

Изучаем ядро *Linux* и библиотеки *C*

2-е издание

Linux

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ



O'REILLY®

Роберт Лав

ПИТЕР®

Robert Love

Linux

System Programming

SECOND EDITION

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Роберт Лав

Linux

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

2-е издание



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара • Новосибирск
Киев • Харьков • Минск

2014

ББК 32.973.2-018.2
УДК 004.451
Л13

Лав Р.

Л13 Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-00747-4

Роберт Лав стоит у истоков создания операционной системы Linux. Он внес существенный вклад в создание ядра Linux и настольной среды GNOME.

Эта книга представляет собой руководство по системному программированию для Linux, справочник по системным вызовам Linux, а также подробный рассказ о том, как писать более быстрый и умный код. Роберт Лав четко разграничивает стандартные функции POSIX и специальные службы, которые предлагаются лишь в Linux. Во втором издании вы изучите эту операционную систему как с теоретической, так и с прикладной точки зрения.

12+ (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 O'Reilly. 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 978-1449339531 англ.

Authorized Russian translation of the English edition of Linux System Programming: Talking Directly to the Kernel and C Library 2nd edition (ISBN 9781449339531) © 2013 Robert Love.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
© Перевод на русский язык ООО Издательство «Питер», 2014
© Издание на русском языке, оформление ООО Издательство «Питер», 2014

ISBN 978-5-496-00747-4

Краткое содержание

Предисловие	17
Вступление	19
Благодарности	25
От издательства	26
Глава 1. Введение и основополагающие концепции	27
Глава 2. Файловый ввод-вывод	54
Глава 3. Буферизованный ввод-вывод	99
Глава 4. Расширенный файловый ввод-вывод	125
Глава 5. Управление процессами	171
Глава 6. Расширенное управление процессами	210
Глава 7. Поточность	245
Глава 8. Управление файлами и каталогами	275
Глава 9. Управление памятью	324
Глава 10. Сигналы	365
Глава 11. Время	394
Приложение А. Расширения GCC для языка C	427
Приложение Б. Библиография	441

Оглавление

Предисловие	17
Вступление	19
Целевая аудитория и необходимые предпосылки	19
Краткое содержание	20
Версии, рассмотренные в книге	21
Условные обозначения	22
Работа с примерами кода	24
Благодарности	25
От издательства	26
Глава 1. Введение и основополагающие концепции	27
Системное программирование	27
Зачем изучать системное программирование	28
Краеугольные камни системного программирования	29
Системные вызовы	29
Библиотека C	30
Компилятор C	31
API и ABI	31
API	32
ABI	32
Стандарты	33
История POSIX и SUS	34
Стандарты языка C	34
Linux и стандарты	35
Стандарты и эта книга	36
Концепции программирования в Linux	37
Файлы и файловая система	37
Процессы	45
Пользователи и группы	47
Права доступа	48

Сигналы	49
Межпроцессное взаимодействие	50
Заголовки	50
Обработка ошибок	50
Добро пожаловать в системное программирование	53
Глава 2. Файловый ввод-вывод	54
Открытие файлов	55
Системный вызов <code>open()</code>	55
Владельцы новых файлов	58
Права доступа новых файлов	58
Функция <code>creat()</code>	60
Возвращаемые значения и коды ошибок	61
Считывание с помощью <code>read()</code>	61
Возвращаемые значения	62
Считывание всех байтов	63
Неблокирующее считывание	64
Другие значения ошибок	64
Ограничения размера для <code>read()</code>	65
Запись с помощью <code>write()</code>	65
Случаи частичной записи	66
Режим дозаписи	67
Неблокирующая запись	67
Другие коды ошибок	68
Ограничения размера при использовании <code>write()</code>	68
Поведение <code>write()</code>	68
Синхронизированный ввод-вывод	70
<code>fsync()</code> и <code>fdatasync()</code>	70
<code>sync()</code>	72
Флаг <code>O_SYNC</code>	73
Флаги <code>O_DSYNC</code> и <code>O_RSYNC</code>	74
Непосредственный ввод-вывод	74
Закрытие файлов	75
Значения ошибок	76
Позиционирование с помощью <code>lseek()</code>	76
Поиск с выходом за пределы файла	78
Ограничения	79
Позиционное чтение и запись	79
Усечение файлов	80

Мультиплексный ввод-вывод	81
select()	83
Системный вызов poll()	88
Сравнение poll() и select()	92
Внутренняя организация ядра	93
Виртуальная файловая система	93
Страничный кэш	94
Страничная отложенная запись	96
Резюме	98
Глава 3. Буферизованный ввод-вывод	99
Ввод-вывод с пользовательским буфером	100
Стандартный ввод-вывод.	102
Открытие файлов	103
Открытие потока данных с помощью файлового дескриптора	104
Закрытие потоков данных	105
Считывание из потока данных	106
Считывание одного символа в момент времени	106
Считывание целой строки	107
Считывание двоичных данных	109
Запись в поток данных	110
Запись отдельного символа	110
Запись строки символов.	111
Запись двоичных данных	111
Пример программы, в которой используется буферизованный ввод-вывод	112
Позиционирование в потоке данных	113
Сброс потока данных.	115
Ошибки и конец файла	116
Получение ассоциированного файлового дескриптора.	117
Управление буферизацией	117
Безопасность программных потоков.	119
Блокировка файлов вручную	120
Неблокируемые потоковые операции.	122
Недостатки стандартного ввода-вывода.	123
Резюме	123
Глава 4. Расширенный файловый ввод-вывод	125
Фрагментированный ввод-вывод	125
Системные вызовы readv() и writev()	126

Возвращаемые значения	127
Пример использования <code>writew()</code>	128
Пример использования <code>readv()</code>	129
Реализация	130
Опрос событий	131
Создание нового экземпляра <code>epoll</code>	131
Управление <code>epoll</code>	132
Ожидание событий с помощью <code>epoll</code>	135
Сравнение событий, запускаемых по фронту и по уровню сигнала	136
Отображение файлов в память	137
<code>mmap()</code>	137
Системный вызов <code>mmap()</code>	142
Пример отображения	143
Преимущества <code>mmap()</code>	144
Недостатки <code>mmap()</code>	145
Изменение размеров отображения	145
Изменение защиты отображения	147
Синхронизация файла с помощью отображения	147
Извещения об отображении	149
Извещения об обычном файловом вводе-выводе	151
Системный вызов <code>posix_fadvise()</code>	151
Системный вызов <code>readahead()</code>	153
Рекомендации почти ничего не стоят	153
Синхронизированные, синхронные и асинхронные операции	154
Планировщики и производительность ввода-вывода	156
Адресация диска	156
Жизненный цикл планировщика ввода-вывода	157
Помощь при считывании	158
Выбор и настройка планировщика ввода-вывода	162
Оптимизация производительности ввода-вывода	163
Резюме	170
Глава 5. Управление процессами	171
Программы, процессы и потоки	171
Идентификатор процесса	172
Выделение идентификатора процесса	173
Иерархия процессов	173
<code>pid_t</code>	174
Получение идентификаторов процесса и родительского процесса	174

Запуск нового процесса	174
Семейство вызовов <code>exec</code>	175
Системные вызовы <code>fork()</code>	179
Завершение процесса	182
Другие способы завершения	183
<code>atexit()</code>	184
<code>on_exit()</code>	185
<code>SIGCHLD</code>	185
Ожидание завершенных дочерних процессов	185
Ожидание определенного процесса	188
Еще больше гибкости при ожидании	190
На сцену выходит BSD: <code>wait3()</code> и <code>wait4()</code>	192
Запуск и ожидание нового процесса	193
Зомби	195
Пользователи и группы	196
Реальные, действительные и сохраненные идентификаторы пользователя и группы	197
Изменение реального или сохраненного идентификатора пользователя или группы	198
Изменение действительного идентификатора пользователя или группы	199
Изменение идентификаторов пользователя и группы согласно стилю BSD	199
Изменение идентификаторов пользователя и группы согласно стилю HP-UX	200
Действия с предпочтительными идентификаторами пользователя или группы	201
Поддержка сохраненных пользовательских идентификаторов	201
Получение идентификаторов пользователя и группы	201
Сессии и группы процессов	202
Системные вызовы сессий	204
Системные вызовы групп процессов	205
Устаревшие функции для группы процессов	206
Демоны	207
Резюме	209
Глава 6. Расширенное управление процессами	210
Планирование процессов	210
Кванты времени	211
Процессы ввода-вывода против ограниченных процессором	212

Приоритетное планирование	213
Completely Fair Scheduler	213
Высвобождение ресурсов процессора	215
Приоритеты процессов	216
nice()	217
getpriority() и setpriority()	218
Приоритеты ввода-вывода	219
Привязка процессов к процессору	220
Системы реального времени	223
Мягкие и жесткие системы реального времени	224
Задержка, колебание и временное ограничение	225
Поддержка реального времени в Linux	226
Политики планирования и приоритеты в Linux	226
Установка параметров планирования	230
sched_rr_get_interval()	233
Предосторожности при работе с процессами реального времени	235
Детерминизм	235
Лимиты ресурсов	238
Лимиты по умолчанию	242
Установка и проверка лимитов	243
Коды ошибок	244
Глава 7. Поточность	245
Бинарные модули, процессы и потоки	245
Многопоточность	246
Издержки многопоточности	248
Альтернативы многопоточности	248
Поточные модели	249
Поточность на уровне пользователя	249
Комбинированная поточность	250
Сопрограммы и фибры	251
Шаблоны поточности	251
Поток на соединение	251
Поток, управляемый событием	252
Конкурентность, параллелизм и гонки	253
Синхронизация	256
Мьютексы	257
Взаимные блокировки	258
P-потоки	260

Реализация поточности в Linux	261
API для работы с P-потокami	261
Связывание P-потокoв	262
Создание потокoв	262
Идентификаторы потокoв	264
Завершение потокoв	265
Самозавершение	265
Завершение других потокoв	266
Присоединение и отсоединение потокoв	268
Присоединение потокoв	268
Отсоединение потокoв	269
Пример поточности	269
Мьютексы P-потокoв	270
Инициализация мьютексов	270
Запирание мьютексов	271
Отпирание мьютексов	271
Пример использования мьютексов	272
Дальнейшее изучение	273
Глава 8. Управление файлами и каталогами	275
Файлы и их метаданные	275
Семейство stat	276
Разрешения	280
Владение	281
Расширенные атрибуты	284
Перечисление расширенных атрибутов файла	289
Каталоги	292
Текущий рабочий каталог	293
Создание каталогов	298
Удаление каталогов	299
Чтение содержимого каталога	300
Ссылки	303
Жесткие ссылки	304
Символические ссылки	305
Удаление ссылки	307
Копирование и перемещение файлов	308
Копирование	308
Перемещение	309
Узлы устройств	311

Специальные узлы устройств	311
Генератор случайных чисел	312
Внеполосное взаимодействие	312
Отслеживание файловых событий	314
Инициализация inotify	315
Стражи	316
События inotify	317
Расширенные события отслеживания	321
Удаление стража inotify	321
Получение размера очереди событий	322
Уничтожение экземпляра inotify	323
Глава 9. Управление памятью	324
Адресное пространство процесса	324
Страницы и их подкачка	324
Области памяти	326
Выделение динамической памяти	327
Выделение массивов	329
Изменение размера выделенных областей	331
Освобождение динамической памяти	332
Выравнивание	335
Управление сегментом данных	339
Анонимные отображения в памяти	340
Создание анонимных отображений в памяти	342
Отображение /dev/zero	344
Расширенное выделение памяти	345
Отладка при операциях выделения памяти	348
Выделение памяти на основе стека	349
Дублирование строк в стеке	351
Массивы переменной длины	352
Выбор механизма выделения памяти	353
Управление памятью	354
Установка байтов	354
Сравнение байтов	355
Перемещение байтов	356
Поиск байтов	357
Перещелкивание байтов	358
Блокировка памяти	358
Блокировка части адресного пространства	359

Блокировка всего адресного пространства	360
Разблокировка памяти	361
Лимиты блокировки	362
Находится ли страница в физической памяти	362
Уступающее выделение памяти	363
Глава 10. Сигналы	365
Концепции, связанные с сигналами	366
Идентификаторы сигналов	366
Сигналы, поддерживаемые в Linux	367
Основы управления сигналами	372
Ожидание любого сигнала	373
Примеры	374
Выполнение и наследование	376
Сопоставление номеров сигналов и строк	377
Отправка сигнала	378
Права доступа	378
Примеры	379
Отправка сигнала самому себе	379
Отправка сигнала целой группе процессов	380
Реентерабельность	380
Наборы сигналов	382
Блокировка сигналов	384
Получение сигналов, ожидающих обработки	385
Ожидание набора сигналов	385
Расширенное управление сигналами	385
Структура <code>siginfo_t</code>	388
Удивительный мир <code>si_code</code>	389
Отправка сигнала с полезной нагрузкой	391
Изъян в UNIX?	393
Глава 11. Время	394
Структуры данных, связанные с представлением времени	397
Оригинальное представление	397
А теперь — с микросекундной точностью!	397
И еще лучше: наносекундная точность	398
Разбиение времени	398
Тип для процессного времени	399

Часы POSIX	400
Получение текущего времени суток	401
Более удобный интерфейс	402
Продвинутый интерфейс	403
Получение процессного времени	404
Установка текущего времени суток	405
Установка времени с заданной точностью	405
Продвинутый интерфейс для установки времени	406
Эксперименты с временем	406
Настройка системных часов	408
Засыпание и ожидание	411
Засыпание с микросекундной точностью	412
Засыпание с наносекундной точностью	413
Продвинутая работа со спящим режимом	415
Переносимый способ засыпания	416
Превышение пределов	417
Альтернативы засыпанию	418
Таймеры	418
Простые варианты сигнализации	418
Интервальные таймеры	419
Функции для расширенной работы с таймерами	421
Приложение А. Расширения GCC для языка C	427
GNU C	427
Встраиваемые функции	428
Подавление встраивания	429
Чистые функции	429
Постоянные функции	430
Невозвращаемые функции	430
Функции, выделяющие память	430
Принудительная проверка возвращаемого значения вызывающей стороной	431
Как пометить функцию как устаревшую	431
Как пометить функцию как используемую	431
Как пометить функции или параметры как неиспользуемые	432
Упаковка структуры	432
Увеличение границы выравнивания переменной	433
Помещение глобальных переменных в регистр	434

Аннотирование ветвей.	434
Получение типа выражения.	435
Получение границы выравнивания типа	436
Смещение члена внутри структуры	437
Получение возвращаемого адреса функции	437
Диапазоны оператора case	438
Арифметика указателей типа void и указателей на функции.	438
Более переносимо и красиво	439
Приложение Б. Библиография	441
Книги по языку программирования C.	441
Книги по программированию в Linux	442
Книги, посвященные ядру Linux.	443
Книги об организации операционных систем	443

Предисловие

Есть старая шутка, что разработчики ядра Linux, рассердившись, могут в сердцах бросить: «Все ваше пользовательское пространство — просто тестовая нагрузка для ядра!»

Говоря такое, разработчики просто пытаются умыть руки и снять с себя ответственность за любые случаи, в которых ядру не удастся обеспечивать максимально эффективную работу пользовательского кода. По мнению создателей ядра, программистам, работающим в пользовательском пространстве, следует просто поосторониться и исправлять собственный код, ведь проблемы могут объясняться чем угодно, но не недостатками ядра.

Уже более трех лет назад один из ведущих разработчиков ядра Linux прочитал лекцию под названием «Почему пользовательское пространство — ерунда». Целью лекции было доказать, что обычно корень проблем лежит не в ядре. Выступив перед переполненной аудиторией, этот специалист привел примеры отвратительного пользовательского кода, на который практически всем пользователям Linux приходится полагаться ежедневно. Другие разработчики ядра создали специальные инструменты, демонстрирующие, как сильно пользовательские программы могут злоупотреблять оборудованием и растрачивать заряд ничего не подозревающего ноутбука.

Однако если пользовательский код и может быть банальной «тестовой нагрузкой», которая порой возмущает разработчиков ядра, необходимо признать, что и сами разработчики ядра ежедневно вынуждены работать с тем самым пользовательским кодом. Если бы его не было, ядро годилось бы для выполнения единственной функции — вывода на экран перемежающихся серий АВАВАВ.

В настоящее время Linux является наиболее гибкой и мощной из когда-либо созданных операционных систем. Linux с успехом применяется как в крошечных сотовых телефонах и внедренных устройствах, так и на 90 % из 500 мощнейших суперкомпьютеров в мире. Ни одна другая операционная система не обладает такими превосходными возможностями масштабирования и адаптации к нюансам разнообразных типов оборудования и экосистем.

Однако код, работающий в пользовательском пространстве Linux, способен взаимодействовать со всеми этими платформами не менее эффективно, чем ядро, обеспечивая функционирование реальных прикладных приложений и утилит, с которыми и приходится работать нам с вами.

В своей книге Роберт Лав (Robert Love) поставил перед собой титаническую задачу — рассказать читателю практически обо всех системных вызовах, происходящих в Linux. В результате получился настоящий фолиант, позволяющий

вам полностью понять, как работает Linux «с пользовательской точки зрения» и как максимально эффективно использовать всю мощь этой операционной системы.

Прочтя данную книгу, вы научитесь писать код, который будет работать во всех дистрибутивах Linux на самом разном оборудовании. Книга поможет вам осознать, как функционирует Linux и как наиболее эффективно задействовать ее гибкость.

В конечном итоге эта книга научит вас писать код, который никто уже не назовет ерундой, а это самое главное.

Грег Кроах-Хартман (Greg Kroah-Hartman)

Вступление

Данная книга рассказывает о системном программировании в Linux. *Системное программирование* — это практика написания *системного ПО*, низкоуровневый код которого взаимодействует непосредственно с ядром и основными системными библиотеками. Иными словами, речь далее пойдет в основном о системных вызовах Linux и низкоуровневых функциях, в частности тех, которые определены в библиотеке C.

Есть немало пособий, посвященных системному программированию для UNIX-систем, но вы почти не найдете таких, которые рассматривают данную тему достаточно подробно и фокусируются именно на Linux. Еще меньше подобных книг учитывают новейшие релизы Linux и продвинутые интерфейсы, ориентированные исключительно на Linux. Эта книга не только лишена всех перечисленных недостатков, но и обладает важным достоинством: дело в том, что я написал массу кода для Linux, как для ядра, так и для системных программ, расположенных непосредственно «над ядром». На самом деле я реализовал на практике ряд системных вызовов и других функций, описанных далее. Соответственно книга содержит богатый материал, рассказывая не только о том, как *должны* работать системные интерфейсы, но и о том, как они *действительно* работают и как вы сможете использовать их с максимальной эффективностью. Таким образом, данная книга одновременно является и руководством по системному программированию для Linux, и справочным пособием, описывающим системные вызовы Linux, и подробным повествованием о том, как создавать более интеллектуальный и быстрый код. Текст написан простым, доступным языком. Независимо от того, является ли создание системного кода вашей основной работой, эта книга научит полезным приемам, которые помогут вам стать по-настоящему высокопрофессиональным программистом.

Целевая аудитория и необходимые предпосылки

Пособие предназначается для читателей, знакомых с программированием на языке C и с применяемой в Linux экосистемой программирования. Не обязательно быть экспертом в этих темах, но в них нужно как минимум ориентироваться. Если вам не приходилось работать с текстовыми редакторами для UNIX — наиболее известными и хорошо себя зарекомендовавшими являются Emacs и vim, — поэкспериментируйте с ними. Кроме того, следует в общих чертах представлять работу с gcc, gdb,

make и др. Существует еще множество инструментов и практикумов по программированию для Linux; в приложении Б в конце перечислены некоторые полезные источники.

Кроме того, я ожидаю от читателя определенных знаний в области системного программирования для Linux и UNIX. Эта книга начинается с самых основ, ее темы постепенно усложняются вплоть до обсуждения наиболее продвинутых интерфейсов и приемов оптимизации. Надеюсь, пособие понравится читателям с самыми разными уровнями подготовки, научит их чему-то ценному и новому. Пока писал книгу, я сам узнал немало интересного.

У меня были определенные предположения об убеждениях и мотивации читателя. Инженеры, желающие (более качественно) программировать на системном уровне, являются основной целевой аудиторией, но книга будет интересна и программистам, которые специализируются на высокоуровневом коде и желают приобрести более солидные базовые знания. Любознательным хакерам пособие также понравится, утолит их жажду нового. Книга задумывалась так, чтобы заинтересовать большинство программистов.

В любом случае, независимо от ваших мотивов, надеюсь, что чтение окажется для вас интересным!

Краткое содержание

Книга разделена на 11 глав и 2 приложения.

- **Глава 1. Введение и основополагающие концепции.** Она — введение в проблему. Здесь делается обзор Linux, системного программирования, ядра, библиотеки C и компилятора C. Главу следует изучить даже самым опытным пользователям.
- **Глава 2. Файловый ввод-вывод.** Тут дается вводная информация о файлах — наиболее важной абстракции в экосистеме UNIX, а также файловом вводе/выводе, который является основой процесса программирования для Linux. Подробно рассматриваются считывание информации из файлов и запись информации в них, а также другие базовые операции файлового ввода-вывода. Итоговая часть главы рассказывает, как ядро Linux внедряет (реализует) концепцию файлов и управляет ими.
- **Глава 3. Буферизованный ввод-вывод.** Здесь обсуждается проблема, связанная с базовыми интерфейсами ввода-вывода — управление размером буфера, — и рассказывается о буферизованном вводе-выводе вообще, а также стандартном вводе-выводе в частности как о возможных решениях.
- **Глава 4. Расширенный файловый ввод-вывод.** Завершает трио тем о вводе-выводе и рассказывает о продвинутых интерфейсах ввода-вывода, способах распределения памяти и методах оптимизации. В заключение главы мы поговорим о том, как избегать подвода головок, и о роли планировщика ввода-вывода, работающего в ядре Linux.

- **Глава 5. Управление процессами.** В ней читатель познакомится со второй по важности абстракцией UNIX — *процессом* — и семейством системных вызовов, предназначенных для базового управления процессами, в частности древним феноменом ветвления (*fork*).
- **Глава 6. Расширенное управление процессами.** Здесь продолжается обсуждение процессов. Глава начинается с рассмотрения продвинутых способов управления процессами, в частности управления в реальном времени.
- **Глава 7. Поточность.** Здесь обсуждаются потоки и многопоточное программирование. Глава посвящена в основном высокоуровневым концепциям проектирования. В частности, в ней читатель познакомится с API многопоточности POSIX, который называется Pthreads.
- **Глава 8. Управление файлами и каталогами.** Тут обсуждаются вопросы создания, перемещения, копирования, удаления и других приемов, связанных с управлением файлами и каталогами.
- **Глава 9. Управление памятью.** В ней рассказывается об управлении памятью. Глава начинается с ознакомления с основными концепциями UNIX, связанными с памятью, в частности с адресным пространством процесса и подкачкой страниц. Далее мы поговорим об интерфейсах, к которым можно обращаться для получения памяти и через которые можно возвращать память обратно в ядро. В заключение мы ознакомимся с продвинутыми интерфейсами, предназначенными для управления памятью.
- **Глава 10. Сигналы.** Здесь рассматриваются сигналы. Глава начинается с обсуждения природы сигналов и их роли в системе UNIX. Затем описываются сигнальные интерфейсы, от самых простых к наиболее сложным.
- **Глава 11. Время.** Она посвящена обсуждению времени, спящего режима и управления часами. Здесь рассмотрены все базовые интерфейсы вплоть до часов POSIX и таймеров высокого разрешения.
- **Приложение А.** В нем рассматриваются многие языковые расширения, предоставляемые gcc и GNU C, в частности атрибуты, позволяющие сделать функцию константной, чистой или интринсичной.
- **Приложение Б.** Здесь собрана библиография работ, которые я рекомендую для дальнейшего изучения. Они служат не только важным дополнением к изложенному в книге материалу, но и рассказывают об обязательных темах, не затронутых в моей работе.

Версии, рассмотренные в книге

Системный интерфейс Linux определяется как бинарный (двоичный) интерфейс приложений и интерфейс программирования приложений, предоставляемый благодаря взаимодействию трех сущностей: ядра Linux (центра операционной системы), библиотеки GNU C (glibc) и компилятора GNU C (gcc — в настоящее время

он официально называется набором компиляторов для GNU и применяется для работы с различными языками, но нас интересует только C). В этой книге рассмотрен системный интерфейс, определенный с применением версии ядра Linux 3.9, версий glibc 2.17 и gcc 4.8. Более новые интерфейсы этих компонентов должны и далее соответствовать интерфейсам и поведением, документированным в данной книге. Аналогично многие интерфейсы, о которых нам предстоит поговорить, давно используются в составе Linux и поэтому обладают обратной совместимостью с более ранними версиями ядра, glibc и gcc.

Если любую развивающуюся операционную систему можно сравнить со скользящей мишенью, то Linux — это просто гепард в прыжке. Прогресс измеряется днями, а не годами, частые релизы ядра и других компонентов постоянно меняют и правила игры, и само игровое поле. Ни в одной книге не удалось бы сделать достаточно долговечный слепок такого динамичного явления.

Тем не менее экосистема, в которой протекает системное программирование, *очень стабильна*. Разработчикам ядра приходится проявлять недюжинную избирательность, чтобы не повредить системные вызовы, разработчики glibc крайне высоко ценят прямую и обратную совместимость, а цепочка инструментов Linux (набор программ для написания кода) создает взаимно совместимый код в различных версиях. Следовательно, при всей динамичности Linux системное программирование для этой операционной системы остается стабильным. Книга, представляющая собой «мгновенный снимок» системы, особенно на современном этапе развития Linux, обладает исключительной фактической долговечностью. Я пытаюсь сказать: не беспокойтесь, что системные интерфейсы вскоре изменятся, и *смело покупайте эту книгу!*

Условные обозначения

В книге применяются следующие условные обозначения.

Курсивный шрифт

Им обозначаются новые термины и понятия.

Шрифт для названий

Используется для обозначения URL, адресов электронной почты, а также сочетаний клавиш и названий элементов интерфейса.

Шрифт для команд

Применяется для обозначения программных элементов — переменных и названий функций, типов данных, переменных окружения, операторов и ключевых слов и т. д.

Шрифт для листингов

Используется в листингах программного кода.

ПРИМЕЧАНИЕ

Данная врезка содержит совет, замечание практического характера или общее замечание.

ВНИМАНИЕ

Такая врезка содержит какое-либо предостережение.

Большинство примеров кода в книге представляют собой краткие фрагменты, которые легко можно использовать повторно. Они выглядят примерно так:

```
while (1) {
    int ret;

    ret = fork ();
    if(ret== -1)
        perror("fork");
}
```

Пришлось проделать огромную работу, чтобы фрагменты кода получились столь краткими и при этом не утратили практической ценности. Для работы вам не потребуется никаких специальных заголовочных файлов, переполненных безумными макросами и сокращениями, о смысле которых остается только догадываться. Я не писал нескольких гигантских программ, а ограничился многочисленными, но сжатыми примерами, которые, будучи практическими и наглядными, сделаны максимально компактными и ясными. Надеюсь, при первом прочтении книги они послужат вам удобным пособием, а на последующих этапах работы станут хорошим справочным материалом.

Почти все примеры являются самодостаточными. Это означает, что вы можете просто скопировать их в текстовый редактор и смело использовать на практике. Если не указано иное, сборка всех фрагментов кода должна происходить без применения каких-либо специальных индикаторов компилятора (в отдельных случаях понадобится связь со специальной библиотекой). Рекомендую следующую команду для компиляции файла исходников:

```
$ gcc -Wall -Wextra -O2 -g -o snippet snippet.c
```

Она собирает файл исходного кода `snippet.c` в исполняемый бинарный файл `snippet`, обеспечивая выполнение многих предупреждающих проверок, значительных, но разумных оптимизаций, а также отладку. Код из книги должен компилироваться без возникновения ошибок или предупреждений — хотя, конечно, вам для начала может потребоваться построить скелетное приложение на базе того или иного фрагмента кода.

Когда в каком-либо разделе вы знакомитесь с новой функцией, она записывается в обычном для UNIX формате справочной страницы такого вида:

```
#include <fcntl.h>
int posix_fadvise (int fd, off_t pos, off_t len, int advice);
```

Все необходимые заголовки и определения находятся сверху, за ними следует полный прототип вызова.

Работа с примерами кода

Эта книга написана, чтобы помочь вам при работе. В принципе, вы можете использовать код, содержащийся в ней, в ваших программах и документации. Можете не связываться с нами и не спрашивать разрешения, если собираетесь воспользоваться небольшим фрагментом кода. Например, если вы пишете программу и кое-где вставляете в нее код из книги, никакого особого разрешения не требуется. Однако если вы запишете на диск примеры из книги и начнете раздавать или продавать такие диски, то на это необходимо получить разрешение. Если вы цитируете это издание, отвечая на вопрос, или воспроизводите код из него в качестве примера, разрешение не нужно. Если вы включаете значительный фрагмент кода из данной книги в документацию по вашему продукту, необходимо разрешение.

Благодарности

Эта книга могла появиться на свет только благодаря участию множества умных и великодушных людей. Конечно, любой их список будет неполным, но я искренне хотел бы поблагодарить за участие всех, кто помогал мне в работе, воодушевлял меня, делился знаниями и поддерживал.

Энди Орам (Andy Oram) — феноменальный редактор и удивительный человек. Работа над книгой никогда бы не завершилась без его упорного труда. Энди относится к редкому типу людей, которые обладают одновременно и глубокими техническими знаниями, и поэтическим языковым чутьем.

У этой книги были превосходные технические рецензенты, истинные мастера своего искусства, без участия которых эта работа была бы бледной тенью того, что вы сейчас читаете. Техническими рецензентами книги выступили Джереми Эллисон (Jeremy Allison), Роберт Пи-Джи Дэй (Robert P. J. Day), Кеннет Гейсшет (Kenneth Geisshirt), Джоуи Шоу (Joey Shaw) и Джеймс Уилкоккс (James Willcox). Они славно поработали, и если в книге и найдутся ошибки, то это только моя вина.

Мои коллеги из Google были и остаются самой умной и самоотверженной командой инженеров, вместе с которыми я имел счастье работать. «Покой нам только снится» — такой фразой в лучшем ее смысле можно охарактеризовать этот коллектив. Спасибо вам за проекты в области системного программирования, которые помогли мне создать этот труд, и за атмосферу, подвигающую человека на творческие свершения, например написание подобной книги.

По многим причинам я хотел бы выразить благодарность и глубокое уважение Полу Амичи (Paul Amici), Майки Бэббиту (Mikey Babbitt), Нату Фридману (Nat Friedman), Мигелю де Иказе (Miguel de Icaza), Грегу Кроаху-Хартману (Greg Kroah-Hartman), Дорис Лав (Doris Love), Линде Лав (Linda Love), Тиму О'Рейли (Tim O'Reilly), Сальваторе РибAUDO (Salvatore Ribaudo) и его семье, Крису Ривере (Chris Rivera), Кэролин Родон (Carolyn Rodon), Саре Стюарт (Sarah Stewart), Питеру Тейчману (Peter Teichman), Линусу Торвальдсу (Linus Torvalds), Джону Трoубриджу (Jon Trowbridge), Джереми ван Дорену (Jeremy van Doren) и его семье, Луису Вилье (Luis Villa), Стиву Вайсбергу (Steve Weisberg) и его семье, а также Хелен Уиснант (Helen Whisnant).

Наконец, благодарю моих родителей — Боба (Bob) и Элен (Elaine).

Роберт Лав, Бостон

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinit ski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

1 Введение и основополагающие концепции

Эта книга рассказывает о *системном программировании*, то есть написании *системного программного обеспечения*. Системные программы являются низкоуровневыми, взаимодействуют непосредственно с ядром и основными системными библиотеками. Ваши командная оболочка и текстовый редактор, компилятор и отладчик, основные утилиты и системные демоны — все это системное программное обеспечение. К данной категории относятся также сетевой сервер, веб-сервер и база данных. Эти компоненты являются классическими образцами системного ПО и взаимодействуют в основном, а то и исключительно с ядром и библиотекой C. Другое программное обеспечение (например, прикладные программы с графическими пользовательскими интерфейсами) находится на более высоком уровне и взаимодействует с низкоуровневыми лишь эпизодически. Некоторые специалисты целыми днями пишут только системное программное обеспечение, другие уделяют таким задачам лишь часть рабочего времени, но понимание системного ПО — это навык, который пригодится любому специалисту. Независимо от того, является такое программирование насущным хлебом конкретного инженера либо просто базисом для создания более высокоуровневых концепций, системное программирование — это сердце всего создаваемого нами софта.

Эта книга посвящена не всему системному программированию, а системному программированию в Linux. Linux — это современная UNIX-подобная операционная система, написанная с нуля Линусом Торвалдсом и стихийным сообществом программистов со всего мира. Linux разделяет цели и философию UNIX, но при этом не является лишь разновидностью UNIX, а идет своим путем, возвращаясь в русло UNIX, где это желательно, и отклоняясь, когда сие целесообразно. Однако, если не считать базового строения, Linux довольно самобытна. По сравнению с традиционными системами UNIX Linux поддерживает множество дополнительных системных вызовов, работает иначе и предлагает новые возможности.

Системное программирование

Программирование для UNIX изначально зарождалось именно как системное. Исторически системы UNIX не включали значительного количества высокоуровневых

концепций. Даже при программировании в среде разработки, например в системе X Window, в полной мере задействовались системные API ядра UNIX. Соответственно, можно сказать, что эта книга — о программировании для Linux вообще. Однако учтите, что в книге не рассматриваются *среды разработки* для Linux, например вообще не затрагивается тема make. Основное содержание книги — это API системного программирования, предоставляемые для использования на современной машине Linux.

Можно сравнить системное программирование с программированием приложений — и мы сразу заметим как значительное сходство, так и важные различия этих областей. Важная черта системного программирования заключается в том, что программист, специализирующийся в этой области, должен обладать глубокими знаниями оборудования и операционной системы, с которыми он имеет дело. Системные программы взаимодействуют в первую очередь с ядром и системными библиотеками, а прикладные опираются и на высокоуровневые библиотеки. Такие высокоуровневые библиотеки *абстрагируют* детальные характеристики оборудования и операционной системы. У подобного абстрагирования есть несколько целей: переносимость между различными системами, совместимость с разными версиями этих систем, создание удобного в использовании (либо более мощного, либо и то и другое) высокоуровневого инструментария. Соотношение, насколько активно конкретное приложение использует высокоуровневые библиотеки и насколько — систему, зависит от уровня стека, для которого было написано приложение. Некоторые приложения создаются для взаимодействия исключительно с высокоуровневыми абстракциями. Однако даже такие абстракции, весьма отдаленные от самых низких уровней системы, лучше всего получаются у специалиста, имеющего навыки системного программирования. Те же проверенные методы и понимание базовой системы обеспечивают более информативное и разумное программирование для всех уровней стека.

Зачем изучать системное программирование

В течение прошедшего десятилетия в написании приложений наблюдалась тенденция к уходу от системного программирования к высокоуровневой разработке. Это делалось как с помощью веб-инструментов (например, JavaScript), так и посредством управляемого кода (Java). Тем не менее такие разработки не свидетельствуют об отмирании системного программирования. Действительно, ведь кому-то приходится писать и интерпретатор JavaScript, и виртуальную машину Java, которые создаются именно на уровне системного программирования. Более того, даже разработчики, которые программируют на Python, Ruby или Scala, только выиграют от знаний в области системного программирования, поскольку будут понимать всю подноготную машины. Качество кода при этом гарантированно улучшится независимо от части стека, для которой он будет создаваться.

Несмотря на описанную тенденцию в программировании приложений, большая часть кода для UNIX и Linux по-прежнему создается на системном уровне. Этот код написан преимущественно на C и C++ и существует в основном на базе

интерфейсов, предоставляемых библиотекой C и ядром. Это традиционное системное программирование с применением Apache, bash, cp, Emacs, init, gcc, gdb, glibc, ls, mv, vim и X. В обозримом будущем эти приложения не сойдут со сцены.

К области системного программирования часто относят и разработку ядра или как минимум написание драйверов устройств. Однако эта книга, как и большинство работ по системному программированию, никак не касается разработки ядра. Ее основной фокус — системное программирование для пользовательского пространства, то есть уровень, который находится выше ядра. Тем не менее знания о ядре будут полезным дополнительным багажом при чтении последующего текста. Написание драйверов устройств — это большая и объемная тема, которая подробно описана в книгах, посвященных конкретно данному вопросу.

Что такое системный интерфейс и как я пишу системные приложения для Linux? Что именно при этом мне предоставляют ядро и библиотека C? Как мне удастся создавать оптимальный код, какие приемы возможны в Linux? Какие интересные системные вызовы есть в Linux, но отсутствуют в других UNIX-подобных системах? Как все это работает? Именно эти вопросы составляют суть данной книги.

Краеугольные камни системного программирования

В системном программировании для Linux можно выделить три основных краеугольных камня: системные вызовы, библиотеку C и компилятор C. О каждом из этих феноменов следует рассказать отдельно.

Системные вызовы

Системные вызовы — это начало и конец системного программирования. Системные вызовы (в англоязычной литературе встречается сокращение *syscall*) — это вызовы функций, совершаемые из пользовательского пространства. Они направлены из приложений (например, текстового редактора или вашей любимой игры) к ядру. Смысл системного вызова — запросить у операционной системы определенную службу или ресурс. Системные вызовы включают как всем знакомые операции, например `read()` и `write()`, так и довольно экзотические, в частности `get_thread_area()` и `set_tid_address()`.

В Linux реализуется гораздо меньше системных вызовов, чем в ядрах большинства других операционных систем. Например, в системах с архитектурой x86-64 таких вызовов насчитывается около 300 — сравните это с Microsoft Windows, где предположительно задействуются тысячи подобных вызовов. При работе с ядром Linux каждая машинная архитектура (например, Alpha, x86-64 или PowerPC) может дополнять этот стандартный набор системных вызовов своими собственными. Следовательно, системные вызовы, доступные в конкретной архитектуре, могут отличаться от доступных в другой. Тем не менее значительное подмножество всех системных вызовов — более 90 % — реализуется во всех архитектурах. К этим разделяемым 90 % относятся и общие интерфейсы, о которых мы поговорим в данной книге.

Активация системных вызовов. Невозможно напрямую связать приложения пользовательского пространства с пространством ядра. По причинам, связанным с обеспечением безопасности и надежности, приложениям пользовательского пространства нельзя разрешать непосредственно исполнять код ядра или манипулировать данными ядра. Вместо этого ядро должно предоставлять механизм, с помощью которого пользовательские приложения будут «сигнализировать» ядру о требовании активировать системный вызов. После этого приложение сможет осуществить *системное прерывание ядра* (trap) в соответствии с этим строго определенным механизмом и выполнить только тот код, который разрешит выполнить ядро. Детали этого механизма в разных архитектурах немного различаются. Например, в процессорах i386 пользовательское приложение выполняет инструкцию программного прерывания `int` со значением `0x80`. Эта инструкция осуществляет переключение на работу с пространством ядра — защищенной областью, — где ядром выполняется обработчик программного прерывания. Что же такое обработчик прерывания `0x80`? Это не что иное, как обработчик системного вызова!

Приложение сообщает ядру, какой системный вызов требуется выполнить и с какими параметрами. Это делается посредством *аппаратных регистров*. Системные вызовы обозначаются по номерам, начиная с 0. В архитектуре i386, чтобы запросить системный вызов 5 (обычно это вызов `open()`), пользовательское приложение записывает 5 в регистр `eax`, после чего выдает инструкцию `int`.

Передача параметров обрабатывается схожим образом. Так, в архитектуре i386 регистр применяется для всех возможных параметров — например, регистры `ebx`, `ecx`, `edx`, `esi` и `edi` в таком же порядке содержат первые пять параметров. В редких случаях, когда системный вызов имеет более пяти параметров, всего один регистр применяется для указания на буфер в пользовательском пространстве, где хранятся все эти параметры. Разумеется, у большинства системных вызовов имеется всего пара параметров.

