователя. Однако в FreeBSD 8.0 и Mac OS X 10.6.8 имя пользователя сохраняется в таблице процессов и копируется в дочерний процесс при вызове функции `fork`. Это означает, что процесс всегда может узнать имя пользователя, если только он не был запущен одним из процессов, которые не имеют имени пользователя (как, например, процесс `init`).

Листинг С.11. Вызов функции `daemonize` и попытка определить имя пользователя

```
#include "apue.h"

int
main(void)
{
    FILE *fp;
    char *p;

    daemonize("getlog");
    p = getlogin();
    fp = fopen("/tmp/getlog.out", "w");
    if (fp != NULL) {
        if (p == NULL)
            fprintf(fp, "процесс не имеет имени пользователя\n");
        else
            fprintf(fp, "имя пользователя: %s\n", p);
    }
    exit(0);
}
```

Глава 14

- 14.1** Тестовая программа приводится в листинге С.12.

Листинг С.12. Проверка поведения механизма блокировки записей в файле

```
#include "apue.h"
#include <fcntl.h>
#include <errno.h>

void
```

```
sigint(int signo)
{
}

int
main(void)
{
    pid_t pid1, pid2, pid3;
    int fd;

    setbuf(stdout, NULL);
    signal_intr(SIGINT, sigint);

    /*
     * Создать файл.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("невозможно открыть/создать файл блокировки");

    /*
     * Установить блокировку для чтения.
     */
    if ((pid1 = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid1 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 1: невозможно заблокировать "
                    "файл для чтения");
        printf("потомок 1: установлена блокировка для чтения\n");
        pause();
        printf("потомок 1: выход после паузы\n");
        exit(0);
    } else { /* предок */
        sleep(2);
    }

    /*
     * Родительский процесс продолжается ...
     * снова установить блокировку для чтения.
     */
    if ((pid2 = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid2 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 2: невозможно заблокировать "
                    "файл для чтения");
        printf("потомок 2: установлена блокировка для чтения\n");
        pause();
        printf("потомок 2: выход после паузы\n");
        exit(0);
    } else { /* родительский процесс */
        sleep(2);
    }
}
```

```

/*
 * Родительский процесс продолжается ... блокируется
 * при попытке установить блокировку для записи.
 */
if ((pid3 = fork()) < 0) {
    err_sys("ошибка вызова функции fork");
} else if (pid3 == 0) { /* потомок */
    if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
        printf("потомок 3: невозможно заблокировать "
               "файл для записи:%s\n",
               strerror(errno));
    printf("потомок 3: останов, пока не получит блокировку...\n");
    if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
        err_sys("потомок 3: невозможно заблокировать "
               "файл для записи");
    printf("потомок 3 сумел установить блокировку для записи???\n");
    pause();
    printf("потомок 3: выход после паузы\n");
    exit(0);
} else { /* родительский процесс */
    sleep(2);
}

/*
 * Проверить, будет ли заблокирована попытка получить
 * блокировку для записи очередной попыткой установки
 * блокировки для чтения.
 */
if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
    printf("родитель: невозможно заблокировать файл для чтения: %s\n",
           strerror(errno));
else
    printf("родитель: установлена дополнительная "
           "блокировка для чтения,"
           " запрос на установку блокировки для записи ожидает\n");
printf("останавливается потомок 1...\n");
kill(pid1, SIGINT);
printf("останавливается потомок 2...\n");
kill(pid2, SIGINT);
printf("останавливается потомок 3...\n");
kill(pid3, SIGINT);
exit(0);
}

```

В FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 был получен одинаковый результат: дополнительные читающие процессы могут оставить пишущие процессы ни с чем. Запустив программу, мы получили следующие результаты:

```

потомок 1: установлена блокировка для чтения
потомок 2: установлена блокировка для чтения
потомок 3: невозможно заблокировать файл для записи: Resource temporarily

```

```
unavailable
потомок 3 пытается установить блокировку для записи
родитель: установлена дополнительная блокировка для чтения, запрос
на установку блокировки для записи ожидает
останавливается потомок 1...
потомок 1: выход после паузы
останавливается потомок 2...
потомок 2: выход после паузы
останавливается потомок 3...
потомок 3: невозможно заблокировать файл для записи: Interrupted system call
```

В Solaris 10 читающие процессы не способны заблокировать пишущие процессы. В данном примере родительский процесс не сможет получить блокировку на чтение, потому что имеются процессы, ожидающие получения блокировки для записи.

- 14.2** В большинстве систем тип данных `fd_set` определен как структура, которая содержит всего одно поле — массив длинных целых чисел. Каждый бит в этом массиве соответствует одному дескриптору. Макросы `FD_` работают с этим массивом длинных целых чисел, включая, выключая и возвращая состояние отдельных битов.

Одна из причин, по которым этот тип данных был объявлен как структура, содержащая массив, а не просто как массив, заключается в том, что это дает возможность присваивать значение одной переменной типа `fd_set` другой переменной типа `fd_set` обычным оператором присваивания языка C.

- 14.3** В старые добрые времена большинство систем допускало возможность определения константы `FD_SETSIZE` перед подключением заголовочного файла `<sys/select.h>`. Например, с помощью инструкций

```
#define FD_SETSIZE 2048
#include <sys/select.h>
```

можно определить размер типа `fd_set` таким, чтобы он мог вместить 2048 дескрипторов. К сожалению, это больше невозможно. Чтобы добиться подобного в современных системах, необходимо:

1. Прежде чем подключать какие-либо заголовочные файлы, необходимо определить символ, предотвращающий подключение `<sys/select.h>`. Некоторые системы могут защищать определение типа `fd_set` отдельным символом. Нам также нужно определить его. Например, чтобы предотвратить подключение `<sys/select.h>` в FreeBSD 8.0, необходимо определить символ `_SYS_SELECT_H_` и также определить символ `_FD_SET`, чтобы предотвратить включение определения типа `fd_set`.
2. Иногда, для совместимости со старыми приложениями, заголовочный файл `<sys/types.h>` определяет размер типа `fd_set`, поэтому необходимо сначала подключить его, а затем удалить символ `FD_SETSIZE`. Обратите внимание, что в некоторых системах вместо `FD_SETSIZE` используется символ `__FD_SETSIZE`.

3. Далее следует переопределить символ `FD_SETSIZE` (или `_FD_SETSIZE`), указав желаемое максимальное значение количества дескрипторов, которое может использоваться с функцией `select`.
4. Затем необходимо удалить символ, определенный на шаге 1.
5. И наконец, подключить `<sys/select.h>`.