В других архитектурах активация системных вызовов обрабатывается иначе, хотя принцип остается тем же. Вам, как системному программисту, обычно не нужно знать, как именно ядро обрабатывает системные вызовы. Эта информация уже интегрирована в стандартные соглашения вызова, соблюдаемые в конкретной архитектуре, и автоматически обрабатывается компилятором и библиотекой C.

Библиотека C

Библиотека C (`libc`) — это сердце всех приложений UNIX. Даже если вы программируете на другом языке, то библиотека C, скорее всего, при этом задействуется. Она обернута более высокоуровневыми библиотеками и предоставляет основные службы, а также способствует активации системных вызовов. В современных системах Linux библиотека C предоставляется в форме `GNUlibc`, сокращенно `glibc` (произносится как «джи-либ-си», реже «глиб-си»).

Библиотека GNU C предоставляет гораздо больше возможностей, чем может показаться из ее названия. Кроме реализации стандартной библиотеки C, `glibc` дает обертки для системных вызовов, поддерживает работу с потоками и основные функции приложений.

Компилятор C

В Linux стандартный компилятор языка C предоставляется в форме коллекции *компиляторов GNU* (GNU Compiler Collection, сокращенно gcc). Изначально gcc представляла собой версию cc (компилятора C) для GNU. Соответственно gcc расшифровывалась как GNU C Compiler. Однако впоследствии добавилась поддержка других языков, поэтому сегодня gcc служит общим названием всего семейства компиляторов GNU. При этом gcc — это еще и двоичный файл, используемый для активации компилятора C. В этой книге, говоря о gcc, я, как правило, имею в виду программу gcc, если из контекста не следует иное.

Компилятор, используемый в UNIX-подобных системах, в частности в Linux, имеет огромное значение для системного программирования, поскольку помогает внедрять стандарт языка C (см. подразд. «Стандарты языка C» разд. «Стандарты» данной главы), а также системный двоичный интерфейс приложений (см. разд. «API и ABI» текущей главы), о которых будет рассказано далее.

C++

В этой главе речь пойдет в основном о языке C — лингва франка системного программирования. Однако C++ также играет важную роль.

В настоящее время C++ уступил ведущие позиции в системном программировании своему старшему собрату C. Исторически разработчики Linux всегда отдавали C предпочтение перед C++: основные библиотеки, демоны, утилиты и, разумеется, ядро Linux написаны на C. Влияние C++ как «улучшенного C» в большинстве «нелинуксовых» систем можно назвать каким угодно, но не универсальным, поэтому в Linux C++ также занимает подчиненное положение относительно C.

Тем не менее далее в тексте в большинстве случаев вы можете заменять «C» на «C++». Действительно, C++ — отличная альтернатива C, подходящая для решения практически любых задач в области системного программирования. C++ может связываться с кодом на C, активизировать системные вызовы Linux, использовать glibc.

При написании на C++ в основу системного программирования закладывается еще два краеугольных камня — стандартная библиотека C++ и компилятор GNUC++. *Стандартная библиотека C++* реализует системные интерфейсы C++ и использует стандарт ISO C++ 11. Он обеспечивается библиотекой libstdc++ (иногда используется название libstdcxx). *Компилятор GNUC++* — это стандартный компилятор для кода на языке C++ в системах Linux. Он предоставляется в двоичном файле g++.

API и ABI

Разумеется, программист заинтересован, чтобы его код работал на всех системах, которые планируется поддерживать, как в настоящем, так и в будущем. Хочется быть уверенными, что программы, создаваемые на определенном дистрибутиве Linux, будут работать на других дистрибутивах, а также иных поддерживаемых архитектурах Linux и более новых (а также ранних) версиях Linux.

На системном уровне существует два отдельных множества определений и описаний, которые влияют на такую переносимость. Одно из этих множеств называется *интерфейсом программирования приложений* (Application Programming Interface, API), а другое — *двоичным интерфейсом приложения* (Application Binary Interface, ABI). Обе эти концепции определяют и описывают интерфейсы между различными компонентами программного обеспечения.

API

API определяет интерфейсы, на которых происходит обмен информацией между двумя компонентами программного обеспечения на уровне исходного кода. API обеспечивает абстракцию, предоставляя стандартный набор интерфейсов — как правило, это функции, — которые один программный компонент (обычно, но не обязательно это более высокоуровневый компонент из пары) может вызывать из другого (обычно более низкоуровневого). Например, API может абстрагировать концепцию отрисовки текста на экране с помощью семейства функций, обеспечивающих все необходимые аспекты для отрисовки текста. API просто определяет интерфейс; тот компонент программы, который обеспечивает работу API, обычно называется *реализацией* этого API.

API часто называют «контрактом». Это неверно как минимум в юридическом смысле этого слова, поскольку API не имеет ничего общего с двусторонним соглашением. Пользователь API (обычно более высокоуровневая программа) располагает нулевым входным сигналом для данного API и реализацией этой сущности. Пользователь может применять API «как есть» или не использовать его вообще: возьми или не трогай! Задача API — просто гарантировать, что, если оба компонента ПО воспользуются этим API, они будут совместимы на уровне исходного кода. Это означает, что пользователь API сможет успешно скомпилироваться с зависимостью от реализации этого API.

Практическим примером API служат интерфейсы, определенные в соответствии со стандартом C и реализуемые стандартной библиотекой C. Этот API определяет семейство простейших и критически важных функций, таких как процедуры для управления памятью и манипуляций со строками.

На протяжении всей книги мы будем опираться на разнообразные API, например стандартную библиотеку ввода-вывода, которая будет подробно рассмотрена в гл. 3. Самые важные API, используемые при системном программировании в Linux, описаны в разд. «Стандарты» данной главы.

ABI

Если API определяет интерфейсы в исходном коде, то ABI предназначен для определения двоичного интерфейса между двумя и более программными компонентами в конкретной архитектуре. ABI определяет, как приложение взаимодействует с самим собой, с ядром и библиотеками. В то время как API обеспечивает совместимость на уровне исходного кода, ABI отвечает за *совместимость*

на двоичном уровне. Это означает, что фрагмент объектного кода будет функционировать в любой системе с таким же ABI без необходимости перекомпиляции.

ABI помогают решать проблемы, связанные с соглашениями на уровне вызовов, порядком следования байтов, использованием регистров, активацией системных вызовов, связыванием, поведением библиотек и форматом двоичных объектов. Например, соглашения на уровне вызовов определяют, как будут вызываться функции, как аргументы передаются функциям, какие регистры сохраняются, а какие — искажаются, как вызывающая сторона получает возвращаемое значение.

Несколько раз предпринимались попытки определить единый ABI для многих операционных систем, взаимодействующих с конкретной архитектурой (в частности, для различных UNIX-подобных систем, работающих на i386), но эти усилия не увенчались какими-либо заметными успехами. Напротив, в операционных системах, в том числе Linux, сохраняется тенденция к определению собственных ABI по усмотрению разработчиков. ABI тесно связаны с архитектурой; абсолютное большинство ABI оперирует машинно-специфичными концепциями, в частности Alpha или x86-64. Таким образом, ABI является как элементом операционной системы (например, Linux), так и элементом архитектуры (допустим, x86-64).

Системные программисты должны ориентироваться в ABI, но запоминать их обычно не требуется. Структура ABI определяется *цепочкой инструментов* — компилятором, компоновщиком и т. д. — и никак иначе обычно не проявляется. Однако знание ABI положительно сказывается на качестве программирования, а также требуется при написании ассемблерного кода или разработке самой цепочки инструментов (последняя — классический пример системного программирования).

Стандарты

Системное программирование для UNIX — старинное искусство. Основы программирования для UNIX остаются незыблемыми в течение десятилетий. Однако сами системы UNIX развиваются достаточно динамично. Поведение изменяется — добавляются новые возможности. Чтобы как-то справиться с хаосом, целые группы, занятые стандартизацией, кодифицируют системные интерфейсы в специальных официальных документах. Существует множество таких стандартов, но фактически Linux официально не подчиняется каким-либо из них. Linux просто *стремится* соответствовать двум наиболее важным и превалирующим стандартам — POSIX и Single UNIX Specification (SUS, единая спецификация UNIX).

В POSIX и SUS, в частности, документирован API языка C для интерфейса, обеспечивающего взаимодействие с UNIX-подобными операционными системами. Фактически эти стандарты определяют системное программирование или как минимум его общее подмножество для UNIX-совместимых систем.

История POSIX и SUS

В середине 1980-х годов Институт инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE) возглавил начинания по стандартизации системных интерфейсов в UNIX-подобных операционных системах. Ричард Столлман (Richard Stallman), основатель движения Free Software, предложил назвать этот стандарт POSIX (произносится «пазикс», Portable Operating System Interface — интерфейс переносимых операционных систем UNIX).

Первым результатом этой работы, обнародованным в 1988 году, стал стандарт IEEE Std 1003.1-1988 (сокращенно POSIX 1988). В 1990 году IEEE пересмотрел стандарт POSIX, выпустив новую версию IEEE Std 1003.1-1990 (POSIX 1990). Необязательная поддержка работы в реальном времени и потоков была документирована соответственно в стандартах IEEE Std 1003.1b-1993 (POSIX 1993 или POSIX.1b) и IEEE Std 1003.1c-1995 (POSIX 1995 или POSIX.1c). В 2001 году необязательные стандарты были объединены с базовым POSIX 1990, образовав единый стандарт IEEE Std 1003.1-2001 (POSIX 2001). Последняя на этот момент версия была выпущена в декабре 2008 года и называется IEEE Std 1003.1-2008 (POSIX 2008). Все основные стандарты POSIX сокращаются до аббревиатур вида POSIX.1, версия от 2008 года является новейшей.

В конце 1980-х — начале 1990-х годов между производителями UNIX-подобных систем бушевали настоящие «юниксовые войны»: каждый старался закрепить за своим продуктом статус *единственной настоящей UNIX-системы*. Несколько крупных производителей сплотились вокруг The Open Group — промышленного консорциума, сформировавшегося в результате слияния Open Software Foundation (OSF) и X/Open. The Open Group стала заниматься сертификацией, публикацией научных статей и тестированием соответствия. В начале 1990-х годов, когда «юниксовые войны» были в самом разгаре, The Open Group выпустила Единую спецификацию UNIX (SUS). Популярность SUS быстро росла, во многом благодаря тому, что она была бесплатной, а стандарт POSIX оставался дорогостоящим. В настоящее время SUS включает в себя новейший стандарт POSIX.

Первая версия SUS была опубликована в 1994 году. Затем последовали пересмотренные версии, выпущенные в 1997-м (SUSv2) и 2002 году (SUSv3). Последний вариант SUS, SUSv4, был опубликован в 2008 году. В SUSv4 пересмотрен стандарт IEEE Std 1003.1-2008, объединяемый в рамках этой спецификации с несколькими другими стандартами. В данной книге я буду делать оговорки, когда системные вызовы и другие интерфейсы стандартизируются по POSIX. Стандартизацию по SUS я отдельно указывать не буду, так как SUS входит в состав POSIX.

Стандарты языка C

Знаменитая книга Денниса Ричи (Dennis Ritchie) и Брайана Кернигана (Brian Kernighan) *The C Programming Language*, впервые опубликованная в 1978 году, в течение многих лет использовалась как неофициальная спецификация языка C. Эта версия C была известна в кругах специалистов под названием *K&R C*. Язык C уже

стремительно заменял BASIC и другие языки того времени, превращаясь в лингва франка микрокомпьютерного программирования, поэтому в 1983 году Американский национальный институт стандартов (ANSI) сформировал специальный комитет. Этот орган должен был разработать официальную версию C и стандартизировать самый популярный на тот момент язык программирования. Новая версия включала в себя разнообразные доработки и усовершенствования, сделанные различными производителями, а также новый язык C++. Это был долгий и трудоемкий процесс, но к 1989 году версия *ANSI C* была готова. В 1990 году Международная организация по стандартизации (ISO) ратифицировала стандарт *ISO C90*, основанный на *ANSI C* с небольшими модификациями.

В 1995 году ISO выпустила обновленную (но редко используемую) версию языка C, которая называется *ISO C95*. В 1999 году последовала новая, значительно пересмотренная версия языка — *ISO C99*. В ней множество нововведений, в частности внутрискладочные функции, новые типы данных, массивы переменной длины, комментарии в стиле C++ и новые библиотечные функции. Последняя версия этого стандарта называется *ISO C11*, в которой следует отметить формализованную модель памяти. Она обеспечивает переносимость при использовании потоков в многоплатформенной среде.

Что касается C++, ISO-стандартизация этого языка протекала медленнее. В 1998 году после долгих лет разработки и выпуска компилятора, не обладавшего прямой совместимостью, был ратифицирован первый стандарт C, *ISO C98*. Он значительно улучшил совместимость между различными компиляторами, однако некоторые аспекты этого стандарта ограничивали согласованность и переносимость. Стандарт *ISO C++03* появился в 2003 году. В нем были исправлены некоторые ошибки, а также добавлены изменения, облегчившие работу разработчикам компиляторов, но незаметные на пользовательском уровне. Следующий стандарт ISO, самый актуальный в настоящее время, называется *C++11* (ранее он обозначался как *C++0x*, поскольку не исключалась более ранняя дата выхода). В этой версии появилось множество дополнений как на уровне языка, так и в стандартных библиотеках. На самом деле их оказалось настолько большое количество, что многие даже считают язык C++11 совершенно самостоятельным, независимым от более ранних версий C++.

Linux и стандарты

Как было сказано выше, Linux стремится соответствовать стандартам POSIX и SUS. В Linux предоставляются интерфейсы, документированные в SUSv4 и POSIX 2008, а также поддерживается работа в реальном времени (POSIX.1b) и работа с потоками (POSIX.1c). Гораздо важнее, что Linux стремится работать в соответствии с требованиями POSIX и SUS. В принципе, любое несоответствие стандартам является ошибкой. Считается, что Linux также соответствует POSIX.1 и SUSv3, но, поскольку никакой официальной сертификации POSIX или SUS не проводилось (в частности, во всех существующих версиях Linux), нельзя сказать, что Linux официально соответствует POSIX или SUS.

Что касается языковых стандартов, в Linux все хорошо. Компилятор gcc языка C соответствует стандарту ISO C99; планируется обеспечить поддержку C11. Компилятор g++ языка C++ соответствует стандарту ISO C++03, поддержка стандарта C++11 находится в разработке. Кроме того, компиляторы gcc и g++ реализуют расширения для языков C и C++. Все эти расширения объединяются под общим названием GNU C и документированы в приложении А.

Linux не может похвастаться большими достижениями в области обеспечения прямой совместимости¹, хотя сегодня и в этой области ситуация значительно улучшилась. Интерфейсы, документированные в соответствии со стандартами, в частности стандартная библиотека C, очевидно, навсегда останутся совместимыми на уровне исходников. Двоичная совместимость поддерживается как минимум на уровне основной, крупной версии glibc, а поскольку язык C стандартизирован, gcc всегда будет компилировать код, написанный на правильном языке C. Правда, различные специфичные расширения gcc могут устаревать и в конце концов исчезать из новых релизов gcc. Важнее всего, что ядро Linux гарантирует стабильность системных вызовов. Если системный вызов реализован в стабильной версии ядра Linux, можно быть уверенными, что такой вызов точно сработает.

В различных дистрибутивах Linux многие компоненты операционной системы определяются в LSB (Linux Standard Base). LSB — это совместный проект нескольких производителей Linux, проводящийся под эгидой Linux Foundation (ранее эта организация называлась Free Standards Group). LSB дополняет POSIX и SUS, а также добавляет собственные стандарты. Организация стремится предоставить двоичный стандарт, позволяющий в неизменном виде выполнять объектный код в системах, отвечающих этому стандарту. Большинство производителей Linux в той или иной степени придерживаются LSB.

Стандарты и эта книга

В данной книге я намеренно стараюсь не разглагольствовать о каком-либо стандарте. Слишком часто авторы книг по системному программированию для UNIX излишне увлекаются сравнениями, как интерфейс работает по одним стандартам и как по другим, как конкретный системный вызов реализован в той или иной системе, — словом, льют воду. Эта книга посвящена именно системному программированию в современных вариантах Linux, в которых используются новейшие версии ядра Linux (3.9), компилятора gcc (4.8) и библиотеки C (2.17).

Системные интерфейсы можно считать практически неизменными — разработчики ядра Linux проделали огромную работу, чтобы, например, никогда не пришлось ломать интерфейсы системных вызовов. Эти интерфейсы обеспечивают известный уровень совместимости на уровне исходного кода и двоичном уровне, поэтому выбранный в данной книге подход позволяет подробно рассмотреть детали системных

¹ Возможно, опытные пользователи Linux помнят переход с a.out на ELF, переход с libc5 на glibc, изменения gcc, фрагментацию шаблонов ABI C++ и т. д. К счастью, эти времена давно позади.

интерфейсов Linux, абстрагируясь от проблем совместимости с многочисленными другими UNIX-подобными системами и не думая о соответствии всем стандартам. Мы будем говорить только о Linux, поэтому можем позволить себе подробно остановиться на ультрасовременных интерфейсах этой операционной системы, которые, несомненно, останутся востребованными и действующими в обозримом будущем. В основу книги положены глубокие знания Linux, информация о реализации и поведении таких компонентов, как ядро и gcc. Эту работу можно считать повествованием разработчика-ветерана, полным проверенных методов и советов по оптимизации.

Концепции программирования в Linux

В этом разделе вы найдете краткий обзор сервисов, предоставляемых в системе Linux. Все UNIX-подобные системы, включая Linux, предлагают общий набор абстракций и интерфейсов. На самом деле в этой взаимосовместимости и заключается *суть* UNIX. Такие абстракции, как файл и процесс, интерфейсы для управления конвейерами и сокетами и т. д., являются главной составляющей систем UNIX.

Этот обзор предполагает, что вы знакомы с системой Linux. Имеется в виду, что вы умеете обращаться с командной оболочкой, использовать базовые команды и компилировать простую программу на C. Это *не* обзор Linux или системы для программирования в ней, а рассказ об основах системного программирования Linux.

Файлы и файловая система

Файл — это самая простая и базовая абстракция в Linux. Linux придерживается философии «*все есть файл*», пусть и не так строго, как некоторые другие системы — достаточно вспомнить Plan 9¹. Следовательно, многочисленные взаимодействия представляют собой считывание из файлов и запись в них, даже если объект, с которым вы имеете дело, совсем не похож на «традиционный» файл.

Вообще, чтобы получить доступ к файлу, его сначала нужно открыть. Файлы можно открывать для чтения, записи или того и другого сразу. На открытый файл указывает уникальный дескриптор, отображающий метаданные, ассоциированные с открытым файлом, обратно на сам этот файл. В ядре Linux такой дескриптор управляется целым числом (в системе типов C целому числу соответствует тип `int`). Эта сущность, называемая *файловым дескриптором*, сокращенно обозначается *fd*. Дескрипторы файлов совместно используются в системе и пользовательском пространстве. Пользовательские программы применяют их непосредственно для доступа к файлам. Значительная часть системного программирования в Linux сводится к открытию файлов, манипуляциям с ними, закрытию файлов и использованию файловых дескрипторов иными способами.

¹ Plan 9 — это операционная система, разработанная в BellLabs и часто характеризующаяся как наследница UNIX. В ней воплощено несколько инновационных идей, и она четко придерживается философии «все есть файл».

Обычные файлы

Сущности, которые большинству из нас известны под названием «файлы», в Linux именуются *обычными файлами*. Обычный файл содержит байты данных, организованные в виде линейного массива, который называется потоком байтов. В Linux для файла не задается никаких других видов упорядочения или форматирования. Байты могут иметь любые значения и быть организованы внутри файла любыми способами. На системном уровне Linux не регламентирует для файлов никакой структуры, кроме организации в виде потока байтов. В некоторых операционных системах, например VMS, используются высокоструктурированные файлы, в которых применяются так называемые *записи*. В Linux такие записи отсутствуют.

Любые байты внутри файла могут использоваться для считывания или записи. Эти операции всегда начинаются с указанного байта, который можно назвать местоположением в файле. Это местоположение называется *файловой позицией* или *смещением файла*. Файловая позиция — это важнейший элемент метаданных, который ядро ассоциирует с каждым открытием файла. Когда файл открывается впервые, его файловая позиция равна нулю. Обычно по мере того, как байты из файла считываются либо в него записывается информация (байт за байтом), значение файловой позиции увеличивается. Файловую позицию можно также вручную устанавливать в определенное значение, причем оно может находиться даже за пределами (за последним байтом) конкретного файла. Когда файловая позиция находится за пределами файла, промежуточные байты будут заполняться нулями. Вполне возможно воспользоваться этим способом и задать файловую позицию дальше конца файла, однако вы никак не сможете установить эту позицию перед началом файла. Правда, такая практика кажется бессмысленной и действительно она почти не применяется. Файловая позиция начинается с нуля; она не может иметь отрицательное значение. При записи в байт в середине файла значение, которое ранее находилось по этому смещению, заменяется новым, поэтому вы не сможете расширить файл, записывая информацию в его середину. Как правило, запись в файл происходит в его конце. Максимальное значение файловой позиции ограничено только размером типа C, используемого для хранения файла. В современных системах Linux максимальное значение этого параметра равно 64 бит.

Размер файла измеряется в байтах и называется его *длиной*. Можно сказать, что длина — это просто количество байтов в линейном массиве, составляющем файл. Длину файла можно изменить с помощью операции, которая называется *усечением*. Файл можно укоротить, уменьшив его размер по сравнению с исходным. В результате будут удалены байты, расположенные в конце файла. Термин «усечение» немного неудачный, поскольку им обозначается и удлинение файла, то есть увеличение его размера по сравнению с исходным. В таком случае новые байты (добавляемые в конце файла) заполняются нулями. Файл может быть пуст (иметь нулевую длину) и, соответственно, не содержать ни одного валидного байта. Максимальная длина файла, как и файловая позиция, ограничена лишь размерами тех типов C, которые применяются ядром Linux для управления файлами. Однако

в конкретных файловых системах могут действовать собственные ограничения, из-за которых потолок длины файла существенно снижается.

Отдельно взятый файл можно одновременно открыть несколько раз как в ином, так и в том же самом процессе. Каждому открытому экземпляру файла присваивается уникальный дескриптор. С другой стороны, процессы могут совместно использовать свои файловые дескрипторы, один дескриптор может применяться в нескольких процессах. Ядро не накладывает никаких ограничений на параллельный доступ к файлу. Множественные процессы вполне могут одновременно считывать информацию из файла и записывать туда новые данные. Результаты такой параллельной работы зависят от упорядочения отдельных операций и, в принципе, непредсказуемы. Программы пользовательского пространства обычно должны взаимно координироваться, чтобы обеспечить правильную синхронизацию параллельных обращений к файлам.

Хотя доступ к файлам обычно осуществляется по их именам, непосредственная связь файла с его названием отсутствует. В действительности ссылка на файл выполняется по *индексному дескриптору*¹. Этому дескриптору присваивается целочисленное значение, уникальное для файловой системы (но не обязательно уникальное во всей системе в целом). Данное значение называется *номером индексного дескриптора*. В индексном дескрипторе сохраняются метаданные, ассоциированные с файлом, например отметка о времени его последнего изменения, владелец файла, тип, длина и местоположение данных файла, но имя файла там не сохраняется! Индексный дескриптор одновременно является и физическим объектом, расположенным на диске в UNIX-подобной файловой системе, и концептуальной сущностью, представленной как структура данных в ядре Linux.

Каталоги и ссылки

Обращение к файлам по их индексным дескрипторам — довольно трудоемкий процесс (а также потенциальная брешь в системе безопасности), поэтому из пользовательского пространства файлы обычно вызываются по имени, а не по индексному дескриптору. Для предоставления имен, по которым можно обращаться к файлам, используются *каталоги*. Каталог представляет собой отображение понятных человеку имен в номера индексных дескрипторов. Пара, состоящая из имени и индексного дескриптора, называется *ссылкой*. Физическая форма этого отображения, присутствующая на диске, например простая таблица или хеш, реализуется и управляется кодом ядра, поддерживающим конкретную файловую систему. В принципе, каталог ничем не отличается от обычного файла, за исключением того, что в нем содержатся лишь отображения имен в индексные дескрипторы. Ядро непосредственно пользуется этими отображениями для разрешения имен в индексные дескрипторы.

Когда из пользовательского пространства приходит запрос на открытие файла с указанным именем, ядро открывает каталог, в котором содержится файл с таким названием, и ищет данное имя. По имени файла ядро получает номер его

¹ См.: <http://ru.wikipedia.org/wiki/Inode>. — *Примеч. пер.*

индексного дескриптора. По этому номеру находится сам индексный дескриптор. Индексный дескриптор содержит метаданные, ассоциированные с файлом, в частности информацию о том, в каком именно фрагменте диска записаны данные этого файла.

Сначала на диске присутствует лишь один *корневой каталог*. К нему обычно ведет путь /. Однако, как известно, в любой системе, как правило, множество каталогов. Как ядро узнает, в каком *именно* нужно искать файл с заданным именем?

Выше мы говорили о том, что каталоги во многом похожи на обычные файлы. Действительно, с ними даже ассоциированы свои индексные дескрипторы, поэтому ссылки внутри каталогов могут указывать на индексные дескрипторы, находящиеся в других каталогах. Это означает, что одни каталоги можно вкладывать в другие, образуя иерархические структуры, что, в свою очередь, позволяет использовать *полные пути к элементам*, знакомые каждому пользователю UNIX, например `/home/blackbeard/concorde.png`.

Когда мы запрашиваем у ядра открытие подобного пути к файлу, оно обходит все *записи каталогов*, указанные в пути к элементу. Так удастся найти индексный дескриптор следующей записи. В предыдущем примере ядро начинает работу с /, получает индексный дескриптор `home`, идет туда, получает индексный дескриптор `blackbeard`, идет туда и, наконец, получает индексный дескриптор `concorde.png`. Эта операция называется *разрешением каталога* или *разрешением пути к элементу*. Кроме того, в ядре Linux используется кэш, называемый *кэшем каталогов*. В кэше каталогов сохраняются результаты разрешения каталогов, впоследствии обеспечивающие более быстрый поиск с учетом временной локальности¹.

Если имя пути начинается с корневого каталога, говорят, что путь *полностью уточнен*. Его называют *абсолютным путем к элементу*. Некоторые имена путей уточнены не полностью, а указываются относительно какого-то другого каталога (например, `todo/plunder`). Такие пути называются *относительными*. Если ядру предоставляется относительный путь, то оно начинает разрешение пути с *текущего рабочего каталога*. Отсюда ядро ищет путь к каталогу `todo`. В каталоге `todo` ядро получает индексный дескриптор `plunder`. В результате комбинации относительного пути к элементу и пути к текущему рабочему каталогу получается полностью уточненный путь.

Хотя каталоги и воспринимаются как обычные файлы, ядро не позволяет их открывать и производить с ними те же манипуляции, что и с обычными файлами. Для работы с каталогами используется специальный набор системных вызовов. Эти системные вызовы предназначаются для добавления и удаления ссылок — в принципе, на этом перечень разумных операций с каталогами заканчивается. Если бы можно было манипулировать каталогами прямо из пользовательского пространства, без посредничества ядра, то единственной простой ошибки хватило бы для повреждения всей файловой системы.

¹ Временная локальность — это высокая вероятность обращения к конкретному ресурсу после другой, более ранней операции доступа к нему же. Временная локальность характерна для многих ресурсов компьютера.

Жесткие ссылки

С учетом всего вышесказанного ничто вроде бы не препятствует разрешению множества имен в один и тот же индексный дескриптор. Действительно, это допускается. Когда множественные ссылки отображают различные имена на один и тот же индексный дескриптор, эти ссылки называются *жесткими*.

Благодаря жестким ссылкам в файловых системах обеспечивается создание сложных структур, где множественные имена путей могут указывать на одни и те же данные. Эти жесткие ссылки могут находиться в одном каталоге, а также в двух и более различных каталогах. В любом случае ядро просто разрешает имя пути в верный индексный дескриптор. Например, можно поставить жесткую ссылку на конкретный индексный дескриптор, ссылающийся на определенный фрагмент данных из двух мест — `/home/bluebeard/treasure.txt` и `/home/blackbeard/to_steal.txt`.

При удалении файла он отсоединяется от структуры каталогов. Для этого нужно просто удалить из каталога пару, в которой содержится имя файла и его индексный дескриптор. Однако, поскольку в Linux поддерживаются жесткие ссылки, файловая система не может просто уничтожать индексный дескриптор и ассоциированные с ним данные при каждой операции удаления. Что, если на этот файл были проставлены и другие жесткие ссылки из файловой системы? Чтобы гарантировать, что файл не будет уничтожен, пока не исчезнут *все* указывающие на него жесткие ссылки, в каждом индексном дескрипторе содержится *счетчик ссылок*, отслеживающий количество ссылок в файловой системе, указывающих на этот дескриптор. Когда путь к элементу отсоединяется от файловой системы, значение этого счетчика уменьшается на 1. Лишь если значение счетчика ссылок достигает нуля, и индексный дескриптор, и ассоциированные с ним данные окончательно удаляются из файловой системы.

Символьные ссылки

Жесткие ссылки не могут связывать файловые системы, поскольку номер индексного дескриптора не имеет смысла вне его собственной файловой системы. Чтобы ссылки могли соединять информацию из различных файловых систем, становясь при этом и более простыми, и менее прозрачными, в системах UNIX применяются так называемые *символьные ссылки*.

Символьные ссылки похожи на обычные файлы. Такая ссылка имеет свой индексный дескриптор и ассоциированный с ним фрагмент данных, содержащий полное имя пути к связанному файлу. Таким образом, символьные ссылки могут указывать куда угодно, в том числе на файлы и каталоги, расположенные в иных файловых системах, и даже на несуществующие файлы и каталоги. Символьная ссылка, указывающая на несуществующий файл, называется *сломанной*.

С использованием символьных ссылок связано больше издержек, чем при работе с жесткими ссылками, так как символьная ссылка, в сущности, требует разрешения двух файлов: самой символьной ссылки и связанного с ней файла. При использовании жестких ссылок такие дополнительные затраты отсутствуют — нет разницы между обращениями к файлам, обладающим одной связью в файловой

системе либо несколькими связями. Издержки при работе с символьными ссылками минимальны, но тем не менее они воспринимаются отрицательно.

Кроме того, символьные ссылки менее прозрачны, чем жесткие. Использование жестких ссылок — совершенно очевидный процесс. Более того, не так просто найти файл, на который проставлено несколько жестких ссылок! Для манипуляций же с символьными ссылками требуются специальные системные вызовы. Эта непрозрачность зачастую воспринимается как положительный момент, так как в символьной ссылке ее структура выражается открытым текстом. Символьные ссылки используются именно как *инструменты быстрого доступа (ярлыки)*, а не как внутрисистемные ссылки.

Специальные файлы

Специальные файлы — это объекты ядра, представленные в виде файлов. С годами в системах UNIX накопилось множество типов поддерживаемых специальных файлов. В Linux поддерживается четыре типа таких файлов: файлы блочных устройств, файлы символьных устройств, именованные каналы¹ и доменные сокеты UNIX. Специальные файлы обеспечивают возможности встраивания определенных абстракций в файловую систему и, таким образом, поддерживают парадигму «все есть файл». Для создания специального файла в Linux предоставляется специальный системный вызов.

Доступ к устройствам в системах UNIX осуществляется через файлы устройств, которые выглядят и действуют как обычные файлы, расположенные в файловой системе. Файлы устройств можно открывать, считывать из них информацию и записывать ее в них. Из пользовательского пространства можно получать доступ к файлам устройств и манипулировать устройствами в системе (как физическими, так и виртуальными). Как правило, все устройства в UNIX подразделяются на две группы — *символьные устройства* и *блочные устройства*. Каждому типу устройства соответствует свой специальный файл устройства.

Доступ к символьному устройству осуществляется как к линейной последовательности байтов. Драйвер устройства ставит байты в очередь, один за другим, а программа из пользовательского пространства считывает байты в порядке, в котором они были помещены в очередь. Типичным примером символьного устройства является клавиатура. Если пользователь наберет на клавиатуре последовательность *рег*, то приложению потребуется считать из файла-устройства клавиатуры сначала *р*, потом *е* и, наконец, *г* — именно в таком порядке. Когда больше не остается символов, которые необходимо прочесть, устройство возвращает «конец файла» (EOF). Если какой-то символ будет пропущен или символы будут прочтены в неправильном порядке, то операция получится фактически бессмысленной. Доступ к символьным устройствам происходит через *файлы символьных устройств*.

¹ Именованный канал называется в оригинале *named pipe*, более точный перевод — именованный конвейер. Тем не менее мы остановимся на варианте «именованный канал» (http://ru.wikipedia.org/wiki/Именованный_канал), как на более употребительном в русском языке, а термин *pipe* будем далее переводить как «конвейер». — *Примеч. пер.*

Напротив, доступ к блочному устройству происходит как к массиву байтов. Драйвер устройства отображает массив байтов на устройство с возможностью позиционирования, и пользовательское пространство может в произвольном порядке обращаться к любым валидным байтам массива, то есть можно сначала прочитать байт 12, потом байт 7, потом опять байт 12. Блочные устройства — это обычно устройства для хранения информации. Жесткие диски, дисководы гибких дисков, CD-ROM, флэш-накопители — все это примеры блочных устройств. Доступ к ним осуществляется через *файлы блочных устройств*.

Именованные каналы (часто обозначаемые аббревиатурой FIFO — «первым пришел, первым обслужен») — это механизм межпроцессного взаимодействия (IPC), предоставляющего канал связи для дескриптора файла. Доступ к именованному каналу выполняется через специальный файл. Обычные конвейеры применяются именно для того, чтобы «перекачивать» вывод одной программы во ввод другой; они создаются в памяти посредством системного вызова и не существуют в какой-либо файловой системе. Именованные каналы действуют как и обычные, но обращение к ним происходит через файл, называемый *специальным файлом FIFO*. Несвязанные процессы могут обращаться к этому файлу и обмениваться информацией.

Последний тип специальных файлов — это *сокеты*. Сокеты обеспечивают усовершенствованную разновидность межпроцессного взаимодействия между двумя несвязанными процессами — не только на одной машине, но и даже на двух разных. На самом деле сокеты являются основополагающей концепцией всего программирования для сетей и Интернета. Существует множество разновидностей сокетов, в том числе доменные сокеты UNIX. Последние используются для взаимодействия на локальной машине. В то время как сокеты, обменивающиеся информацией по Интернету, могут использовать пару из хост-имени и порта для идентификации «цели» взаимодействия, доменные сокеты используют для этого специальный файл, расположенный в файловой системе. Часто его называют просто сокет-файлом.

Файловые системы и пространства имен

Linux, как и все системы UNIX, предоставляет глобальное и единое *пространство имен* для файлов и каталогов. В некоторых операционных системах отдельные физические и логические диски разделяются на самостоятельные пространства имен. Например, для доступа к файлу на диске может использоваться путь A:\plank.jpg, а пути ко всем файлам на жестком диске будут начинаться с C:\. В UNIX тот же файл с дискеты может быть доступен через /media/floppy/plank.jpg или даже через /home/captain/stuff/plank.jpg, в одном ряду с файлами с других носителей. В этом и выражается единство пространства имен в UNIX.

Файловая система — это набор файлов и каталогов формальной и валидной иерархии. Файловые системы можно по отдельности добавлять к глобальному пространству имен и удалять их из этого глобального пространства файлов и каталогов. Данные операции называются *монтированием* и *размонтированием*. Каждая файловая система может быть индивидуально смонтирована к конкретной точке пространства имен. Она обычно называется *точкой монтирования*. В дальнейшем из точки монтирования открывается доступ к корневому

каталогу файловой системы. Например, CD может быть монтирован в точке `/media/cdrom`, в результате чего корневой каталог файловой системы компакт-диска также будет доступен через `/media/cdrom`. Файловая система, которая была монтирована первой, находится в корне пространства имен, `/`, и называется *корневой файловой системой*. Во всех системах Linux всегда имеется корневая файловая система. Монтирование других файловых систем в тех или иных точках в Linux не является обязательным.

Файловые системы обычно существуют физически (то есть сохраняются на диске), хотя в Linux также поддерживаются *виртуальные файловые системы*, существующие лишь в памяти, и *сетевые файловые системы*, существующие одновременно на нескольких машинах, работающих в сети. Физические файловые системы находятся на блочных устройствах хранения данных, в частности на компакт-дисках, дискетах, картах флэш-памяти, жестких дисках. Некоторые из этих устройств являются *сегментируемыми* — это означает, что их дисковое пространство можно разбить на несколько файловых систем, каждой из которых можно управлять отдельно. В Linux поддерживаются самые разные файловые системы, решительно любые, которые могут встретиться среднему пользователю на практике. В частности, Linux поддерживает файловые системы, специфичные для определенных носителей (например, ISO9660), сетевые файловые системы (*NFS*), нативные файловые системы (*ext4*), файловые системы из других разновидностей UNIX (*XFS*), а также файловые системы, не относящиеся к семейству UNIX (*FAT*).

Наименьшим адресуемым элементом блочного устройства является *сектор*. Сектор — это физический атрибут устройства. Объем секторов может быть равен различным степеням двойки, довольно распространены секторы размером 512 байт. Блочное устройство не может передавать элемент данных размером меньше, чем сектор этого устройства, а также не может получать доступ к такому мелкому фрагменту данных. При операциях ввода-вывода должны задействоваться один или несколько секторов.

Аналогично, наименьшим логически адресуемым элементом файловой системы является *блок*. Блок — это абстракция, применяемая в файловой системе, а не на физическом носителе, на котором эта система находится. Обычно размер блока равен степени двойки от размера сектора. В Linux блоки, как правило, крупнее сектора, но они должны быть меньше *размера страницы* (под страницей в модели памяти Linux понимается наименьший элемент, адресуемый *блоком управления памятью* — аппаратным компонентом)¹. Размеры большинства блоков составляют 512 байт, 1 Кбайт и 4 Кбайт.

Исторически в системах UNIX было только одно разделяемое пространство имен, видимое для всех пользователей и всех процессов в системе. В Linux используется инновационный подход и поддерживаются *пространства имен отдельных процессов*. В таком случае каждый процесс может иметь уникальное представление

¹ Это искусственное ограничение возможностей ядра, установленное ради обеспечения простоты с учетом, что в будущем, возможно, системы значительно усложнятся.

файла системы и иерархии каталогов¹. По умолчанию каждый процесс наследует пространство имен своего родительского процесса, но процесс также может создать собственное пространство имен со своим набором точек монтирования и уникальным корневым каталогом.

Процессы

Если файлы являются самой фундаментальной абстракцией системы UNIX, то следующая по важности — *процесс*. Процессы — это объектный код, находящийся в процессе исполнения: активные, работающие программы. Однако процессы — это не просто объектный код, так как они состоят из данных, ресурсов, состояния и виртуализованного процессора.

Процесс начинает свой жизненный цикл в качестве исполняемого объектного кода. Это код в формате, пригодном для исполнения на машине и понятный ядру. Наиболее распространенный подобный формат в Linux называется форматом исполняемых и компонуемых файлов (ELF). Исполняемый формат содержит метаданные и множество *разделов* с кодом и данными. Разделы — это линейные фрагменты объектного кода, именно в такой линейной форме загружаемые в память. Все байты одного раздела обрабатываются одинаково, имеют одни и те же права доступа и, как правило, используются в одних и тех же целях.

К самым важным и распространенным разделам относятся текстовый раздел, раздел данных и раздел `bss`. В текстовом разделе содержатся исполняемый код и данные только для чтения, в частности константные переменные. Обычно этот раздел помечается как доступный только для чтения и исполняемый. В разделе данных хранятся инициализированные данные (например, переменные `C` с определенными значениями). Обычно этот раздел помечается как доступный для чтения и записи. Раздел `bss` содержит неинициализированные глобальные данные. Стандарт `C` требует, чтобы все глобальные переменные `C` по умолчанию инициализировались нулями, поэтому нет необходимости хранить нули в объектном коде на диске. Вместо этого объектный код может просто перечислять неинициализированные переменные в разделе `bss`, а при загрузке раздела в память ядро отобразит на него *нулевую страницу* (страницу, содержащую только нули). Раздел `bss` задумывался исключительно в качестве оптимизации. Название `bss`, в сущности, является пережитком, оно означает `block started by symbol` (блок, начинающийся с символа). Другими примерами распространенных разделов исполняемых файлов в формате ELF являются *абсолютный раздел* (содержащий непереключаемые символы) и *неопределенный раздел*, также называемый общей корзиной (`catchall`).

Кроме того, процесс ассоциирован с различными системными ресурсами, которые выделяются и управляются ядром. Как правило, процессы запрашивают ресурсы и манипулируют ими только посредством системных вызовов. К ресурсам относятся таймеры, ожидающие сигналы, открытые файлы, сетевые соединения,

¹ Этот подход был впервые реализован в операционной системе Plan 9 производства BellLabs.

аппаратное обеспечение и механизмы межпроцессного взаимодействия. Ресурсы процесса, а также относящиеся к нему данные и статистика сохраняются внутри ядра в *дескрипторе процесса*.

Процесс — это абстракция виртуализации. Ядро Linux поддерживает как вытесняющую многозадачность, так и виртуальную память, поэтому предоставляет каждому процессу и виртуализованный процессор, и виртуализованное представление памяти. Таким образом, с точки зрения процесса система выглядит так, как будто только он ею управляет. Соответственно, даже если конкретный процесс может быть диспетчеризован наряду со многими другими процессами, он работает так, как будто он один обладает полным контролем над системой. Ядро незаметно и прозрачно вытесняет и переназначает процессы, совместно используя системные процессоры на всех работающих процессах данной системы. Процессы этого даже не знают. Аналогичным образом каждый процесс получает отдельное линейное адресное пространство, как если бы он один контролировал всю память, доступную в системе. С помощью виртуальной памяти и подкачки страниц ядро обеспечивает одновременное сосуществование сразу многих процессов в системе. Каждый процесс при этом работает в собственном адресном пространстве. Ядро управляет такой виртуализацией, опираясь на аппаратную поддержку, обеспечиваемую многими современными процессорами. Таким образом, операционная система может параллельно управлять состоянием множественных, не зависящих друг от друга процессов.

Потоки

Каждый процесс состоит из одного или нескольких *потоков выполнения*, обычно называемых просто *потоками*. Поток — это единица активности в процессе. Можно также сказать, что поток — это абстракция, отвечающая за выполнение кода и поддержку процесса в рабочем состоянии.

Большинство процессов состоят только из одного потока и именуются *однопоточными*. Если процесс содержит несколько потоков, его принято называть *многопоточным*. Традиционно программы UNIX являются однопоточными, это связано с присущей UNIX простотой, быстрым созданием процессов и надежными механизмами межпроцессного взаимодействия. По всем этим причинам многопоточность не имеет в UNIX существенного значения.

Поток состоит из *стека* (в котором хранятся локальные переменные этого процесса, точно как в стеке процесса, используемом в однопоточных системах), состояния процесса и актуального местоположения в объектном коде. Информация об этом местоположении обычно хранится в *указателе команд* процесса. Большинство остальных элементов процесса разделяются между всеми его потоками, особенно это касается адресного пространства. Таким образом, потоки совместно используют абстракцию виртуальной памяти, поддерживая абстракцию виртуализованного процессора.

На внутрисистемном уровне в ядре Linux реализуется уникальная разновидность потоков: фактически они представляют собой обычные процессы, которые по мере необходимости разделяют определенные ресурсы. В пользовательском пространстве Linux реализует потоки в соответствии со стандартом POSIX 1003.1c (также

называемым Pthreads). Самый современный вариант реализации потоков в Linux именуется Native POSIX Threading Library (NPTL). Эта библиотека входит в состав glibc. Подробнее мы поговорим о потоках в гл. 7.

Иерархия процессов

Для идентификации каждого процесса применяется уникальное положительное целое число, называемое *идентификатором процесса*, или ID процесса (pid). Таким образом, первый процесс имеет идентификатор 1, а каждый последующий процесс получает новый, уникальный pid.

В Linux процессы образуют строгую иерархию, называемую *деревом процессов*. Корень дерева находится в первом процессе, называемом *процессом инициализации* и обычно принадлежащем программе init. Новые процессы создаются с помощью системного вызова fork(). В результате этого вызова создается дубликат вызывающего процесса. Исходный процесс называется *предком*, а порожденный — *потомком*. У каждого процесса, кроме самого первого, есть свой предок. Если родительский процесс завершается раньше дочернего (потомка), то ядро *переназначает* предка для потомка, делая его потомком процесса инициализации.

Когда процесс завершается, он еще какое-то время остается в системе. Ядро сохраняет фрагменты процесса в памяти, обеспечивая процессу-предку доступ к информации о процессе-потомке, актуальной на момент завершения потомка. Такое запрашивание называется *обслуживанием* завершенного процесса. Когда родительский процесс обслужит дочерний, последний полностью удаляется. Процесс, который уже завершился, но еще не был обслужен, называется *зомби*. Инициализирующий процесс по порядку обслуживает все свои дочерние процессы, гарантируя, что процессы с переназначенными родителями не останутся в состоянии зомби на неопределенный срок.

Пользователи и группы

Авторизация в Linux обеспечивается с помощью системы *пользователей и групп*. Каждому пользователю присваивается уникальное положительное целое число, называемое *пользовательским ID* (uid). Каждый процесс, в свою очередь, ассоциируется ровно с одним uid, обозначающим пользователя, который запустил процесс. Этот идентификатор называется *реальным uid* процесса. Внутри ядра Linux uid является единственной концепцией, представляющей пользователя. Однако пользователи называют себя и обращаются к другим пользователям по *именам пользователей* (username), а не по числовым значениям. Пользовательские имена и соответствующие им uid сохраняются в каталоге /etc/passwd, а библиотечные процедуры сопоставляют предоставленные пользователями имена и соответствующие uid.

При входе в систему пользователь вводит свое имя и пароль в программу входа (login). Если эта программа получает верную пару, состоящую из логина и пароля, то login порождает пользовательскую *оболочку входа*, также указываемую в /etc/passwd, и делает uid оболочки равным uid вошедшего пользователя. Процессы-потомки наследуют uid своих предков.

Uid 0 соответствует особому пользователю, который называется *root*. Этот пользователь обладает особыми привилегиями и может делать в системе практически что угодно. Например, только root-пользователь имеет право изменить uid процесса. Следовательно, программа входа (*login*) работает как *root*.

Кроме реального uid, каждый процесс также обладает *действительным uid*, *сохраненным uid* и *uid файловой системы*. В то время как реальный uid процесса всегда совпадает с uid пользователя, запустившего процесс, действительный uid может изменяться по определенным правилам, чтобы процесс мог выполняться с правами, соответствующими различным пользователям. В сохраненном uid записывается исходный действительный uid; его значение используется при определении, на какие значения действительного uid может переключаться пользователь. Uid файловой системы, который, как правило, равен действительному uid, используется для верификации доступа к файловой системе.

Каждый пользователь может принадлежать к одной или нескольким группам, в частности к *первичной группе*, она же *группа входа в систему*, указанной в */etc/passwd*, а также к нескольким дополнительным группам, которые перечисляются в */etc/group*. Таким образом, каждый процесс ассоциируется с соответствующим *групповым ID (gid)* и имеет *действительный gid*, *сохраненный gid* и *gid файловой системы*. Обычно процессы ассоциированы с пользовательской группой входа, а не с какими-либо дополнительными группами.

Определенные проверки безопасности позволяют процессам выполнять те или иные операции лишь при условии, что процесс соответствует заданным критериям. Исторически это решение в UNIX было жестко детерминированным: процессы с uid 0 имели доступ, а остальные — нет. Сравнительно недавно в Linux эта система была заменена более универсальной, в которой используется концепция *возможностей* (*capabilities*). Вместо обычной двоичной проверки ядро, с учетом возможностей, может предоставлять доступ на базе гораздо более филигранных настроек.

Права доступа

Стандартный механизм прав доступа и безопасности в Linux остался таким же, каким изначально был в UNIX.

Каждый файл ассоциирован с пользователем-владельцем, владеющей группой и тремя наборами битов доступа. Эти биты описывают права пользователя-владельца, владеющей группы и всех остальных, связанные с чтением файла, записью в него и его исполнением. На каждый из этих трех классов действий приходится по три бита, всего имеем девять бит. Информация о владельце файла и правах хранится в индексном дескрипторе файла.

В случае с обычными файлами назначение прав доступа кажется очевидным: они регламентируют возможности открытия файла для чтения, открытия для записи и исполнения файла. Права доступа, связанные с чтением и записью, аналогичны для обычных и специальных файлов, но сама информация, которая считывается из специального файла или записывается в него, зависит от разновидности специального файла. Например, если предоставляется доступ к каталогу для чтения, это означает,

что пользователь может открыть каталог и увидеть список его содержимого. Право записи позволяет добавлять в каталог новые ссылки, а право исполнения разрешает открыть каталог и ввести его название в имя пути. В табл. 1.1 перечислены все девять бит доступа, их восьмеричные значения (распространенный способ представления девяти бит), их текстовые значения (отображаемые командой `ls`) и их права.

Таблица 1.1. Биты доступа и их значения

Бит	Восьмеричное значение	Текстовое значение	Соответствующие права
8	400	r-----	Владелец может читать
7	200	-w-----	Владелец может записывать
6	100	--x-----	Владелец может исполнять
5	040	---r----	Члены группы могут читать
4	020	---w----	Члены группы могут записывать
3	010	---x----	Члены группы могут исполнять
2	004	-----r--	Любой может читать
1	002	-----w-	Любой может записывать
0	001	-----x	Любой может исполнять

Кроме исторически сложившихся в UNIX прав доступа, Linux также поддерживает списки контроля доступа (ACL). Они позволяют предоставлять гораздо более детализированные и точные права, соответственно, более полный контроль над безопасностью. Эти преимущества приобретаются за счет общего усложнения прав доступа и увеличения требуемого для этой информации дискового пространства.

Сигналы

Сигналы — это механизм, обеспечивающий односторонние асинхронные уведомления. Сигнал может быть отправлен от ядра к процессу, от процесса к другому процессу либо от процесса к самому себе. Обычно сигнал сообщает процессу, что произошло какое-либо событие, например возникла ошибка сегментации или пользователь нажал `Ctrl+C`.

Ядро Linux реализует около 30 разновидностей сигналов (точное количество зависит от конкретной архитектуры). Каждый сигнал представлен числовой константой и текстовым названием. Например, сигнал `SIGHUP` используется, чтобы сообщить о зависании терминала. В архитектуре `x86-64` этот сигнал имеет значение 1.

Сигналы *прерывают* исполнение работающего процесса. В результате процесс откладывает любую текущую задачу и немедленно выполняет заранее определенное действие. Все сигналы, за исключением `SIGKILL` (всегда завершает процесс) и `SIGSTOP` (всегда останавливает процесс), оставляют процессам возможность выбора того, что должно произойти после получения конкретного сигнала. Так, процесс может совершить действие, заданное по умолчанию (в частности, завершение процесса, завершение процесса с созданием дампа, остановка процесса или отсутствие действия), — в зависимости от полученного сигнала. Кроме того, процессы могут явно выбирать, будут они обрабатывать сигнал или проигнорируют его.

Проигнорированные сигналы бесшумно удаляются. Если сигнал решено обработать, выполняется предоставляемая пользователем функция, которая называется *обработчиком сигнала*. Программа переходит к выполнению этой функции, как только получит сигнал. Когда обработчик сигнала возвращается, контроль над программой передается обратно инструкции, работа которой была прервана. Сигналы являются асинхронными, поэтому обработчики сигналов не должны срывать выполнение прерванного кода. Таким образом, речь идет о выполнении только функций, которые *безопасны для выполнения в асинхронной среде*, они также называются *сигналобезопасными*.

Межпроцессное взаимодействие

Самые важные задачи операционной системы — это обеспечение обмена информацией между процессами и предоставление процессам возможности уведомлять друг друга о событиях. Ядро Linux реализует большинство из исторически сложившихся в UNIX механизмов межпроцессного взаимодействия. В частности, речь идет о механизмах, определенных и стандартизированных как в SystemV, так и в POSIX. Кроме того, в ядре Linux имеется пара собственных механизмов подобного взаимодействия.

К механизмам межпроцессного взаимодействия, поддерживаемым в Linux, относятся именованные каналы, семафоры, очереди сообщений, разделяемая память и фьютексы.

Заголовки

Системное программирование в Linux всецело зависит от нескольких заголовков. И само ядро, и glibc предоставляют заголовки, применяемые при системном программировании. К ним относятся стандартный джентльменский набор C (например, `<string.h>`) и обычные заголовки UNIX (к примеру, `<unistd.h>`).

Обработка ошибок

Само собой разумеется, что проверка наличия ошибок и их обработка — задача первостепенной важности. В системном программировании ошибка характеризуется возвращаемым значением функции и описывается с помощью специальной переменной `errno`. glibc явно предоставляет поддержку `errno` как для библиотечных, так и для системных вызовов. Абсолютное большинство интерфейсов, рассмотренных в данной книге, используют для сообщения об ошибках именно этот механизм.

Функции уведомляют вызывающую сторону об ошибках посредством специального возвращаемого значения, которое обычно равно `-1` (точное значение, используемое в данном случае, зависит от конкретной функции). Значение ошибки предупреждает вызывающую сторону, что возникла ошибка, но никак не объясняет, почему она произошла. Переменная `errno` используется для выяснения причины ошибки.

Эта переменная объявляется в `<errno.h>` следующим образом:

```
extern int errno;
```

Ее значение является валидным лишь непосредственно после того, как функция, задающая `errno`, указывает на ошибку (обычно это делается путем возвращения `-1`). Дело в том, что после успешного выполнения функции значение этой переменной вполне может быть изменено.

Переменная `errno` доступна для чтения или записи напрямую; это модифицируемое именуемое выражение. Значение `errno` соответствует текстовому описанию конкретной ошибки. Препроцессор `#define` также ассоциирует переменную `errno` с числовым значением. Например, препроцессор определяет `EACCES` равным `1` и означает «доступ запрещен». В табл. 1.2 перечислены стандартные определения и соответствующие им описания ошибок.

Таблица 1.2. Ошибки и их описание

Обозначение препроцессора	Описание
E2BIG	Список аргументов слишком велик
EACCES	Доступ запрещен
EAGAIN	Повторить попытку (ресурс временно недоступен)
EBADF	Недопустимый номер файла
EBUSY	Заняты ресурс или устройство
EINVAL	Процессы-потомки отсутствуют
EDOM	Математический аргумент вне области функции
EEXIST	Файл уже существует
EFAULT	Недопустимый адрес
EFBIG	Файл слишком велик
EINTR	Системный вызов был прерван
EINVAL	Недействительный аргумент
EIO	Ошибка ввода-вывода
EISDIR	Это каталог
EMFILE	Слишком много файлов открыто
EMLINK	Слишком много ссылок
ENFILE	Переполнение таблицы файлов
ENODEV	Такое устройство отсутствует
ENOENT	Такой файл или каталог отсутствует
ENOEXEC	Ошибка формата исполняемого файла
ENOMEM	Недостаточно памяти
ENOSPC	Израсходовано все пространство на устройстве
ENOTDIR	Это не каталог
ENOTTY	Недопустимая операция управления вводом-выводом
ENXIO	Такое устройство или адрес не существует
EPERM	Операция недопустима
EPIPE	Поврежденный конвейер
ERANGE	Результат слишком велик
EROFS	Файловая система доступна только для чтения
ESPIPE	Недействительная операция позиционирования
ESRCH	Такой процесс отсутствует
ETXTBSY	Текстовый файл занят
EXDEV	Неверная ссылка

Библиотека C предоставляет набор функций, позволяющих представлять конкретное значение `errno` в виде соответствующего ему текстового представления. Эта возможность необходима только для отчетности об ошибках и т. п. Проверка наличия ошибок и их обработка может выполняться на основе одних лишь определений препроцессора, а также напрямую через `errno`.

Первая подобная функция такова:

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

Эта функция печатает на устройстве `stderr` (`standard error` — «стандартная ошибка») строковое представление текущей ошибки, взятое из `errno`, добавляя в качестве префикса строку, на которую указывает параметр `str`. Далее следует двоеточие. Для большей информативности имя отказавшей функции следует включать в строку. Например:

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

В библиотеке C также предоставляются функции `strerror()` и `strerror_r()`, прототипированные как:

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

и

```
#include <string.h>
int strerror_r (int errnum, char *buf, size_t len);
```

Первая функция возвращает указатель на строку, описывающую ошибку, выданную `errnum`. Эта строка не может быть изменена приложением, однако это можно сделать с помощью последующих вызовов `perror()` и `strerror()`. Соответственно, такая функция не является потокобезопасной.

Функция `strerror_r()`, напротив, является потокобезопасной. Она заполняет буфер, имеющий длину `len`, на который указывает `buf`. При успешном вызове `strerror_r()` возвращается 0, при сбое — -1. Забавно, но при ошибке она выдает `errno`.

Для некоторых функций к допустимым возвращаемым значениям относится вся область возвращаемого типа. В таких случаях перед вызовом функции `errno` нужно присвоить значение 0, а по завершении проверить ее новое значение (такие функции могут вернуть ненулевое значение `errno`, только когда действительно возникла ошибка). Например:

```
errno = 0;
arg = strtoul (buf, NULL, 0);
if (errno)
    perror ("strtoul");
```

При проверке `errno` мы зачастую забываем, что это значение может быть изменено любым библиотечным или системным вызовом. Например, в следующем коде есть ошибка:

```
if (fsync (fd) == -1) {
    fprintf (stderr, "fsyncfailed!\n");
    if (errno == EIO)
        fprintf (stderr, "I/O error on %d!\n", fd);
}
```

Если необходимо сохранить значение `errno` между вызовами нескольких функций, делаем это:

```
if (fsync (fd) == -1) {
    const int err = errno;
    fprintf (stderr, "fsync failed: %s\n", strerror (errno));
    if (err == EIO) {
        /* если ошибка связана с вводом-выводом — уходим */
        fprintf (stderr, "I/O error on %d!\n", fd);
        exit (EXIT_FAILURE);
    }
}
```

В однопоточных программах `errno` является глобальной переменной, как было показано выше в этом разделе. Однако в многопоточных программах `errno` сохраняется отдельно в каждом потоке для обеспечения потокобезопасности.

Добро пожаловать в системное программирование

В этой главе мы рассмотрели основы системного программирования в Linux и сделали обзор операционной системы Linux с точки зрения программиста. В следующей главе мы обсудим основы файлового ввода-вывода, в частности поговорим о чтении файлов и записи информации в них. Многие интерфейсы в Linux реализованы как файлы, поэтому файловый ввод-вывод имеет большое значение для решения разных задач, а не только для работы с обычными файлами.

Итак, все общие моменты изучены, настало время приступить к настоящему системному программированию. В путь!

2 Файловый ввод-вывод

В этой и трех следующих главах мы поговорим о файлах. В UNIX-подобных системах много сущностей представлено именно в виде файлов, поэтому данные главы критически важны для понимания UNIX. В этой главе мы обсудим основы ввода-вывода, подробно рассмотрим системные вызовы, которые представляют простейшие и наиболее распространенные способы работы с файлами. Следующая глава посвящена стандартному вводу-выводу из библиотеки C. В гл. 4 эта тема получает дальнейшее развитие, в ней обсуждаются более сложные и специализированные интерфейсы файлового ввода-вывода. Наконец, в гл. 8 данные темы получают логическое завершение: в ней рассматриваются манипуляции, выполняемые при работе с файлами и каталогами.

Прежде чем из файла можно будет считать информацию или записать туда новые данные, файл нужно открыть. Ядро поддерживает попроцессный список открытых файлов, называемый *файловой таблицей*. Она индексируется с помощью неотрицательных целых чисел, называемых *файловыми дескрипторами* (часто они именуются сокращенно *fd*). Каждая запись в списке содержит информацию об открытом файле, в частности указатель на хранимую в памяти копию файлового дескриптора и ассоциированных с ним метаданных. К метаданным относятся, в частности, файловая позиция и режимы доступа. Как пользовательское пространство, так и пространство ядра пользуются файловыми дескрипторами как уникальными cookie. При открытии файла возвращается его дескриптор, а при последующих операциях (считывание информации, запись и т. д.) дескриптор файла принимается как первичный аргумент.

В языке C дескрипторы файлов относятся к типу `int`. Факт, что для них не выделен специальный тип, зачастую считается странным, но исторически именно такая ситуация сложилась в UNIX. Для каждого процесса Linux задается максимальное количество файлов, которые он может открыть. Файловые дескрипторы начинаются со значения 0 и могут дойти до значения, на единицу меньше максимального. По умолчанию максимальное значение равно 1024, но при необходимости его можно повысить до 1 048 576. Файловые дескрипторы не могут иметь отрицательных значений, поэтому значение -1 часто применяется для индикации ошибки, полученной от функции. При отсутствии ошибки функция вернула бы валидный (допустимый) дескриптор файла.

Каждый процесс традиционно имеет не менее трех открытых файловых дескрипторов: 0, 1 и 2, если, конечно, процесс явно не закрывает один из них. Файловый дескриптор 0 соответствует *стандартному вводу* (stdin), дескриптор 1 — *стандартному выводу* (stdout), дескриптор 2 — *стандартной ошибке* (stderr). Библиотека C не ссылается непосредственно на эти целые числа, а предоставляет препроцессорные определения STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO для каждого из вышеописанных вариантов соответственно. Как правило, stdin подключен к терминальному устройству ввода (обычно это пользовательская клавиатура), а stdout и stderr — к дисплею терминала. Пользователи могут переназначать эти стандартные файловые дескрипторы и даже направлять по конвейеру вывод одной программы во ввод другой. Именно так оболочка реализует переназначения и конвейеры.

Дескрипторы могут ссылаться не только на обычные файлы, но и на файлы устройств и конвейеры, каталоги и фьютексы, FIFO и сокет. Это соответствует парадигме «все есть файл». Практически любая информация, которую можно читать или записывать, доступна по файловому дескриптору.

По умолчанию процесс-потомок получает копию таблицы файлов своего процесса-предка. Список открытых файлов и режимы доступа к ним, актуальные файловые позиции и другие метаданные не меняются. Однако изменение, связанное с одним процессом, например закрытие файла процессом-потомком, не затрагивает таблиц файлов других процессов. В гл. 5 будет показано, что такое поведение является типичным, но предок и потомок могут совместно использовать таблицу файлов предка (как потоки обычно и делают).

Открытие файлов

Наиболее простой способ доступа к файлу связан с использованием системных вызовов read() и write(). Однако прежде, чем к файлу можно будет получить доступ, его требуется открыть с помощью системного вызова open() или creat(). Когда работа с файлом закончена, его нужно закрыть посредством системного вызова close().

Системный вызов open()

Открытие файла и получение файлового дескриптора осуществляются с помощью системного вызова open():

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

```
int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

Системный вызов open() ассоциирует файл, на который указывает имя пути name с файловым дескриптором, возвращаемым в случае успеха. В качестве файловой

позиции указывается его начало (нуль), и файл открывается для доступа в соответствии с заданными флагами (параметр `flags`).

Флаги для `open()`. Аргумент `flags` — это поразрядное ИЛИ, состоящее из одного или нескольких флагов. Он должен указывать режим доступа, который может иметь одно из следующих значений: `O_RDONLY`, `O_WRONLY` или `O_RDWR`. Эти аргументы соответственно означают, что файл может быть открыт только для чтения, только для записи или одновременно для того и другого.

Например, следующий код открывает каталог `/home/kidd/madagascar` для чтения:

```
int fd;

fd = open ("/home/kidd/madagascar", O_RDONLY);
if (fd == -1)
    /* ошибка */
```

Если файл открыт только для чтения, в него *невозможно* что-либо записать, и наоборот. Процесс, осуществляющий системный вызов `open()`, должен иметь довольно широкие права, чтобы получить запрашиваемый доступ. Например, если файл доступен определенному пользователю только для чтения, то процесс, запущенный этим пользователем, может открыть этот файл как `O_RDONLY`, но не как `O_WRONLY` или `O_RDWR`.

Перед указанием режима доступа задается вариант побитового «ИЛИ» для аргумента `flags`. Вариант описывается одним или несколькими из следующих значений, изменяющих поведение запроса на открытие файла.

- `O_APPEND`. Файл будет открыт в *режиме дозаписи*. Это означает, что перед каждым актом записи файловая позиция будет обновляться и устанавливаться в текущий конец файла. Это происходит, даже когда другой процесс что-то записал в файл уже после последнего акта записи от процесса, выполнившего вызов (таким образом, процесс, осуществивший вызов, уже изменил позицию конца файла).
- `O_ASYNC`. Когда указанный файл станет доступным для чтения или записи, генерируется специальный сигнал (по умолчанию `SIGIO`). Этот флаг может использоваться только при работе с FIFO, каналами, сокетами и терминалами, но не с обычными файлами.
- `O_CLOEXEC`. Задаёт флаг `close-on-exec` для открытого файла. Как только начнется выполнение нового процесса, этот файл будет автоматически закрыт. Таким образом, очевидна необходимость вызова `fcntl()` для задания флага и исключается возможность возникновения условий гонки. Этот флаг доступен лишь в версии ядра Linux 2.6.23 и выше.
- `O_CREAT`. Если файл, обозначаемый именем `name`, не существует, то ядро создаст его. Если же файл уже есть, то этот флаг не даёт никакого эффекта (кроме случаев, в которых также задан `O_EXCL`).
- `O_DIRECT`. Файл будет открыт для непосредственного ввода-вывода.
- `O_DIRECTORY`. Если файл `name` не является каталогом, то вызов `open()` не удастся. Этот флаг используется внутрисистемно библиотечным вызовом `opendir()`.

- `O_EXCL`. При указании вместе с `O_CREAT` этот флаг предотвратит срабатывание вызова `open()`, если файл с именем `name` уже существует. Данный флаг применяется для предотвращения условий гонки при создании файлов. Если `O_CREAT` не указан, то данный флаг не имеет значения.
- `O_LARGEFILE`. Указанный файл будет открыт с использованием 64-битных смещений. Таким образом, становится возможно манипулировать файлами крупнее 2 Гбайт. Для таких операций требуется 64-битная архитектура.
- `O_NOATIME+`. Последнее значение времени доступа к файлу не обновляется при его чтении. Этот флаг удобно применять при индексировании, резервном копировании и использовании других подобных программ, считывающих все файлы в системе. Так предотвращается учет значительной записывающей активности, возникающей при обновлении индексных дескрипторов каждого считываемого файла. Этот флаг доступен лишь в версии ядра Linux 2.6.8 и выше.
- `O_NOCTTY`. Если файл с именем `name` указывает на терминальное устройство (например, `/dev/tty`), это устройство не получает контроля над процессом, даже если в настоящий момент этот процесс не имеет контролирующего устройства. Этот флаг используется редко.
- `O_NOFOLLOW`. Если `name` — это символьная ссылка, то вызов `open()` окончится ошибкой. Как правило, происходит разрешение ссылки, после чего открывается целевой файл. Если остальные компоненты заданного пути также являются ссылками, то вызов все равно удастся. Например, при `name`, равном `/etc/ship/plank.txt`, вызов не удастся в случае, если `plank.txt` является символьной ссылкой. При этом если `plank.txt` не является символьной ссылкой, а `etc` или `ship` являются, то вызов будет выполнен успешно.
- `O_NONBLOCK`. Если это возможно, файл будет открыт в неблокирующем режиме. Ни вызов `open()`, ни какая-либо иная операция не вызовут блокирования процесса (перехода в спящий режим) при вводе-выводе. Такое поведение может быть определено лишь для FIFO.
- `O_SYNC`. Файл будет открыт для синхронного ввода-вывода. Никакие операции записи не завершатся, пока данные физически не окажутся на диске. Обычные действия по считыванию уже протекают синхронно, поэтому данный флаг никак не влияет на чтение. Стандарт POSIX дополнительно определяет `O_DSYNC` и `O_RSYNC`, но в Linux эти флаги эквивалентны `O_SYNC`.
- `O_TRUNC`. Если файл уже существует, является обычным файлом и заданные для него флаги допускают запись, то файл будет усечен до нулевой длины. Флаг `O_TRUNC` для FIFO или терминального устройства игнорируется. Использование с файлами других типов не определено. Указание `O_TRUNC` с `O_RDONLY` также является неопределенным, так как для усечения файла вам требуется разрешение на доступ к нему для записи.

Например, следующий код открывает для записи файл `/home/teach/pearl`. Если файл уже существует, то он будет усечен до нулевой длины. Флаг `O_CREAT` не указывается, когда файл еще не существует, поэтому вызов не состоится:

```
int fd;  
  
fd = open ("/home/teach/pearl", O_WRONLY | O_TRUNC);  
if (fd == -1)  
    /* ошибка */
```

Владельцы новых файлов

Определить, какой пользователь является владельцем файла, довольно просто: `uid` владельца файла является действительным `uid` процесса, создавшего файл.

Определить владеющую группу уже сложнее. Как правило, принято устанавливать значение группы файла в значение действительного `uid` процесса, создавшего файл. Такой подход практикуется в System V; вообще такая модель и такой образ действия очень распространены в Linux и считаются стандартными.

Тем не менее операционная система BSD вносит здесь лишнее усложнение и определяет собственный вариант поведения: группа файлов получает `gid` родительского каталога. Такое поведение можно обеспечить в Linux с помощью одного из параметров времени монтирования¹. Именно такое поведение будет срабатывать в Linux по умолчанию, если для родительского каталога данного файла задан бит смены индикатора группы (`setgid`). Хотя в большинстве систем Linux предпочитается поведение System V (при котором новые файлы получают `gid` родительского каталога), возможность поведения в стиле BSD (при котором новые файлы приобретают `gid` родительского каталога) подразумевает следующее: код, занятый работой с владеющей группой нового файла, должен самостоятельно задать эту группу посредством системного вызова `fchown()` (подробнее об этом — в гл. 8).

К счастью, вам нечасто придется заботиться о том, к какой группе принадлежит файл.

Права доступа новых файлов

Обе описанные выше формы системного вызова `open()` допустимы. Аргумент `mode` игнорируется, если файл не создается. Этот аргумент требуется, если задан флаг `O_CREAT`. Если вы забудете указать аргумент `mode` при использовании `O_CREAT`, то результат будет неопределенным и зачастую неприятным, поэтому лучше не забываете.

При создании файла аргумент `mode` задает права доступа к этому новому файлу. Режим доступа не проверяется при данном конкретном открытии файла, поэтому вы можете выполнить операции, противоречащие присвоенным правам доступа, например открыть для записи файл, для которого указаны права доступа только для чтения.

Аргумент `mode` является знакомой UNIX-последовательностью битов, регламентирующей доступ. К примеру, он может представлять собой восьмеричное значение `0644` (владелец может читать файл и записывать в него информацию, все остальные — только читать). С технической точки зрения POSIX указывает, что точные значения зависят от конкретной реализации. Соответственно, различные

¹ Есть два параметра времени монтирования — `bsdgroups` или `sysvgroups`.

UNIX-подобные системы могут компоновать биты доступа по собственному усмотрению. Однако во всех системах UNIX биты доступа реализованы одинаково, поэтому пусть с технической точки зрения биты 0644 или 0700 и не являются переносимыми, они будут иметь одинаковый эффект в любой системе, с которой вы теоретически можете столкнуться.

Тем не менее, чтобы компенсировать непереносимость позиций битов в режиме доступа, POSIX предусматривает следующий набор констант, которые можно указывать в виде двоичного «ИЛИ» и добавлять к аргументу `mode`:

- `S_IRWXU` — владелец имеет право на чтение, запись и исполнение файла;
- `S_IRUSR` — владелец имеет право на чтение;
- `S_IWUSR` — владелец имеет право на запись;
- `S_IXUSR` — владелец имеет право на исполнение;
- `S_IRWXG` — владеющая группа имеет право на чтение, запись и исполнение файла;
- `S_IRGRP` — владеющая группа имеет право на чтение;
- `S_IWGRP` — владеющая группа имеет право на запись;
- `S_IXGRP` — владеющая группа имеет право на исполнение;
- `S_IRWXO` — любой пользователь имеет право на чтение, запись и исполнение файла;
- `S_IROTH` — любой пользователь имеет право на чтение;
- `S_IWOTH` — любой пользователь имеет право на запись;
- `S_IXOTH` — любой пользователь имеет право на исполнение.

Конкретные биты доступа, попадающие на диск, определяются с помощью двоичного «И», объединяющего аргумент `mode` с пользовательской маской создания файла (`umask`). Такая маска — это специфичный для процесса атрибут, обычно задаваемый интерактивной оболочкой входа. Однако маску можно изменять с помощью вызова `umask()`, позволяющего пользователю модифицировать права доступа, действующие для новых файлов и каталогов. Биты пользовательской маски файла *отключаются* в аргументе `mode`, сообщаемом вызову `open()`. Таким образом, обычная пользовательская маска 022 будет преобразовывать значение 0666, сообщенное `mode`, в 0644. Вы, как системный программист, обычно не учитываете воздействия пользовательских масок, когда задаете права доступа. Смысл подобной маски в том, чтобы сам пользователь мог ограничить права доступа, присваиваемые программами новым файлам.

Рассмотрим пример. Следующий код открывает для записи файл, указанный в `file`. Если файл не существует, то при действующей пользовательской маске 022 он создается с правами доступа 0644 (несмотря на то что в аргументе `mode` указано значение 0664). Если он существует, то этот файл усекается до нулевой длины:

```
int fd;
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
    /* ошибка */
```

Если мы готовы частично пожертвовать переносимостью (как минимум теоретически) в обмен на удобочитаемость, то можем написать следующий код, функционально идентичный предыдущему:

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0664);
if (fd == -1)
    /* ошибка */
```

Функция creat()

Комбинация `O_WRONLY | O_CREAT | O_TRUNC` настолько распространена, что существует специальный системный вызов, обеспечивающий именно такое поведение:

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

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

ПРИМЕЧАНИЕ

Да, в названии этой функции не хватает буквы «е». Кен Томпсон (Ken Thompson), автор UNIX, как-то раз пошутил, что пропуск этой буквы был самым большим промахом, допущенным при создании данной операционной системы.

Следующий типичный вызов `creat()`:

```
int fd;

fd = creat (filename, 0644);
if (fd == -1)
    /* ошибка */

идентичен такому:

int fd;

fd = open (filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == -1)
    /* ошибка */
```

В большинстве архитектур Linux¹ `creat()` является системным вызовом, хотя его можно легко реализовать и в пользовательском пространстве:

```
int creat (const char *name, int mode)
{
    return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

¹ Не забывайте, что системные вызовы определяются в зависимости от архитектуры. Таким образом, в архитектуре x86-64 есть системный вызов `creat()`, а в Alpha — нет. Функцию `creat()` можно, разумеется, использовать в любой архитектуре, но она может быть реализована как библиотечная функция, а не как самостоятельный системный вызов.

Такое дублирование исторически сохранилось с тех пор, когда вызов `open()` имел только два аргумента. В настоящее время `creat()` остается системным вызовом для обеспечения обратной совместимости. Новые архитектуры могут реализовывать `creat()` как библиотечный вызов. Он активирует `open()`, как показано выше.

Возвращаемые значения и коды ошибок

При успешном вызове как `open()`, так и `creat()` возвращаемым значением является дескриптор файла. При ошибке оба этих вызова возвращают `-1` и устанавливают `errno` в нужное значение ошибки (выше (см. гл. 1) подробно обсуждается `errno` и перечисляются возможные значения ошибок). Обработать ошибку при открытии файла несложно, поскольку перед открытием обычно выполняется совсем мало шагов, которые необходимо отменить (либо вообще не совершается никаких). Типичный ответ — это предложение пользователю выбрать новое имя файла или просто завершение программы.

Считывание с помощью read()

Теперь, когда вы знаете, как открывать файл, давайте научимся его читать, а в следующем разделе поговорим о записи.

Самый простой и распространенный механизм чтения связан с использованием системного вызова `read()`, определенного в POSIX.1:

```
#include <unistd.h>
```

```
ssize_t read (int fd, void *buf, size_t len);
```

Каждый вызов считывает не более `len` байт в памяти, на которые содержится указание в `buf`. Считывание происходит с текущим значением смещения, в файле, указанном в `fd`. При успешном вызове возвращается количество байтов, записанных в `buf`. При ошибке вызов возвращает `-1` и устанавливает `errno`. Файловая позиция продвигается в зависимости от того, сколько байтов было считано с `fd`. Если объект, указанный в `fd`, не имеет возможности позиционирования (например, это файл символьного устройства), то считывание всегда начинается с «текущей» позиции.

Принцип использования прост. В данном примере информация считывается из файлового дескриптора `fd` в `word`. Количество байтов, которые будут считаны, равно размеру типа `unsigned long`, который (как минимум в Linux) имеет размер 4 байт на 32-битных системах и 8 байт на 64-битных системах. При возврате `nr` содержит количество считанных байтов или `-1` при ошибке:

```
unsigned long word;  
ssize_t nr;
```

```
/* считываем пару байт в 'word' из 'fd' */  
nr = read (fd, &word, sizeof (unsigned long));  
if (nr == -1)  
    /* ошибка */
```

В данной упрощенной реализации есть две проблемы. Во-первых, вызов может вернуться, считав не все байты из `len`; во-вторых, он может допустить ошибки, требующие исправления, но не проверяемые и не обрабатываемые в коде. К сожалению, код, подобный показанному выше, встречается очень часто. Давайте посмотрим, как его можно улучшить.

Возвращаемые значения

Системный вызов `read()` может вернуть положительное ненулевое значение, меньшее чем `len`. Это может произойти по нескольким причинам: доступно меньше байтов, чем указано в `len`, системный вызов прерван сигналом, конвейер оказался поврежден (если `fd` ссылается на конвейер) и т. д.

Еще одна возможная проблема при использовании `read()` — получение возвращаемого значения 0. Возвращая 0, системный вызов `read()` указывает *конец файла* (end-of-file, EOF). Разумеется, в данном случае никакие байты считаны не будут. EOF не считается ошибкой (соответственно, не дает возвращаемого значения -1). Эта ситуация попросту означает, что файловая позиция превысила последнее допустимое значение смещения в этом файле и читать больше нечего. Однако если сделан вызов на считывание `len` байт, а необходимое количество байтов для считывания отсутствует, то вызов *блокируется* (переходит в спящий режим), пока нужные байты не будут доступны. При этом предполагается, что файл был открыт в неблокирующем режиме. Обратите внимание: эта ситуация отличается от возврата EOF, то есть существует разница между «данные отсутствуют» и «достигнут конец данных». В случае с EOF достигнут конец файла. При блокировании считывающая функция будет дожидаться дополнительных данных — такие ситуации возможны при считывании с сокета или файла устройства.

Некоторые ошибки исправимы. Например, если вызов `read()` прерван сигналом еще до того, как было считано какое-либо количество байтов, то возвращается -1 (0 можно было бы спутать с EOF) и `errno` присваивается значение `EINTR`. В таком случае вы можете и должны повторить считывание.

На самом деле последствия вызова `read()` могут быть разными.

- Вызов возвращает значение, равное `len`. Все `len` считанных байтов сохраняются в `buf`. Именно это нам и требовалось.
- Вызов возвращает значение, меньшее чем `len`, но большее чем нуль. Считанные байты сохраняются в `buf`. Это может случиться потому, что во время выполнения считывания этот процесс был прерван сигналом. Ошибка возникает в середине процесса, становится доступно значение, большее 0, но меньшее `len`. Конец файла был достигнут ранее, чем было прочитано заданное количество байтов. При повторном вызове (в котором соответствующим образом обновлены значения `len` и `buf`) оставшиеся байты будут считаны в оставшуюся часть буфера либо укажут на причину проблемы.
- Вызов возвращает 0. Это означает конец файла. Считывать больше нечего.
- Вызов блокируется, поскольку в текущий момент данные недоступны. Этого не происходит в неблокирующем режиме.

- Вызов возвращает -1, а `errno` присваивается `EINTR`. Это означает, что сигнал был получен прежде, чем были считаны какие-либо байты. Вызов будет повторен.
- Вызов возвращает -1, а `errno` присваивается `EAGAIN`. Это означает, что вызов блокировался потому, что в настоящий момент нет доступных данных, и запрос следует повторить позже. Это происходит только в неблокирующем режиме.
- Вызов возвращает -1, а `errno` присваивается иное значение, нежели `EINTR` или `EAGAIN`. Это означает более серьезную ошибку. Простое повторение вызова в данном случае, скорее всего, не поможет.

Считывание всех байтов

Все описанные возможности подразумевают, что приведенный выше упрощенный вариант использования `read()` не подходит, если вы желаете обработать все ошибки и действительно прочитывать все байты до достижения `len` (или по крайней мере до достижения конца файла). Для такого решения требуется применить цикл и несколько условных операторов:

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("read");
        break;
    }

    len -= ret;
    buf += ret;
}
```

В этом фрагменте кода обрабатываются все пять условий. Цикл считывает `len` байт с актуальной файловой позиции, равной значению `fd`, и записывает их в `buf`. Разумеется, значение `buf` должно быть как минимум равно значению `len`. Чтение продолжается, пока не будут получены все `len` байт или до достижения конца файла. Если прочитано ненулевое количество байтов, которое, однако, меньше `len`, то значение `len` уменьшается на количество прочитанных байтов, `buf` увеличивается на то же количество и вызов повторяется. Если вызов возвращает -1 и значение `errno`, равное `EINTR`, то вызов повторяется без обновления параметров. Если вызов возвращает -1 с любым другим значением `errno`, вызывается `perror()`. Он выводит описание возникшей проблемы в стандартную ошибку, и выполнение цикла прекращается.