Прежде чем запускать программу, следует настроить систему, чтобы она позволяла открывать столько дескрипторов, сколько потребуется, и мы действительно могли использовать `FD_SETSIZE` дескрипторов.

- 14.4** В следующей таблице перечислены функции, которые решают сходные задачи.

<code>FD_ZERO</code>	<code>sigemptyset</code>
<code>FD_SET</code>	<code>sigaddset</code>
<code>FD_CLR</code>	<code>sigdelset</code>
<code>FD_ISSET</code>	<code>sigismember</code>

В семействе `FD_XXX` нет функции, которая соответствовала бы функции `sigfillset`. При работе с сигналами указатель на набор сигналов всегда передается в первом аргументе, а номер сигнала — во втором. При работе с наборами дескрипторов в первом аргументе передается номер дескриптора, а в следующем — указатель на набор дескрипторов.

- 14.5** В листинге С.13 показана реализация с использованием функции `select`.

Листинг С.13. Реализация функции `sleep_us` на основе функции `select`

```
#include "apue.h"
#include <sys/select.h>

void
sleep_us(unsigned int nusecs)
{
    struct timeval tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

В листинге С.14 показана аналогичная реализация с использованием функции `poll`.

Листинг С.14. Реализация функции `sleep_us` на основе функции `poll`

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
```

```
struct pollfd dummy;
int timeout;

if ((timeout = nusecs / 1000) <= 0)
    timeout = 1;
poll(&dummy, 0, timeout);
}
```

Как утверждает страница справочного руководства `usleep(3)` в BSD, функция `usleep` использует в своей работе `nanosleep`. Она корректно взаимодействует с другими таймерами, установленными вызывающим процессом, и не прерывается в случае перехвата сигнала.

- 14.6** Нет. В этом случае `TELL_WAIT` должна была бы создать временный файл длиной в два байта, где один байт отводится для родительского и один байт — дочернего процесса. Функция `WAIT_CHILD` могла бы заставить родительский процесс ожидать снятия блокировки с байта дочернего процесса, а `TELL_PARENT` — снимать блокировку с байта дочернего процесса. Проблема, однако, в том, что функция `fork` снимает все блокировки в дочернем процессе, поэтому дочерний процесс не может быть запущен с какими-либо установленными блокировками.

- 14.7** Решение приводится в листинге C.15.

Листинг C.15. Подсчет емкости неименованного канала с помощью неблокирующей операции записи

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    set_fl(fd[1], O_NONBLOCK);

    /*
     * Записывать по 1 байту, пока канал не заполнится.
     */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("функция write вернула число %d, ", i);
            break;
        }
    }
    printf("емкость канала = %d\n", n);
    exit(0);
}
```

В следующей таблице показаны значения, полученные на наших четырех платформах.

Платформа	Емкость канала в байтах
FreeBSD 8.0	65 536
Linux 3.2.0	65 536
Mac OS X 10.6.8	16 384
Solaris 10	16 384

Эти значения могут отличаться от значения константы `PIPE_BUF`, поскольку эта константа определяет максимальный объем данных, которые можно записать в канал атомарно. Здесь же мы получили объем данных, которые могут находиться в канале, не принимая во внимание атомарность их записи.

- 14.10** Изменит ли программа из листинга 14.10 время последнего обращения к исходному файлу, зависит от операционной системы и типа файловой системы, в которой размещается файл. На всех четырех платформах, рассматриваемых в книге, время последнего обращения к файлу обновляется, если он располагается в файловой системе, используемой по умолчанию этими платформами.

Глава 15

- 15.1** Если конец канала, открытый для записи, не будет закрыт, процесс, читающий данные из канала, никогда не увидит признака конца файла. Поэтому программа постраничного просмотра окажется «навечно» заблокированной в операции чтения со стандартного ввода.
- 15.2** Родительский процесс завершится сразу после записи в канал последней строки. Конец канала, открытый для чтения, автоматически закроется по завершении родительского процесса. Но родительский процесс наверняка опережает потомка на один буфер, поскольку дочерний процесс (программа постраничного просмотра) ожидает, пока пользователь не просмотрит выведенную перед ним страницу. Если запустить программу в командной оболочке, такой как Korn shell, которая работает в диалоговом режиме, оболочка наверняка изменит режим терминала по завершении работы родительского процесса и выведет свое приглашение. Это несомненно повлияет на программу постраничного просмотра, так как она тоже изменяет режим терминала. (Большинство программ постраничного просмотра в ожидании перехода к следующей странице переводят терминал в неканонический режим.)
- 15.3** Функция `ropen` вернет указатель на структуру `FILE`, потому что она запустит командную оболочку. Но сама командная оболочка не сможет выполнить несуществующую команду и потому выведет строку

```
sh: line 1: ./a.out: No such file or directory
```

в стандартное устройство вывода сообщений об ошибках и завершится с кодом 127 (впрочем, код завершения зависит от типа командной оболочки). Функция `pclose` вернет код завершения команды, который будет получен от функции `waitpid`.

- 15.4** После завершения родительского процесса посмотрите код его завершения. В Bourne shell, Bourne-again shell и Korn shell это можно сделать с помощью команды `echo $?`. Она выведет число, равное сумме числа 128 и номера сигнала.
- 15.5** Прежде всего, нужно добавить объявление

```
FILE *fpin, *fpout;
```

Затем с помощью функции `fdopen` связать дескриптор канала с потоком ввода/вывода и назначить ему построчный режим буферизации. Сделать это необходимо перед входом в цикл `while`, где производится чтение из стандартного ввода:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
```

Обращения к функциям `read` и `write` в цикле заменить строками

```
if (fputs(line, fpout) == EOF)
    err_sys("ошибка вывода в канал");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("дочерний процесс закрыл канал");
    break;
}
```

- 15.6** Функция `system` вызовет `wait`, и первым завершится дочерний процесс, запущенный функцией `popen`. Поскольку это не тот потомок, который был запущен функцией `system`, она снова вызовет функцию `wait` и заблокируется, пока не завершится работа команды `sleep`. После этого функция `system` вернет управление. Когда `pclose` вызовет `wait`, она вернет признак ошибки, поскольку все дочерние процессы уже завершили работу. Вслед за ней и сама `pclose` вернет признак ошибки.
- 15.7** Функция `select` отметит дескриптор как доступный для чтения. Когда функция `read` будет вызвана после чтения всех данных из канала, она вернет 0 в качестве признака конца файла. В случае с функцией `poll` будетозвращено событие `POLLHUP`, а оно может быть возвращено, даже если в канале еще имеются данные, доступные для чтения. Однако когда функция `read` прочитает все данные, она вернет 0 как признак конца файла. После чтения всех данных событие `POLLIN` возвращено не будет, даже если нам еще только

предстоит прочитать признак конца файла (возвращаемое значение 0 функции `read`).