Случаи частичного считывания не только допустимы, но и вполне обычны. Из-за программистов, не озаботившихся правильной проверкой и обработкой неполных операций считывания, возникают бесчисленные ошибки. Старайтесь не пополнять их список!

Неблокирующее считывание

Иногда программист не собирается блокировать вызов `read()` при отсутствии доступных данных. Вместо этого он предпочитает немедленный возврат вызова, указывающий, что данных действительно нет. Такой прием называется *неблокирующим вводом-выводом*. Он позволяет приложениям выполнять ввод-вывод, потенциально применимый сразу ко многим файлам, вообще без блокирования, поэтому недостающие данные могут быть взяты из другого файла.

Следовательно, будет целесообразно проверять еще одно значение `errno` — `EAGAIN`. Как было сказано выше, если определенный дескриптор файла был открыт в неблокирующем режиме (вызову `open()` сообщен флаг `O_NONBLOCK`) и данных для считывания не оказалось, то вызов `read()` возвратит `-1` и вместо блокирования установит значение `errno` в `EAGAIN`. При выполнении неблокирующего считывания нужно выполнять проверку на наличие `EAGAIN`, иначе вы рискуете перепутать серьезную ошибку с тривиальным отсутствием данных. Например, вы можете использовать примерно следующий код:

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
    if (errno == EINTR)
        goto start; /* вот незадача */
    if (errno == EAGAIN)
        /* повторить вызов позже */
    else
        /* ошибка */
}
```

ПРИМЕЧАНИЕ

Если бы мы попытались обработать случай с `EAGAIN` так же, как и с `EAGAIN` (с применением `goto start`), это практически не имело бы смысла. Мы могли бы и не применять неблокирующий ввод-вывод. Весь смысл использования неблокирующего ввода-вывода заключается в том, чтобы перехватить `EAGAIN` и выполнить другую полезную работу.

Другие значения ошибок

Другие коды относятся к ошибкам, допускаемым при программировании или (как ЕЮ) к низкоуровневым проблемам. Возможные значения `errno` после неуспешного вызова `read()` таковы:

- `EBADF` — указанный дескриптор файла недействителен или не открыт для чтения;
- `EFAULT` — указатель, предоставленный `buf`, не относится к адресному пространству вызывающего процесса;

- EINVAL — дескриптор файла отображается на объект, не допускающий считывания;
- EIO — возникла ошибка низкоуровневого ввода-вывода.

Ограничения размера для read()

Типы `size_t` и `ssize_t` types предписываются POSIX. Тип `size_t` используется для хранения значений, применяемых для измерения размера в байтах. Тип `ssize_t` — это вариант `size_t`, имеющий знак (отрицательные значения `ssize_t` используются для дополнительной характеристики ошибок). В 32-битных системах базовыми типами C для этих сущностей являются соответственно `unsigned int` и `int`. Эти два типа часто используются вместе, поэтому потенциально более узкий диапазон `ssize_t` лимитирует и размер `size_t`.

Максимальное значение `size_t` равно `SIZE_MAX`, максимальное значение `ssize_t` составляет `SSIZE_MAX`. Если значение `len` превышает `SSIZE_MAX`, то результаты вызова `read()` не определены. В большинстве систем Linux значение `SSIZE_MAX` соответствует `LONG_MAX`, которое, в свою очередь, равно `2 147 483 647` на 32-битной машине. Это относительно большое значение для однократного считывания, но о нем необходимо помнить. Если вы использовали предыдущий считывающий цикл как обобщенный суперсчитыватель, то, возможно, решите сделать нечто подобное:

```
if (len > SSIZE_MAX)
    len = SSIZE_MAX;
```

При вызове `read()` с длиной, равной нулю, вызов немедленно вернется с возвращаемым значением 0.

Запись с помощью write()

Самый простой и распространенный системный вызов для записи информации называется `write()`. Это парный вызов для `read()`, он также определен в POSIX.1:

```
#include <unistd.h>
```

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

При вызове `write()` записывается некоторое количество байтов, меньшее или равное тому, что указано в `count`. Запись начинается с `buf`, установленного в текущую файловую позицию. Ссылка на нужный файл определяется по файловому дескриптору `fd`. Если в основе файла лежит объект, не поддерживающий позиционирования (так, в частности, выглядит ситуация с символьными устройствами), запись всегда начинается с текущей позиции «курсора».

При успешном выполнении возвращается количество записанных байтов, а файловая позиция обновляется соответственно. При ошибке возвращается -1 и устанавливается соответствующее значение `errno`. Вызов `write()` может вернуть 0, но

это возвращаемое значение не имеет никакой специальной трактовки, а всего лишь означает, что было записано 0 байт.

Как и в случае с `read()`, простейший пример использования тривиален:

```
const char *buf = "My ship is solid!";
ssize_t nr;

/* строка, находящаяся в 'buf', записывается в 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
    /* ошибка */
```

Однако, как и в случае с `read()`, вышеуказанный код написан не совсем грамотно. Вызывающая сторона также должна проверять возможное наличие частичной записи:

```
unsigned long word = 1720;
size_t count;
ssize_t nr;

count = sizeof (word);
nr = write (fd, &word, count);
if (nr == -1)
    /* ошибка, проверить errno */
else if (nr != count)
    /* возможна ошибка, но значение 'errno' не установлено */
```

Случаи частичной записи

Системный вызов `write()` выполняет частичную запись не так часто, как системный вызов `read()` — частичное считывание. Кроме того, в случае с `write()` отсутствует условие EOF. В случае с обычными файлами `write()` гарантированно выполняет всю запрошенную запись, если только не возникает ошибка.

Следовательно, при записи информации в обычные файлы не требуется использовать цикл. Однако при работе с файлами других типов, например сокетами, цикл может быть необходим. С его помощью можно гарантировать, что вы *действительно* записали все требуемые байты. Еще одно достоинство использования цикла заключается в том, что второй вызов `write()` может вернуть ошибку, проясняющую, по какой причине при первом вызове удалось осуществить лишь частичную запись (правда, вновь следует оговориться, что такая ситуация не слишком распространена). Вот пример:

```
ssize_t ret, nr;

while (len != 0 && (ret = write (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("write");
    }
}
```

```
        break;
    }

    len -= ret;
    buf += ret;
}
```

Режим дозаписи

Когда дескриптор `fd` открывается в режиме дозаписи (с флагом `O_APPEND`), запись начинается не с текущей позиции дескриптора файла, а с точки, в которой в данный момент находится конец файла.

Предположим, два процесса пытаются записать информацию в конец одного и того же файла. Такая ситуация распространена: например, она может возникать в журнале событий, разделяемом многими процессами. Перед началом работы файловые позиции установлены правильно, каждая из них соответствует концу файла. Первый процесс записывает информацию в конец файла. Если режим дозаписи не используется, то второй процесс, попытавшись сделать то же самое, начнет записывать свои данные уже не в конце файла, а в точке с тем смещением, где конец файла находился до операции записи, выполненной первым процессом. Таким образом, множественные процессы не могут дозаписывать информацию в конец одного и того же файла без явной синхронизации между ними, поскольку при ее отсутствии наступают условия гонки.

Режим дозаписи избавляет нас от таких неудобств. Он гарантирует, что файловая позиция всегда устанавливается в его конец, поэтому все добавляемые информационные фрагменты всегда дозаписываются правильно, даже если они поступают от нескольких записывающих процессов. Эту ситуацию можно сравнить с атомарным обновлением файловой позиции, предшествующим каждому запросу на запись информации. Затем файловая позиция обновляется и устанавливается в точку, соответствующую окончанию последних записанных данных. Это совершенно не мешает работе следующего вызова `write()`, поскольку он обновляет файловую позицию автоматически, но может быть критичным, если по какой-то причине далее следует вызов `read()`, а не `write()`.

При решении определенных задач режим дозаписи целесообразен, например, при упомянутой выше операции записи файлов журналов, но в большинстве случаев он не находит применения.

Неблокирующая запись

Когда дескриптор `fd` открывается в неблокирующем режиме (с флагом `O_NONBLOCK`), а запись в том виде, в котором она выполнена, в нормальных условиях должна быть заблокирована, системный вызов `write()` возвращает `-1` и устанавливает `errno` в значение `EAGAIN`. Запрос следует повторить позже. С обычными файлами этого, как правило, не происходит.

Другие коды ошибок

Следует отдельно упомянуть следующие значения `errno`.

- `EBADF` — указанный дескриптор файла недопустим или не открыт для записи.
- `EFAULT` — указатель, содержащийся в `buf`, ссылается на данные, расположенные вне адресного пространства процесса.
- `EFBIG` — в результате записи файл превысил бы максимум, допустимый для одного процесса по правилам, действующим в системе или во внутренней реализации.
- `EINVAL` — указанный дескриптор файла отображается на объект, не подходящий для записи.
- `EIO` — произошла ошибка низкоуровневого ввода-вывода.
- `ENOSPC` — файловая система, к которой относится указанный дескриптор файла, не обладает достаточным количеством свободного пространства.
- `EPIPE` — указанный дескриптор файла ассоциирован с конвейером или сокетом, чей считывающий конец закрыт. Этот процесс также получит сигнал `SIGPIPE`. Стандартное действие, выполняемое при получении сигнала `SIGPIPE`, — завершение процесса-получателя. Следовательно, такие процессы получают данное значение `errno`, только когда они явно запрашивают, как поступить с этим сигналом — игнорировать, блокировать или обработать его.

Ограничения размера при использовании `write()`

Если значение `count` превышает `SSIZE_MAX`, то результат вызова `write()` не определен.

При вызове `write()` со значением `count`, равным нулю, вызов возвращается немедленно и при этом имеет значение 0.

Поведение `write()`

При возврате вызова, отправленного к `write()`, ядро уже располагает данными, скопированными из предоставленного буфера в буфер ядра, но нет гарантии, что рассматриваемые данные были записаны именно туда, где им следовало оказаться. Действительно, вызовы `write` возвращаются слишком быстро и о записи в нужное место, скорее всего, не может быть и речи. Производительность современных процессоров и жестких дисков несравнима, поэтому на практике данное поведение было бы не только ощутимым, но и весьма неприятным.

На самом деле, после того как приложение из пользовательского пространства осуществляет системный вызов `write()`, ядро Linux выполняет несколько проверок, а потом просто копирует данные в свой буфер. Позже в фоновом режиме ядро собирает все данные из *грязных буферов* — так именуются буферы, содержащие более актуальные данные, чем записанные на диске. После этого ядро оптимальным образом сортирует информацию, добытую из грязных буферов, и записывает их

содержимое на диск (этот процесс называется *отложенной записью*). Таким образом, вызовы write работают быстро и возвращаются практически без задержки. Кроме того, ядро может откладывать такие операции записи на сравнительно неактивные периоды и объединять в «пакеты» несколько отложенных записей.

Подобная запись с отсрочкой не меняет семантики POSIX. Например, если выполняется вызов для считывания только что записанных данных, находящихся в буфере, но отсутствующих на диске, в ответ на запрос поступает именно информация из буфера, а не «устаревшие» данные с диска. Такое поведение, в свою очередь, повышает производительность, поскольку в ответ на запрос о считывании поступают данные из хранимого в памяти кэша, а диск вообще не участвует в операции. Запросы о чтении и записи чередуются верно, а мы получаем ожидаемый результат — конечно, если система не откажет прежде, чем данные окажутся на диске! При аварийном завершении системы наша информация на диск так и не попадет, пусть приложение и будет считать, что запись прошла успешно.

Еще одна проблема, связанная с отложенной записью, заключается в невозможности принудительного *упорядочения записи*. Конечно, приложению может требоваться, чтобы запросы записи обрабатывались именно в том порядке, в котором они попадают на диск. Однако ядро может переупорядочивать запросы записи так, как считает целесообразным в первую очередь для оптимизации производительности. Как правило, это несоответствие приводит к проблемам лишь в случае аварийного завершения работы системы, поскольку в нормальном рабочем процессе содержимое всех буферов рано или поздно попадает в конечную версию файла, содержащуюся на диске, — отложенная запись срабатывает правильно. Абсолютное большинство приложений никак не регулируют порядок записи. На практике упорядочение записи применяется редко, наиболее распространенные примеры подобного рода связаны с базами данных. В базах данных важно обеспечить порядок операций записи, при котором база данных гарантированно не окажется в несогласованном состоянии.

Последнее неудобство, связанное с использованием отложенной записи, — это сообщения системы о тех или иных ошибках ввода-вывода. Любая ошибка ввода-вывода, возникающая при отложенной записи, — допустим, отказ физического диска — не может быть сообщена процессу, который сделал вызов о записи. На самом деле грязные буферы, расположенные в ядре, вообще никак не ассоциированы с процессами. Данные, находящиеся в конкретном грязном буфере, могли быть доставлены туда несколькими процессами, причем выход процесса может произойти как раз в промежутки времени, когда данные уже записаны в буфер, но еще не попали на диск. Кроме того, как в принципе можно сообщить процессу о неуспешной записи *уже постфактум*?

Учитывая все потенциальные проблемы, которые могут возникать при отложенной записи, ядро стремится минимизировать связанные с ней риски. Чтобы гарантировать, что все данные будут своевременно записаны на диск, ядро задает *максимальный возраст буфера* и переписывает все данные из грязных буферов до того, как их срок жизни превысит это значение. Пользователь может сконфигурировать данное значение в /proc/sys/vm/dirty_expire_centisecs. Значение указывается в сантисекундах (сотых долях секунды).

Кроме того, можно принудительно выполнить отложенную запись конкретного файлового буфера и даже синхронизировать все операции записи. Эти вопросы будут рассмотрены в следующем разделе («Синхронизированный ввод-вывод») данной главы.

Далее в этой главе, в разд. «Внутренняя организация ядра», подробно описана подсистема буферов ядра Linux, используемая при отложенной записи.

Синхронизированный ввод-вывод

Конечно, синхронизация ввода-вывода — это важная тема, однако не следует преувеличивать проблемы, связанные с отложенной записью. Буферизация записи обеспечивает *значительное* повышение производительности. Следовательно, любая операционная система, хотя бы претендующая на «современность», реализует отложенную запись именно с применением буферов. Тем не менее в определенных случаях приложению нужно контролировать, когда именно записанные данные попадают на диск. Для таких случаев Linux предоставляет возможности, позволяющие пожертвовать производительностью в пользу синхронизации операций.

fsync() и fdatasync()

Простейший метод, позволяющий гарантировать, что данные окажутся на диске, связан с использованием системного вызова `fsync()`. Этот вызов стандартизирован в POSIX.1b:

```
#include <unistd.h>
```

```
int fsync (int fd);
```

Вызов `fsync()` гарантирует, что все грязные данные, ассоциированные с конкретным файлом, на который отображается дескриптор `fd`, будут записаны на диск. Файловый дескриптор `fd` должен быть открыт для записи. В ходе отложенной записи вызов заносит на диск как данные, так и метаданные. К метаданным относятся, в частности, цифровые отметки о времени создания файла и другие атрибуты, содержащиеся в индексном дескрипторе. Вызов `fsync()` не вернется, пока жесткий диск не сообщит, что все данные и метаданные оказались на диске.

В настоящее время существуют жесткие диски с кэшами (обратной) записи, поэтому вызов `fsync()` не может однозначно определить, оказались ли данные на диске физически к определенному моменту. Жесткий диск может сообщить, что данные были записаны на устройство, но на самом деле они еще могут находиться в кэше записи этого диска. К счастью, все данные, которые оказываются в этом кэше, должны отправляться на диск в срочном порядке.

В Linux присутствует и системный вызов `fdatasync()`:

```
#include <unistd.h>
```

```
int fdatasync (int fd);
```

Этот системный вызов функционально идентичен `fsync()`, с оговоркой, что он лишь сбрасывает на диск данные и метаданные, которые потребуются для корректного доступа к файлу в будущем. Например, после вызова `fdatasync()` на диске окажется информация о размере файла, так как она необходима для верного считывания файла. Этот вызов не гарантирует, что несущественные метаданные будут синхронизированы с диском, поэтому потенциально он быстрее, чем `fsync()`. В большинстве практических случаев метаданные (например, отметка о последнем изменении файла) не считаются существенной частью транзакции, поэтому бывает достаточно применить `fdatasync()` и получить выигрыш в скорости.

ПРИМЕЧАНИЕ

При вызове `fsync()` всегда выполняется как минимум две операции ввода-вывода: в ходе одной из них осуществляется отложенная запись измененных данных, а в ходе другой обновляется временная метка изменения индексного дескриптора. Данные из индексного дескриптора и данные, относящиеся к файлу, могут находиться в несмежных областях диска, поэтому может потребоваться затратная операция позиционирования. Однако в большинстве случаев, когда основная задача сводится к верной передаче транзакции, можно не включать в эту транзакцию метаданные, несущественные для правильного доступа к файлу в будущем. Примером таких метаданных является отметка о последнем изменении файла. По этой причине в большинстве случаев вызов `fdatasync()` является допустимым, а также обеспечивает выигрыш в скорости.

Обе функции используются одинаковым простым способом:

```
int ret;
```

```
ret = fsync (fd);
if (ret == -1)
    /* ошибка */
```

Вот пример с использованием `fdatasync()`:

```
int ret;
```

```
/* аналогично fsync, но на диск не сбрасываются несущественные метаданные */
ret = fdatasync (fd);
if (ret == -1)
    /* ошибка */
```

Ни одна из этих функций не гарантирует, что все обновившиеся записи каталогов, в которых содержится файл, будут синхронизированы на диске. Имеется в виду, что если ссылка на файл недавно была обновлена, то информация из данного файла может успешно попасть на диск, но еще не отразиться в ассоциированной с файлом записи из того или иного каталога. В таком случае файл окажется недоступен. Чтобы гарантировать, что на диске окажутся и все обновления, касающиеся записей в каталогах, `fsync()` нужно вызывать и к дескриптору файла, открытому для каталога, содержащего интересующий нас файл.

Возвращаемые значения и коды ошибок. В случае успеха оба вызова возвращают 0. В противном случае оба вызова возвращают -1 и устанавливают `errno` в одно из следующих трех значений:

- EBADE — указанный дескриптор файла не является допустимым дескриптором, открытым для записи;
- EINVAL — указанный дескриптор файла отображается на объект, не поддерживающий синхронизацию;
- EIO — при синхронизации произошла низкоуровневая ошибка ввода-вывода; здесь мы имеем дело с реальной ошибкой ввода-вывода, более того — именно тут обычно отлавливаются подобные ошибки.

В некоторых версиях Linux вызов `fsync()` может оказаться неуспешным потому, что вызов `fsync()` не реализован в базовой файловой системе этой версии, даже если `fdatasync()` реализован. Некоторые параноидальные приложения пытаются сделать вызов `fdatasync()`, если `fsync()` вернул `EINVAL`, например:

```
if (fsync (fd) == -1) {
    /*
     * Предпочтителен вариант cfsync(), но мы пытаемся сделать и fdatasync(),
     * если fsync() окажется неуспешным — так, на всякий случай.
     */
    if (errno == EINVAL) {
        if (fdatasync (fd) == -1)
            perror ("fdatasync");
    } else
        perror ("fsync");
}
```

POSIX требует использовать `fsync()`, а `fdatasync()` расценивает как необязательный, поэтому системный вызов `fsync()` непременно должен быть реализован для работы с обычными файлами во всех распространенных файловых системах Linux. Файлы необычных типов (например, в которых отсутствуют метаданные, требующие синхронизации) или малораспространенные файловые системы могут, конечно, реализовывать только `fdatasync()`.

sync()

Дедовский системный вызов `sync()` не оптимален для решения описываемых задач, зато гораздо более универсален. Этот вызов обеспечивает синхронизацию *всех* буферов, имеющихся на диске:

```
#include <unistd.h>
```

```
void sync (void);
```

Эта функция не имеет ни параметров, ни возвращаемого значения. Она всегда завершается успешно, и после ее возврата все буферы — содержащие как данные, так и метаданные — гарантированно оказываются на диске¹.

¹ Здесь мы сталкиваемся с теми же подводными камнями, что и раньше: жесткий диск может солгать и сообщить ядру, что содержимое буферов записано на диске, тогда как на самом деле эта информация еще может оставаться в кэше диска.

Согласно действующим стандартам от `sync()` не требуется дожидаться, пока все буферы будут сброшены на диск, и только потом возвращаться. Требуется лишь следующее: вызов должен инициировать процесс отправки на диск содержимого всех буферов, поэтому часто рекомендуется делать вызов `sync()` неоднократно, чтобы гарантировать надежную доставку всех данных на диск. Однако как раз Linux *действительно дожидается*, пока информация из всех буферов отправится на диск, поэтому в данной операционной системе достаточно будет и одного вызова `sync()`.

Единственный практически важный пример использования `sync()` — реализация утилиты *sync*. Приложения, в свою очередь, должны применять `fsync()` и `fdatasync()` для отправки на диск только данных, которые обладают требуемыми файловыми дескрипторами. Обратите внимание: в активно эксплуатируемой системе на завершение `sync()` может потребоваться несколько минут или даже больше времени.

Флаг `O_SYNC`

Флаг `O_SYNC` может быть передан вызову `open()`. Этот флаг означает, что все операции ввода-вывода, осуществляемые с этим файлом, должны быть синхронизированы:

```
int fd;

fd = open (file, O_WRONLY | O_SYNC);
if (fd == -1) {
    perror ("open");
    return -1;
}
```

Запросы на считывание всегда синхронизированы. Если бы такая синхронизация отсутствовала, то мы не могли бы судить о допустимости данных, считанных из предоставленного буфера. Тем не менее, как уже упоминалось выше, вызовы `write()`, как правило, не синхронизируются. Нет никакой связи между возвратом вызова и отправкой данных на диск. Флаг `O_SYNC` принудительно устанавливает такую связь, гарантируя, что вызовы `write()` будут выполнять синхронизированный ввод-вывод.

Флаг `O_SYNC` можно рассмотреть в следующем ключе: он принудительно выполняет неявный вызов `fsync()` после каждой операции `write()` перед возвратом вызова. Этот флаг обеспечивает именно такую семантику, хотя ядро реализует вызов `O_SYNC` немного эффективнее.

При использовании `O_SYNC` несколько ухудшаются два показателя операций записи: *время, затрачиваемое ядром*, и *пользовательское время*. Это соответственно периоды, затраченные на работу в пространстве ядра и в пользовательском пространстве. Более того, в зависимости от размера записываемого файла `O_SYNC` общее истекшее время также может увеличиваться на один-два порядка, поскольку все *время ожидания при вводе-выводе* (время, необходимое для завершения операций ввода-вывода) суммируется со временем, затрачиваемым на работу процесса. Налицо огромное увеличение издержек, поэтому синхронизированный ввод-вывод следует использовать только при отсутствии альтернатив.

Как правило, если приложению требуется гарантировать, что информация, записанная с помощью `write()`, попала на диск, обычно используются вызовы `fsync()` или `fdatasync()`. С ними, как правило, связано меньше издержек, чем с `O_SYNC`, так как их требуется вызывать не столь часто (то есть только после завершения определенных критически важных операций).

Флаги `O_DSYNC` и `O_RSYNC`

Стандарт POSIX определяет еще два флага для вызова `open()`, связанных с синхронизированным вводом-выводом, — `O_DSYNC` и `O_RSYNC`. В Linux эти флаги определяются как синонимичные `O_SYNC`, они предоставляют аналогичное поведение.

Флаг `O_DSYNC` указывает, что после каждой операции должны синхронизироваться только обычные данные, но не метаданные. Ситуацию можно сравнить с неявным вызовом `fdatasync()` после каждого запроса на запись. `O_SYNC` предоставляет более надежные гарантии, поэтому совмещение `O_DSYNC` с ним не влечет за собой никакого функционального ухудшения. Возможно лишь потенциальное снижение производительности, связанное с тем, что `O_SYNC` предъявляет к системе более строгие требования.

Флаг `O_RSYNC` требует синхронизации запросов как на считывание, так и на запись. Его нужно использовать вместе с `O_SYNC` или `O_DSYNC`. Как было сказано выше, операции считывания синхронизируются изначально — если уж на то пошло, они не возвращаются, пока получают какую-либо полезную информацию, которую можно будет предоставить пользователю. Флаг `O_RSYNC` регламентирует, что все побочные эффекты операции считывания также должны быть синхронизированы. Это означает, что обновления метаданных, происходящие в результате считывания, должны быть записаны на диск прежде, чем вернется вызов. На практике данное требование обычно всего лишь означает, что до возврата вызова `read()` должно быть обновлено время доступа к файлу, фиксируемое в копии индексного дескриптора, находящейся на диске. В Linux флаг `O_RSYNC` определяется как аналогичный `O_SYNC`, пусть это и кажется нецелесообразным (ведь `O_RSYNC` не являются подмножеством `O_SYNC`, в отличие от `O_DSYNC`, которые таким подмножеством являются). В настоящее время в Linux отсутствует способ, позволяющий обеспечить функциональность `O_RSYNC`. Максимум, что может сделать разработчик, — инициировать `fdatasync()` после каждого вызова `read()`. Правда, такое поведение требуется редко.

Непосредственный ввод-вывод

Ядро Linux, как и ядро любых других современных операционных систем, реализует между устройствами и приложениями сложный уровень архитектуры, отвечающий за кэширование, буферизацию и управление вводом-выводом (см. разд. «Внутренняя организация ядра» данной главы). Высокопроизводительным приложениям, возможно, потребуется обходить этот сложный уровень и применять собственную систему управления вводом-выводом. Правда, обычно эксплуатация такой системы

не оправдывает затрачиваемых на нее усилий. Вероятно, инструменты, которые уже доступны вам на уровне операционной системы, позволят обеспечить значительно более высокую производительность, чем подобные им существующие на уровне приложений. Тем не менее в системах баз данных обычно предпочтительнее использовать собственный механизм кэширования и свести к минимуму участие операционной системы в рабочих процессах, насколько это возможно.

Когда мы снабжаем вызов `open()` флагом `O_DIRECT`, мы предписываем ядру свести к минимуму активность управления вводом-выводом. При наличии этого флага операции ввода-вывода будут инициироваться непосредственно из буферов пользовательского пространства на устройство, минуя страничный кэш. Все операции ввода-вывода станут синхронными, вызовы не будут возвращаться до завершения этих действий.

При выполнении непосредственного ввода-вывода длина запроса, выравнивание буфера и смещения файлов должны представлять собой целочисленные значения, кратные размеру сектора на базовом устройстве. Как правило, размер сектора составляет 512 байт. До выхода версии ядра Linux 2.6 это требование было еще строже. Так, в версии 2.4 все эти значения должны были быть кратны размеру логического блока файловой системы (обычно 4 Кбайт). Для обеспечения совместимости приложения должны соответствовать более крупной (и потенциально менее удобной) величине — размеру логического блока.

Заккрытие файлов

После того как программа завершит работу с дескриптором файла, она может разорвать связь, существующую между дескриптором и файлом, который с ним ассоциирован. Это делается с помощью системного вызова `close()`:

```
#include <unistd.h>
```

```
int close (int fd);
```

Вызов `close()` отменяет отображение открытого файлового дескриптора `fd` и разрывает связь между файлом и процессом. Данный дескриптор файла больше не является допустимым, и ядро свободно может переиспользовать его как возвращаемое значение для последующих вызовов `open()` или `creat()`. При успешном выполнении вызов `close()` возвращает 0. При ошибке он возвращает -1 и устанавливает `errno` в соответствующее значение. Пример использования прост:

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

Обратите внимание: закрытие файла никак не связано с актом сбрасывания файла на диск. Чтобы перед закрытием файла убедиться, что он уже присутствует на диске, в приложении необходимо задействовать одну из возможностей синхронизации, рассмотренных выше (см. разд. «Синхронизированный ввод-вывод» данной главы).

Правда, с закрытием файла связаны некоторые побочные эффекты. Когда закрывается последний из открытых файловых дескрипторов, ссылавшийся на данный файл, в ядре высвобождается структура данных, с помощью которой обеспечивалось представление файла. Когда эта структура высвобождается, она «расцепляется» с хранимой в памяти копией индексного дескриптора, ассоциированного с файлом. Если индексный дескриптор ни с чем больше не связан, он также может быть высвобожден из памяти (конечно, этот дескриптор может остаться доступным, так как ядро кэширует индексные дескрипторы из соображений производительности, но это не гарантируется). В некоторых случаях разрывается связь между файлом и диском, но файл остается открытым вплоть до этого разрыва. В таком случае физического удаления данного файла с диска не происходит, пока файл не будет закрыт, а его индексный дескриптор удален из памяти, поэтому вызов `close()` также может привести к тому, что ни с чем не связанный файл окажется физически удаленным с диска.

Значения ошибок

Распространена порочная практика — не проверять возвращаемое значение `close()`. В результате можно упустить критическое условие, приводящее к ошибке, так как подобные ошибки, связанные с отложенными операциями, могут не проявиться вплоть до момента, как о них сообщит `close()`.

При таком отказе вы можете встретить несколько возможных значений `errno`. Кроме `EBADF` (заданный дескриптор файла оказался недопустимым), наиболее важным значением ошибки является `EIO`. Оно соответствует низкоуровневой ошибке ввода-вывода, которая может быть никак не связана с самим актом закрытия. Если файловый дескриптор допустим, то при выдаче сообщения об ошибке он всегда закрывается, независимо от того, какая именно ошибка произошла. Ассоциированные с ним структуры данных высвобождаются.

Вызов `close()` никогда не возвращает `EINTR`, хотя POSIX это допускает. Разработчики ядра Linux знают, что делают.

Позиционирование с помощью `lseek()`

Как правило, операции ввода-вывода происходят в файле линейно и все позиционирование сводится к неявным обновлениям файловой позиции, происходящим в результате операций чтения и записи. Однако некоторые приложения перемещаются по файлу скачками, выполняя произвольный, а не линейный доступ к данным. Системный вызов `lseek()` предназначен для установки в заданное значение файловой позиции конкретного файлового дескриптора. Этот вызов не осуществляет никаких других операций, кроме обновления файловой позиции, в частности не иницирует каких-либо действий, связанных с вводом-выводом.

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek (int fd, off_t pos, int origin);
```

Поведение вызова lseek() зависит от аргумента origin, который может иметь одно из следующих значений.

- SEEK_CUR — текущая файловая позиция дескриптора fd установлена в его текущее значение плюс pos. Последний может иметь отрицательное, положительное или нулевое значение. Если pos равен нулю, то возвращается текущее значение файловой позиции.
- SEEK_END — текущая файловая позиция дескриптора fd установлена в текущее значение длины файла плюс pos, который может иметь отрицательное, положительное или нулевое значение. Если pos равен нулю, то смещение устанавливается в конец файла.
- SEEK_SET — текущая файловая позиция дескриптора fd установлена в pos. Если pos равен нулю, то смещение устанавливается в начало файла.

В случае успеха этот вызов возвращает новую файловую позицию. При ошибке он возвращает -1 и присваивает errno соответствующее значение.

В следующем примере файловая позиция дескриптора fd получает значение 1825:

```
off_t ret;

ret = lseek (fd, (off_t) 1825, SEEK_SET);
if (ret == (off_t) -1)
    /* ошибка */
```

В качестве альтернативы можно установить файловую позицию дескриптора fd в конец файла:

```
off_t ret;

ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) -1)
    /* ошибка */
```

Вызов lseek() возвращает обновленную файловую позицию, поэтому его можно использовать для поиска текущей файловой позиции. Нужно установить значение SEEK_CUR в ноль:

```
int pos;

pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
    /* ошибка */
else
    /* 'pos' — это текущая позиция fd */
```

По состоянию на настоящий момент `lseek()` чаще всего применяется для поиска относительно начала файла, конца файла или для определения текущей позиции файлового дескриптора.

Поиск с выходом за пределы файла

Можно указать `lseek()` переставить указатель файловой позиции за пределы файла (дальше его конечной точки). Например, следующий код устанавливает позицию на 1688 байт после конца файла, на который отображается дескриптор `fd`:

```
int ret;

ret = lseek (fd, (off_t) 1688, SEEK_END);
if (ret == (off_t) -1)
    /* ошибка */
```

Само по себе позиционирование с выходом за пределы файла не дает результата — запрос на считывание такой новой файловой позиции вернет значение EOF (конец файла). Однако если затем сделать запрос на запись, указав такую конечную позицию, то между старым и новым значениями длины файла будет создано дополнительное пространство, которое программа заполнит нулями.

Такое заполнение нулями называется *дырой*. В UNIX-подобных файловых системах дыры не занимают на диске никакого пространства. Таким образом, общий размер всех файлов, содержащихся в файловой системе, может превышать физический размер диска. Файлы, содержащие дыры, называются *разреженными*. При использовании разреженных файлов можно экономить значительное пространство на диске, а также оптимизировать производительность, ведь при манипулировании дырами не происходит никакого физического ввода-вывода.

Запрос на считывание фрагмента файла, полностью находящегося в пределах дыры, вернет соответствующее количество нулей.

Значения ошибок. При ошибке `lseek()` возвращает `-1` и присваивает `errno` одно из следующих значений.

- `EBADF` — указанное значение дескриптора не ссылается на открытый файловый дескриптор.
- `EINVAL` — значение аргумента `origin` не является `SEEK_SET`, `SEEK_CUR` или `SEEK_END` либо результирующая файловая позиция получится отрицательной. Факт, что `EINVAL` может соответствовать обоим подобным ошибкам, конечно, неудобен. В первом случае мы наверняка имеем дело с ошибкой времени компиляции, а во втором, возможно, наличествует более серьезная ошибка в логике исполнения.
- `EOVERFLOW` — результирующее файловое смещение не может быть представлено как `off_t`. Такая ситуация может возникнуть лишь в 32-битных архитектурах. В момент получения такой ошибки файловая позиция *уже* обновлена; данная ошибка означает лишь, что новую позицию файла невозможно вернуть.
- `ESPIPE` — указанный дескриптор файла ассоциирован с объектом, который не поддерживает позиционирования, например с конвейером, FIFO или сокетом.

Ограничения

Максимальные значения файловых позиций ограничиваются типом `off_t`. В большинстве машинных архитектур он определяется как тип `long` языка C. В Linux размер этого вида обычно равен *размеру машинного слова*. Как правило, под размером машинного слова понимается размер универсальных аппаратных регистров в конкретной архитектуре. Однако на внутрисистемном уровне ядро хранит файловые смещения в типах `long` языка C. На машинах с 64-битной архитектурой это не представляет никаких проблем, но такая ситуация означает, что на 32-битных машинах ошибка `EOVERFLOW` может возникать при выполнении операций относительного поиска.

Позиционное чтение и запись

Linux позволяет использовать вместо `lseek()` два варианта системных вызовов — `read()` и `write()`. Оба эти вызова получают файловую позицию, с которой требуется начинать чтение или запись. По завершении работы эти вызовы *не обновляют* позицию файла.

Данная форма считывания называется `pread()`:

```
#define _XOPEN_SOURCE 500
```

```
#include <unistd.h>
```

```
ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

Этот вызов считывает до `count` байт в `buf`, начиная от файлового дескриптора `fd` на файловой позиции `pos`.

Данная форма записи называется `pwrite()`:

```
#define _XOPEN_SOURCE 500
```

```
#include <unistd.h>
```

```
ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

Этот вызов записывает до `count` байт в `buf`, начиная от файлового дескриптора `fd` на файловой позиции `pos`.

Функционально эти вызовы практически идентичны своим собратьям без букв `p`, за исключением того, что они игнорируют текущую позицию файла. Вместо использования `pos` они прибегают к значению, указанному в `pos`. Кроме того, выполнив свою работу, они не обновляют позицию файла. Таким образом, если смешивать позиционные вызовы с обычными `read()` и `write()`, последние могут полностью испортить всю работу, выполненную позиционными вызовами.

Оба позиционных вызова применимы только с файловыми дескрипторами, которые поддерживают поиск, в частности с обычными файлами. С семантической точки зрения их можно сравнить с вызовами `read()` или `write()`, которым предшествует

вызов `lseek()`, но с тремя оговорками. Во-первых, позиционные вызовы проще в использовании, особенно если нас интересует какая-либо хитрая манипуляция, например перемещение по файлу в обратном направлении или произвольном порядке. Во-вторых, завершив работу, они не обновляют указатель на файл. Наконец, самое важное — они исключают возможность условий гонки, которые могут возникнуть при использовании `lseek()`.

Потоки совместно используют файловые таблицы, и текущая файловая позиция хранится в такой разделяемой таблице, поэтому один поток программы может обновить файловую позицию уже после вызова `lseek()`, поступившего к файлу от другого потока, но прежде, чем закончится выполнение операции считывания или записи. Налицо потенциальные условия гонки, если в вашей программе присутствуют два и более потока, оперирующие одним и тем же файловым дескриптором. Таких условий гонки можно избежать, работая с системными вызовами `pread()` и `pwrite()`.

Значения ошибок. В случае успеха оба вызова возвращают количество байтов, которые соответственно были прочитаны или записаны. Возвращаемое значение 0, полученное от `pread()`, означает конец файла; возвращаемое значение 0 от `pwrite()` указывает, что вызов ничего не записал. При ошибке оба вызова возвращают -1 и устанавливают `errno` соответствующее значение. В случае `pread()` это может быть любое допустимое значение `errno` для `read()` или `lseek()`. В случае `pwrite()` это может быть любое допустимое значение `errno` для `write()` или `lseek()`.

Усечение файлов

В Linux предоставляется два системных вызова для усечения длины файла. Оба они определены и обязательны (в той или иной степени) согласно различным стандартам POSIX. Вот эти вызовы:

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);

и

#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

Оба системных вызова выполняют усечение заданного файла до длины, указанной в `len`. Системный вызов `ftruncate()` оперирует файловым дескриптором `fd`, который должен быть открыт для записи. Системный вызов `truncate()` оперирует именем файла, указанным в `path`, причем этот файл должен быть пригоден для записи. Оба вызова при успешном выполнении возвращают 0. При ошибке оба вызова возвращают -1 и присваивают `errno` соответствующее значение.

Как правило, эти системные вызовы используются для усечения файла до длины, меньшей чем текущая. При успешном возврате вызова длина файла равна `len`.

Все данные, прежде находившиеся между `len` и неусеченным показателем длины, удаляются и становятся недоступны для запросов на считывание.

Эти функции могут также выполнять «усечение» файла с увеличением его размера, как при комбинации позиционирования и записи, описанной выше (см. разд. «Позиционирование с выходом за пределы файла» данной главы). Дополнительные байты заполняются нулями.

Ни при одной из этих операций файловая позиция не обновляется.

Рассмотрим, например, файл `pirate.txt` длиной 74 байт со следующим содержанием:

```
Edward Teach was a notorious English pirate.  
He was nicknamed Blackbeard
```

Не выходя из каталога с этим файлом, запустим следующую программу:

```
#include <unistd.h>  
#include <stdio.h>  
  
int main()  
{  
    int ret;  
  
    ret = truncate ("./pirate.txt", 45);  
    if (ret == -1) {  
        perror ("truncate");  
        return -1;  
    }  
  
    return 0;  
}
```

В результате получаем файл следующего содержания длиной 45 байт:

```
Edward Teach was a notorious English pirate.
```

Мультиплексный ввод-вывод

Зачастую приложениям приходится блокироваться на нескольких файловых дескрипторах, перемежая ввод-вывод от клавиатуры (`stdin`), межпроцессное взаимодействие и оперируя при этом несколькими файлами. Однако современные приложения с событийно управляемыми графическими пользовательскими интерфейсами (GUI) могут справляться без малого с сотнями событий, ожидающими обработки, так как в этих интерфейсах используется *основной цикл*¹.

¹ Основные циклы должны быть знакомы каждому, кто когда-либо писал приложения с графическими интерфейсами. Например, в приложениях системы GNOME используется основной цикл, предоставляемый `glib` — базовой библиотекой GNOME. Основной цикл позволяет отслеживать множество событий и реагировать на них из одной и той же точки блокирования.

Не прибегая к потокам — в сущности, обслуживая каждый файловый дескриптор отдельно, — одиночный процесс, разумеется, может фиксироваться только на одном дескрипторе в каждый момент времени. Работать с множественными файловыми дескрипторами удобно, если они всегда готовы к считыванию или записи. Однако если программа встретит файловый дескриптор, который еще не готов к взаимодействию (допустим, мы выполнили системный вызов `read()`, а данные для считывания пока отсутствуют), то процесс блокируется и не сможет заняться работой с какими-либо другими файловыми дескрипторами. Он может блокироваться даже на несколько секунд, из-за чего приложение станет неэффективным и будет только раздражать пользователя. Более того, если нужные для файлового дескриптора данные так и не появятся, то блокировка может длиться вечно. Операции ввода-вывода, связанные с различными файловыми дескрипторами, зачастую взаимосвязаны (вспомните, например, работу с конвейерами), поэтому один из файловых дескрипторов вполне может оставаться не готовым к работе, пока не будет обслужен другой. В частности, при работе с сетевыми приложениями, в которых одновременно бывает открыто большое количество сокетов, эта проблема может стать весьма серьезной.

Допустим, произошла блокировка на файловом дескрипторе, относящемся к межпроцессному взаимодействию. В то же время в режиме ожидания остаются данные, введенные с клавиатуры (`stdin`). Пока заблокированный файловый дескриптор, отвечающий за межпроцессное взаимодействие, не вернет данные, приложение так и не узнает, что еще остаются необработанные данные с клавиатуры. Однако что делать, если возврата от заблокированной операции так и не произойдет?

Ранее в данной главе мы обсуждали неблокирующий ввод-вывод в качестве возможного решения этой проблемы. Приложения, работающие в режиме неблокирующего ввода-вывода, способны выдавать запросы на ввод-вывод, которые в случае подвисания не блокируют всю работу, а возвращают особое условие ошибки. Это решение неэффективно по двум причинам. Во-первых, процессу приходится постоянно осуществлять операции ввода-вывода в каком-то произвольном порядке, дожидаясь, пока один из открытых файловых дескрипторов не будет готов выполнить операцию ввода-вывода. Это некачественная конструкция программы. Во-вторых, программа работала бы эффективнее, если бы могла ненадолго засыпать, высвобождая процессор для решения других задач. Просыпаться программа должна, только когда один файловый дескриптор или более будут готовы к обработке ввода-вывода.

Пора познакомиться с *мультиплексным вводом-выводом*. Мультиплексный ввод-вывод позволяет приложениям параллельно блокировать несколько файловых дескрипторов и получать уведомления, как только любой из них будет готов к чтению или записи без блокировки, поэтому мультиплексный ввод-вывод оказывает настоящим стержнем приложения, выстраиваемым примерно по следующему принципу.

1. Мультиплексный ввод-вывод: сообщите мне, когда любой из этих файловых дескрипторов будет готов к операции ввода-вывода.
2. Ни один не готов? Перехожу в спящий режим до готовности одного или нескольких дескрипторов.

3. Проснулся! Где готовый дескриптор?
4. Обрабатываю без блокировки все файловые дескрипторы, готовые к вводу-выводу.
5. Возвращаюсь к шагу 1.

В Linux предоставляется три сущности для различных вариантов мультиплексного ввода-вывода. Это интерфейсы для выбора (`select`), опроса (`poll`) и расширенного опроса (`epoll`). Здесь мы рассмотрим первые два решения. Последний вариант — продвинутый, специфичный для Linux. Его мы обсудим в гл. 4.

select()

Системный вызов `select()` обеспечивает механизм для реализации синхронного мультиплексного ввода-вывода:

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

```
int select (int n,  
            fd_set *readfds,  
            fd_set *writefds,  
            fd_set *exceptfds,  
            struct timeval *timeout);
```

```
FD_CLR(int fd, fd_set *set);  
FD_ISSET(int fd, fd_set *set);  
FD_SET(int fd, fd_set *set);  
FD_ZERO(fd_set *set);
```

Вызов к `select()` блокируется, пока указанные файловые дескрипторы не будут готовы к выполнению ввода-вывода либо пока не истечет необязательный интервал задержки.

Отслеживаемые файловые дескрипторы делятся на три группы. Дескрипторы из каждой группы ждут событий определенного типа. Файловые дескрипторы, перечисленные в `readfds`, отслеживают, появились ли данные, доступные для чтения, то есть они проверяют, удастся ли совершить операцию считывания без блокировки. Файловые дескрипторы, перечисленные в группе `writefds`, аналогичным образом ждут возможности совершить неблокирующую операцию записи. Наконец, файловые дескрипторы из группы `exceptfds` следят, не было ли исключения либо не появились ли в доступе внеполосные данные (в таком состоянии могут находиться только сокеты). Одна из групп может иметь значение `NULL`; это означает, что `select()` не отслеживает события данного вида.

При успешном возврате каждая группа изменяется таким образом, что в ней остаются только дескрипторы, готовые к вводу-выводу определенного типа, соответствующего конкретной группе. Предположим, у нас есть два файловых дескриптора со значениями 7 и 9, которые относятся к группе `readfds`. Возвращается вызов. Если к этому моменту дескриптор 7 не покинул эту группу, то, следовательно, он готов к неблокирующему считыванию. Если 9 уже не находится в этой

группе, то он, вероятно, не сможет выполнить считывание без блокировки. Под «вероятно» здесь подразумевается, что данные для считывания могли стать доступны уже после того, как произошел возврат вызова. В таком случае при последующем вызове `select()` этот файловый дескриптор будет расцениваться как готовый для считывания¹.

Первый параметр, `n`, равен наивысшему значению файлового дескриптора, присутствующему во всех группах, плюс 1. Следовательно, сторона, вызывающая `select()`, должна проверить, какой из заданных файловых дескрипторов имеет наивысшее значение, а затем передать сумму (это значение плюс 1) первому параметру.

Параметр `timeout` является указателем на структуру `timeval`, определяемую следующим образом:

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

```
struct timeval {
    long tv_sec;          /* секунды */
    long tv_usec;         /* микросекунды */
};
```

Если этот параметр не равен `NULL`, то вызов `select()` вернется через `tv_sec` секунд и `tv_usec` микросекунд, даже если ни один из файловых дескрипторов не будет готов к вводу-выводу. После возврата состояние этой структуры в различных UNIX-подобных системах не определено, поэтому должно инициализироваться заново (вместе с группами файловых дескрипторов) перед каждой активацией. На самом деле современные версии Linux автоматически изменяют этот параметр, устанавливая значения в оставшееся количество времени. Таким образом, если величина задержки была установлена в 5 секунд и истекло 3 секунды с момента, как файловый дескриптор перешел в состояние готовности, `tv.tv_sec` после возврата вызова будет иметь значение 2.

Если оба значения задержки равны нулю, то вызов вернется немедленно, сообщив обо всех событиях, которые находились в режиме ожидания на момент вызова. Однако этот вызов не будет дожидаться никаких последующих событий.

Манипуляции с файловыми дескрипторами осуществляются не напрямую, а посредством вспомогательных макрокоманд. Благодаря этому системы UNIX могут реализовывать группы дескрипторов так, как считается целесообразным. В большинстве систем, однако, эти группы реализованы как простые битовые массивы.

`FD_ZERO` удаляет все файловые дескрипторы из указанной группы. Эта команда должна вызываться перед каждой активизацией `select()`:

```
fd_set writefds;
```

```
FD_ZERO(&writefds);
```

¹ Дело в том, что вызовы `select()` и `poll()` являются обрабатываемыми по уровню, а не по фронту. Вызов `epoll()`, о котором мы поговорим в гл. 4, может работать в любом из этих режимов. Операции, обрабатываемые по фронту, проще, но если пользоваться ими неаккуратно, то некоторые события могут быть пропущены.

FD_SET добавляет файловый дескриптор в указанную группу, а FD_CLR удаляет дескриптор из указанной группы:

```
FD_SET(fd, &writefds);    /* добавляем 'fd' к группе */
FD_CLR(fd, &writefds);    /* ой, удаляем 'fd' из группы */
```

В качественном коде практически не должно встречаться случаев, в которых приходится воспользоваться FD_CLR, поэтому данная команда действительно используется очень редко.

FD_ISSET проверяет, принадлежит ли определенный файловый дескриптор к конкретной группе. Если дескриптор относится к группе, то эта команда возвращает ненулевое целое число, а если не относится, возвращает 0. FD_ISSET используется после возврата вызова от select(). С его помощью мы проверяем, готов ли определенный файловый дескриптор к действию:

```
if (FD_ISSET(fd, &readfds))
    /* 'fd' доступен для неблокирующего считывания! */
```

Группы файловых дескрипторов создаются в статическом режиме, поэтому устанавливается лимит на максимальное количество дескрипторов, которые могут находиться в группах. Кроме того, задается максимальное значение, которое может иметь какой-либо из этих дескрипторов. Оба значения определяются командой FD_SETSIZE. В Linux данное значение равно 1024. Далее в этой главе мы рассмотрим случаи отклонения от данного максимального значения.

Возвращаемые значения и коды ошибок

В случае успеха select() возвращает количество файловых дескрипторов, готовых для ввода-вывода, во всех трех группах. Если была задана задержка, то возвращаемое значение может быть равно нулю. При ошибке вызов возвращает значение -1, а errno устанавливается в одно из следующих значений:

- EBADF — в одной из трех групп оказался недопустимый файловый дескриптор;
- EINVAL — сигнал был получен в период ожидания, и вызов можно повторить;
- ENOMEM — запрос не был выполнен, так как не был доступен достаточный объем памяти.

Пример использования select()

Рассмотрим пример тривиальной, но полностью функциональной программы. На нем вы увидите использование вызова select(). Эта программа блокируется, дожидаясь поступления ввода на stdin, блокировка может продолжаться вплоть до 5 секунд. Эта программа отслеживает лишь один файловый дескриптор, поэтому здесь отсутствует мультиплексный ввод-вывод как таковой. Однако данный пример должен прояснить использование этого системного вызова:

```
#include <stdio.h>
#include <sys/time.h>
```

```

#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5      /* установка тайм-аута в секундах */
#define BUF_LEN 1024   /* длина буфера считывания в байтах */

int main (void)
{
    struct timeval tv;
    fd_set readfds;
    int ret;

    /* Ждем ввода на stdin. */
    FD_ZERO(&readfds);
    FD_SET(STDIN_FILENO, &readfds);

    /* Ожидаем не дольше 5 секунд. */
    tv.tv_sec = TIMEOUT;
    tv.tv_usec = 0;

    /* Хорошо, а теперь блокировка! */
    ret = select (STDIN_FILENO + 1,
                  &readfds,
                  NULL,
                  NULL,
                  &tv);

    if (ret == -1) {
        perror ("select");
        return 1;
    } else if (!ret) {
        printf ("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    /*
     * Готов ли наш файловый дескриптор к считыванию?
     * (Должен быть готов, так как это был единственный fd,
     * предоставленный нами, а вызов вернулся ненулевым,
     * но мы же тут просто развлекаемся.)
     */
    if (FD_ISSET(STDIN_FILENO, &readfds)) {
        char buf[BUF_LEN+1];
        int len;

        /* блокировка гарантированно отсутствует */
        len = read (STDIN_FILENO, buf, BUF_LEN);
        if (len == -1) {
            perror ("read");
            return 1;
        }

        if (len) {

```

```

        buf[len] = '\0';
        printf ("read: %s\n", buf);
    }

    return 0;
}

fprintf(stderr, "Этого быть не должно!\n");
return 1;
}

```

Использование select() для краткого засыпания

Исторически на различных UNIX-подобных системах вызов select() был более распространен, чем механизм засыпания с разрешающей способностью менее секунды, поэтому данный вызов часто используется как механизм для кратковременного засыпания. Чтобы использовать select() в таком качестве, достаточно указать ненулевую задержку, но задать NULL для всех трех групп:

```

struct timeval tv;

tv.tv_sec = 0;
tv.tv_usec = 500;

/* засыпаем на 500 микросекунд */
select (0, NULL, NULL, NULL, &tv);

```

В Linux предоставляются интерфейсы для засыпания с высоким разрешением. О них мы подробно поговорим в гл. 11.

Вызов pselect()

Системный вызов select(), впервые появившийся в 4.2BSD, достаточно популярен, но в POSIX есть и собственный вариант решения — вызов pselect(). Он был описан сначала в POSIX 1003.1g-2000, а затем в POSIX 1003.1-2001:

```

#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            const struct timespec *timeout,
            const sigset_t *sigmask);
/* эти же значения используются и cselect() */
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);

```

Между системными вызовами pselect() и select() есть три различия.

1. Вызов pselect() использует для своего параметра timeout структуру timespec, а не timeval. Структура timespec может иметь значения в секундах и наносекундах,

а не в секундах и микросекундах. Теоретически `timespec` должна создавать задержки с более высоким разрешением, но на практике ни одна из этих структур не может надежно обеспечивать даже разрешение в микросекундах.

2. При вызове `pselect()` параметр `timeout` не изменяется. Следовательно, не требуется заново инициализировать его при последующих вызовах.
3. Системный вызов `select()` не имеет параметра `sigmask`. Если при работе с сигналами установить этот параметр в значение `NULL`, то `pselect()` станет функционально аналогичен `select()`.

Структура `timespec` определяется следующим образом:

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

struct timespec {
    long tv_sec;          /* секунды */
    long tv_nsec;         /* наносекунды */
};
```

Основная причина, по которой вызов `pselect()` был добавлен в инструментарий UNIX, связана с появлением параметра `sigmask`. Этот параметр призван справляться с условиями гонки, которые могут возникать при ожидании файловых дескрипторов и сигналов. Подробнее о сигналах мы поговорим в гл. 10. Предположим, что обработчик сигнала устанавливает глобальный флаг (большинство из них именно так и делают), а процесс проверяет этот флаг перед вызовом `select()`. Далее предположим, что сигнал приходит в период после проверки, но до вызова. Приложение может оказаться заблокированным на неопределенный срок и так и не отреагировать на установленный флаг. Вызов `pselect()` позволяет решить эту проблему: приложение может вызвать `pselect()`, предоставив набор сигналов для блокирования. Заблокированные сигналы не обрабатываются, пока не будут разблокированы. Как только `pselect()` вернется, ядро восстановит старую маску сигнала.

До версии ядра Linux 2.6.16 `pselect()` был реализован в этой операционной системе не как системный вызов, а как обычная обертка для вызова `select()`, предоставляемая `glibc`. Такая обертка сводила к минимуму риск возникновения условий гонки, но не исключала его полностью. Когда `pselect()` стал системным вызовом, проблема с условиями гонки была решена.

Несмотря на (относительно незначительные) улучшения, характерные для `pselect()`, в большинстве приложений продолжает использоваться вызов `select()`. Это может делаться как по привычке, так и для обеспечения оптимальной переносимости.

Системный вызов `poll()`

Вызов `poll()` является в System V как раз тем решением, которое обеспечивает мультиплексный ввод-вывод. Он компенсирует некоторые недостатки, имеющиеся у `select()`, хотя `select()` по-прежнему используется очень часто (как по привычке, так и для обеспечения оптимальной переносимости):


```
#include <poll.h>
```

```
int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

В отличие от вызова `select()`, применяющего неэффективный метод с тремя группами дескрипторов на основе битовых масок, `poll()` работает с единым массивом структур `pollfd`, на которые указывают файловые дескрипторы. Такая структура определяется следующим образом:

```
#include <poll.h>
```

```
struct pollfd {
    int fd;           /* файловый дескриптор */
    short events;     /* запрашиваемые события для отслеживания */
    short revents;    /* зафиксированные возвращаемые события */
};
```

В каждой структуре `pollfd` указывается один файловый дескриптор, который будет отслеживаться. Можно передавать сразу несколько структур, указав `poll()` отслеживать несколько файловых дескрипторов. Поле `events` каждой структуры представляет собой битовую маску событий, которые мы собираемся отслеживать на данном файловом дескрипторе. Ядро устанавливает это поле после возврата значения. Все события, возвращенные в поле `events`, могут быть возвращены в поле `revents`. Допустимы следующие события:

- `POLLIN` — имеются данные для считывания;
- `POLLRDNORM` — имеются обычные данные для считывания;
- `POLLRDBAND` — имеются приоритетные данные для считывания;
- `POLLPRI` — имеются срочные данные для считывания;
- `POLLOUT` — запись блокироваться не будет;
- `POLLWRNORM` — запись обычных данных блокироваться не будет;
- `POLLWRBAND` — запись приоритетных данных блокироваться не будет;
- `POLLMSG` — доступно сообщение `SIGPOLL`.

Кроме того, в поле `revents` могут быть возвращены следующие события:

- `POLLER` — возникла ошибка на указанном файловом дескрипторе;
- `POLLHUP` — событие зависания на указанном файловом дескрипторе;
- `POLLVAL` — указанный файловый дескриптор не является допустимым.

Эти события не имеют никакого значения в поле `events`, и их не следует передавать в данное поле, поскольку при необходимости система всегда их возвращает. При использовании `poll()`, чего не скажешь о `select()`, не требуется явно задавать необходимость отчета об исключениях.

Сочетание `POLLIN` | `POLLPRI` эквивалентно событию считывания в вызове `select()`, а событие `POLLOUT` | `POLLWRBAND` идентично событию записи в вызове `select()`. Значение `POLLIN` эквивалентно `POLLRDNORM` | `POLLRDBAND`, а `POLLOUT` эквивалентно `POLLWRNORM`.

Например, чтобы одновременно отслеживать на файловом дескрипторе возможность считывания и возможность записи, следует задать для параметра `events` значение `POLLIN | POLLOUT`. Получив возвращаемое значение, мы проверим поле `revents` на наличие этих флагов в структуре, соответствующей интересующему нас файловому дескриптору. Если бы флаг `POLLOUT` был установлен, то файловый дескриптор был бы доступен для записи без блокирования. Эти флаги не являются взаимоисключающими: можно установить сразу оба, обозначив таким образом, что возможен возврат и при считывании, и при записи. Блокирования при считывании и записи на этом файловом дескрипторе не будет.

Параметр `timeout` указывает задержку (длительность ожидания) в миллисекундах перед возвратом независимо от наличия или отсутствия готового ввода-вывода. Отрицательное значение соответствует неопределенно долгой задержке. Значение 0 предписывает вызову вернуться незамедлительно, перечислив все файловые дескрипторы, на которых имеется ожидающий обработки ввод-вывод, но не дожидаясь каких-либо дальнейших событий. Таким образом, `poll()` оправдывает свое название: он совершает один акт опроса и немедленно возвращается.

Возвращаемые значения и коды ошибок

В случае успеха `poll()` возвращает количество файловых дескрипторов, чьи структуры содержат ненулевые поля `revents`. В случае возникновения задержки до каких-либо событий этот вызов возвращает 0. При ошибке возвращается -1, а `errno` устанавливается в одно из следующих значений:

- `EBADF` — для одной или нескольких структур были заданы недопустимые файловые дескрипторы;
- `EFAULT` — значение указателя на файловые дескрипторы находится за пределами адресного пространства процесса;
- `EINTR` — до того как произошло какое-либо из запрошенных событий, был выдан сигнал; вызов можно повторить;
- `EINVAL` — параметр `nfds` превысил значение `RLIMIT_NOFILE`;
- `ENOMEM` — для выполнения запроса оказалось недостаточно памяти.

Пример использования `poll()`

Рассмотрим пример программы, использующей вызов `poll()` для одновременной проверки двух условий: не возникнет ли блокировка при считывании с `stdin` и записи в `stdout`:

```
#include <stdio.h>
#include <unistd.h>
#include <poll.h>

#define TIMEOUT 5 /* задержка poll, значение в секундах */

int main (void)
{
    struct pollfd fds[2];
```

```

int ret;

/* отслеживаем ввод на stdin */
fds[0].fd = STDIN_FILENO;
fds[0].events= POLLIN;

/* отслеживаем возможность записи на stdout (практически всегда true) */
fds[1].fd = STDOUT_FILENO;
fds[1].events = POLLOUT;

/* Все установлено, блокируем! */
ret= poll(fds, 2, TIMEOUT* 1000);
if (ret == -1) {
    perror ("poll");
    return 1;
}

if (!ret) {
    printf ("%d seconds elapsed.\n", TIMEOUT);
    return 0;
}

if (fds[0].revents &POLLIN)
    printf ("stdin is readable\n");

if (fds[1].revents &POLLOUT)
    printf ("stdout is writable\n");
    return 0;
}

```

Запустив этот код, мы получим следующий результат, как и ожидалось:

```

$ ./poll
stdout is writable

```

Запустим его еще раз, но теперь перенаправим файл в стандартный ввод, и мы увидим оба события:

```

$ ./poll<ode_to_my_parrot.txt
stdin is readable
stdout is writable

```

Если бы мы использовали `poll()` в реальном приложении, то нам не требовалось бы реконструировать структуры `pollfd` при каждом вызове. Одну и ту же структуру можно передавать многократно; при необходимости ядро будет заполнять поле `revents` нулями.

Системный вызов `ppoll()`

Linux предоставляет вызов `ppoll()`, напоминающий `poll()`. Сходство между ними примерно такое же, как между `select()` и `pselect()`. Однако в отличие от `pselect()` вызов `ppoll()` является специфичным для Linux интерфейсом.

```
#define _GNU_SOURCE

#include <poll.h>

int ppoll (struct pollfd *fds,
           nfd_t nfds,
           const struct timespec *timeout,
           const sigset_t *sigmask);
```

Как и в случае с `pselect()`, параметр `timeout` задает значение задержки в секундах и наносекундах. Параметр `sigmask` содержит набор сигналов, которых следует ожидать.

Сравнение `poll()` и `select()`

Системные вызовы `poll()` и `select()` выполняют примерно одну и ту же задачу, однако `poll()` удобнее, чем `select()`, по нескольким причинам.

- Вызов `poll()` не требует от пользователя вычислять и передавать в качестве параметра значение «максимальный номер файлового дескриптора плюс один».
- Вызов `poll()` более эффективно оперирует файловыми дескрипторами, имеющими крупные номера. Предположим, мы отслеживаем с помощью `select()` всего один файловый дескриптор со значением 900. В этом случае ядру пришлось бы проверять каждый бит в каждой из переданных групп вплоть до 900-го.
- Размер групп файловых дескрипторов, используемых с `select()`, задается статически, что вынуждает нас идти на компромисс: либо делать их небольшими и, следовательно, ограничивать максимальное количество файловых дескрипторов, которые может отслеживать `select()`, либо делать их большими, но неэффективными. Операции с крупными масками битов неэффективны, особенно если заранее не известно, применялось ли в них разреженное заполнение¹. С помощью `poll()` мы можем создать массив именно того размера, который нам требуется. Будем наблюдать только за одним элементом? Хорошо, передаем всего одну структуру.
- При работе с `select()` группы файловых дескрипторов реконструируются уже после возврата значения, поэтому каждый последующий вызов должен повторно их инициализировать. Системный вызов `poll()` разграничивает ввод (поле `events`) и вывод (поле `revents`), позволяя переиспользовать массив без изменений.
- Параметр `timeout` вызова `select()` на момент возврата значения имеет неопределенное значение. Переносимый код должен заново инициализировать его. При работе с вызовом `pselect()` такая проблема отсутствует.

¹ Если битовая маска после заполнения получилась разреженной, то каждое слово, содержащееся в ней, можно проверить, сравнив его с нулем. Только если эта операция возвратит «ложно», потребуется отдельно проверять каждый бит. Тем не менее, если маска не разреженная, то это совершенно напрасная работа.

Однако у системного вызова `select()` есть определенные преимущества:

- для `select()` характерна значительная переносимость, а вот `poll()` в некоторых системах UNIX не поддерживается;
- вызов `select()` обеспечивает более высокое разрешение задержки — до микросекунд, тогда как `poll()` гарантирует разрешение лишь с миллисекундной точностью; и `ppoll()`, и `pselect()` теоретически должны обеспечивать разрешение с точностью до наносекунд, но на практике ни один из этих вызовов не может предоставлять разрешение даже на уровне микросекунд.

Оба — и `poll()`, и `select()` — уступают по качеству интерфейсу `epoll`. Это специфичное для Linux решение, предназначенное для мультиплексного ввода-вывода. Подробнее мы поговорим о нем в гл. 4.

Внутренняя организация ядра

В этом разделе мы рассмотрим, как ядро Linux реализует ввод-вывод. Нас в данном случае интересуют три основные подсистемы ядра: *виртуальная файловая система* (VFS), *страничный кэш* и *страничная отложенная запись*. Взаимодействуя, эти подсистемы обеспечивают гладкий, эффективный и оптимальный ввод-вывод.

ПРИМЕЧАНИЕ

В гл. 4 мы рассмотрим и четвертую подсистему — планировщик ввода-вывода.

Виртуальная файловая система

Виртуальная файловая система, которую иногда также называют *виртуальным файловым коммутатором*, — это механизм абстракции, позволяющий ядру Linux вызывать функции файловой системы и оперировать данными файловой системы, не зная — и даже не пытаясь узнать, — какой именно тип файловой системы при этом используется.

VFS обеспечивает такую абстракцию, предоставляя *общую модель файлов* — основу всех используемых в Linux файловых систем. С помощью указателей функций, а также с использованием различных объектно-ориентированных приемов¹ общая файловая модель образует структуру, которой должны соответствовать файловые системы в ядре Linux. Таким образом, виртуальная файловая система может делать обобщенные запросы в фактически применяемой файловой системе. В данном фреймворке предоставляются привязки для поддержки считывания, создания ссылок, синхронизации и т. д. Затем каждая файловая система регистрирует функции для обработки операций, которые в ней обычно приходится выполнять.

Такой подход требует определенной схожести между файловыми системами. Так, VFS работает в контексте индексных дескрипторов, суперблоков и записей

¹ Да, на языке C.

каталогов. Если приходится иметь дело с файловой системой, не относящейся к семейству UNIX, то в ней могут отсутствовать некоторые важные концепции UNIX, например индексные дескрипторы, и необходимо как-то с этим справляться. Пока это удастся. Например, Linux может без проблем взаимодействовать с файловыми системами FAT и NTFS.

Преимущество использования VFS множество. Единственного системного вызова достаточно для считывания информации из *любой* файловой системы, с *любого* носителя. Отдельно взятая утилита может копировать информацию из любой файловой системы в какую угодно другую. Все файловые системы поддерживают одни и те же концепции, интерфейсы и системные вызовы. Все просто работает — и работает хорошо.

Если определенное приложение выдает системный вызов `read()`, то путь этого вызова получается довольно интересным. Библиотека C содержит определения системного вызова, которые во время компиляции преобразуются в соответствующие операторы ловушки. Как только ядро поймает процесс из пользовательского пространства, переданный обработчиком системного вызова и «врученный» системному вызову `read()`, ядро определяет, какой объект *лежит в основе* конкретного файлового дескриптора. Затем ядро вызывает функцию считывания, ассоциированную с этим базовым объектом. Данная функция входит в состав кода файловой системы. Затем функция выполняет свою задачу — например, физически считывает данные из файловой системы — и возвращает полученные данные вызову `read()`, относящемуся к пользовательскому пространству. После этого `read()` возвращает информацию обработчику вызова, который, в свою очередь, копирует эти данные обратно в пользовательское пространство, где происходит возврат системного вызова `read()` и выполнение процесса продолжается.

Системный программист должен разбираться и в разновидностях файловой системы VFS. Обычно ему не приходится беспокоиться о типе файловой системы или носителя, на котором находится файл. Универсальные системные вызовы — `read()`, `write()` и т. д. — могут оперировать файлами в любой поддерживаемой файловой системе и на каком угодно поддерживаемом носителе.

Страничный кэш

Страничный кэш — это находящееся в памяти хранилище данных, которые взяты из файловой системы, расположенной на диске, и к которым недавно выполнялись обращения. Доступ к диску удручающе медленный, особенно по сравнению со скоростями современных процессоров. Благодаря хранению востребованных данных в памяти ядро может выполнять последующие запросы к тем же данным, уже не обращаясь к диску.

В страничном кэше задействуется концепция *временной локальности*. Это разновидность *локальности ссылок*, согласно которой ресурс, запрошенный в определенный момент времени, с большой вероятностью будет снова запрошен в ближайшем будущем. Таким образом, компенсируется память, затрачиваемая на кэширование

данных после первого обращения: мы избавляемся от необходимости последующих затратных обращений к диску.

Если ядру требуется найти данные о файловой системе, то оно первым делом обращается именно в страничный кэш. Ядро приказывает подсистеме памяти получить данные с диска, только если они не будут обнаружены в страничном кэше. Таким образом, при первом считывании любого элемента данных этот элемент переносится с диска в системный кэш и возвращается к приложению уже из кэша. Все операции прозрачно выполняются через страничный кэш. Так гарантируется актуальность и правильность используемых данных.

Размер страничного кэша Linux является динамическим. В результате операций ввода-вывода все больше информации с диска оседает в памяти, страничный кэш растет, пока наконец не израсходует всю свободную память. Если места для роста страничного кэша не осталось, но система снова совершает операцию выделения, требующую дополнительной памяти, страничный кэш *урежается*, высвобождая страницы, которые используются реже всего, и «оптимизируя» таким образом использование памяти. Такое урезание происходит гладко и автоматически. Размеры кэша задаются динамически, поэтому Linux может пользоваться всей памятью в системе и кэшировать столько данных, сколько возможно.

Однако иногда бывает более целесообразно *подкачивать* на диск редко используемую страницу процессной памяти, а не обрезать востребованные части страничного кэша, которые вполне могут быть заново перенесены в память уже при следующем запросе на считывание (благодаря возможности подкачки ядро может хранить сравнительно большой объем данных на диске, таким образом выделяя больший объем памяти, чем общий объем RAM на данной машине). Ядро Linux использует эвристику, чтобы добиться баланса между подкачкой данных и урезанием страничного кэша (а также других резервных областей памяти). В рамках такой эвристики может быть решено выгрузить часть данных из памяти на диск ради урезания страничного кэша, особенно если выгружаемые таким образом данные сейчас не используются.

Баланс подкачки и кэширования можно настроить в `/proc/sys/vm/swappiness`. Этот виртуальный файл может иметь значение в диапазоне от 0 до 100. По умолчанию оно равно 60. Чем выше значение, тем выше приоритет урезания страничного кэша относительно подкачки.

Еще одна разновидность локальности ссылок — это *сосредоточенность последовательности*. В соответствии с данным принципом ссылка часто делается на ряд данных, следующих друг за другом. Чтобы воспользоваться преимуществом данного принципа, ядро также реализует *опережающее считывание* страничного кэша. Опережающее считывание — это акт чтения с диска дополнительных данных с перемещением их в страничный кэш. Опережающее считывание сопутствует каждому запросу и обеспечивает постоянное наличие в памяти некоторого избытка данных. Когда ядро читает с диска фрагмент данных, оно также считывает и один-два последующих фрагмента. Одновременное считывание сравнительно большой последовательности данных оказывается эффективным, так как обычно обходится без позиционирования (подвода головок). Кроме того, ядро может выполнить запрос

на опережающее считывание, пока процесс обрабатывает первый фрагмент полученных данных. Если (как часто бывает) процесс должен вот-вот выдать новый запрос на считывание последующего фрагмента, то ядро может передать данные от исходного опережающего считывания, даже не запрашивая лишний раз дисковый ввод-вывод.

Ядро управляет опережающим считыванием динамически, как и работой со страничным кэшем. Если система заметит, что процесс постоянно использует данные, полученные в ходе опережающего считывания, то ядро увеличивает окно такого считывания, забирая с диска все больше и больше дополнительных данных. Окно опережающего считывания может быть совсем маленьким (16 Кбайт), но в некоторых случаях достигает и 128 Кбайт. Верно и обратное: если ядро замечает, что опережающее считывание не дает значительного эффекта, это означает, что приложение выбирает данные из файла в произвольном, а не последовательном порядке. В таком случае опережающее считывание может быть полностью отключено.

Работа страничного кэша должна быть совершенно прозрачной. Как правило, системные программисты не могут оптимизировать свой код так, чтобы он учитывал существование страничного кэша и задействовал его на пользу программе. Единственный вариант использования таких преимуществ предполагает, что программист реализует подобный кэш самостоятельно уже в пользовательском пространстве. Как правило, для оптимального использования страничного кэша требуется просто писать эффективный код. Опять же можно пользоваться опережающим считыванием. Последовательный файловый ввод-вывод всегда предпочтительнее произвольного доступа, хотя организовать его удается не всегда.

Страничная отложенная запись

Как уже упоминалось в разд. «Запись с помощью `write()`», ядро откладывает операции записи, используя систему буферов. Когда процесс выдает запрос на запись, данные копируются в буфер, который с этого момента считается *грязным*. Это означает, что копия информации, находящаяся в памяти, новее копии, имеющейся на диске. После этого запрос на запись сразу возвращается. Если был сделан другой запрос на запись, обращенный к тому же фрагменту файла, то буфер заполняется новыми данными. Если к одному и тому же файлу обращено несколько запросов записи, все они генерируют новые буферы.

Рано или поздно информация из грязных буферов должна быть сброшена на диск, синхронизируя, таким образом, дисковые файлы с данными, находящимися в памяти. Этот процесс называется *отложенной записью*. Она происходит в двух случаях.

- Когда объем свободной памяти становится ниже определенного порога (эту величину можно конфигурировать), содержимое грязных буферов записывается

на диск. Очищенные таким образом буферы можно удалить и высвободить часть памяти.

- Если возраст буфера превышает определенный порог (эту величину можно конфигурировать), информация из данного буфера записывается на диск. Благодаря этому данные не остаются в грязном буфере на неопределенно долгий срок.

Отложенная запись выполняется группой потоков ядра, которые называются *промывочными*. Если удовлетворяется хотя бы одно из двух вышеназванных условий, то такие потоки активизируются и начинают сбрасывать информацию из грязных буферов на диск. Это происходит, пока оба условия не станут ложными.

В некоторых ситуациях сразу несколько промывочных потоков одновременно начинают выполнять отложенную запись. Это делается, чтобы максимально эффективно задействовать сильные стороны параллелизма и для реализации техники *избегания скученности*. Выполняя ее, мы стараемся исключить накопление записей в период ожидания отдельно взятого блочного устройства. Если имеются грязные буферы, относящиеся к разным блочным устройствам, то промывочные потоки будут работать с расчетом на максимальное использование каждого из блочных устройств. Таким образом компенсируется один недостаток, имевшийся в сравнительно старых ядрах: предшественники промывочных потоков (потоки `pdflush`, а еще раньше `bdflush`) могли потратить все свое время, дожидаясь единственного блочного устройства, тогда как другие блочные устройства простаивали. На современных машинах ядро Linux может одновременно подпитывать множество дисков.

Буферы представлены в ядре структурой данных `buffer_head`. Она отслеживает различные метаданные, ассоциированные с буфером, в частности информацию о том, чист данный буфер или грязен. Кроме того, в этой структуре содержится указатель на сами данные. Они находятся в страничном кэше. Так обеспечивается объединение подсистемы буферов и страничного кэша.

В ранних версиях ядра Linux — до 2.4 — подсистема буферов была отделена от страничного кэша, существовал как страничный, так и буферный кэш. Таким образом, данные могли одновременно находиться и в буферном кэше (в грязном буфере), и в страничном (в качестве кэшированных данных). Естественно, синхронизация этих двух отдельных кэшей потребовала определенных усилий. Единый страничный кэш, появившийся в ядре Linux 2.4, был встречен восторженными отзывами.

Отложенная запись и действующая в Linux подсистема буферов обеспечивают быструю запись, но повышают риск потери данных при резком сбое питания. Для устранения такой опасности в критически важных приложениях (а также приложениях программистов-параноиков) можно использовать синхронизированный ввод-вывод (о нем мы говорили выше в этой главе).

Резюме

В данной главе мы обсудили одну из фундаментальных тем системного программирования в Linux — ввод-вывод. В таких системах, как Linux, которые стремятся представить все, что можно, в виде файлов, очень важно уметь правильно открывать, считывать, записывать и закрывать последние. Все эти операции относятся к классике UNIX и описаны во многих стандартах.

В следующей главе мы поговорим о буферизованном вводе-выводе и стандартных интерфейсах ввода-вывода, предоставляемых в библиотеке C. Стандартная библиотека C используется в данном случае не только для удобства программиста: буферизация ввода-вывода в пользовательском пространстве позволяет значительно повысить производительность.

3 Буферизованный ВВОД-ВЫВОД

Как вы помните из гл. 1, блок — это абстракция, представляющая мельчайший компонент системы, предназначенный для хранения данных в файловой системе. Внутри ядра все операции файловой системы трактуются именно в контексте блоков. Действительно, блок — это настоящий общий знаменатель всего ввода-вывода. Следовательно, любые действия ввода-вывода не могут осуществляться над объемом информации, не превышающим размер одного блока, а также над объемом данных, который не выражается целым числом, кратным размеру блока. Если вам требуется прочитать всего один байт — ничего не поделаешь, придется считывать целый блок. Хотите записать 4,5 единицы данных? Нужно будет записать пять блоков, причем впоследствии тот, который записан частично, будет считываться целиком. При этом вы обновите лишь половину блока, а отложенная запись займет целый блок.

Вы понимаете, к чему я клоню: операции с частичной записью блоков неэффективны. Операционной системе приходится «вписывать» ваш ввод-вывод в имеющиеся рамки и гарантировать, что все действия укладываются в блочные границы. Это делается с помощью округления с увеличением до ближайшего целого блока. К сожалению, именно в таком ключе обычно пишутся приложения для пользовательского пространства. Большинство приложений при работе оперируют более высокоуровневыми абстракциями — например, полями и строками. Размер таких сущностей варьируется независимо от размера блока. Очень плохие пользовательские приложения могут одновременно считывать всего один байт! Это расточительно. Такая однобайтовая запись займет целый блок дискового пространства.

Ситуация дополнительно усугубляется из-за лишних системных вызовов, которые необходимы, например, для 1024 операций считывания блоков, содержащих по одному байту каждый, ведь можно было бы обойтись всего одним вызовом, если бы эта информация компактно располагалась в блоке размером 1024 байт. Решение этой проблемы с производительностью заключается в использовании *ввода-вывода с пользовательским буфером*. Этот способ позволяет приложениям естественным образом считывать и записывать любые объемы данных, но осуществлять ввод-вывод в размере целых блоков данной файловой системы.