Таблица С.1. Поведение функций `select` и `poll` при работе с каналами

Операция	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Вызов <code>select</code> в читающем процессе, когда пишущий процесс закрыл свой дескриптор	R/W/E	R	R/W	R/W/E
Вызов <code>poll</code> в читающем процессе, когда пишущий процесс закрыл свой дескриптор	R/HUP	HUP	INV	HUP
Вызов <code>select</code> в пишущем процессе, когда читающий процесс закрыл свой дескриптор	R/W/E	R/W	R/W	R/W
Вызов <code>poll</code> в пишущем процессе, когда читающий процесс закрыл свой дескриптор	R/HUP	W/ERR	INV	HUP

Сокращения в табл. С.1: R (readable — доступно для чтения), W (writable — доступно для записи), E (exception — исключение), HUP (hangup — разрыв связи), ERR (error — ошибка) и INV (invalid file descriptor — недопустимый дескриптор файла). Для дескриптора, ссылающегося на канал, который был закрыт читающим процессом, `select` сообщает, что дескриптор доступен для записи. Но когда будет вызвана функция `write`, система генерирует сигнал `SIGPIPE`. Если сигнал игнорируется программой или обработчик сигнала вернет управление, `write` вернет признак ошибки с кодом `EPIPE` в переменной `errno`. Однако поведение функции `poll` в подобной ситуации может отличаться в разных системах.

- 15.8 Все, что будет выведено дочерним процессом в стандартный вывод сообщений об ошибках, будет отправлено в стандартный вывод сообщений об ошибках родительского процесса. Чтобы отправить данные со стандартного вывода сообщений об ошибках родительскому процессу, включите в `cmdstring` операцию перенаправления `2>&1`.
- 15.9 Функция `ropen` создает дочерний процесс, а он запускает командный интерпретатор. Командный интерпретатор, в свою очередь, вызывает `fork`, и новый дочерний процесс командного интерпретатора запускает командную строку. Родительский командный интерпретатор дожидается, когда `cmdstring` завершится, и также завершает работу, чего, в свою очередь, ожидает функция `waitpid` в `pclose`.
- 15.10 Хитрость заключается в том, что канал FIFO надо открыть дважды: один раз для чтения и один раз для записи. Мы вообще не используем дескриптор, открытый для записи, но оставляем его открытым для предотвращения генерации признака конца файла, когда количество клиентов

уменьшается с 1 до 0. Открытие FIFO в два приема требует некоторых дополнительных действий, так как оно должно производиться в неблокирующем режиме. Сначала мы должны открыть FIFO только для чтения в неблокирующем режиме, а затем вызвать `open` в блокирующем режиме, чтобы открыть канал только для записи. (Если мы сначала попытаемся открыть FIFO в неблокирующем режиме только для записи, функция `open` вернет признак ошибки.) Затем мы должныбросить флаг неблокирующего режима в дескрипторе, открытому для чтения. В листинге C.16 показано, как это делается.

Листинг C.16. Открытие канала FIFO для чтения и записи без блокировки процесса

```
#include "apue.h"
#include <fcntl.h>

#define FIFO "temp fifo"

int
main(void)
{
    int fdread, fdwrite;

    unlink(FIFO);
    if ((mkfifo(FIFO, FILE_MODE) < 0))
        err_sys("ошибка вызова функции mkfifo");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("ошибка открытия для чтения");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("ошибка открытия для записи");
    clr_f1(fdread, O_NONBLOCK);
    exit(0);
}
```

15.11 Беспорядочное чтение сообщений из активной очереди может повлечь за собой конфликты между сервером и клиентом из-за несоблюдения протокола обмена, так как в этом случае могут быть утеряны либо запросы клиента, либо отклики сервера. Чтобы получить возможность чтения из очереди, процесс должен знать ее идентификатор, а очередь должна иметь установленный бит `world-read` (доступ на чтение для всех остальных).

15.12 Мы никогда не должны хранить фактические адреса в сегменте разделяемой памяти, поскольку существует вероятность, что сервер и все клиенты подключают этот сегмент к различным адресам. Вместо адресов в связанном списке, который строится в сегменте разделяемой памяти, следует использовать величины смещений объекта от начала сегмента разделяемой памяти. Эти смещения формируются путем вычитания адреса начала сегмента разделяемой памяти из адреса объекта.

15.14 В табл. C.2 приводится схема происходящих событий.

Таблица С.2. Чередование периодов работы родительского и дочернего процессов из листинга 15.12

Значение i в роди- теле	Значение i в потомке	Разделяемое значение	Возвращаемое значение update	Комментарий
		0		Инициализируется функ- цией mmap
	1			Потомок запускается пер- вым и затем блокируется
0				Запускается родитель
		1		
			0	Затем родитель блокиру- ется
		2		Потомок возобновляет работу
			1	
	3			Затем потомок блокиру- ется
2				Родитель возобновляет работу
		3		
			2	Затем родитель блокиру- ется
		4		
			3	
	5			Затем потомок блокиру- ется
4				Родитель возобновляет работу

Глава 16

16.1 В листинге С.17 приводится программа, которая определяет порядок бай-
тов для аппаратной архитектуры, на которой она запущена.

Листинг С.17. Определение порядка байтов

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
```

```

{
    uint32_t i = 0x04030201;
    unsigned char *cp = (unsigned char *)&i;

    if (*cp == 1)
        printf("обратный (little-endian)\n");
    else if (*cp == 4)
        printf("прямой (big-endian)\n");
    else
        printf("неизвестный?\n");
    exit(0);
}

```

- 16.3** Каждый из сокетов, который будет принимать запросы на соединение, должен быть привязан к своему адресу, и для каждого дескриптора должна быть создана соответствующая запись в структуре `fd_set`. Для приема запросов на соединение на нескольких адресах мы будем использовать функцию `select`. В разделе 16.4 уже говорилось, что по прибытии запроса на соединение дескриптор сокета будет отмечен как доступный для чтения. Прибывающие запросы на соединение мы будем принимать и обслуживать, как и прежде.
- 16.5** Для этого нужно установить обработчик сигнала `SIGCHLD`, обратившись к функции `signal` (листинг 10.12), которая устанавливает обработчик сигнала с помощью функции `sigaction`, позволяющей определить возможность перезапуска прерванных системных вызовов. Затем следует убрать вызов `waitpid` из функции `serve`. После запуска дочернего процесса родитель закрывает новый дескриптор и переходит к ожиданию новых запросов на соединение. И наконец, нам нужен сам обработчик сигнала `SIGCHLD`:

```

void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}

```

- 16.6** Чтобы разрешить асинхронный режим работы сокета, необходимо назначить процесс владельцу сокета с помощью команды `F_SETOWN` функции `fcntl` и разрешить асинхронную доставку сигнала с помощью команды `FIOASYNC` функции `ioctl`. Чтобы запретить асинхронный режим работы сокета, достаточно просто запретить асинхронную доставку сигнала. Смешивание вызовов функций `fcntl` и `ioctl` необходимо для обеспечения переносимости. Код функций приводится в листинге C.18.

Листинг C.18. Функции разрешения и запрещения асинхронного режима работы сокета

```

#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

```

```

#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}

int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}

```

Глава 17

- 17.1** Обычные каналы обеспечивают доступ к данным как к потоку байтов. Для определения границ сообщений в каждое из них необходимо добавить заголовок, где указать длину сообщения. Но при этом все еще сохраняется необходимость выполнять две дополнительные операции копирования: одну — для записи в канал и одну — для чтения из канала. Намного эффективнее использовать канал только для передачи главному потоку сигнала о доступности нового сообщения. Для этого достаточно использовать один байт. При использовании такого решения нам потребуется переместить структуру `mymsg` в структуру `threadinfo` и задействовать мьютекс и переменную состояния, чтобы помешать вспомогательному потоку повторно использовать структуру `mymsg`, пока она не будет обработана главным потоком. Реализация этого решения представлена в листинге С.19.

Листинг С.19. Проверка наличия сообщений XSI с использованием каналов

```

#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3      /* количество очередей */

```

```
#define MAXMSZ 512      /* максимальный размер сообщения */
#define KEY      0x123    /* ключ для первой очереди сообщений */

struct mymsg {
    long      mtype;
    char     mtext[MAXMSZ+1];
};

struct threadinfo {
    int      qid;
    int      fd;
    int      len;
    pthread_mutex_t mutex;
    pthread_cond_t ready;
    struct mymsg   m;
};

void *
helper(void *arg)
{
    int n;
    struct threadinfo  *tip = arg;

    for(;;) {
        memset(&tip->m, 0, sizeof(struct mymsg));
        if ((n = msgrcv(tip->qid, &tip->m, MAXMSZ, 0,
                         MSG_NOERROR)) < 0)
            err_sys("ошибка вызова функции msgrcv");
        tip->len = n;
        pthread_mutex_lock(&tip->mutex);
        if (write(tip->fd, "a", sizeof(char)) < 0)
            err_sys("ошибка вызова функции write");
        pthread_cond_wait(&tip->ready, &tip->mutex);
        pthread_mutex_unlock(&tip->mutex);
    }
}

int
main()
{
    char      c;
    int       i, n, err;
    int       fd[2];
    int       qid[NQ];
    struct pollfd  pfd[NQ];
    struct threadinfo  ti[NQ];
    pthread_t    tid[NQ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("ошибка вызова функции msgget");

        printf("очередь %d получила идентификатор %d\n", i, qid[i]);
    }
}
```

```

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("ошибка вызова функции socketpair");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if (pthread_cond_init(&ti[i].ready, NULL) != 0)
            err_sys("ошибка вызова функции pthread_cond_init");
        if (pthread_mutex_init(&ti[i].mutex, NULL) != 0)
            err_sys("ошибка вызова функции pthread_mutex_init");
        if ((err = pthread_create(&tid[i], NULL, helper,
                                &ti[i])) != 0)
            err_exit(err, "ошибка вызова функции pthread_create");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("ошибка вызова функции poll");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, &c, sizeof(char))) < 0)
                    err_sys("ошибка вызова функции read");
                ti[i].m.mtext[ti[i].len] = 0;
                printf("очередь: %d, сообщение: %s\n", qid[i],
                       ti[i].m.mtext);
                pthread_mutex_lock(&ti[i].mutex);
                pthread_cond_signal(&ti[i].ready);
                pthread_mutex_unlock(&ti[i].mutex);
            }
        }
    }
    exit(0);
}

```

- 17.3** *Объявление* определяет атрибуты (такие, как тип данных) набора идентификаторов. Если объявление предполагает выделение памяти под объявленные объекты, то такое объявление называется *определением*.

В заголовочном файле `opend.h` мы объявляем три глобальные переменные с классом хранения `extern`. Эти объявления не подразумевают выделения памяти для хранения значений переменных. В файле `main.c` мы определяем три глобальные переменные. Иногда определение глобальной переменной может сопровождаться ее инициализацией, но мы, как правило, позволяем языку C инициализировать ее значением по умолчанию.

- 17.5** Обе функции, `select` и `poll`, возвращают количество дескрипторов, готовых к выполнению операции. Цикл обхода массива `client` может быть завершен раньше, когда число обработанных дескрипторов достигнет значения, полученного от функции `select` или `poll`.

- 17.6** Первая проблема заключается в состоянии гонки в интервале времени между вызовами `stat` и `unlink`, в течение которого файл может изменить-

ся. Вторая проблема проявляется, когда имя представляет символьическую ссылку на файл сокета домена UNIX — функция `stat` сообщит, что это сокет (функция `stat` следует по символьическим ссылкам), но функция `unlink` удалит саму символьическую ссылку, а не файл сокета. Решить последнюю проблему можно, заменив вызов функции `stat` вызовом `lstat`, но это не решает первую проблему.

- 17.7 Первый способ — отправить оба дескриптора в одном управляющем сообщении. Все файловые дескрипторы хранятся в смежных областях памяти. Это демонстрирует следующий код:

```
struct msghdr msg;
struct cmsghdr *cmptr;
int *ip;

if ((cmptr = calloc(1, CMSG_LEN(2*sizeof(int)))) == NULL)
    err_sys("ошибка вызова функции calloc");

msg.msg_control = cmptr;
msg.msg_controllen = CMSG_LEN(2*sizeof(int));
/* продолжение инициализации msghdr... */
cmptr->cmsg_len = CMSG_LEN(2*sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
ip = (int *)CMSG_DATA(cmptr);
*ip++ = fd1;
*ip = fd2;
```

Данный прием можно использовать на всех четырех платформах, обсуждаемых в этой книге. Второй способ — упаковать две отдельные структуры `cmsghdr` в одно сообщение:

```
struct msghdr msg;
struct cmsghdr *cmptr;

if ((cmptr = calloc(1, 2*CMSG_LEN(sizeof(int)))) == NULL)
    err_sys("ошибка вызова функции calloc");
msg.msg_control = cmptr;
msg.msg_controllen = 2*CMSG_LEN(sizeof(int));
/* продолжение инициализации msghdr... */
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd1;
cmptr = CMPTN_NXTHDR(&msg, cmptr);
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd2;
```

В отличие от первого, этот способ работает только в FreeBSD 8.0.

Глава 18

- 18.1 Обратите внимание: поскольку терминал находится в неканоническом режиме, ввод команды `reset` должен завершаться символом перевода строки, а не символом возврата каретки.
- 18.2 Она строит таблицу для каждого из 128 символов и затем устанавливает самый старший бит (бит четности) в соответствии с указаниями пользователя. После этого она использует 8-разрядный ввод/вывод, самостоятельно обслуживая бит четности.
- 18.3 Если вы используете терминал с оконной системой, вам не нужно входить в систему дважды. Вы можете проделать этот эксперимент в двух отдельных окнах. В Solaris запустите команду `stty -a`, перенаправив стандартный ввод окна, в котором запущен редактор `vi`. Это позволит увидеть, что `vi` устанавливает параметры `MIN` и `TIME` в значение 1. Вызов функции `read` будет ожидать ввода хотя бы одного символа, но когда символ будет введен, функция `read`, прежде чем вернуть управление, будет ждать ввода дополнительных символов не дольше одной десятой доли секунды.

Глава 19

- 19.1 Оба сервера, `telnetd` и `rlogind`, работают с привилегиями суперпользователя, поэтому могут без ограничений пользоваться функциями `chown` и `chmod`.
- 19.2 Запустите `pty -n stty -a`, чтобы предотвратить инициализацию структур `termios` и `winsize` подчиненного терминала.
- 19.4 К сожалению, команда `F_SETFL` функции `fcntl` не позволяет изменять состояние режима «для чтения и для записи».
- 19.5 Здесь присутствуют три группы процессов: (1) командная оболочка входа, (2) дочерний и родительский процессы программы `pty`, (3) процесс `cat`. Первые две группы составляют единый сеанс, в котором в качестве лидера выступает командная оболочка входа. Второй сеанс содержит только процесс `cat`. Первая группа процессов (командная оболочка входа) является группой процессов фонового режима, а две другие — группами процессов переднего плана.
- 19.6 Первым завершится процесс `cat`, когда получит от своего модуля дисциплины обслуживания терминала признак конца файла. Это приведет к завершению ведомого PTY, что вызовет завершение ведущего PTY. Это, в свою очередь, приведет к тому, что родительский процесс, который получает ввод от ведущего PTY, получит признак конца файла. Родительский процесс пошлет сигнал `SIGTERM` дочернему процессу, вследствие чего дочерний процесс прекратит работу. (Дочерний процесс не перехватывает этот сигнал.) И наконец, родительский процесс вызовет функцию `exit(0)` в конце функции `main`.

Ниже приводится вывод программы из листинга 8.16, соответствующий данному случаю.

```
cat      e =      270, chars =      274, stat =      0:
pty      e =      262, chars =       40, stat =     15: F      X
pty      e =      288, chars =     188, stat =      0:
```

- 19.7** Сделать это можно с помощью команд `echo` и `date(1)`, запустив их в подоболочке:

```
#!/bin/sh
( echo "Сбор данных запущен " `date`;
  pty "${SHELL:/bin/sh}";
  echo " Сбор данных завершен " `date` ) | tee typescript
```

- 19.8** В модуле дисциплины обслуживания терминала, расположенному выше ведомого PTY, разрешен эхо-вывод, поэтому все, что читает PTY со своего стандартного ввода и записывает в ведущий PTY, по умолчанию выводится в виде эха. Эхо-вывод производится модулем дисциплины обслуживания терминала, расположенным выше подчиненного PTY, даже если программа (`ttynname`) не читает данные.

Глава 20

- 20.1** Наш консерватизм в установке блокировки в функции `_db_dodelete` обусловлен стремлением избежать состояния гонки в функции `db_nextrec`. Если вызов `_db_writedat` не будет защищен блокировкой, может возникнуть ситуация, когда запись с данными окажется стерта во время ее чтения функцией `db_nextrec`: функция `db_nextrec` может прочитать индексную запись, убедиться, что она не пуста, и приступить к чтению записи с данными, которая может быть стерта функцией `_db_dodelete` между вызовами `_db_readidx` и `_db_readdat` в `db_nextrec`.
- 20.2** Предположим, что `db_nextrec` вызывает `_db_readidx`, которая читает индекс в буфер процесса. Затем процесс приостанавливается ядром, и управление передается другому процессу. Другой процесс вызывает `db_delete` и удаляет запись, прочитанную первым процессом. Обе записи — ключ и данные — оказываются затертymi пробелами. Затем управление возвращается первому процессу, который вызывает `_db_readdat` (из `db_nextrec`) и читает запись с данными, затертую пробелами. Блокировка для чтения, устанавливаемая в `db_nextrec`, позволяет выполнить чтение индексной записи и записи с данными атомарно (относительно других процессов, использующих ту же самую базу данных).
- 20.3** Использование принудительных блокировок окажет влияние на другие читающие и пишущие процессы. Они заблокируются ядром, пока не будут сняты блокировки, установленные функциями `_db_writeidx` и `_db_writedat`.
- 20.5** Используя такой порядок записи (сначала данные, потом индекс), мы защищаем файлы базы данных от повреждения в случае, если процесс завершится между двумя операциями записи. Если процесс сначала запишет индексную запись и будет неожиданно завершен перед записью данных, мы

получим корректную индексную запись, которая указывает на некорректные данные.

Глава 21

- 21.5** Несколько подсказок. Проверять наличие заданий можно в двух местах: в очереди демона печати и во внутренней очереди сетевого принтера. Вы должны не допустить, чтобы один пользователь получил возможность отменить задание печати другого пользователя. Разумеется, суперпользователь должен иметь возможность отменить печать любого задания.
- 21.7** Этого не требуется, потому что демону не нужно повторно читать конфигурационный файл, пока не появится задание для печати. Функция `printer_thread` проверяет необходимость повторного чтения файла конфигурации перед каждой попыткой отправить задание принтеру.
- 21.9** Достаточно записать строку, оканчивающуюся нулевым байтом (не забывайте, что функция `strlen` не учитывает нулевой байт при определении длины строки). Существует два простых решения: либо добавлять 1 к счетчику записываемых байтов, либо использовать функцию `dprintf` вместо пары функций `sprintf` и `write`.

Список литературы

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. «Mach: A New Kernel Foundation for UNIX Development», Proceedings of the 1986 Summer USENIX Conference, pp. 93–113, Atlanta, GA.

Введение в операционную систему Mach.

Adams, J., Bustos, D., Hahn, S., Powell, D., and Praza, L. 2005. «Solaris Service Management Facility: Modern System Startup and Administration», Proceedings of the 19th Large Installation System Administration Conference (LISA'05), pp. 225–236, San Diego, CA.

Описание механизма управления службами (Service Management Facility, SMF) в ОС Solaris, обеспечивающего основу для запуска и мониторинга администрируемых процессов, а также восстановления служб после сбоев.

Adobe Systems Inc. 1999. PostScript Language Reference Manual, Third Edition. Addison-Wesley, Reading, MA.

Справочное руководство по языку PostScript.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. The AWK Programming Language. Addison-Wesley, Reading, MA.

Замечательная книга по языку программирования awk. Версия awk, описанная в книге, иногда называется nawk («new awk»).

Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. «Building a Transaction Processing System on UNIX Systems», Proceedings of the 1989 USENIX Transaction Processing Workshop, vol. May, pp. 13–22, Pittsburgh, PA.

Описание системы обработки запросов AT&T Tuxedo.

Arnold, J. Q. 1986. «Shared Libraries on UNIX System V», Proceedings of the 1986 Summer USENIX Conference, pp. 395–404, Atlanta, GA.

Описание реализации разделяемых библиотек в SVR3.

AT&T. 1989. System V Interface Definition, Third Edition. Addison-Wesley, Reading, MA.

Этот четырехтомник описывает интерфейсы исходного кода System V и ее поведение во время выполнения. Третья редакция соответствует SVR4. Пятый том содержит обновленные версии команд и функций из томов 1–4, был издан в 1991 году. В настоящее время тираж распродан.

AT&T. 1990a. UNIX Research System Programmer's Manual, Tenth Edition, Volume I. Saunders College Publishing, Fort Worth, TX.

Версия «Руководства программиста UNIX» для 10-й редакции Research UNIX System (V10). В этой книге содержатся традиционные для UNIX страницы справочного руководства (разделы 1–9).

AT&T. 1990b. UNIX Research System Papers, Tenth Edition, Volume II. Saunders College Publishing, Fort Worth, TX.

Том II руководства программиста для UNIX Version 10 (V10) содержит 40 статей, описывающих различные аспекты системы.

AT&T. 1990c. UNIX System V Release 4 BSD/XENIX Compatibility Guide. Prentice Hall, Englewood Cliffs, NJ.

Содержит страницы справочного руководства, описывающие библиотеку совместимости.

AT&T. 1990d. UNIX System V Release 4 Programmer's Guide: STREAMS. Prentice Hall, Englewood Cliffs, NJ.

Описывает систему STREAMS в SVR4.

AT&T. 1990e. UNIX System V Release 4 Programmer's Reference Manual. Prentice Hall, Englewood Cliffs, NJ.

Справочное руководство программиста к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 2 (системные вызовы), 3 (подпрограммы), 4 (форматы файлов) и 5 (различные возможности).

AT&T. 1991. UNIX System V Release 4 System Administrator's Reference Manual. Prentice Hall, Englewood Cliffs, NJ.

Справочное руководство системного администратора к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 4 (форматы файлов), 5 (различные возможности) и 7 (специальные файлы).

Bach, M. J. 1986. The Design of the UNIX Operating System. Prentice Hall, Englewood Cliffs, NJ.

Книга подробно описывает архитектуру и реализацию операционной системы UNIX. Хотя исходный код UNIX и не приводится (поскольку в то время он был собственностью AT&T), все же в книге представлено большое количество алгоритмов и структур данных, используемых ядром UNIX. Эта книга описывает SVR2.

Bolsky, M. I., and Korn, D. G. 1995. The New KornShell Command and Programming Language, Second Edition. Prentice Hall, Englewood Cliffs, NJ.

Книга описывает работу с командной оболочкой Korn shell — как с командным интерпретатором, так и с языком программирования.