Ввод-вывод с пользовательским буфером

Программы, которым приходится выполнять множество мелких системных вызовов к обычным файлам, часто осуществляют ввод-вывод с пользовательским буфером. Так называется буферизация, выполняемая в пользовательском пространстве. Ее можно организовывать в приложении вручную либо прозрачно выполнять в библиотеке. Однако такая буферизация никак не связана с ядром. Как было сказано в гл. 2, на внутрисистемном уровне буферизация данных в ядре происходит посредством отложенных записей, объединения смежных запросов ввода-вывода и опережающего считывания. Пользовательская буферизация выполняется иначе, но также нацелена на улучшение производительности.

Рассмотрим пример с использованием программы `dd`, работающей в пользовательском пространстве:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

Мы имеем аргумент `bs=1`, поэтому данная команда будет копировать 2 Мбайт информации с устройства `/dev/zero` (это виртуальное устройство, выдающее бесконечное количество нулей) в файл `pirate`. Операция будет передана в виде 2 097 152 однокбайтовых фрагментов. Таким образом, на копирование этих данных мы затратим примерно 2 000 000 операций считывания и записи — по байту за раз.

Рассмотрим копирование тех же 2 Мбайт информации, но уже с использованием блоков размером по 1024 байт каждый:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

Эта операция позволяет скопировать те же 2 Мбайт информации в тот же файл, но требует в 1024 раза меньше операций считывания и записи. Налицо радикальное улучшение производительности, как видно из табл. 3.1. В ней я записал затраченное время (измеренное тремя разными способами) четырьмя командами `dd`, которые отличались лишь размером блока. Реальное время — это общее время, истекшее на настенных часах. Пользовательское время — это время, потраченное на исполнение программного кода в пользовательском пространстве. Системное время — это время, затраченное на выполнение инициированных процессом системных вызовов в пространстве ядра.

Таблица 3.1. Влияние размера блока на производительность

Размер блока, байты	Реальное время, секунды	Пользовательское время, секунды	Системное время, секунды
1	18,707	1,118	17,549
1024	0,025	0,002	0,023
1130	0,035	0,002	0,027

При использовании фрагментов данных по 1024 байт достигается грандиозное улучшение производительности по сравнению с передачей информации по одному байту. Тем не менее данные, приведенные выше (см. табл. 3.1), также свидетельствуют, что при использовании блоков с большим размером — и, соответственно, при

дальнейшем сокращении количества системных вызовов — производительность может и снижаться, если используемые при работе фрагменты не кратны размеру блока диска. Несмотря на то что запросы 1130-байтовых фрагментов требуют меньшего количества системных вызовов, они оказываются менее эффективными, чем 1024-байтовые запросы, кратные размеру блока.

Чтобы обратить эти аспекты производительности себе на пользу, необходимо заранее знать размер физического блока на данном устройстве. Результаты, приведенные выше (см. табл. 3.1), показывают, что размер блока, скорее всего, будет равен 1024 байт, целочисленному кратному 1024 или делителю 1024. В случае с `/dev/zero` точный размер блока равен 4096 байт.

Размер блока. На практике размер блока обычно составляет 512, 1024, 2048, 4096 или 8192 байт.

Как показано выше (см. табл. 3.1), для значительного повышения производительности достаточно просто выполнять операции фрагментами, которые являются целочисленными кратными или делителями размера блока. Дело в том, что и ядро, и аппаратное обеспечение работает в контексте блоков. Соответственно, если оперировать либо размером блока, либо значением, которое аккуратно вписывается в блок, то все запросы ввода-вывода будут выполняться в пределах целых блоков. Лишняя работа в ядре исключается.

Узнать размер блока на конкретном устройстве довольно просто: для этого используется системный вызов `stat()`, рассмотренный в гл. 8, либо команда `stat(1)`. Однако оказывается, что в большинстве случаев не требуется знать точный размер блока.

Выбирая размер для будущих операций ввода-вывода, главное — не получить какую-нибудь заведомо неровную величину, например 1130. Ни один блок в истории UNIX не имел размер 1130 байт — если вы выберете такой объем для ваших операций, то уже после первого запроса выравнивание ввода-вывода будет нарушено. Если же выбранный вами размер операции позволяет сохранять выравнивание по границам блоков, то производительность будет высокой. Чем больше кратное, тем меньше системных вызовов вам потребуется.

Соответственно, самый простой вариант — использовать при вводе-выводе достаточно большой буфер, являющийся общим кратным типичных размеров блоков. Очень удобны значения 4096 и 8192 байт.

Получается, достаточно осуществлять весь ввод-вывод фрагментами по 4 или 8 Кбайт, и проблемы решены? Не все так просто. Проблема заключается в том, что сами программы редко оперируют цельными блоками — они работают с отдельными полями, строками, символами, а не с такой абстракцией, как блок. Ввод-вывод с пользовательским буфером устраняет разрыв между файловой системой, работающей с блоками, и приложением, оперирующим собственными абстракциями. Принцип работы пользовательского буфера одновременно простой и мощный: по мере того как данные записываются, они сохраняются в буфере в пределах адресного пространства конкретной программы. Когда размер буфера достигает установленного предела, называемого *размером буфера*, содержимое этого буфера переносится на диск за одну операцию записи. Считывание данных также происходит фрагментами, равными размеру буфера и выровненными по границам блоков.

Поступающие от приложения запросы на считывание имеют разные размеры и обслуживаются не из файловой системы напрямую, а удовлетворяются фрагментами, получаемыми через буфер. По мере того как приложение считывает все больше и больше информации, данные выдаются из буфера кусками. Наконец, как только буфер пустеет, начинается считывание следующего сравнительно крупного фрагмента, выровненного по границам блоков. Таким образом, приложение может считывать и записывать любые произвольные фрагменты данных, как потребуется, но буферизация данных всегда выполняется лишь сравнительно большими кусками. Эти крупные операции, выровненные по границам блоков, уже направляются в файловую систему. В конечном итоге нам удастся обойтись меньшим количеством системных вызовов при работе со значительными объемами данных, выровненными строго по границам блоков. В результате мы имеем серьезное повышение производительности.

Можете самостоятельно реализовать пользовательскую буферизацию в ваших программах. Кстати, именно так и делается во многих критически важных приложениях. Однако в абсолютном большинстве программ используется популярная *стандартная библиотека ввода-вывода*, входящая в состав стандартной библиотеки C, либо, как вариант, *библиотека классов ввода-вывода* языка C++, с помощью которых легко создавать надежные и практичные решения с пользовательской буферизацией.

Стандартный ввод-вывод

В составе стандартной библиотеки C есть стандартная библиотека ввода-вывода, иногда сокращенно именуемая `stdio`. Она, в свою очередь, предоставляет независимое от платформы решение для пользовательской буферизации. Стандартная библиотека ввода-вывода проста в использовании, но ей не занимать мощности.

В отличие от таких языков программирования, как FORTRAN, язык C не содержит никакой встроенной поддержки ключевых слов, которая обеспечивала бы более сложный функционал, чем управление выполнением программы, арифметические действия и т. д. Естественно, в языке нет и встроенной поддержки ввода-вывода. По эволюции языка программирования C его пользователи разработали стандартные наборы процедур, обеспечивающих основную функциональность. В частности, речь идет о манипуляции строками, математических процедурах, работе с датой и временем, а также о вводе-выводе. Со временем эти процедуры совершенствовались. В 1989 году, когда был ратифицирован стандарт ANSI C (C89), все эти функции были объединены в стандартную библиотеку C. В C95, C99 и C11 добавилось несколько новых интерфейсов, однако стандартная библиотека ввода-вывода осталась практически нетронутой со времени появления в C89.

Оставшаяся часть этой главы посвящена вводу-выводу с пользовательским буфером, тому, как он относится к файловому вводу-выводу и как реализован в стандартной библиотеке C. Иными словами, нас интересует открытие, закрытие, считывание и запись файлов с помощью стандартной библиотеки C. Решение,

будут ли в приложении использоваться стандартный ввод-вывод, собственное решение с пользовательским буфером либо обычные системные вызовы, принимает сам разработчик. Это решение должно быть тщательно взвешенным и приниматься с учетом всех потребностей приложения и его поведения.

В стандартах C некоторые детали всегда остаются на усмотрение конкретной реализации, и в разных реализациях действительно часто добавляются новые возможности. В этой главе, а также во всей оставшейся книге описаны интерфейсы и поведения в виде, в котором они реализованы в библиотеке glibc в современной системе Linux. Если в Linux делается отступление от стандартов, я это особо указываю.

Указатели файлов. Процедуры стандартного ввода-вывода не работают непосредственно с файловыми дескрипторами. Вместо этого каждая использует свой уникальный идентификатор, обычно называемый *указателем файла*. В библиотеке C указатель файла ассоциируется с файловым дескриптором (отображается на него). Указатель файла представлен как указатель на определение типа FILE, определяемый в `<stdio.h>`.

ПРИМЕЧАНИЕ

Название FILE часто критикуют за то, что оно записывается прописными буквами, что кажется особенно непривлекательным в стандарте C, ведь в нем (а следовательно, и в большинстве стилей написания кода в конкретных приложениях) имена типов и функций записываются только в нижнем регистре. Эта странность объясняется историческими причинами: изначально стандартный ввод-вывод был написан в виде макрокоманд. Не только FILE, но и все методы библиотеки были реализованы в виде наборов макрокоманд. Сегодня также сохраняется традиция давать всем макрокомандам названия прописными буквами. По мере того как язык C развивался и стандартный ввод-вывод наконец был регламентирован как официальная часть языка, большинство методов были переписаны в виде обычных функций, а FILE стал определением типа, но он так и продолжает записываться в верхнем регистре.

В терминологии ввода-вывода открытый файл называется *поток данных*¹. Потоки могут быть открыты для чтения (поток данных ввода), для записи (поток данных вывода) или для того и другого (поток данных ввода/вывода).

Открытие файлов

Файлы открываются для чтения или записи с помощью функции `fopen()`:

```
#include <stdio.h>
```

```
FILE * fopen (const char *path, const char *mode);
```

Эта функция открывает файл `path`, поведение которого определено в `mode`, и ассоциирует с ним новый поток данных.

¹ В русском языке сложилась омонимия между понятиями «поток» (thread) и «поток» (stream). Во избежание двусмысленности в этом разделе слово stream будет переводиться как «поток данных», а слово thread — как «программный поток». Данная терминология сохраняется вплоть до конца главы. — *Примеч. пер.*

Режимы. Аргумент `mode` описывает, как открывать конкретный файл. Данный аргумент может быть представлен одной из следующих строк.

- `r` — файл открывается для чтения. Поток данных устанавливается в начале файла.
- `r+` — файл открывается как для чтения, так и для записи. Поток данных устанавливается в начале файла.
- `w` — файл открывается для записи. Если файл существует, то он усекается до нулевой длины. Если файл не существует, он создается. Поток данных устанавливается в начале файла.
- `w+` — файл открывается для чтения и для записи. Если файл существует, то он усекается до нулевой длины. Если файл не существует, он создается. Поток данных устанавливается в начале файла.
- `a` — файл открывается для дополнения в режиме дозаписи. Если файл не существует, то он создается. Поток данных устанавливается в конце файла. Все вводимые данные дозаписываются в файл.
- `a+` — файл открывается для дополнения и считывания в режиме дозаписи. Если файл не существует, то он создается. Поток данных устанавливается в конце файла. Все вводимые данные дозаписываются в файл.

ПРИМЕЧАНИЕ

В указанном режиме также может содержаться символ `b`, хотя в Linux это значение всегда игнорируется. Некоторые операционные системы по-разному обрабатывают текст и двоичные файлы. Символ `b` означает, что файл должен быть открыт именно в двоичном режиме. Linux, как и все операционные системы, соответствующие стандарту POSIX, воспринимает текст и двоичные файлы одинаково.

В случае успеха функция `fopen()` возвращает допустимый указатель `FILE`. При ошибке она возвращает `NULL` и устанавливает `errno` соответствующее значение.

Например, следующий код открывает для чтения файл `/etc/manifest` и ассоциирует его с потоком данных `stream`:

```
FILE *stream;
```

```
stream = fopen ("/etc/manifest", "r");  
if (!stream)  
    /* ошибка */
```

Открытие потока данных с помощью файлового дескриптора

Функция `fdopen()` преобразует уже открытый файловый дескриптор (`fd`) в поток данных:

```
#include <stdio.h>
```

```
FILE * fdopen (int fd, const char *mode);
```


Могут использоваться те же режимы, что и с функцией `fopen()`, при этом они должны быть совместимы с режимами, которые изначально применялись для открытия файлового дескриптора. Режимы `w` и `w+` можно указывать, но они не будут приводить к усечению файла. Поток данных устанавливается в файловую позицию, которая соответствует данному файловому дескриптору.

После преобразования файлового дескриптора в поток данных ввод-вывод больше не выполняется напрямую с этим файловым дескриптором. Тем не менее это не возбраняется. Обратите внимание: файловый дескриптор не дублируется, а просто ассоциируется с новым потоком данных. При закрытии потока данных закрывается и файловый дескриптор.

В случае успеха `fdopen()` возвращает допустимый указатель файла, при ошибке она возвращает `NULL` и присваивает `errno` соответствующее значение.

Например, следующий код открывает файл `/home/kidd/map.txt` с помощью системного вызова `open()`, а потом создает ассоциированный поток данных, опираясь на базовый файловый дескриптор:

```
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
    /* ошибка */

stream = fdopen (fd, "r");
if (!stream)
    /* ошибка */
```

Заккрытие потоков данных

Функция `fclose()` закрывает конкретный поток данных:

```
#include <stdio.h>

int fclose(FILE*stream);
```

Сначала сбрасываются на диск все буферизованные, но еще не записанные данные. В случае успеха `fclose()` возвращает `0`. При ошибке она возвращает `EOF` (конец файла) и устанавливает `errno` в соответствующее значение.

Заккрытие всех потоков данных. Функция `fcloseall()` закрывает все потоки данных, ассоциированные с конкретным процессом, в частности используемые для стандартного ввода, стандартного вывода и стандартных ошибок:

```
#define _GNU_SOURCE

#include <stdio.h>

int fcloseall(void);
```

Перед закрытием все потоки данных сбрасываются на диск. Функция является специфичной для Linux и всегда возвращает `0`.

Считывание из потока данных

Теперь, когда мы умеем открывать и закрывать потоки данных, поговорим о том, как сделать что-то полезное — как считать поток данных, а затем как записать в него информацию.

Стандартная библиотека C реализует множество функций для считывания из открытого потока данных. Среди них есть как общеизвестные, так и мало-распространенные. В этом разделе мы рассмотрим три наиболее популярных варианта считывания: считывание одного символа в момент времени, считывание целой строки в момент времени, считывание двоичных данных. Чтобы из потока данных можно было считать информацию, он должен быть открыт как поток данных ввода в подходящем режиме, то есть в любом допустимом режиме, кроме `w` и `a`.

Считывание одного символа в момент времени

Зачастую идеальный принцип ввода-вывода сводится к считыванию одного символа в момент времени. Функция `fgetc()` используется для считывания отдельного символа из потока данных:

```
#include <stdio.h>
```

```
int fgetc(FILE*stream);
```

Эта функция считывает следующий символ из `stream` и возвращает его как `unsigned char`, приведенный к `int`. Такое приведение осуществляется, чтобы получить достаточно широкий диапазон для уведомлений EOF или описания ошибок: в таких случаях возвращается EOF. Возвращаемое значение `fgetc()` должно быть сохранено в `int`. Сохранение в `char` — распространенный и опасный промах, ведь в таком случае вы не можете обнаруживать ошибки.

В следующем коде мы считываем отдельно взятый символ из `stream`, проверяем наличие ошибок и выводим результат как `char`:

```
int c;
```

```
c = fgetc (stream);
```

```
if (c == EOF)
    /* ошибка */
```

```
else
```

```
    printf ("c=%c\n", (char) c);
```

Поток данных, указанный в `stream`, должен быть открыт для чтения.

Возврат символа в поток данных. В рамках стандартного ввода-вывода предоставляется функция для перемещения символа обратно в поток данных. С ее помощью вы можете «заглянуть» в поток данных и вернуть символ обратно, если окажется, что он вам не подходит:

```
#include<stdio.h>

int ungetc(int c, FILE*stream);
```

При каждом вызове мы отправляем обратно в поток данных `stream` символ `c`, приведенный к `unsigned char`. В случае успеха возвращается `c`, в случае ошибки — `EOF`. При следующем считывании из `stream` будет возвращен `c`. Если мы возвращаем в поток данных несколько символов, то они приходят туда в обратном порядке: *последним пришел — первым вернулся*. Образуется своеобразный стек. Согласно стандарту С лишь один такой возврат должен гарантированно быть успешным при отсутствии «вклинивающихся» запросов на считывание. В свою очередь, в некоторых реализациях разрешается всего один такой возврат. В Linux допускается любое количество возвратов при условии наличия достаточного объема памяти. Как минимум один такой возврат в Linux, разумеется, гарантированно будет успешным.

Если в ходе возврата вы делаете «вклинивающийся» вызов функции позиционирования (см. разд. «Позиционирование в потоке данных» текущей главы) после вызова `ungetc()`, но еще до выдачи запроса на считывание, то все символы, отправленные обратно, будут удалены. Именно это происходит и при работе с программными потоками в одном процессе, так как программные потоки в процессе совместно используют общий буфер.

Считывание целой строки

Функция `fgets()` считывает строку из указанного потока данных:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

Эта функция может считать из потока данных `stream` количество байтов, как максимум *на один меньше*, чем количество `size` байт. Результат считывания сохраняется в `str`. Символ нуля (`\0`) сохраняется в буфере после последнего считанного байта. Считывание прекращается, когда достигнут конец файла или первый символ новой строки. Если начинается считывание новой строки, то в `str` сохраняется `\n`.

В случае успеха возвращается `str`, в случае ошибки — `NULL`.

Например:

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
    /* ошибка */
```

POSIX определяет `LINE_MAX` в `<limits.h>`: это максимальный размер, который может иметь строка ввода, чтобы интерфейсы POSIX для манипуляций со строками могли ею оперировать. В библиотеке С такое ограничение отсутствует — строки могут быть любого размера, — но мы никак не можем сообщить об этом в самом

определении `LINE_MAX`. Для обеспечения надежности переносимых программ в них можно обходиться `LINE_MAX`; в Linux значение этого параметра задано достаточно высоким. В специфичных для Linux программах нет необходимости беспокоиться о границах размеров строк.

Считывание произвольных строк. Зачастую построчное считывание `fgets()` — как раз то, что требуется. Однако не реже оно только мешает. Иногда разработчик предпочитает воспользоваться разделителем, а не переходить на новую строку. В других случаях разделитель вообще не нужен, и уж совсем редко разработчику может понадобиться, чтобы разделитель сохранялся в буфере! Практика показывает, что решение сохранять переход на новую строку в возвращаемом буфере чаще всего оказывается ошибочным.

В качестве замены несложно написать функцию `fgets()`, которая будет использовать `fgetc()`. Например, в следующем фрагменте считывается $n - 1$ байт из `stream` в `str`, после чего к полученной информации дозаписывается символ `\0`:

```
char *s;
int c;

s = str;
while (--n > 0 &&(c = fgetc (stream)) != EOF)
    *s++ = c;
*s = '\0';
```

Этот пример можно изменить, чтобы не считывался и разделитель (в качестве разделителя выступает целое число `d`, в этом примере мы не можем использовать в таком качестве символ нуля).

```
char *s;
int c = 0;

s = str;
while (--n > 0 &&(c = fgetc (stream)) != EOF &&(*s++ = c) != d)
    ;

if (c == d)
    *--s = '\0';
else
    *s = '\0';
```

Если установить значение `d` в `\n`, получим примерно такое же поведение, как и у `fgets()`, но уже не будем сохранять в буфере переход на новую строку.

В зависимости от реализации `fgets()` этот вариант может работать медленнее, так как он неоднократно делает вызовы функции `fgetc()`. Однако это немного иная проблема, чем показанная в нашем исходном примере с `dd`! Да, в последнем фрагменте мы имеем издержки, связанные с дополнительными вызовами функций, но избавляемся от нагрузки, которая была обусловлена избыточными системными вызовами, а также от неудобств, связанных с невыровненным вводом-выводом при `dd c bs=1`, — последняя проблема намного серьезнее.

Считывание двоичных данных

В некоторых приложениях считывать отдельные символы или строки недостаточно. Иногда разработчику требуется считывать и записывать сложные двоичные данные, в частности структуры C. Для этой цели в стандартной библиотеке ввода-вывода предоставляется функция `fread()`:

```
#include <stdio.h>
```

```
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

При вызове `fread()` мы можем прочитать вплоть до `nr` фрагментов данных, каждый размером `size` байт. Считывание происходит из потока данных `stream` в буфер, указанный `buf`. Значение файлового указателя увеличивается на число, равное количеству прочитанных байтов.

Возвращается количество считанных элементов (но не количество считанных байтов!). При ошибке или достижении конца файла функция возвращает значение, меньшее чем `nr`. К сожалению, мы не можем узнать, какое именно из двух условий наступило, — для этого потребуется специально применить функции `ferror()` и `feof()` (см. разд. «Ошибки и конец файла» этой главы).

Из-за различий, связанных с переменным размером, выравниванием, заполнением и порядком байтов, двоичные данные, записанные одним приложением, могут быть непригодны для считывания другим приложением либо даже тем же самым приложением на другой машине.

Проблемы, связанные с выравниванием

Во всех машинных архитектурах действуют свои требования к выравниванию данных. Обычно программисты представляют себе память просто как массив буферов. Однако процессор читает и записывает память не фрагментами, равными энному количеству байтов. При обращении к памяти процессор делает это с конкретной детализацией — например, по 2, 4, 8 или 16 байт. Адресное пространство каждого процесса начинается с 0, поэтому процессы должны инициировать доступ с адреса, который является целочисленным кратным единицы детализации.

Соответственно, сохранять переменные C и обращаться к ним нужно с выровненных адресов. В принципе, переменным свойственно естественное выравнивание, то есть выравнивание, соответствующее размеру типа данных в языке C. Например, 32-битное целое число выровнено по 4-байтовым границам. Иными словами, в большинстве архитектур `int` должно сохраняться в памяти по адресу, значение которого без остатка делится на 4.

При доступе к невыровненным данным возникают различные проблемы, зависящие от машинной архитектуры. Некоторые процессоры могут обращаться к невыровненным данным, но за счет снижения производительности. Другие процессоры вообще не умеют обращаться к невыровненным данным, при попытке это сделать возникает аппаратное исключение. Хуже того, некоторые процессоры просто бесшумно удаляют младшие биты, чтобы принудительно добиться выравнивания адреса. Практически всегда это приводит к непредвиденным последствиям.

Как правило, компилятор естественным образом выравнивает все данные, и это не представляет заметной проблемы для программиста. Однако при работе со структурами, управлении памятью вручную, сохранении двоичных данных на диск и обмене информацией по сети вопросы выравнивания данных могут стать актуальными, поэтому системный программист должен уметь справляться с подобными сложностями.

Мы подробнее поговорим о выравнивании в гл. 9.

Простейший образец `fread()` применяется для считывания одного элемента линейных байтов из указанного потока данных:

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream);
if(nr== 0)
    /* ошибка */
```

Ниже мы рассмотрим более сложные примеры — они встретятся вам при изучении записывающей функции `fwrite()`, парной для `fread()`.

Запись в поток данных

Стандартная библиотека C содержит функции не только для чтения, но и для записи в открытый поток данных. В этом разделе мы рассмотрим три наиболее популярных метода записи: запись отдельного символа, запись строки символов и запись двоичных данных. Такие разнообразные варианты записи идеально подходят для буферизованного ввода-вывода. Чтобы мы могли записать информацию в поток данных, он должен быть открыт как поток данных вывода в нужном режиме. Подойдут любые допустимые режимы, кроме `r`.

Запись отдельного символа

Функция, парная `fgetc()`, называется `fputc()`:

```
#include <stdio.h>

int fputc (int c, FILE *stream);
```

Функция `fputc()` записывает байт, указанный в `c` (приведенный к `unsigned char`), в поток данных, указанный в `stream`. При успешном завершении операции функция возвращает `c`. В противном случае она возвращает `EOF` и присваивает `errno` соответствующее значение.

Пример использования прост:

```
if(fputc('p', stream) == EOF)
    /* ошибка */
```

В данном примере мы записываем символ `p` в поток данных `stream`, который должен быть открыт для записи.

Запись строки символов

Функция `fputs()` используется для записи целой строки в заданный поток данных:

```
#include <stdio.h>
```

```
int fputs (const char *str, FILE *stream);
```

При вызове `fputs()` все содержимое строки, оканчивающейся нулем, записывается в поток данных, указанный в `stream`. Сама эта строка указывается в `str`. В случае успеха `fputs()` возвращает неотрицательное число. При ошибке она возвращает `EOF`.

В следующем примере файл открывается для внесения данных в режиме дозаписи. Указанная строка записывается в ассоциированный поток данных, после чего этот поток данных закрывается:

```
FILE *stream;
```

```
stream = fopen ("journal.txt", "a");
```

```
if (!stream)
```

```
    /* ошибка */
```

```
if (fputs ("The ship is made of wood.\n", stream) == EOF)
```

```
    /* ошибка */
```

```
if (fclose (stream) == EOF)
```

```
    /* ошибка */
```

Запись двоичных данных

Отдельных символов и строк недостаточно, если программе требуется записывать сложные данные. Для непосредственного сохранения двоичных данных, например переменных языка `C`, в рамках стандартного ввода-вывода предоставляется функция `fwrite()`:

```
#include <stdio.h>
```

```
size_t fwrite (void *buf,  
               size_t size,  
               size_t nr,  
               FILE *stream);
```

При вызове `fwrite()` в поток данных `stream` записывается вплоть до `nr` элементов, каждый до `size` в длину. Берутся данные из буфера, указанного в `buf`. Значение файлового указателя увеличивается на число, равное общему количеству записанных байтов.

Возвращается количество успешно записанных элементов (но не количество байтов!). Возвращаемое значение меньше `nr` означает ошибку.

Пример программы, в которой используется буферизованный ввод-вывод

Давайте рассмотрим пример — фактически полнофункциональную программу, включающую в себя многие интерфейсы, уже изученные нами в этой главе. Сначала программа определяет структуру `struct pirate`, а потом определяет две переменные такого типа. Программа инициализирует одну из переменных, а потом записывает ее на диск через поток данных вывода в файл `data`. В другом потоке данных программа вновь считывает данные из файла `data` непосредственно в другой экземпляр `struct pirate`. Наконец, программа записывает содержимое структуры в стандартный вывод:

```
#include <stdio.h>

int main (void)
{
    FILE *in, *out;
    struct pirate {
        char          name[100];    /* настоящее имя */
        unsigned long  booty;        /* вознаграждение в фунтах стерлингов */
        unsigned int   beard_len;    /* длина бороды в дюймах */
    } p, blackbeard= { "EdwardTeach", 950, 48 };

    out = fopen ("data", "w");
    if (!out) {
        perror ("fopen");
        return 1;
    }

    if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
        perror ("fwrite");
        return 1;
    }

    if (fclose (out)) {
        perror ("fclose");
        return 1;
    }

    in = fopen ("data", "r");
    if (!in) {
        perror ("fopen");
        return 1;
    }

    if (!fread (&p, sizeof (struct pirate), 1, in)) {
        perror ("fread");
        return 1;
    }
}
```



```

    }

    if (fclose (in)) {
        perror ("fclose");
        return 1;
    }

    printf ("name=\"%s\" booty=%lu beard_len=%u\n",
           p.name, p.booty, p.beard_len);

    return 0;
}

```

На выходе, разумеется, имеем исходные значения:

```
name="Edward Teach" booty=950 beard_len=48
```

Опять же важно помнить, что из-за различий размеров переменных, при выравнивании и т. д. двоичные данные, записанные в одном приложении, могут оказаться нечитаемыми в других. Это означает, что иное приложение — или даже то же самое приложение на другой машине — может оказаться не в состоянии правильно считать данные, записанные с помощью `fwrite()`. В нашем примере представьте себе варианты развития событий, если изменится размер `unsigned long` или степень заполнения. Эти параметры гарантированно остаются постоянными лишь на машине определенного типа с конкретным интерфейсом ABI.

Позиционирование в потоке данных

Часто бывает полезно манипулировать текущей позицией в потоке данных. Допустим, приложение считывает сложный файл, требующий манипулирования записями, и программа должна произвольно перемещаться по этому файлу. В другой ситуации, возможно, придется сбрасывать поток данных на нулевую позицию. Для подобных случаев в стандартной библиотеке ввода-вывода предоставляется семейство интерфейсов, функционально эквивалентных системному вызову `lseek()` (он был рассмотрен в гл. 2). Функция `fseek()`, наиболее распространенный интерфейс позиционирования из инструментов стандартного ввода-вывода, управляет файловой позицией в потоке данных `stream` в зависимости от значений `offset` и `whence`:

```
#include <stdio.h>
```

```
int fseek (FILE *stream, long offset, int whence);
```

Если аргумент `whence` имеет значение `SEEK_SET`, то файловая позиция устанавливается в `offset`. Если `whence` равен `SEEK_CUR`, файловая позиция получает значение, равное «текущая позиция плюс `offset`». Если `whence` имеет значение `SEEK_END`, то файловая позиция устанавливается в значение, равное «конец файла плюс `offset`».

При успешном завершении функция `fseek()` возвращает 0, стирает индикатор EOF и отменяет любые эффекты функции `ungetc()` (при их наличии). При ошибке она возвращает -1 и устанавливает `errno` в соответствующее значение. Самые распространенные ошибки — это недействительный поток данных (EBADF) и недействительный аргумент `whence` (EINVAL).

В качестве альтернативы для стандартного ввода-вывода предоставляется функция `fsetpos()`:

```
#include <stdio.h>
```

```
int fsetpos(FILE*stream, fpos_t*pos);
```

Эта функция устанавливает позицию в потоке данных `stream` в значение `pos`. Она действует так же, как и функция `fseek()` с аргументом `whence`, равным `SEEK_SET`. В случае успеха эта функция возвращает 0. В противном случае она возвращает -1 и присваивает `errno` соответствующее значение. Эта функция (а также парная ей функция `fgetpos()`, которую мы вскоре рассмотрим) предоставлена исключительно для работы с не относящимися к UNIX платформами, где для представления позиции в потоке данных используются сложные типы. При взаимодействии с такими платформами данная функция — единственный инструмент, позволяющий установить произвольную позицию в потоке данных, поскольку возможностей типа `long` языка C недостаточно. В специфичных для Linux приложениях этот интерфейс использовать нет необходимости, хотя и можно, если требуется обеспечить максимальную межплатформенную совместимость.

В стандартной библиотеке ввода-вывода также предоставляется функция быстрого доступа `rewind()`:

```
#include <stdio.h>
```

```
void rewind (FILE *stream);
```

Вызов этой функции:

```
rewind (stream);
```

сбрасывает значение позиции обратно на начало потока данных. Код эквивалентен следующему:

```
fseek (stream, 0, SEEK_SET);
```

с оговоркой, что она также очищает индикатор ошибки.

Обратите внимание: функция `rewind()` не имеет возвращаемого значения и поэтому не может напрямую сообщать о возникающих ошибках. Если вызывающей стороне требуется убедиться в наличии ошибки, то понадобится очистить `errno` до вызова, а потом проверить, сохранила ли переменная после этого ненулевое значение. Например:

```
errno= 0;
rewind(stream);
if(errno)
    /* ошибка */
```

Получение актуальной позиции в потоке данных. В отличие от `lseek()`, `fseek()` не возвращает обновленную позицию. Для этого предоставляется отдельный интерфейс. Функция `ftell()` возвращает текущую позицию из потока данных `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

При ошибке она возвращает `-1` и устанавливает `errno` в соответствующее значение.

В рамках стандартного ввода-вывода предоставляется и альтернативная функция `fgetpos()`:

```
#include <stdio.h>

int fgetpos(FILE*stream, fpos_t*pos);
```

В случае успеха `fgetpos()` возвращает `0` и устанавливает текущую позицию потока данных `stream` в значение `pos`. При ошибке она возвращает `-1` и присваивает `errno` соответствующее значение. Подобно `fsetpos()`, `fgetpos()` предоставляется лишь для работы с платформами не-UNIX, где для представления файловой позиции используются сложные типы.

Сброс потока данных

В стандартной библиотеке ввода-вывода есть интерфейс, позволяющий выписать содержимое пользовательского буфера в ядро. Он гарантирует, что все данные, записанные в поток данных, будут сброшены к ядру с помощью `write()`. Функция `fflush()` действует следующим образом:

```
#include <stdio.h>

int fflush (FILE *stream);
```

При вызове этой функции все незаписанные данные из потока данных, указанного в `stream`, сбрасываются в буфер ядра. Если значение `stream` равно `NULL`, то *все* открытые потоки данных этого процесса записываются в буфер ядра. В случае успеха `fflush()` возвращает `0`. В случае ошибки эта функция возвращает `EOF` и присваивает `errno` соответствующее значение.

Чтобы понять принцип действия `fflush()`, следует понимать разницу между буфером, поддерживаемым библиотекой C, и буферизацией, выполняемой в самом ядре. Все вызовы, описанные в данной главе, работают с буфером, поддерживаемым библиотекой C. Этот буфер располагается в пользовательском пространстве, и, следовательно, в нем работает пользовательский код, а не выполняются системные вызовы. Системный вызов выдается, только когда необходимо обратиться к диску или какому-нибудь другому носителю.

Функция `fflush()` просто записывает данные из пользовательского буфера в буфер ядра. Получается эффект, как будто пользовательская буферизация вообще не задействовалась и мы напрямую применили вызов `write()`. В данной ситуации не гарантируется физическая отправка данных на тот или иной носитель — для полной уверенности в благополучной отправке данных потребуется использовать что-то вроде `fsync()`. В ситуациях, когда необходимо знать, что ваши данные успешно отправлены в резервное хранилище, целесообразно вызвать `fsync()` непосредственно после `fflush()`. Таким образом, сначала мы убеждаемся, что информация из пользовательского буфера перенесена в буфер ядра, а потом гарантируем, что информация из буфера ядра попадет на диск.

Ошибки и конец файла

Некоторые интерфейсы стандартного ввода-вывода, например `fread()`, плохо справляются с информированием вызывающей стороны о сбоях, так как в них отсутствует механизм, позволяющий отличать ошибку от конца файла. При работе с такими вызовами, а также в некоторых других случаях бывает полезно проверить статус конкретного потока данных и определить, установлен ли на потоке данных `stream` индикатор ошибки:

```
#include<stdio.h>
```

```
int ferror(FILE*stream);
```

Индикатор ошибки устанавливается другими интерфейсами стандартного ввода-вывода в ответ на возникновение условия ошибки. Функция возвращает ненулевое значение, если индикатор установлен, а в других случаях возвращает 0.

Функция `feof()` проверяет, установлен ли на потоке данных `stream` индикатор EOF:

```
#include<stdio.h>
```

```
int feof(FILE*stream);
```

Индикатор EOF устанавливается другими интерфейсами стандартного ввода-вывода, когда достигается конец файла. Функция возвращает ненулевое значение, если индикатор установлен, а в других случаях возвращает 0.

Функция `clearerr()` удаляет с потока данных `stream` индикаторы ошибки и EOF:

```
#include <stdio.h>
```

```
void clearerr(FILE*stream);
```

Она не имеет возвращаемого значения, поэтому не может закончиться неудачно (мы не можем отследить случаи, в которых нами был получен недопустимый поток данных). Вызов `clearerr()` следует делать только после проверки индикаторов ошибок и EOF, так как после `clearerr()` они будут необратимо удалены. Например:

```
/* 'f' – допустимый поток данных */  
  
if(ferror(f))  
    printf("Error on f!\n");  
if (feof (f))  
    printf ("EOF on f!\n");  
  
clearerr (f);
```

Получение ассоциированного файлового дескриптора

Иногда предпочтительно получить файловый дескриптор, лежащий в основе конкретного потока данных. Например, нам может потребоваться выполнить применительно к потоку данных системный вызов. Это будет выполняться через файловый дескриптор потока данных, когда ассоциированной функции стандартного ввода-вывода не существует. Чтобы получить файловый дескриптор, лежащий в основе потока данных, используйте `fileno()`:

```
#include <stdio.h>  
  
int fileno(FILE*stream);
```

В случае успеха `fileno()` возвращает файловый дескриптор, ассоциированный с потоком данных `stream`. В случае ошибки она возвращает `-1`. Это может произойти, только когда заданный поток данных является недопустимым: в такой ситуации функция устанавливает `errno` значение `EBADF`.

Как правило, не рекомендуется смешивать вызовы стандартного ввода-вывода и системные вызовы. При использовании `fileno()` программист должен проявлять осторожность и гарантировать, что операции, в которых задействованы файловые дескрипторы, не будут конфликтовать с пользовательской буферизацией. В частности, целесообразно сначала сбрасывать поток данных, а потом начинать манипулировать лежащим в его основе файловым дескриптором. Практически ни в каких случаях не следует смешивать операции, связанные с файловыми дескрипторами и потоковым вводом-выводом.

Управление буферизацией

При стандартном вводе-выводе реализуется три типа пользовательской буферизации, и она предоставляет разработчикам интерфейс, позволяющий контролировать тип и размер буфера. Различные виды пользовательской буферизации применяются для разных целей и идеально подходят каждый для своей ситуации.

- **Без буферизации.** Пользовательская буферизация не применяется. Данные отправляются прямо в ядро. В таком случае нам приходится отказываться и от пользовательской буферизации, и от всех ее преимуществ, поэтому данный

подход используется редко, с одним исключением: по умолчанию без буферизации работает стандартная ошибка.

- **Построчная буферизация.** Буферизация выполняется построчно. На каждом символе перехода на новую строку содержимое буфера отправляется в ядро. Буферизация строк целесообразна при работе с потоками данных, чей вывод попадает на экран, поскольку выводимые на монитор сообщения всегда разделяются по строкам. Следовательно, этот способ буферизации по умолчанию используется с потоками данных, подключенными к терминалам, — например, со стандартным выводом.
- **Поблочная буферизация.** Буферизация выполняется поблочно. В данном случае блок — это фиксированное количество байтов. Именно о таком типе буферизации мы говорили в начале этой главы, такая буферизация идеально подходит для работы с файлами. По умолчанию поблочная буферизация применяется со всеми потоками данных, которые ассоциированы с файлами. В контексте стандартного ввода-вывода поблочная буферизация называется *полной буферизацией*.

В большинстве случаев заданный по умолчанию тип буферизации является оптимальным. Тем не менее стандартный ввод-вывод предоставляет интерфейс для управления типом применяемой буферизации:

```
#include <stdio.h>
```

```
int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

Функция `setvbuf()` устанавливает тип буферизации для потока данных `stream` в значение `mode`, которое может быть одним из следующих:

- `_IONBF` — без буферизации;
- `_IOLBF` — построчная буферизация;
- `_IOFBF` — поблочная буферизация.

За исключением значения `_IONBF`, при котором `buf` и `size` игнорируются, `buf` может указывать на буфер размером `size` байт. Стандартный ввод-вывод будет использовать этот буфер для данных указанного потока данных. Если `buf` равен `NULL`, то буфер указанного вами размера автоматически выделяется библиотекой `glibc`.

Функция `setvbuf()` может быть вызвана после открытия потока данных, но до того, как с ним будут выполняться какие-либо иные манипуляции. При успехе она возвращает 0, в другом случае — ненулевое значение.

Если буфер предоставлен, то он должен существовать и после закрытия потока данных. Распространенная ошибка — когда буфер определяется как автоматическая переменная в области видимости, заканчивающейся до закрытия потока данных. В частности, не следует создавать буфер, локальный для `main()`, так как впоследствии вы не сможете явно закрывать потоки данных. Например, следующий код содержит ошибку:

```
#include <stdio.h>

int main (void)
{
    charbuf[BUFSIZ];

    /* задаем для stdin поблочную буферизацию с буфером BUFSIZ */
    setvbuf(stdout, buf, _IOFBF, BUFSIZ);
    printf ("Arrr!\n");

    return 0;
    /* 'buf' выходит из области видимости и высвобождается, но stdout закрывается
уже позже */
}
```

Чтобы избежать ошибок такого типа, нужно явно закрывать поток данных до выхода из этого контекста либо делать `buf` глобальной переменной.