Bovet, D. P. и Cesati, M. Understanding the Linux Kernel, Third Edition. O'Reilly Media, Sebastopol, CA.¹

Chen, D., Barkley, R. E., and Lee, T. P. 1990. «Insuring Improved VM Performance: Some No-Fault Policies», Proceedings of the 1990 Winter USENIX Conference, pp. 11–22, Washington, D.C.

Описывает изменения, внесенные в реализацию виртуальной памяти SVR4 для повышения производительности (главным образом функций `fork` и `exec`).

Comer, D. E. 1979. «The Ubiquitous B-Tree», ACM Computing Surveys, vol. 11, no. 2, pp. 121–137 (June).

Хорошая подробная статья о двоичных деревьях.

Date, C. J. 2004. An Introduction to Database Systems, Eighth Edition. Addison-Wesley, Boston, MA.²

Обширный обзор систем управления базами данных.

Evans, J. 2006. «A Scalable Concurrent malloc Implementation for FreeBSD», Proceedings of BSDCan.

Описание реализации `jemalloc` – библиотеки для работы с динамической памятью, используемой в ОС FreeBSD.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. «Extendible Hashing – A Fast Access Method for Dynamic Files», ACM Transactions on Databases, vol. 4, no. 3, pp. 315–344 (September).

Статья, описывающая методику расширяемого хеширования.

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. «An Efficient File Hierarchy Walker», Proceeding of the 1989 Summer USENIX Conference, pp. 173–188, Baltimore, MD.

Описывает альтернативную библиотеку функций для обхода дерева каталогов файловой системы.

Gallmeister, B. O. 1995. POSIX.4: Programming for the Real World. O'Reilly & Associates, Sebastopol, CA.

Описывает интерфейсы реального времени стандарта POSIX.

Garfinkel, S., Spafford, G., and Schwartz, A. 2003. Practical UNIX & Internet Security, Third Edition. O'Reilly & Associates, Sebastopol, CA.

Подробная книга о безопасности операционной системы UNIX.

¹ Даниель Бовет, М. Чезати. Ядро Linux. 3-е изд. ВНВ-Санкт-Петербург, 2007, ISBN: 978-5-94157-957-0.

² К. Дж. Дейт. Введение в системы баз данных. 8-е изд. Вильямс, 2006, ISBN: 5-8459-0788-8.

Ghemawat, S., and Menage, P. 2005. «TCMalloc: Thread-Caching Malloc».

Краткое описание диспетчера памяти TCMalloc, разработанного в компании Google. Доступно также в электронном виде по адресу <http://goog-perf-tools.sourceforge.net/doc/tcmalloc.html>.

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. «Shared Libraries in SunOS», Proceedings of the 1987 Summer USENIX Conference, pp. 131–145, Phoenix, AZ.

Описывает реализацию разделяемых библиотек в SunOS.

Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. «Virtual Memory Architecture in SunOS», Proceedings of the 1987 Summer USENIX Conference, pp. 81–94, Phoenix, AZ.

Описывает первоначальную реализацию функции `mmap` и проблемы, связанные с архитектурой виртуальной памяти.

Goodheart, B. 1991. UNIX Curses Explained. Prentice Hall, Englewood Cliffs, NJ.

Полное руководство по `terminfo` и библиотеке `curses`. В настоящее время тираж распродан.

Hume, A. G. 1988. «A Tale of Two Greps», Software Practice and Experience, vol. 18, no. 11, pp. 1063–1072.

Интересная статья, в которой обсуждается вопрос повышения производительности утилиты `grep`.

IEEE. 1990. Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]. IEEE (Dec.).

Это был первый из стандартов POSIX, и он определял стандартные системные интерфейсы языка программирования С на основе ОС UNIX. Нередко он называется POSIX.1. В настоящее время входит в состав стандарта Single UNIX Specification, опубликованного The Open Group [2008].

ISO. 1999. International Standard ISO/IEC 9899 – Programming Language C. ISO/IEC.

Официальный стандарт языка программирования С и его библиотек. В 2011 году вышла новая версия стандарта, тем не менее все системы, описываемые в этой книге, все еще соответствуют версии стандарта 1999 года.

Электронную версию стандарта в формате PDF можно получить по адресам <http://www.ansi.org> и <http://www.iso.org>.

ISO. 2011. International Standard ISO/IEC 9899, Information Technology – Programming Languages – C. ISO/IEC.

Последняя версия официального стандарта языка программирования С и его библиотек. Эта версия пришла на смену версии 1999 года.

Электронную версию стандарта в формате PDF можно получить по адресам <http://www.ansi.org> и <http://www.iso.org>.

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, NJ.¹

Общее руководство по программированию в UNIX. Книга охватывает множество команд и утилит UNIX, таких как `grep`, `sed`, `awk` и `Bourne shell`.

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, NJ.²

Книга о версии ANSI языка программирования С. Приложение В содержит описание библиотек, определяемых стандартом ANSI.

Kerrisk, M. 2010. *The Linux Programming Interface*. No Starch Press, San Francisco, CA.

Если данная книга показалась вам слишком длинной, то эта книга вдвое больше, но она описывает только программные интерфейсы Linux.

Kleiman, S. R. 1986. «Vnodes: An Architecture for Multiple File System Types in Sun Unix», Proceedings of the 1986 Summer USENIX Conference, pp. 238–247, Atlanta, GA.

Описание оригинальной реализации концепции виртуальных узлов.

Knuth, D. E. 1998. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, Boston, MA.³

Описывает алгоритмы сортировки и поиска.

Korn, D. G., and Vo, K. P. 1991. «SFIO: Safe/Fast String/File IO», Proceedings of the 1991 Summer USENIX Conference, pp. 235–255, Nashville, TN.

Описание альтернативной библиотеки ввода/вывода. Библиотека доступна по адресу <http://www.research.att.com/sw/tools/sfio>.

Krieger, O., Stumm, M., and Unrau, R. 1992. «Exploiting the Advantages of Mapped Files for Stream I/O», Proceedings of the 1992 Winter USENIX Conference, pp. 27–42, San Francisco, CA.

Альтернатива стандартной библиотеке ввода/вывода, основанная на отображаемых файлах.

¹ Б. Керниган, Р. Пайк. UNIX. Программное окружение. Символ-Плюс, 2012, ISBN: 5-93286-029-4.

² Б. Керниган, Д. Ричи. Язык программирования Си». 2-е изд. Вильямс, 2017, ISBN: 978-5-8459-1874-1, 0-13-110362-8.

³ Дональд Э. Кнут. Искусство программирования. Т. 3. Сортировка и поиск». 2-е изд. Вильямс, 2014, ISBN: 978-5-8459-0082-1, 0-201-89685-0.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, Reading, MA.

Книга целиком посвящена операционной системе 4.3BSD. Описывает версию Tahoe 4.3BSD. В настоящее время тираж распродан.

Lennert, D. 1987. «How to Write a UNIX Daemon», *;login;*, vol. 12, no. 4, pp. 17–23 (July/August).

Рассказывает о написании демонов для UNIX.

Libes, D. 1990. «expect: Curing Those Uncontrollable Fits of Interaction», Proceedings of the 1990 Summer USENIX Conference, pp. 183–192, Anaheim, CA.

Описание программы `expect` и ее реализации.

Libes, D. 1991. «expect: Scripts for Controlling Interactive Processes», Computing Systems, vol. 4, no. 2, pp. 99–125 (Spring).

В статье представлены многочисленные сценарии для программы `expect`.

Libes, D. 1994. Exploring Expect. O'Reilly & Associates, Sebastopol, CA.

Книга по работе с программой `expect`.

Lions, J. 1977. A Commentary on the UNIX Operating System. AT&T Bell Laboratories, Murray Hill, NJ.

Описывает исходные тексты 6-й редакции UNIX (6th Edition UNIX System). Доступна только для специалистов и служащих AT&T, хотя некоторые копии просочились за пределы AT&T.