Как правило, разработчику не приходится задумываться о буферизации потоков. Если не считать стандартных ошибок, на всех терминалах используется построчная буферизация, и это имеет смысл. При работе с файлами используется поблочная буферизация, и это тоже верно. При поблочной буферизации задаваемый по умолчанию размер буфера равен `BUFSIZ`. Он определяется в `<stdio.h>`, и выбранное для него значение обычно также оптимальное (представляет собой кратное типичному размеру блока).

Безопасность программных потоков

Программные потоки (threads) — это рабочие единицы внутри процесса. Большинство процессов содержат всего один программный поток. Тем не менее процессы могут включать в себя и множество программных потоков, в каждом из которых выполняется своя задача. Такой процесс называется *многопоточным*. Многопоточный процесс можно представить как совокупность мелких процессов, делящих общее адресное пространство. Без явной координации программные потоки могут запускаться когда угодно и перемежаться друг с другом всевозможными способами. В многопроцессорной системе два и более программных потока, относящихся к одному процессу, могут даже работать параллельно. Программные потоки могут перезаписывать совместно используемые данные, если разработчик специально не организует синхронизацию доступа к данным (такая практика называется *блокировкой*) либо не сделает данные локальными для программного потока (в этом случае принято говорить о *привязке к потоку*).

Операционные системы, поддерживающие программные потоки, также предоставляют механизмы блокировки (специальные программные конструкторы, обеспечивающие взаимное исключение). Блокировка нужна, чтобы разные программные потоки не «наступали друг другу на пятки». Эти механизмы используются при стандартном вводе-выводе. В результате гарантируется, что отдельные программные

потоки, относящиеся к одному процессу, могут выполнять параллельные вызовы стандартного ввода-вывода — даже по отношению к одному и тому же потоку данных! При этом параллельные операции нисколько не мешают друг другу. Правда, этих механизмов иногда оказывается недостаточно, например, если требуется заблокировать группу вызовов, расширив таким образом *критический участок кода* (так называется фрагмент кода, работающий без всякой интерференции со стороны другого программного потока) с одной операции до нескольких. В других ситуациях может понадобиться вообще избавиться от блокировок для повышения эффективности¹. В этом разделе мы обсудим оба упомянутых варианта.

Функции стандартного ввода-вывода по определению являются *потокобезопасными*. На внутрисистемном уровне они ассоциируют с каждым открытым потоком данных саму блокировку, подсчет блокировок и владеющий программный поток. Любой программный поток должен получить блокировку и стать владельцем, лишь после этого он сможет выдавать какие-либо запросы на ввод-вывод. Два и более программных потока, оперирующих одним и тем же потоком данных, не могут перемежать операции стандартного ввода-вывода, поэтому в контексте отдельных вызовов функций операции стандартного ввода-вывода являются атомарными.

Разумеется, на практике многие приложения требуют большей атомарности, чем на уровне отдельных вызовов функций. Допустим, что множественные программные потоки, работающие в одном процессе, выдают запросы на запись. Функции стандартного ввода-вывода являются потокобезопасными, поэтому отдельные операции записи не будут накладываться друг на друга и приводить к искажению вывода. Таким образом, даже если два программных потока одновременно выдают запрос на запись, блокировка гарантирует, что один из этих запросов завершится раньше другого. Однако что, если процессу требуется выдать несколько запросов на запись подряд и исключить риск пересечения с запросами на запись, идущими от другого программного потока, не только для отдельного запроса, но и для всей этой последовательности запросов? Для таких нужд в стандартном вводе-выводе предоставляется семейство функций для индивидуальных операций с блокировкой, ассоциированной с конкретным потоком данных.

Блокировка файлов вручную

Функция `flockfile()` дожидается, пока не будет снята блокировка с потока данных `stream`, увеличивает количество блокировок на единицу, а потом получает блокировку и становится владельцем программным потоком этого потока данных. Затем она возвращается:

```
#include <stdio.h>
```

```
void flockfile (FILE *stream);
```

¹ Как правило, если избавиться от блокировок, сразу возникнет целый набор проблем. Однако в некоторых программах для обеспечения безопасности программных потоков весь ввод-вывод может быть делегирован только одному программному потоку. Это вариант привязки к потоку. В таком случае блокировки действительно не требуются.

Функция `funlockfile()` уменьшает количество блокировок, ассоциированное с потоком данных `stream`:

```
#include <stdio.h>
```

```
void funlockfile (FILE*stream);
```

Если количество блокировок достигает нуля, то текущий программный поток прекращает владеть соответствующим потоком данных. Теперь блокировку сможет получить другой программный поток.

Такие вызовы могут быть вложенными. Это означает, что отдельно взятый программный поток может выдавать множественные вызовы `flockfile()` и поток данных не разблокируется, пока процесс не выдаст ровно такое же количество вызовов `funlockfile()`.

Функция `ftrylockfile()` представляет собой неблокирующую разновидность `flockfile()`:

```
#include <stdio.h>
```

```
int ftrylockfile (FILE*stream);
```

Если поток данных `stream` в конкретный момент заблокирован, то `ftrylockfile()` ничего не делает и сразу же возвращает ненулевое значение. Если поток данных `stream` в этот момент не заблокирован, функция получает блокировку, увеличивает количество блокировок, становится владеющим программным потоком потока данных `stream` и возвращает 0.

Рассмотрим пример. Допустим, мы хотим записать несколько строк в файл и гарантировать, что при записи они не будут перемежаться с операциями записи, идущими от других потоков:

```
flockfile (stream);
```

```
fputs ("List of treasure:\n", stream);
```

```
fputs (" (1) 500 gold coins\n", stream);
```

```
fputs (" (2) Wonderfully ornate dishware\n", stream);
```

```
funlockfile (stream);
```

Хотя отдельные операции `fputs()` никогда не вступают в условия гонки с другими операциями ввода/вывода — например, у вас никогда не возникнет пересечения чего бы то ни было с `List of treasure`, — в работу такой функции может вклиниться другая операция стандартного ввода-вывода, поступающая от другого программного потока к тому же потоку данных. Это может происходить в промежутке между двумя вызовами `fputs()`. В идеальном случае приложение должно быть построено так, чтобы множественные программные потоки не направляли ввод-вывод в один и тот же поток данных. Если же в вашем приложении такая отправка необходима, но при этом нужен и атомарный регион шире одной функции, то `flockfile()` со товарищи вам очень пригодятся.

Неблокируемые потоковые операции

Есть и другая причина, по которой может потребоваться блокировать потоки данных вручную. Обеспечив более тонкий и точный контроль блокировок (это под силу только системному программисту), вы сможете минимизировать издержки, связанные с блокировками, и улучшить таким образом производительность. Для этой цели в Linux предоставляется семейство функций, родственных стандартным интерфейсам ввода-вывода. Функции из этого семейства отличаются тем, что вообще не выполняют блокировок. Фактически это неблокируемые аналоги функций стандартного ввода-вывода:

```
#define _GNU_SOURCE

#include <stdio.h>

int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf, size_t size, size_t nr,
                       FILE *stream);
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr,
                       FILE *stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked (FILE *stream);
int fileno_unlocked (FILE *stream);
void clearerr_unlocked (FILE *stream);
```

Все эти функции работают идентично своим блокируемым аналогам, но не проверяют наличия блокировок и не получают блокировку, ассоциированную с конкретным потоком `stream`. Если блокировка необходима, то именно программист должен вручную обеспечить ее получение и последующее высвобождение.

ВНИМАНИЕ

При использовании неблокируемых стандартных функций ввода-вывода значительно повышается производительность. Более того, код радикально упрощается, если не приходится беспокоиться о блокировании сложных операций с помощью `flockfile()`. Создавая приложение, попробуйте передать весь ввод-вывод в один программный поток (либо перепоручить его пулу программных потоков, в котором каждый поток данных отображается ровно на один программный поток).

Хотя в POSIX и определяется несколько неблокируемых вариантов функций стандартного ввода-вывода, ни одна из вышеприведенных функций в POSIX не описана. Все они специфичны для Linux, хотя это подмножество и поддерживается в некоторых других системах UNIX.

Мы подробно поговорим о программных потоках в гл. 7.

Недостатки стандартного ввода-вывода

Как ни часто используется стандартный ввод-вывод, часть экспертов указывают на определенные его недочеты. Некоторые функции, например `fgets()`, бывают неэффективными или плохо сработанными. Другие функции, к примеру `gets()`, настолько ужасны, что их практически изгнали из всех стандартов.

Самые серьезные претензии к стандартному вводу-выводу связаны с его негативным влиянием на производительность, которое объясняется необходимостью двойного копирования. При считывании данных стандартный ввод-вывод направляет к ядру системный вызов `read()`, копируя данные из ядра в буфер стандартного ввода-вывода. Если после этого приложение с помощью стандартного ввода-вывода делает запрос на запись, применяя, скажем, `fgetc()`, то данные копируются вновь — на этот раз из буфера стандартного ввода-вывода в указанный буфер. Запросы на запись работают прямо противоположным образом: сначала данные копируются из предоставленного буфера в буфер стандартного ввода/вывода, а оттуда — в ядро посредством вызова `write()`.

Альтернативная реализация позволяет избежать двойного копирования. Для этого нужно при каждом запросе на считывание возвращать указатель на буфер стандартного ввода-вывода. После этого данные можно считывать напрямую внутри буфера стандартного ввода-вывода без необходимости дополнительного копирования. Если приложению действительно потребуются данные из его собственного локального буфера — например, чтобы что-то дописать к ним, — то мы всегда можем организовать копирование вручную. Такая реализация предоставляла бы «свободный» интерфейс, позволяющий приложениям сигнализировать о завершении обработки конкретного фрагмента кода из буфера считывания.

Запись протекала бы несколько сложнее, но двойного копирования по-прежнему не было бы. При выдаче запроса на запись эта реализация требовала бы использовать указатель. В конечном итоге, когда данные были бы готовы к сбросу в ядро, такая реализация могла пройти по всему списку сохраненных указателей, записывая данные. Это можно сделать посредством фрагментирующего/дефрагментирующего ввода-вывода, для которого требуется лишь один системный вызов `writew()`. Мы поговорим о фрагментирующем/дефрагментирующем вводе-выводе в следующей главе.

Есть исключительно удобные пользовательские библиотеки, решающие проблему двойного копирования посредством реализаций, очень напоминающих предложенную выше. В качестве альтернативы многие разработчики реализуют собственные решения, обеспечивающие буферизацию. Однако, несмотря на альтернативы, стандартный ввод-вывод остается популярным.

Резюме

Стандартный ввод-вывод — это библиотека для пользовательской буферизации, предоставляемая в рамках стандартной библиотеки C. Несмотря на определенные

недостатки, это мощное и очень популярное решение. На самом деле многие С-программисты умеют обращаться только со стандартным вводом-выводом. Для терминального ввода-вывода, которому идеально подходит строчная буферизация, стандартный ввод-вывод является практически единственным подходящим решением. Нечасто ведь приходится использовать `write()` для передачи информации в стандартный вывод!

Стандартный ввод-вывод и, если уж на то пошло, пользовательская буферизация вообще оправданны, лишь если выполняются все следующие условия:

- предположительно вы будете делать много системных вызовов и хотите минимизировать сопутствующие издержки, объединив несколько таких вызовов в один;
- очень важна высокая производительность, и вы хотите гарантировать, что весь ввод-вывод будет осуществляться поблочными фрагментами, четко ограниченными размерами блоков;
- интересующие вас шаблоны доступа основаны на символах или строках, и вам нужны интерфейсы, обеспечивающие такой доступ без выполнения излишних системных вызовов;
- вы предпочитаете использовать сравнительно высокоуровневый интерфейс, а не низкоуровневые системные вызовы Linux.

Однако максимальная гибкость обеспечивается, когда вы работаете непосредственно с системными вызовами Linux. В следующей главе мы рассмотрим продвинутые формы ввода-вывода и связанные с ними системные вызовы.

4 Расширенный файловый ввод-вывод

В гл. 2 мы рассмотрели простейшие системные вызовы ввода-вывода, применяемые в Linux. Эти вызовы являются не только базисом файлового ввода-вывода, но и основой практически любого обмена информацией в Linux. В гл. 3 мы изучали, зачем может потребоваться пользовательская буферизация, зачастую выстраиваемая на базе простейших системных вызовов ввода-вывода. Мы также ознакомились с решением, специально предназначенным для выполнения пользовательской буферизации, — стандартной библиотекой C. В этой главе мы займемся расширенными системными вызовами ввода-вывода, имеющимися в Linux.

- **Фрагментированный ввод-вывод.** Позволяет отдельно взятому вызову записывать данные или считывать их из многих буферов одновременно. Полезен для объединения в пакеты полей из разных структур данных и для формирования транзакции ввода-вывода.
- **Интерфейс `epoll`.** Усовершенствованный вариант системных вызовов `poll()` и `select()`, описанных в гл. 2. Полезен, когда в ходе опроса требуется получить из одного потока сотни файловых дескрипторов.
- **Ввод-вывод с отображением в память.** Отображает файл на пространство памяти, обеспечивая выполнение файлового ввода-вывода путем простых операций в памяти. Полезен в некоторых шаблонах ввода-вывода.
- **Файловые извещения.** Эта техника позволяет процессу давать ядру подсказки, как предполагается использовать файл в данном процессе; может улучшить производительность ввода-вывода.
- **Асинхронный ввод-вывод.** Позволяет процессу выдавать запросы на ввод-вывод, не дожидаясь завершения этих операций. Полезен при оперировании большими объемами ввода-вывода без использования потоков.

В конце этой главы мы обсудим вопросы производительности и подсистемы ввода-вывода, входящие в состав ядра.

Фрагментированный ввод-вывод

Фрагментированный ввод-вывод — это способ ввода-вывода, при котором отдельно взятый системный вызов позволяет записать вектор буферов из одного потока

данных. Данный тип ввода-вывода получил такое название, поскольку данные в виде небольших фрагментов рассыпаны в векторе буферов, откуда программе приходится их собирать. Альтернативное название этого подхода — *векторный ввод/вывод*. Стандартные системные вызовы, применяемые для чтения и записи, рассмотренные нами в гл. 2, обеспечивают *линейный ввод/вывод*.

Фрагментированный ввод/вывод имеет преимущества перед линейным.

- **Более логичный принцип написания кода.** Если ваши данные сегментированы естественным образом — допустим, представлены в виде полей, имеющих заранее определенную структуру, — векторный ввод-вывод обеспечивает интуитивно понятные манипуляции с кодом.
- **Эффективность.** Всего одна операция векторного ввода-вывода может заменить множество операций линейного.
- **Производительность.** Кроме снижения количества выполняемых системных вызовов, векторная реализация ввода-вывода может обеспечить более высокую производительность по сравнению с линейной, что достигается посредством внутрисистемной оптимизации.
- **Атомарность.** В отличие от ситуации с множественными линейными операциями ввода-вывода, процесс может выполнять всего одну операцию векторного ввода-вывода без риска наложения на ввод-вывод, происходящий в другом процессе.

Системные вызовы `readv()` и `writew()`

Стандарт POSIX 1003.1-2001 описывает, а Linux реализует пару системных вызовов, обеспечивающих фрагментированный ввод-вывод. Реализация Linux соответствует всем преимуществам, перечисленным в предыдущем разделе.

Функция `readv()` считывает `count` сегментов из файлового дескриптора `fd` в буферы, описанные `iov`:

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

ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

Функция `writew()` записывает не более `count` сегментов из буферов, описанных в `iov`, в файловый дескриптор `fd`:

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

ssize_t writew (int fd,
                const struct iovec *iov,
                int count);
```

Функции `readv()` и `writew()` работают так же, как и `read()` и `write()` соответственно, за тем исключением, что считывание или запись в первых двух случаях затрагивает сразу множество буферов.

Каждая структура `iovec` описывает самостоятельный отдельный буфер, называемый *сегментом*:

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

struct iovec{
    void *iov_base; /* указатель на начало буфера */
    size_t iov_len; /* размер буфера в байтах */
};
```

Набор сегментов называется *вектором*. Каждый сегмент в векторе описывает адрес и длину того буфера в памяти, в который должны быть записаны данные (или из которого они должны быть считаны). Функция `readv()` целиком заполняет каждый буфер, имеющий длину `iov_len` байт, после чего переходит к следующему буферу. Функция `writev()` всегда записывает полный буфер `iov_len`, прежде чем перейти к следующему буферу. Обе функции всегда обрабатывают сегменты по порядку, начиная с `iov[0]`, затем `iov[1]` и т. д. до `iov[count-1]`.

Возвращаемые значения

В случае успеха функции `readv()` и `writev()` возвращают количество считанных или записанных байт соответственно. Это количество должно быть суммой всех значений `count iov_len`. При ошибке эти вызовы возвращают `-1` и устанавливают `errno` в соответствующее значение. Эти системные вызовы могут сталкиваться с любыми из ошибок, которые характерны для `read()` и `write()`, и в случае получения таких ошибок будут выдавать те же коды `errno`. Кроме того, стандарты определяют для `readv()` и `writev()` еще два варианта ошибки.

Во-первых, поскольку возвращаемый тип равен `ssize_t`, то, если сумма всех значений `iov_len` превышает `SSIZE_MAX`, никакие данные не будут возвращены, а просто будет возвращено значение `-1`, а `errno` будет установлено в `EINVAL`.

Во-вторых, POSIX требует, чтобы `count` было больше нуля, но меньше или равно `IOV_MAX`, что определяется в `<limits.h>`. В современных системах Linux значение `IOV_MAX` составляет 1024. Если `count` равно нулю, то эти системные вызовы возвращают `0`¹. Если значение `count` превышает `IOV_MAX`, то никакие данные не передаются, вызовы возвращают `-1`, а `errno` устанавливается в значение `EINVAL`.

ПРИМЕЧАНИЕ

В ходе операций векторного ввода-вывода ядро Linux должно выделять внутренние структуры данных для представления каждого сегмента. Как правило, такое выделение происходит динамически, в зависимости от значения `count`. Однако в целях оптимизации ядро Linux создает в стеке небольшой массив сегментов, которым пользуется, когда значение `count` достаточно невелико. В результате исчезает необходимость динамического выделения сегментов и производительность несколько улучшается. В настоящее время данный порог равен 8, поэтому если значение `count` меньше или равно 8, то векторные операции ввода-вывода очень экономно расходуют память и происходят за рамками стека ядра.

Скорее всего, вы не сможете выбрать, сколько сегментов информации придется одновременно передавать в рамках одной векторной операции ввода-вывода. Однако если вы работаете гибко и всерьез раздумываете над такими мелкими значениями, то при выборе значения 8 и менее производительность можно существенно увеличить.

¹ Обратите внимание: в других системах UNIX `errno` может устанавливаться в `EINVAL`, если `count` равно нулю. Это явно разрешено в стандартах, указывающих, что при нулевом значении можно устанавливать `EINVAL`, если значение действительно равно нулю, и система может обработать такой нулевой случай иным (неошибочным) способом.

Пример использования writev()

Рассмотрим простой пример, в котором записывается вектор из трех сегментов, причем все три содержат по строке разного размера. Эта самодостаточная программа вполне детально описывает writev(), оставаясь при этом компактной:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/uio.h>

int main ()
{
    struct iovec iov[3];
    ssize_t nr;
    int fd, i;

    char *buf[] = {
        "The term buccaneer comes from the word boucan.\n",
        "A boucan is a wooden frame used for cooking meat.\n",
        "Buccaneer is the West Indies name for a pirate.\n" };

    fd = open ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC);
    if (fd == -1) {
        perror ("open");
        return 1;
    }

    /* заполняем три структуры iovec */
    for(i= 0; i<3; i++) {
        iov[i].iov_base = buf[i];
        iov[i].iov_len = strlen(buf[i]) + 1;
    }
    /* записываем все данные в рамках единственного вызова */
    nr = writev (fd, iov, 3);
    if (nr == -1) {
        perror ("writev");
        return 1;
    }
    printf ("wrote %d bytes\n", nr);

    if (close (fd)) {
        perror ("close");
        return 1;
    }

    return 0;
}
```


При запуске программы получаем желаемый результат

```
$ ./writev
wrote 148 bytes
```

а также файл следующего содержания:

```
$ cat buccaneer.txt
The term buccaneer comes from the word boucan.
A boucan is a wooden frame used for cooking meat.
Buccaneer is the West Indies name for a pirate.
```

Пример использования `readv()`

Теперь рассмотрим пример программы, в которой используется системный вызов `readv()`. Здесь он будет считывать информацию из предварительно сгенерированного текстового файла, используя векторный ввод-вывод. Этот самодостаточный пример получается не сложнее приведенного выше:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>

int main ()
{
    char foo[48], bar[51], baz[49];
    struct iovec iov[3];
    ssize_t nr;
    int fd, i;

    fd = open ("buccaneer.txt", O_RDONLY);
    if (fd == -1) {
        perror ("open");
        return 1;
    }

    /* задаем наши структуры iovec */
    iov[0].iov_base= foo;
    iov[0].iov_len = sizeof (foo);
    iov[1].iov_base = bar;
    iov[1].iov_len = sizeof (bar);
    iov[2].iov_base = baz;
    iov[2].iov_len = sizeof (baz);

    /* считываем все структуры в рамках одного вызова */
    nr = readv (fd, iov, 3);
    if (nr == -1) {
        perror ("readv");
    }
}
```

```

        return 1;
    }

    for (i = 0; i < 3; i++)
        printf ("%d: %s", i, (char *) iov[i].iov_base);

    if (close (fd)) {
        perror ("close");
        return 1;
    }

    return 0;
}

```

Если запустить эту программу после предыдущей, то получим следующий результат:

```

$ ./readv
0: The term buccaneer comes from the word boucan.
1: A boucan is a wooden frame used for cooking meat.
2: Buccaneer is the West Indies name for a pirate.

```

Реализация

Упрощенную реализацию `readv()` и `writew()` можно выполнить в пользовательском пространстве как простой цикл, например:

```

#include <unistd.h>
#include <sys/uio.h>

ssize_t naive_writew (int fd, const struct iovec *iov, int count)
{
    ssize_t ret = 0;
    int i;

    for (i = 0; i < count; i++) {
        ssize_t nr;

        errno = 0;
        nr = write (fd, iov[i].iov_base, iov[i].iov_len);
        if (nr == -1) {
            if (errno == EINTR)
                continue;
            ret = -1;
            break;
        }
        ret += nr;
    }

    return ret;
}

```

К счастью, в Linux эта реализация выполнена *иначе*. В Linux `readv()` и `writew()` являются системными вызовами и на внутрисистемном уровне выполняют фрагментированный ввод-вывод. На самом деле весь ввод-вывод в ядре Linux является векторным. Даже вызовы `read()` и `write()` выполняют векторный ввод-вывод, где каждый вектор содержит всего один сегмент.

Опрос событий

Учитывая ограничения, свойственные вызовам `poll()` и `select()`, разработчики ядра Linux 2.6¹ внедрили возможность *опроса событий*. Интерфейс `epoll` сложнее, чем два рассмотренных выше, но он решает фундаментальную проблему производительности, характерную для `poll()` и `select()`, а также добавляет ряд новых возможностей.

Системные вызовы `poll()` и `select()` (рассмотренные в гл. 2) должны наблюдать за всеми действующими файловыми дескрипторами. После этого ядро проходит по всему списку дескрипторов, которые необходимо отслеживать. Когда этот список становится слишком велик — в нем могут содержаться сотни и даже тысячи файловых дескрипторов, — то проход по всему списку при каждом вызове превращается в то узкое место, которое осложняет масштабируемость.

При опросе событий эту проблему удастся обойти, так как регистрация наблюдателя открепляется от самого акта наблюдения. Один системный вызов инициализирует контекст опроса событий, другой добавляет в контекст наблюдаемые файловые дескрипторы или удаляет их оттуда, третий выполняет само ожидание события.

Создание нового экземпляра `epoll`

Контекст опроса событий создается с помощью вызова `epoll_create1()`:

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

```
int epoll_create1 (int flags);  
/* устарело. В новом коде используйте epoll_create1() */  
int epoll_create (int size);
```

При успешном вызове `epoll_create1()` создается новый экземпляр `epoll` и возвращается файловый дескриптор, ассоциированный с данным экземпляром. Этот файловый дескриптор не имеет отношения к реальному файлу; это просто указатель, который должен применяться с последующими вызовами, задействующими возможность опроса событий. Параметр `flags` позволяет модифицировать поведение опроса событий. В настоящее время допустимым считается только один флаг — `EPOLL_CLOEXEC`. Он обеспечивает автоматическое закрытие дескриптора во время вызова.

¹ Опрос событий появился в рабочей версии ядра 2.5.44, а интерфейс `epoll` был закончен в версии 2.5.66. Эта возможность является специфичной для Linux.

При ошибке этот вызов возвращает -1 и устанавливает `errno` в одно из следующих значений:

- `EINVAL` — недействительный параметр `flags`;
- `EMFILE` — пользователь достиг допустимого лимита на количество открытых файлов;
- `ENFILE` — система достигла допустимого лимита на количество открытых файлов;
- `ENOMEM` — недостаточно памяти для завершения операции.

Вызов `epoll_create()` — это устаревший и выходящий из употребления вариант `epoll_create1()`. Он не принимает никаких флагов. Вместо этого он принимает аргумент `size`, который в настоящее время уже не используется. Раньше `size` применялся в качестве подсказки о том, сколько файловых дескрипторов мы собираемся наблюдать. В настоящее время ядро динамически задает размер для обрабатываемых структур данных, и этот параметр просто должен быть больше нуля. В противном случае возвращается `EINVAL`. Новые приложения должны использовать старый вариант лишь при условии, что целевая система работает с версией Linux, в которой еще не появился `epoll_create1()`. Соответственно, система должна иметь версию ядра ниже 2.6.27 и `glibc 2.9`.

Как правило, этот вызов выглядит так:

```
int epfd;
```

```
epfd = epoll_create1 (0);
if (epfd < 0)
    perror ("epoll_create1");
```

Файловый дескриптор, возвращаемый от `epoll_create1()`, должен уничтожаться с помощью вызова `close()` по окончании опроса.

Управление `epoll`

Системный вызов `epoll_ctl()` можно использовать для добавления файловых дескрипторов в заданный контекст опроса и удаления этих дескрипторов из контекста:

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

int epoll_ctl (int epfd,
               int op,
               int fd,
               struct epoll_event *event);
```

Заголовок `<sys/epoll.h>` определяет структуру `epoll_event` так:

```
struct epoll_event {
    __u32 events; /* события */
    union {
```

```

        void *ptr;
        int fd;
        __u32 u32;
        __u64 u64;
    } data;
};

```

Успешный вызов `epoll_ctl()` управляет экземпляром `epoll`, ассоциированным с файловым дескриптором `epfd`. Параметр `op` указывает операцию, которую нужно применить к файлу, ассоциированному с `fd`. Параметр `event` подробнее описывает ход операции.

Вот допустимые значения, которые может принимать параметр `op`:

- `EPOLL_CTL_ADD` — добавляет монитор для файла, ассоциированного с файловым дескриптором `fd`, к экземпляру `epfd`, при этом идет наблюдение за событиями, определенными в `event`;
- `EPOLL_CTL_DEL` — удаляет монитор для файла, ассоциированного с файловым дескриптором `fd`, из экземпляра `epfd`;
- `EPOLL_CTL_MOD` — изменяет существующий монитор файлового дескриптора `fd`, задавая для него обновленные события, указанные в `event`.

В поле `events` структуры `epoll_event` перечисляется, какие события нужно отслеживать на заданном файловом дескрипторе. Множественные события могут быть перечислены с помощью побитового «ИЛИ». Ниже приведены допустимые значения.

- `EPOLLERR` — в файле возникло условие ошибки. Это событие отслеживается всегда, даже без вашего указания.
- `EPOLLET` — обеспечивает запуск по фронту сигнала для монитора файла (см. подразд. «Сравнение событий, запускаемых по фронту и по уровню сигнала» этого раздела). По умолчанию используется запуск по уровню сигнала.
- `EPOLLHUP` — произошло зависание файла. Это событие отслеживается всегда, даже без вашего указания.
- `EPOLLIN` — файл доступен для считывания без блокировки.
- `EPOLLONESHOT` — после того как событие сгенерировано и прочтено, наблюдение за файлом автоматически прекращается. Для возобновления наблюдения следует указать новую маску события с помощью `EPOLL_CTL_MOD`.
- `EPOLLOUT` — файл доступен для записи без блокировки.
- `EPOLLPRI` — имеются внеполосные данные для срочного считывания.

Поле `data` в структуре `event_poll` предназначено для частного применения пользователями. Содержимое возвращается пользователю после получения запрошенного события. Как правило, `event.data.fd` устанавливается в значение `fd`, благодаря чему становится просто найти файловый дескриптор, вызвавший событие.

В случае успеха `epoll_ctl()` возвращает 0. При ошибке вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- EBADF — `epfd` не является допустимым экземпляром опроса событий либо `fd` не является допустимым файловым дескриптором;
- EEXIST — значение `op` было равно `EPOLL_CTL_ADD`, но `fd` уже ассоциирован с `epfd`;
- EINVAL — `epfd` не является допустимым экземпляром опроса событий, `epfd` равно `fd` или `op` является недопустимым;
- ENOENT — `op` было равно `EPOLL_CTL_MOD` или `EPOLL_CTL_DEL`, но `fd` не ассоциирован с `epfd`;
- ENOMEM — недостаточно памяти для обработки запроса;
- EPERM — `fd` не поддерживает опрос событий.

Например, чтобы добавить к экземпляру опроса событий `epfd` новое наблюдение для файла, связанного с дескриптором `fd`, можно написать:

```
struct epoll_event event;
int ret;
```

```
event.data.fd = fd; /* вернуть нам fd позже (от epoll_wait) */
event.events = EPOLLIN | EPOLLOUT;
```

```
ret = epoll_ctl (epfd, EPOLL_CTL_ADD, fd, &event);
if (ret)
    perror ("epoll_ctl");
```

Чтобы изменить имеющееся событие в файле, ассоциированном с файловым дескриптором `fd` в экземпляре опроса событий `epfd`, можно использовать такой код:

```
struct epoll_event event;
int ret;
```

```
event.data.fd = fd; /* вернуть нам fd позже */
event.events = EPOLLIN;
```

```
ret = epoll_ctl (epfd, EPOLL_CTL_MOD, fd, &event);
if (ret)
    perror ("epoll_ctl");
```

Напротив, чтобы удалить имеющееся событие из файла, связанного с `fd`, из экземпляра опроса событий `epfd`, можно написать:

```
struct epoll_event event;
int ret;
```

```
ret = epoll_ctl (epfd, EPOLL_CTL_DEL, fd, &event);
if (ret)
    perror ("epoll_ctl");
```

Обратите внимание: параметр `event` может быть равен `NULL`, если `op` равно `EPOLL_CTL_DEL`, поскольку в таком случае мы не предоставляем маску событий. Однако в версиях ядра ниже 2.6.9 проводится ненужная проверка, требующая, чтобы этот

параметр был неравен нулю. Чтобы переносить ваш код на такие старые ядра, нужно передавать здесь допустимый ненулевой указатель; он не будет изменен. В ядре 2.6.9 эта ошибка устранена.

Ожидание событий с помощью `epoll`

Системный вызов `epoll_wait()` ожидает событий на файловых дескрипторах, ассоциированных с заданным экземпляром опроса событий:

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

int epoll_wait (int epfd,
                struct epoll_event *events,
                int maxevents,
                int timeout);
```

Вызов `epoll_wait()` ожидает в течение не более `timeout` миллисекунд. Он отслеживает события, которые могут происходить в файлах, ассоциированных с экземпляром опроса событий `epfd`. В случае успеха `events` будет указывать на область памяти, содержащую структуры `epoll_event`, описывающие каждое событие (например, готовность файла к считыванию или записи) — не более `maxevents` событий. Возвращаемое значение соответствует количеству найденных событий или `-1` в случае ошибки. При ошибке `errno` устанавливается в одно из следующих значений:

- `EBADF` — `epfd` не является допустимым файловым дескриптором;
- `EFAULT` — у процесса нет доступа для записи в область памяти, на которую указывает `events`;
- `EINTR` — системный вызов был прерван сигналом до того, как он успел завершиться, или истекла заданная задержка;
- `EINVAL` — `epfd` не является допустимым экземпляром опроса событий либо значение `maxevents` меньше или равно нулю.

Если значение `timeout` равно нулю, то вызов возвращается немедленно, даже если доступные события отсутствуют — в таком случае возвращаемое значение этого вызова будет равно нулю. Если значение `timeout` равно `-1`, то вызов не вернется, пока не появится доступное событие.

После возврата вызова поле `events` структуры `epoll_event` будет описывать произошедшие события. Поле `data` содержит значение, которое в него установил пользователь до вызова `epoll_ctl()`.

Полный пример с `epoll_wait()` выглядит так:

```
#define MAX_EVENTS 64

struct epoll_event *events;
int nr_events, i, epfd;

events = malloc (sizeof (struct epoll_event) * MAX_EVENTS);
if (!events) {
```

```

        perror ("malloc");
        return 1;
    }

    nr_events = epoll_wait (epfd, events, MAX_EVENTS, -1);
    if (nr_events < 0) {
        perror ("epoll_wait");
        free (events);
        return 1;
    }

    for (i = 0; i < nr_events; i++) {
        printf ("event=%ld on fd=%d\n",
                events[i].events,
                events[i].data.fd);

        /*
         * Теперь мы можем с помощью events[i].events оперировать
         * events[i].data.fd в неблокирующем режиме
         */
    }

    free (events);

```

Функции `malloc()` и `free()` будут рассмотрены в гл. 9.

Сравнение событий, запускаемых по фронту и по уровню сигнала

Если в поле `events` параметра `event`, передаваемого вызову `epoll_ctl()`, установлено значение `EPOLLET`, то наблюдение `fd` ведется *по фронту сигнала*, а не *по уровню сигнала*.

Рассмотрим следующие события, которые могут происходить между производителем и потребителем информации, взаимодействие которых осуществляется по конвейеру UNIX:

- производитель записывает 1 Кбайт данных в конвейер;
- потребитель выполняет в конвейере вызов `epoll_wait()`, дожидаясь, пока в конвейере окажутся данные, которые, таким образом, можно будет прочитать.

При наблюдении по уровню сигнала вызов `epoll_wait()`, сделанный на этапе 2, вернется немедленно, показывая, что конвейер готов к считыванию. При запуске по фронту этот вызов вернется не ранее, чем завершится этап 1. Таким образом, даже если конвейер будет готов для считывания в момент вызова `epoll_wait()`, этот вызов не вернется, пока в конвейер не будут записаны данные.

По умолчанию применяется запуск по уровню сигнала. Именно так работают `poll()` и `select()`, именно этого ожидает большинство разработчиков. При отслеживании событий по фронту требуется иной подход к программированию, обычно предполагающий использование неблокирующего ввода-вывода и тщательную проверку наличия `EAGAIN`.

Запуск по фронту сигнала

Термин «запуск по фронту сигнала» заимствован из электротехники. Прерывание для запуска по уровню сигнала выполняется всякий раз, когда происходит суждение о строке. Прерывание для запуска по фронту выполняется только на заднем или на переднем фронте интересующего нас изменения. Прерывания, запускаемые по фронту, полезны, когда нас интересует событие как таковое (строка, подвергаемая суждению).

Предположим, у нас есть файловый дескриптор, с которого мы считываем информацию. При опросе событий, запускаемом по уровню сигнала, мы получим уведомление при условии, что файловый дескриптор готов для считывания. Здесь речь идет об уровне строки, вызывающей уведомление. При запуске по фронту вы также получите уведомление, но только после того, как появятся доступные для чтения данные. Здесь речь идет о фронте, то есть изменении, в результате которого дается уведомление.

Отображение файлов в память

В качестве альтернативы для стандартного файлового ввода-вывода ядро предоставляет интерфейс, позволяющий отображать файл в память. Это означает, что существует однозначное соответствие между адресом в памяти и словом в файле. Благодаря этому программист может обращаться к файлу напрямую через память точно так же, как и к любому другому фрагменту данных, находящихся в памяти. Можно даже разрешить запись в область памяти так, чтобы записываемые данные прозрачно отображались на файл, расположенный на диске.

Стандарт POSIX.1 определяет — а Linux реализует — системный вызов `mmap()`, предназначенный для отображения объектов в память. В этом разделе вызов `mmap()` будет рассмотрен в контексте отображения файлов в память при выполнении ввода-вывода. В гл. 9 мы обсудим другие возможности применения `mmap()`.

`mmap()`

При вызове `mmap()` мы указываем ядру отобразить в память `len` байт объекта, представленного файловым дескриптором `fd` и начинающийся в файле со смещения в `offset` байт. Если включить `addr`, то данное значение указывает адрес в памяти, который имеет приоритет при выборе начальной точки отображения. Права доступа регламентируются в `prot`, а дополнительное поведение можно определить с помощью `flags`:

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

```
void * mmap (void *addr,  
             size_t len,  
             int prot,  
             int flags,
```

```
int fd,  
off_t offset);
```

Параметр `addr` подсказывает ядру, с какого места в памяти лучше всего начать отображение файла. Это всего лишь рекомендация; большинство пользователей передают здесь значение 0. Вызов возвращает конкретный адрес из памяти — тот, с которого будет начинаться отображение.

Параметр `prot` описывает желательную степень защиты памяти при отображении. Он может иметь значение `PROT_NONE`, при котором запрещается доступ к страницам, участвующим в процессе отображения (почти не применяется!), либо иметь один или несколько следующих флагов, перечисляемых методом побитового «ИЛИ»:

- `PROT_READ` — страницы доступны для чтения;
- `PROT_WRITE` — страницы доступны для записи;
- `PROT_EXEC` — страницы доступны для выполнения.

Желаемая степень защиты памяти не должна противоречить режиму, в котором был открыт файл. Например, если программа открывает файл только для чтения, то в `prot` нельзя указывать `PROT_WRITE`.

Флаги защиты, варианты архитектуры и безопасность

В то время как POSIX определяет три бита защиты (на чтение, запись и выполнение), в некоторых архитектурах два таких бита могут быть представлены одним. Например, процессоры часто не отличают чтение от выполнения. В этом случае в процессоре может действовать лишь один флаг считывания. В подобных системах `PROT_READ` подразумевает `PROT_EXEC`. До недавнего времени именно такая ситуация наблюдалась в архитектуре x86.

Разумеется, мы не можем полагаться на такое поведение, если хотим гарантировать переносимость поведения. В переносимых программах при намерении выполнять код в процессе отображения необходимо устанавливать флаг `PROT_EXEC`.

Обратная ситуация отчасти объясняет, почему множество атак на систему связано с переполнением буфера: даже если при заданном варианте отображения выполнение кода в процессе отображения запрещается, сам процессор вполне может допускать такое исполнение.

В новейших процессорах x86 появился бит NX (без исполнения), допускающий отображение, при котором считывание данных разрешается, а выполнение — нет. В этих новых системах `PROT_READ` больше не подразумевает `PROT_EXEC`.

Аргумент `flags` описывает тип отображения и некоторые элементы соответствующего поведения. Он представляет собой побитовое «ИЛИ» следующих значений.

- `MAP_FIXED` — приказывает `mmap()` считать параметр `addr` обязательным, а не рекомендательным. Если ядро не может начать отображение с указанного адреса, то вызов завершается ошибкой. Если параметры адреса и длины отображаемой области накладываются на имеющееся отображение, то отображенные

ранее страницы сбрасываются и заменяются новыми. Такой вариант требует точнейшего ориентирования в адресном пространстве процесса, поэтому данный флаг не является переносимым и использовать его настоятельно не рекомендуется.