Lions, J. 1996. Lions' Commentary on UNIX 6th Edition. Peer-to-Peer Communications, San Jose, CA.

Общедоступная версия классического издания 1977 года описывает 6-ю редакцию ОС UNIX.

Litwin, W. 1980. «Linear Hashing: A New Tool for File and Table Addressing», Proceedings of the 6th International Conference on Very Large Databases, pp. 212–223, Montreal, Canada.

Статья, описывающая метод линейного хеширования.

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, MA.

Книга целиком посвящена операционной системе 4.4BSD.

McKusick, M. K., and Neville-Neil, G. V. 2005. The Design and Implementation of the FreeBSD Operating System. Addison-Wesley, Boston, MA.¹

Книга целиком посвящена операционной системе FreeBSD 5.2.

McDougall, R., and Mauro, J. 2007. Solaris Internals. Solaris 10 and OpenSolaris Kernel Architecture, Second Edition. Prentice Hall, Upper Saddle River, NJ.

Книга о внутреннем устройстве операционной системы Solaris 10. Охватывает также версию OpenSolaris.

Morris, R., and Thompson, K. 1979. «UNIX Password Security», Communications of the ACM, vol. 22, no. 11, pp. 594–597 (Nov.).

Описание истории развития схемы паролей, используемой в системах UNIX.

Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. UNIX System Administration Handbook, Third Edition. Prentice Hall, Upper Saddle River, NJ.²

Книга, в которой подробно рассматривается администрирование UNIX.

The Open Group. 2008. The Single UNIX Specification, Version 4. The Open Group, Berkshire, UK.

Стандарты POSIX и X/Open, объединенные в один справочник.

Электронную версию в формате HTML можно найти по адресу <http://www.opengroup.org>.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. 1995. «Plan 9 from Bell Labs», Plan 9 Programmer's Manual Volume 2. AT&T, Reading, MA.

Описание операционной системы Plan 9, разработанной в том же подразделении, что и система UNIX.

Plauger, P. J. 1992. The Standard C Library. Prentice Hall, Englewood Cliffs, NJ.

Книга о библиотеке ANSI C. Содержит полную реализацию библиотеки языка C.

Presotto, D. L., and Ritchie, D. M. 1990. «Interprocess Communication in the Ninth Edition UNIX System», Software Practice and Experience, vol. 20, no. S1, pp. S1/3–S1/17 (June).

Описывает возможности IPC, предоставляемые 9-й редакцией UNIX (Ninth Edition Research UNIX System), разработанной в AT&T Bell Laboratories. Функциональные возможности основаны на потоковой системе ввода/вывода и включают дуплексные каналы, передачу файловых дескрипторов между про-

¹ Маршалл К. Маккузик, Джордж В. Невилл-Нил. FreeBSD: архитектура и реализация. КУДИЦ-Образ, 2006, ISBN: 5-9579-0103-2.

² Э. Немет, Г. Снайдер, Т. Р. Хейн. UNIX: Руководство системного администратора. Для профессионалов / 4-е изд. Вильямс, 2017, ISBN: 978-5-8459-2006-5.

цессами и создание уникальных соединений между клиентами и серверами. Копия этой статьи имеется также в [8].

Rago, S. A. 1993. UNIX System V Network Programming. Addison-Wesley, Reading, MA.

Книга описывает программирование в сетевом окружении UNIX System V Release 4, основанное на использовании механизмов STREAMS.

Raymond, E. S., ed. 1996. The New Hacker's Dictionary, Third Edition. MIT Press, Cambridge, MA.

Определения множества терминов из лексикона хакера.

Salus, P. H. 1994. A Quarter Century of UNIX. AddisonWesley, Reading, MA.

История развития UNIX с 1969 по 1994 год.

Seltzer, M., and Olson, M. 1992. «LIBTP: Portable Modular Transactions for UNIX», Proceedings of the 1992 Winter USENIX Conference, pp. 9–25, San Francisco, CA.

Модификация библиотеки `db(3)` из 4.4BSD, которая реализует механизм транзакций.

Seltzer, M., and Yigit, O. 1991. «A New Hashing Package for UNIX», Proceedings of the 1991 Winter USENIX Conference, pp. 173–184, Dallas, TX.

Описание библиотеки `dbm(3)` и ее реализации, а также новейшего пакета хеширования.

Singh, A. 2006. Mac OS X Internals: A Systems Approach. Addison-Wesley, Upper Saddle River, NJ.

Примерно 1600 страниц с описанием архитектуры операционной системы Mac OS X.

Stevens, W. R. 1990. UNIX Network Programming. Prentice Hall, Englewood Cliffs, NJ.¹

Книга подробно описывает программирование сетевых приложений для UNIX. Первое издание очень сильно отличается по своему содержанию от более поздних изданий.

Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. UNIX Network Programming, Volume 1, Third Edition. Addison-Wesley, Boston, MA.²

Подробно описывается программирование сетевых приложений для UNIX. Переработана и разбита на два тома во втором издании, дополнена в третьем.

¹ Стивенс У. UNIX: Разработка сетевых приложений. Питер, 2003, ISBN: 5-318-00535-7.

² Стивенс У., Феннер Б., Рудофф Э. UNIX: Разработка сетевых приложений. Питер, 2006, ISBN: 5-94723-991-4.

Stonebraker, M. R. 1981. «Operating System Support for Database Management», Communications of the ACM, vol. 24, no. 7, pp. 412–418 (July).

Описывает службы операционной системы и их влияние на работу базы данных.

Strang, J. 1986. Programming with curses. O'Reilly & Associates, Sebastopol, CA.

Книга о версии библиотеки `curses` из Беркли.

Strang, J., Mui, L., and O'Reilly, T. 1988. termcap & terminfo, Third Edition. O'Reilly & Associates, Sebastopol, CA.

Книга посвящена `termcap` и `terminfo`.

Sun Microsystems. 2005. STREAMS Programming Guide. Sun Microsystems, Santa Clara, CA.

Описывает STREAMS программирование на платформе Solaris.

Thompson, K. 1978. «UNIX Implementation», The Bell System Technical Journal, vol. 57, no. 6, pp. 1931–1946 (July–Aug.).

Описывает некоторые аспекты реализации Version 7.

Vo, Kiem-Phong. 1996. «Vmalloc: A General and Efficient Memory Allocator», Software Practice and Experience, vol. 26, no. 3, pp. 357–374.

Описывает гибкий диспетчер динамической памяти.

Wei, J., and Pu, C. 2005. «TOCTTOU Vulnerabilities in UNIX_Style File Systems: An Anatomical Study», Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05), pp. 155–167, San Francisco, CA.

Описывает недостатки TOCTTOU в интерфейсе файловой системы UNIX.

Weinberger, P. J. 1982. «Making UNIX Operating Systems Safe for Databases», The Bell System Technical Journal, vol. 61, no. 9, pp. 2407–2422 (Nov.).

Описывает некоторые проблемы реализации баз данных в ранних версиях UNIX.

Weinstock, C. B., and Wulf, W. A. 1988. «Quick Fit: An Efficient Algorithm for Heap Storage Allocation», SIGPLAN Notices, vol. 23, no. 10, pp. 141–148.

Описывает алгоритм управления динамической памятью, который подходит для широкого круга приложений.

Williams, T. 1989. «Session Management in System V Release 4», Proceedings of the 1989 Winter USENIX Conference, pp. 365–375, San Diego, CA.

Описывает архитектуру сеанса в SVR4, на которой были основаны интерфейсы POSIX.1. Рассматриваются группы процессов, управление заданиями, управляющие терминалы и вопросы безопасности существующих механизмов.

X/Open. 1989. X/Open Portability Guide. Prentice Hall, Englewood Cliffs, NJ.

Издание состоит из семи томов, которые охватывают команды и утилиты (том 1), системные интерфейсы и заголовочные файлы (том 2), дополнительные определения (том 3), языки программирования (том 4), управление данными (том 5), управление окнами (том 6), сетевые службы (том 7). Хотя это издание в настоящее время отсутствует в продаже, его заменяет Single UNIX Specification [Open Group, 2008].