- `MAP_PRIVATE` — указывает, что отображение не может использоваться совместно. Файл отображается только в режиме копирования при записи, и любые изменения, сделанные в памяти данным процессом, не учитываются ни в самом файле, ни в отображениях, используемых другими процессами¹.
- `MAP_SHARED` — процесс использует отображение совместно со всеми другими процессами, отображающими тот же файл. Запись в отображение эквивалентна записи в файл. Считывание из отображения будет учитывать записи, сделанные другими процессами.

Необходимо задать `MAP_SHARED` или `MAP_PRIVATE`, но не два этих флага одновременно. Другие, более сложные флаги будут рассмотрены в гл. 9.

Когда вы отображаете файловый дескриптор, увеличивается количество ссылок, указывающих на файл, поэтому вы можете закрыть файловый дескриптор после отображения файла, и ваш процесс по-прежнему будет иметь доступ к нему. Соответствующее уменьшение количества ссылок произойдет, когда отображение файла на память прекратится либо когда завершится процесс.

Рассмотрим пример. Следующий фрагмент кода отображает файл с дескриптором `fd`, начиная с первого байта и до `len` байт. При этом отображаемая информация доступна только для чтения:

```
void* p;  
  
p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);  
if (p == MAP_FAILED)  
    perror ("mmap");
```

На рис. 4.1 показано, как параметры, предоставленные в вызове `mmap()`, действуют при отображении между файлом и адресным пространством процесса.

Размер страницы

Страница — это единица детализации (разбивки) в блоке управления памятью (Memory Management Unit, MMU). Следовательно, это мельчайшая единица памяти, для которой могут быть заданы собственное поведение и права доступа. Страница — это основной кирпичик отображений информации в памяти, которые, в свою очередь, являются составными блоками адресного пространства процесса.

Системный вызов `mmap()` оперирует страницами. Значения параметров `addr` и `offset` должны быть выровнены по размерам страницы. Таким образом, они должны быть целочисленными и кратными размеру страницы.

¹ Концепция копирования при записи связана с созданием процессов и подробно описана в разд. «Запуск нового процесса» гл. 5.

Адресное пространство процесса

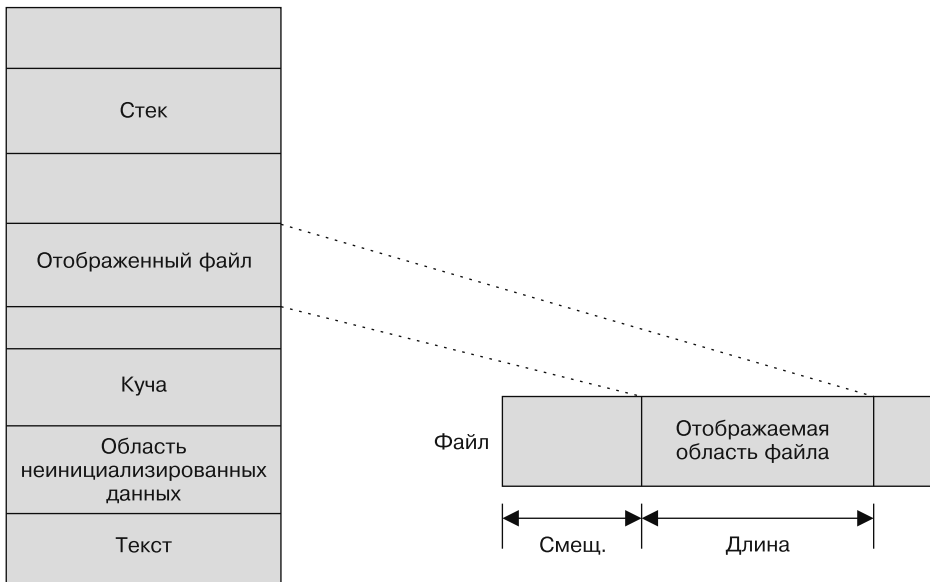


Рис. 4.1. Отображение файла в адресное пространство процесса

Итак, отображения являются целочисленными кратными страниц. Если параметр `len`, предоставленный вызывающей стороной, не выровнен по размерам страницы — например, потому, что размер описываемого им файла не кратен размеру страницы, — то отображения округляются до ближайшей полной страницы. Байты этой добавленной памяти, расположенные между последним допустимым файлом и концом отображаемой области, заполняются нулями. Каждое считывание из такой добавленной области возвратит только нули. Любые попытки записать информацию в эту область никак не отразятся на базовом файле, даже если он отображается с флагом `MAP_SHARED`. В файл всегда записывается столько байтов, сколько сразу было указано в параметре `len`.

Стандартный метод получения размера страницы, определяемый в POSIX, называется `sysconf()`. Он может сообщить вам разнообразную информацию о состоянии системы:

```
#include <unistd.h>
```

```
long sysconf(int name);
```

Метод `sysconf()` возвращает значение конфигурационного элемента `name` либо значение `-1`, если элемент `name` невалиден. В случае ошибки вызов устанавливает `errno` в значение `EINVAL`. В некоторых ситуациях значение `-1` может оказаться допустимым (например, при определении лимита, где `-1` означает «лимит отсутствует»), поэтому перед вызовом целесообразно очищать `errno`, а после вызова — проверять это значение.

POSIX определяет конфигурационный элемент `_SC_PAGESIZE` (и синонимичный ему `_SC_PAGE_SIZE`), значение которого соответствует размеру страницы в байтах, поэтому получить размер страницы во время выполнения не составляет труда:

```
long page_size = sysconf (_SC_PAGESIZE);
```

Linux также предоставляет функцию `getpagesize()`:

```
#include <unistd.h>
```

```
int getpagesize (void);
```

Метод `getpagesize()` аналогично возвращает размер страницы в байтах. Работать с этим методом еще проще, чем с `sysconf()`:

```
int page_size = getpagesize ();
```

Не все системы UNIX поддерживают данную функцию. Она была исключена из версии стандарта POSIX 1003.1-2001. Я упоминаю ее для полноты картины.

Размер страницы также статически сохраняется в макросе `PAGE_SIZE`, определяемом в `<sys/user.h>`. Таким образом, есть и третий способ получения размера страницы:

```
int page_size= PAGE_SIZE;
```

В отличие от первых двух вариантов, здесь мы получаем размер страницы во время компиляции, а не выполнения. Определенные архитектуры поддерживают множественные машинные типы с разными размерами страницы, а некоторые машинные типы даже сами поддерживают несколько вариантов размера страницы! Отдельно взятый двоичный файл должен нормально работать на любом машинном типе конкретной архитектуры. Это означает, что вы должны обеспечить его сборку и запуск на любой машине. Если жестко запрограммировать размер страницы, такая возможность будет исключена. Следовательно, размер страницы должен определяться во время выполнения. Поскольку `addr` и `offset` обычно равны нулю, соблюсти данное требование не так сложно.

Более того, новые версии ядра, вероятно, не будут экспортировать этот макрос в пользовательское пространство. В данной главе я упоминаю его, так как он часто встречается в коде UNIX, но не следует использовать его в собственных программах. Лучше работать с `sysconf()` — этот метод более всего отвечает целям переносимости и в будущем обеспечит максимально широкую совместимость.

Возвращаемые значения и коды ошибок

В случае успеха вызов `mmap()` возвращает расположение отображаемой области. При ошибке он возвращает `MAP_FAILED` и устанавливает `errno` соответствующее значение. Вызов `mmap()` никогда не возвращает 0.

Переменная `errno` может принимать одно из следующих значений:

- `EACCES` — файл, соответствующий данному дескриптору, не является обычным либо режим, в котором был открыт файл, конфликтует с `prot` или `flags`;
- `EAGAIN` — доступ к файлу был закрыт посредством файловой блокировки;
- `EBADF` — указанный файловый дескриптор не является допустимым;

- EINVAL — как минимум один из параметров `addr`, `len` или `off` является недопустимым;
- ENFILE — система достигла лимита на количество открытых файлов;
- ENODEV — файловая система, в которой находится предназначенный для отображения файл, не поддерживает отображения в память;
- ENOMEM — процесс не обладает достаточным объемом памяти;
- EOVERFLOW — сумма `addr + len` превышает размер адресного пространства;
- EPERM — задан флаг `PROT_EXEC`, но файловая система смонтирована как `noexec`.

Ассоциированные сигналы

С отображаемыми областями ассоциируются два сигнала:

- SIGBUS — выдается при попытке процесса обратиться к области отображения, которая уже недействительна — например, потому, что после отображения файл был обрезан;
- SIGSEGV — генерируется, когда процесс пытается что-либо записать в область, отображаемую только для чтения.

Системный вызов `munmap()`

В Linux предоставляется вызов `munmap()`, позволяющий удалять отображение, созданное при вызове `mmap()`:

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

```
int munmap (void *addr, size_t len);
```

Вызов удаляет все отображения, в которых содержатся страницы, расположенные в любой части адресного пространства процесса, начинающейся с `addr` (здесь должна начинаться новая страница) и на протяжении `len` байт. Как только отображение удалено, область памяти, которая была с ним ассоциирована, больше не является допустимой и при последующих попытках обращения к ней генерируется сигнал `SIGSEGV`.

Как правило, мы передаем `munmap()` возвращаемое значение и параметр `len`, использовавшиеся при последнем вызове `mmap()`.

При успехе `munmap()` возвращает 0. При ошибке он возвращает -1 и присваивает `errno` соответствующее значение. Единственное стандартное значение `errno` в данном случае — `EINVAL`, оно указывает, что один или несколько параметров имеют недопустимые значения.

В качестве примера следующий фрагмент кода прекращает отображение в памяти всех страниц, содержащихся в интервале `[addr, addr+len]`:

```
if (munmap (addr, len) == -1)
    perror ("munmap");
```

Пример отображения

Рассмотрим простую программу, которая использует `mmap()` для передачи выбранного пользователем файла в стандартный вывод:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
    struct stat sb;
    off_t len;
    char *p;
    int fd;

    if (argc < 2) {
        fprintf (stderr, "usage: %s <file>\n", argv[0]);
        return 1;
    }

    fd = open (argv[1], O_RDONLY);
    if (fd == -1) {
        perror ("open");
        return 1;
    }

    if (fstat (fd, &sb) == -1) {
        perror ("fstat");
        return 1;
    }

    if (!S_ISREG (sb.st_mode)) {
        fprintf (stderr, "%s is not a file\n", argv[1]);
        return 1;
    }

    p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror ("mmap");
        return 1;
    }

    if (close (fd) == -1) {
        perror ("close");
        return 1;
    }
}
```

```

    for (len = 0; len < sb.st_size; len++)
        putchar (p[len]);

    if (munmap (p, sb.st_size) == -1) {
        perror("munmap");
        return 1;
    }

    return 0;
}

```

Единственный системный вызов в этом примере, который мог показаться вам незнакомым, — это `fstat()`. О нем мы поговорим в гл. 8. На данном этапе достаточно знать, что `fstat()` возвращает информацию о заданном файле. Макрокоманда `S_ISREG()` способна проверять часть данной информации, поэтому перед отображением мы можем убедиться, что заданный файл является обычным (а не файлом устройства и не каталогом). Если файл не обычный, то при отображении его поведение зависит от базового устройства. Некоторые файлы устройств могут работать с `mmap`, другие такой способностью не обладают и устанавливают `errno` в значение `EACCES`.

Оставшаяся часть примера должна быть понятна. Мы передаем программе имя файла в качестве аргумента. Она открывает файл, убеждается, что этот файл является обычным, закрывает его, байт за байтом печатает файл в стандартном выводе, а потом прекращает отображение файла в память.

Преимущества `mmap()`

Если манипулировать файлами с помощью вызова `mmap()`, мы получаем некоторые преимущества по сравнению с использованием стандартных системных вызовов `read()` и `write()`, в частности следующие.

- При чтении файла, отображенного в память, или при его записи мы обходимся без лишнего копирования, которое выполняется при применении системных вызовов `read()` или `write()`, требующих переноса информации в пользовательский буфер и из него.
- Кроме возможных отказов страниц при записи в файл, отображаемый в память, и при его считывании нет никаких издержек, связанных с дополнительными системными вызовами и переключением контекста. Эти операции не сложнее, чем обычные обращения к памяти.
- Когда несколько процессов отображают в память один и тот же объект, информация из этого объекта совместно используется всеми процессами. Отображения, предназначенные только для чтения, а также разделяемые отображения для записи совместно используют страницы, которые еще не были скопированы при записи.
- Позиционирование в ходе отображения сводится к обычным манипуляциям с указателем. Системный вызов `lseek()` в данном случае не нужен.

По вышеперечисленным причинам `mmap()` действительно хорош во многих приложениях.

Недостатки `mmap()`

Есть несколько негативных моментов, которые необходимо учитывать при использовании `mmap()`.

- По своей величине отображения в памяти всегда являются целочисленными кратными размеру страницы, поэтому разница между размером базового файла и целым количеством страниц приходится на напрасно расходуемое пустое место. Если вы работаете с небольшими файлами, то на такое потерянное пространство может приходиться существенный процент отображаемой области. Например, при работе с 4-килобайтными страницами при отображении 7 байт мы теряем 4089 байт при каждом отображении.
- Отображения в памяти должны вписываться в адресное пространство процесса. Если процесс работает в 32-битном пространстве, то при значительном количестве отображений разного размера адресное пространство может сильно дробиться и в нем сложно будет найти сравнительно крупную непрерывающуюся свободную область. Эта проблема сохраняется и в 64-битном адресном пространстве, но там она проявляется значительно слабее.
- Возникают издержки, связанные с созданием и поддержкой отображений, находящихся в памяти, и ассоциированных с ними структур данных, расположенных в ядре. Эта проблема решается, как только удастся избавиться от двойного копирования, упомянутого в предыдущем разделе. Особенно это касается больших и часто используемых файлов.

По этим причинам преимущества `mmap()` становятся наиболее ощутимыми, когда отображаемый файл сравнительно велик (и поэтому потерянное пространство становится незначительным) либо когда общий размер отображаемого файла без остатка делится на размер страницы (таким образом, пространство впустую вообще не тратится).

Изменение размеров отображения

Linux предоставляет системный вызов `mremap()`, позволяющий увеличивать или уменьшать размер заданного отображения. Эта функция специфична для Linux:

```
#define _GNU_SOURCE
```

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

```
void * mremap (void *addr, size_t old_size,  
               size_t new_size, unsigned long flags);
```

Вызов `mremap()` расширяет или сужает площадь отображения в области памяти `[addr, addr+old_size]`, устанавливая новое значение `new_size`. Потенциально ядро

может одновременно переместить отображение — в зависимости от того, имеется ли в адресном пространстве процесса необходимое для этого место, а также в зависимости от значения `flags`.

ПРИМЕЧАНИЕ

Открывающая квадратная скобка в выражении `[addr,addr+old_size)` означает, что эта область начинается с нижнего адреса (и включает его). Закрывающая круглая скобка обозначает, что область заканчивается прямо перед верхним адресом (но не включает его). Такое соглашение называется обозначением интервала.

Параметр `flags` может иметь значение 0 или `MREMAP_MAYMOVE`. Второе значение указывает, что при необходимости ядро может переместить область отображения, если это нужно для изменения размеров. При значительном изменении размеров для успешного выполнения такой операции ядро должно иметь возможность перемещать область отображения.

Возвращаемые значения и коды ошибок. В случае успеха `mremap()` возвращает указатель на область отображения памяти, размер которой только что был изменен. При ошибке эта функция возвращает `MAP_FAILED` и устанавливает `errno` одно из следующих значений:

- `EAGAIN` — данная область памяти ограничена, и изменить ее размер невозможно;
- `EFAULT` — некоторые страницы в заданном диапазоне не являются допустимыми страницами из адресного пространства процесса либо возникла проблема при переотображении страниц;
- `EINVAL` — аргумент оказался недопустимым;
- `ENOMEM` — заданный диапазон невозможно расширить без перемещения (причем значение `MREMAP_MAYMOVE` не задано) либо в адресном пространстве процесса недостаточно свободного места.

Библиотеки, в частности `glibc`, часто используют `mremap()` для реализации эффективной функции `realloc()`, которая представляет собой интерфейс для изменения размера блока памяти, изначально полученного с помощью `malloc()`. Например:

```
void * realloc (void *addr, size_t len)
{
    size_t old_size = look_up_mapping_size (addr);
    void *p;

    p = mremap (addr, old_size, len, MREMAP_MAYMOVE);
    if (p == MAP_FAILED)
        return NULL;
    return p;
}
```

Этот код сработает, только если все операции `malloc()` являются уникальными анонимными отображениями. Тем не менее данный пример демонстрирует, как можно при необходимости значительно повысить производительность. В примере предполагается, что в библиотеке `libc` предоставляется функция `look_up_mapping_size()`.

Библиотека GNU C использует вызов `mmap()` и его семейство для выполнения некоторых операций выделения памяти. Подробнее мы рассмотрим эту тему в гл. 9.

Изменение защиты отображения

POSIX определяет интерфейс `mprotect()`, позволяющий программам изменять права доступа к имеющимся областям памяти:

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

```
int mprotect (const void *addr,  
              size_t len,  
              int prot);
```

Вызов `mprotect()` изменяет режим защиты для страниц памяти, содержащихся в области `[addr, addr+len]`, где параметр `addr` выровнен по размеру страниц. Параметр `prot` принимает те же значения, что и при передаче этого параметра `mmap()`: `PROT_NONE`, `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`. Эти значения не складываются; если область памяти доступна для чтения, а `prot` установлен только как `PROT_WRITE`, то данный вызов сделает область доступной лишь для записи.

В некоторых системах `mprotect()` может оперировать только такими отображениями в памяти, которые были созданы с помощью `mmap()`. В Linux `mprotect()` может работать с любой областью памяти.

Возвращаемые значения и коды ошибок. При успехе `mprotect()` возвращает 0. При ошибке эта функция возвращает -1 и устанавливает `errno` в одно из следующих значений:

- `EACCES` — память не может получить права доступа, запрошенные `prot`; это может произойти, например, если вы пытаетесь установить отображение с возможностью записи для файла, открытого только для чтения;
- `EINVAL` — параметр `addr` является недопустимым либо не выровнен по границам страниц;
- `ENOMEM` — недостаточно памяти ядра для удовлетворения запроса либо одна страница или более в указанной области памяти не являются допустимой частью адресного пространства процесса.

Синхронизация файла с помощью отображения

POSIX предоставляет отображаемый в память эквивалент системного вызова `fsync()`, рассмотренного в гл. 2:

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

```
int msync (void *addr, size_t len, int flags);
```

Вызов `msync()` сбрасывает обратно на диск любые изменения, сделанные в файле, отображенном с помощью `mmap()`, синхронизируя отображенный файл с самим

отображением. В данном случае файл или его подмножество, ассоциированное с отображением (от адреса в памяти `addr` до `len` байт), синхронизируется с диском. Аргумент `addr` должен быть выровнен по размерам страниц. Обычно он представляет собой значение, возвращаемое от предыдущего вызова `mmap()`.

Без вызова `msync()` не гарантируется, что система успеет записать грязную информацию обратно на диск прежде, чем отображение файла прекратится. Это отличается от поведения `write()`, когда буфер загрязняется в процессе записи и ставится в очередь для переноса на диск. При записи в отображение в памяти процесс изменяет страницы непосредственно в страничном кэше ядра, причем само ядро в этом не участвует. После этого ядро может сразу не синхронизировать страничный кэш и информацию на диске.

Параметр `flags` управляет ходом процесса синхронизации. Он может представлять собой побитовое «ИЛИ» либо следующие значения.

- `MS_SYNC` — указывает, что синхронизация должна быть синхронной. Вызов `msync()` не вернется, пока все страницы не будут записаны на диск.
- `MS_ASYNC` — определяет, что синхронизация должна быть асинхронной. Обновление назначается планировщиком, но вызов `msync()` возвращается сразу, не дожидаясь завершения всех операций записи.
- `MS_INVALIDATE` — указывает, что все другие копии отображения должны быть признаны недействительными. Любые последующие обращения к каким-либо отображениям этого файла будут отражать наиболее актуальную синхронизированную информацию с диска.

Можно указать `MS_ASYNC` или `MS_SYNC`, но не два этих флага одновременно.

Работать с этим вызовом просто:

```
if (msync (addr, len, MS_ASYNC) == -1)
    perror ("msync");
```

В данном примере с диском асинхронно синхронизируется (быстро произнесите это словосочетание 10 раз подряд) файл, который отображен на область памяти `[addr, addr+len]`.

Возвращаемые значения и коды ошибок. В случае успеха `msync()` возвращает 0. При ошибке вызов возвращает -1 и устанавливает `errno` в соответствующее значение. Далее приведены допустимые значения `errno`.

- `EINVAL` — у параметра `flags` одновременно установлены значения `MS_SYNC` и `MS_ASYNC` либо установлен иной бит, нежели три допустимых флага, либо значение `addr` не выровнено по размерам страниц.
- `ENOMEM` — указанная область памяти (или ее часть) не отображается. Обратите внимание: Linux вернет `ENOMEM` потому, что этого требует POSIX в подобных случаях (когда необходимо синхронизировать область памяти, отображаемую лишь частично). Тем не менее все действительные отображения в этой области памяти будут синхронизированы.

До выхода версии ядра Linux 2.4.19 вызов `msync()` возвращал `EFAULT`, а не `ENOMEM`.

Извещения об отображении

В Linux предоставляется системный вызов `madvise()`, с помощью которого процессы могут извещать ядро или подсказывать ему, как предполагается использовать данное отображение. На основании этой информации ядро может оптимизировать поведение, чтобы пользоваться преимуществами конкретного варианта отображения. Конечно, ядро Linux динамически корректирует свое поведение и, как правило, обеспечивает оптимальную производительность без явных извещений такого рода. Тем не менее, давая подсказки, мы можем добиться желаемого кэширования и опережающего считывания при некоторых типах нагрузки.

Вызов `madvise()` сообщает ядру, как следует обращаться со страницами, находящимися в памяти по адресу `addr` и далее до `len` байт:

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

```
int madvise (void *addr,  
            size_t len,  
            int advice);
```

Если `len` равно 0, то ядро применит рекомендацию ко всей области отображения, начиная с адреса `addr`. Параметр `advice` ограничивает область действия рекомендации и может иметь одно из следующих значений:

- `MADV_NORMAL` — приложение не получает конкретной рекомендации, как работать с данной областью памяти; обработка должна проходить в обычном режиме;
- `MADV_RANDOM` — приложение планирует обращаться к страницам из указанного диапазона в случайном (произвольном) порядке;
- `MADV_SEQUENTIAL` — приложение будет обращаться к страницам из указанного диапазона последовательно, от самого нижнего до самого верхнего адреса;
- `MADV_WILLNEED` — в ближайшем будущем приложение обратится к страницам из указанного диапазона;
- `MADV_DONTNEED` — в ближайшем будущем приложение не планирует обращаться к страницам из указанного диапазона.

Конкретные модификации поведения, выбираемые ядром в ответ на такую рекомендацию, зависят от реализации системы: POSIX определяет только значение совета, а не его потенциальные последствия. Ядро Linux версии 2.6 и выше следующим образом реагирует на перечисленные значения `advice`:

- `MADV_NORMAL` — ядро работает в обычном режиме, выполняя опережающее считывание в умеренном объеме;
- `MADV_RANDOM` — ядро отключает опережающее считывание, при каждой физической операции считывания захватывается лишь минимальный объем данных;
- `MADV_SEQUENTIAL` — ядро выполняет агрессивное опережающее считывание;
- `MADV_WILLNEED` — ядро инициирует опережающее считывание, копируя указанные страницы в память;

- `MADV_DONTNEED` — ядро высвобождает все ресурсы, связанные с указанными страницами в памяти, а также сбрасывает все грязные и еще не синхронизированные страницы; при последующих обращениях к отображенным данным эта информация будет подкачиваться из файла с резервной копией или (при анонимных отображениях) заполнять нулями запрошенные страницы;
- `MADV_DONTFORK` — указанные здесь страницы не копируются в дочерний процесс на протяжении всего ветвления; этот флаг доступен только в версии ядра Linux 2.6.16 и выше, он нужен при управлении DMA-страницами, в других случаях используется редко;
- `MADV_DOFORK+` — отменяет поведение `MADV_DONTFORK`.

Типичный пример использования:

```
int ret;
```

```
ret = madvise (addr, len, MADV_SEQUENTIAL);  
if (ret < 0)  
    perror("madvise");
```

Этот вызов сообщает ядру, что процесс собирается последовательно обратиться к области памяти `[addr, addr+len]`.

Опережающее считывание

Когда ядро Linux считывает файлы с диска, оно выполняет оптимизацию, называемую опережающим считыванием. Когда делается запрос к определенному фрагменту файла, ядро также считывает и следующий за ним фрагмент. Если затем делается запрос к этому заблаговременно прочитанному фрагменту файла (именно это и происходит при последовательном считывании файла), то ядро вернет запрошенные данные без какой-либо задержки. Поскольку у дисков есть буферы дорожек (так как диски выполняют и собственное опережающее считывание), а еще потому, что файлы располагаются на диске в той или иной последовательности, данная оптимизация получается малозатратной.

Небольшое опережающее считывание обычно целесообразно, но для достижения оптимального результата необходимо определить, в каком именно объеме требуется такое считывание. Если файл обрабатывается последовательно, то окно опережающего считывания лучше сделать шире. Если же доступ к файлу происходит в произвольном порядке, то опережающее считывание будет практически бесполезным.

В разд. «Внутренняя организация ядра» гл. 2 мы говорили, что ядро динамически корректирует размер окна опережающего считывания, реагируя на частоту обращений к этому окну. Если обращений много, это значит, что окно целесообразно увеличить; если мало — окно стоит уменьшить. Системный вызов `madvise()` позволяет приложениям оперативно влиять на размер окна опережающего считывания.

Возвращаемые значения и коды ошибок. В случае успеха `madvise()` возвращает 0. При ошибке он возвращает -1 и присваивает `errno` соответствующее значение. Допустимы следующие значения ошибок:

- EAGAIN — внутренний ресурс ядра (вероятно, память) оказался недоступен; процесс может повторить попытку;
- EBADF — область существует, но не отображает файл;
- EINVAL — параметр `len` имеет отрицательное значение, либо `addr` не выровнен по размеру страниц, либо параметр `advise` недопустим, либо страницы заблокированы или совместно используются с `MADV_DONTNEED`;
- EIO — при применении `MADV_WILLNEED` возникла внутренняя ошибка ввода-вывода;
- ENOMEM — заданная область не является допустимым отображением в адресном пространстве процесса либо указан `MADV_WILLNEED`, но отсутствует достаточный объем памяти, необходимый для подкачки заданных областей.

Извещения об обычном файловом вводе-выводе

В предыдущем подразделе мы рассмотрели, как извещать систему об отображениях в памяти. Здесь мы рассмотрим, как извещать ядро об обычном файловом вводе-выводе. Linux предоставляет для таких извещений два интерфейса: `posix_fadvise()` и `readahead()`.

Системный вызов `posix_fadvise()`

Как понятно из названия, данный интерфейс извещений стандартизирован в рамках POSIX 1003.1-2003:

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

```
int posix_fadvise (int fd,  
                  off_t offset,  
                  off_t len,  
                  int advice);
```

При вызове `posix_fadvise()` ядро получает подсказку `advise` в файловом дескрипторе `fd` в интервале `[offset, offset+len]`. Если `len` равно 0, подсказка будет применяться к диапазону `[offset, length of file]`. Как правило, в таких случаях `len` и `offset` действительно получают значения 0, и эта подсказка применяется ко всему файлу.

Варианты `advise` напоминают допустимые значения `madvice()`. Для `advise` следует задать в точности одно из следующих значений:

- `POSIX_FADV_NORMAL` — приложение не дает никаких рекомендаций относительно обращения с данным разделом файла; раздел должен обрабатываться как обычно;
- `POSIX_FADV_RANDOM` — предполагается, что приложение будет обращаться к данным из указанного диапазона в произвольном (случайном) порядке;
- `POSIX_FADV_SEQUENTIAL` — приложение собирается обращаться к данным из указанного диапазона последовательно, от низшего адреса до высшего;

- `POSIX_FADV_WILLNEED` — предполагается, что приложение будет обращаться к данным из указанного диапазона в ближайшем будущем;
- `POSIX_FADV_NOREUSE` — приложение обратится к данным из указанного диапазона в ближайшем будущем, но только один раз;
- `POSIX_FADV_DONTNEED` — приложению не потребуется обращаться к данным из указанного диапазона в ближайшем будущем.

Как и в случае с `madvise()`, точная реакция на конкретное извещение зависит от реализации системы. Различия могут наблюдаться даже в разных версиях ядра Linux. В настоящее время ядро реагирует на конкретные извещения следующим образом.

- `POSIX_FADV_NORMAL` — ядро работает как обычно, выполняя опережающее считывание в умеренном объеме.
- `POSIX_FADV_RANDOM` — ядро отключает опережающее считывание и при каждой физической операции считывания захватывает лишь минимальное количество данных.
- `POSIX_FADV_SEQUENTIAL` — ядро очень активно задействует опережающее считывание, вдвое увеличивая размер окна для этой операции.
- `POSIX_FADV_NOREUSE` — в настоящее время ядро реагирует на такое извещение так же, как и на `POSIX_FADV_WILLNEED`. Возможно, в более новых версиях ядра в подобных случаях будет задействоваться дополнительная оптимизация, опирающаяся на поведение однократного обращения. Эта подсказка не имеет аналога у вызова `madvise()`.
- `POSIX_FADV_DONTNEED` — ядро извлекает из страничного кэша все данные, кэшированные для этого диапазона. Обратите внимание: реакция ядра на эту подсказку функционально отличается от реакции на одноименную подсказку из вызова `madvise()`.

Ниже приведен фрагмент кода, сообщающий ядру, что доступ ко всему файлу, представленному дескриптором `fd`, будет происходить в случайном, произвольном порядке.

```
int ret;

ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
if (ret == -1)
    perror ("posix_fadvise");
```

Возвращаемые значения и коды ошибок. В случае успеха `posix_fadvise()` возвращает 0. При ошибке он возвращает -1 и присваивает `errno` одно из следующих значений:

- `EBADF` — указанный дескриптор файла является недопустимым;
- `EINVAL` — указанное извещение недопустимо, либо заданный дескриптор файла ссылается на конвейер, либо указанное извещение не может быть применено к данному файлу.

Системный вызов `readahead()`

Системный вызов `posix_fadvise()` появился только в версии ядра Linux 2.6. Ранее применялся вызов `readahead()`, обеспечивающий такое же поведение, как и при использовании подсказки `POSIX_FADV_WILLNEED`. В отличие от `posix_fadvise()`, `readahead()` — это специфичный для Linux интерфейс.

```
#define _GNU_SOURCE
#include <fcntl.h>
```

```
ssize_t readahead (int fd,
                  off64_t offset,
                  size_t count);
```

При вызове `readahead()` мы заносим в страничный кэш область `[offset, offset + count]` из файла, обозначенного дескриптором `fd`.

Возвращаемые значения и коды ошибок. В случае успеха `readahead()` возвращает 0. При ошибке он возвращает -1 и присваивает `errno` одно из следующих значений:

- `EBADF` — указанный файловый дескриптор является недопустимым или не открыт для чтения;
- `EINVAL` — указанный файловый дескриптор отображается на файл, не поддерживающий опережающее считывание.

Рекомендации почти ничего не стоят

В некоторых распространенных прикладных ситуациях мы можем добиться значительных успехов, просто дав ядру несколько хорошо продуманных рекомендаций. Такие подсказки могут значительно облегчить даже очень тяжеловесный ввод-вывод. В наши дни процессоры стали очень быстрыми, а жесткие диски все еще остаются медленными. В таких условиях ядру пригодится любая ваша подсказка, а хорошая рекомендация определенно будет нелишней.

Перед тем как считать фрагмент файла, процесс может выполнить извещение `POSIX_FADV_WILLNEED`, после которого ядро считает файл в страничный кэш. Ввод-вывод будет происходить асинхронно, в фоновом режиме. Когда приложение наконец дойдет до интересующего нас файла, операцию удастся завершить, не провоцируя блокирующего ввода-вывода.

Верно и обратное. После считывания или записи большого объема данных, например при потоковой передаче видео на диск, процесс может дать подсказку `POSIX_FADV_DONTNEED`, чтобы ядро извлекло указанный фрагмент файла из страничного кэша. Когда выполняется большая потоковая операция, страничный кэш может быть постоянно заполнен. Если приложение больше никогда не будет обращаться к конкретному фрагменту данных, страничный кэш может оказаться забитым ненужной информацией, на месте которой могла бы быть полезная. Соответственно, в приложении, работающем с потоковой передачей видео, целесообразно регулярно очищать кэш от находящихся там данных.

Процесс, который должен считать весь файл, может дать подсказку `POSIX_FADV_SEQUENTIAL`, после которой ядро включает агрессивное опережающее считывание. Если же сразу известно, что процесс будет обращаться к файлу в произвольном порядке, считывая разрозненные фрагменты, то можно воспользоваться подсказкой `POSIX_FADV_RANDOM`. В таком случае ядро не будет выполнять опережающее считывание, совершенно бесполезное в данной ситуации.

Синхронизированные, синхронные и асинхронные операции

В системах UNIX термины «синхронизированный», «несинхронизированный», «синхронный» и «асинхронный» употребляются достаточно вольно, и из-за этого возникает определенная путаница (в английском языке слова «синхронный» и «синхронизированный» вообще практически одинаковы).

Синхронная (synchronous) операция записи не возвращается до тех пор, пока записанные данные (как минимум) не будут сохранены в кэше ядра буфера. Синхронная операция считывания не возвращается до тех пор, пока полученные данные не будут сохранены в пользовательском буфере, предоставляемом приложением. С другой стороны, *асинхронная* (asynchronous) операция записи может вернуться еще до того, как данные покинут пользовательское пространство; асинхронная операция считывания может вернуться еще до того, как полученные данные будут доступны. Таким образом, при запросе на операцию система вполне может не выполнить ее сразу, а поставить в очередь. Разумеется, в данном случае нам нужен какой-то механизм, позволяющий определить, когда именно завершится операция и насколько успешно.

Синхронизированная (synchronized) операция более строгая и безопасная, чем синхронная. При синхронизированной операции записи данные сбрасываются на диск. Так можно гарантировать, что находящиеся на диске данные всегда будут синхронизированы с информацией, расположенной в соответствующих буферах ядра. Синхронизированная операция считывания всегда возвращает наиболее актуальную копию данных, которая, как предполагается, должна быть взята с диска.

Итак, термины «синхронный» и «асинхронный» описывают следующий аспект: будут ли операции ввода-вывода дожидаться определенного события (например, сохранения данных на диске) и лишь потом возвращаться. В свою очередь, термины «синхронизированный» и «несинхронизированный» конкретно указывают, какая операция должна произойти (например, запись данных на диск).

Как правило, операции записи UNIX являются синхронными и несинхронизированными, а операции считывания — синхронными и синхронизированными¹. При операциях записи возможны любые комбинации этих характеристик (табл. 4.1).

¹ С технической точки зрения операции считывания также являются несинхронизированными, как и операции записи, но ядро гарантирует, что страничный кэш содержит актуальные данные. Это означает, что информация из страничного кэша всегда идентична информации с диска либо новее ее. Таким образом, поведение на практике всегда является синхронизированным. Строить работу иначе нецелесообразно.

Таблица 4.1. Синхронность операций записи

	Синхронизированные	Несинхронизированные
Синхронные	Операции записи не возвращаются до тех пор, пока данные не будут сброшены на диск. Такое поведение обеспечивается путем указания O_SYNC при открытии файла	Операции записи не возвращаются, пока данные не будут сохранены в буферах ядра. Это обычное поведение
Асинхронные	Операции записи возвращаются, как только запрос поставлен в очередь. Когда операция записи наконец будет выполнена, данные гарантированно уже будут находиться на диске	Операции записи возвращаются, как только запрос будет поставлен в очередь. Когда операция записи наконец будет выполнена, данные гарантированно будут как минимум сохранены в буферах ядра

Операции считывания всегда являются синхронизированными, так как считывать неактуальные данные практически бессмысленно. Однако такие операции бывают как синхронными, так и асинхронными (табл. 4.2).

Таблица 4.2. Синхронность операций считывания

	Синхронизированные
Синхронные	Операции считывания не возвращаются, пока информация, являющаяся актуальной, не будет сохранена в предоставленном буфере (это обычное поведение)
Асинхронные	Операции считывания возвращаются, как только запрос поставлен в очередь, но когда операция наконец завершится, возвращаемые данные будут максимально актуальными

В гл. 2 было рассмотрено, как сделать операции записи синхронизированными (с помощью флага O_SYNC), а также как гарантировать синхронизированность всего ввода-вывода в определенной точке посредством вызова fsync() и ему подобных). Теперь рассмотрим, что нужно сделать, чтобы добиться асинхронности всех операций считывания и записи.

Асинхронный ввод-вывод. Для выполнения асинхронного ввода-вывода требуется поддержка ядра на низших уровнях. POSIX 1003.1-2003 определяет интерфейсы aio, которые Linux, к счастью, реализует. Библиотека aio предоставляет семейство функций, предназначенных для отправки асинхронного ввода-вывода и получения уведомлений об их завершении:

```
#include<aio.h>

/* блок для управления асинхронным вводом-выводом */
struct aiocb{
    int aio_fildes;      /* дескриптор файла */
    int aio_lio_opcode;  /* операция для выполнения */
    int aio_reqprio;     /* смещение приоритета запроса */
    volatile void*aio_buf; /* указатель на буфер */
    size_t aio_nbytes;   /* длина операции */
    struct sigevent aio_sigevent; /* номер и значение сигнала */

    /* далее следуют внутренние закрытые члены... */
};
```

```
};  
  
int aio_read (struct aiocb *aiocbp);  
int aio_write (struct aiocb *aiocbp);  
int aio_error (const struct aiocb *aiocbp);  
int aio_return (struct aiocb *aiocbp);  
int aio_cancel (int fd, struct aiocb *aiocbp);  
int aio_fsync (int op, struct aiocb *aiocbp);  
int aio_suspend (const struct aiocb * const cblist[],  
                 int n,  
                 const struct timespec *timeout);
```

Планировщики и производительность ввода-вывода

В современных системах относительная разница в производительности между дисками и другим аппаратным обеспечением довольно велика, причем она продолжает увеличиваться. Самый медленный компонент диска — это процесс перемещения считывающей/записывающей головки от одного сектора к другому. Эта операция называется *позиционированием*. В мире, где длительность большинства операций укладывается в пару процессорных циклов (каждый из которых может занимать треть наносекунды), средняя длительность операции позиционирования может в среднем составлять около 8 миллисекунд. Совсем немного, но тем не менее *в 25 миллионов раз дольше, чем длится один цикл процессора!*

При такой огромной разнице в производительности между жесткими дисками было бы крайне нерационально и неэффективно отправлять запросы ввода-вывода на диск именно в том порядке, в котором они выдаются, поэтому в современных ядрах операционных систем используются *планировщики ввода-вывода*. Их задача — свести к минимуму количество операций позиционирования. Для этого они манипулируют порядком обслуживания запросов на ввод-вывод и временем, в которое они удовлетворяются. Планировщики ввода-вывода проделывают огромную работу, направленную на снижение издержек при доступе к диску, негативно влияющих на производительность.

Адресация диска

Чтобы понять роль планировщика ввода-вывода, необходимы некоторые базовые знания. Жесткие диски адресуют расположенные на них данные с помощью знакомой читателю геометрической схемы, в которой применяются цилиндры, головки и секторы — такой механизм называется *CHS-адресацией*. Жесткий диск состоит из множества *блинов*. В состав каждого блина входят один диск, шпиндель и считывающе-записывающая головка. Каждый блин похож на CD (или грампластинку), а совокупность блинов жесткого диска напоминает стопку CD. Каждый блин разделен на кольцевидные *дорожки*, как и CD. Каждая дорожка, в свою очередь, делится на целое число *секторов*.

Чтобы найти на диске конкретный элемент данных, логика дисководов опирается на три фрагмента информации: значения цилиндра, головки и сектора. Значение цилиндра указывает дорожку, на которой находятся данные. Если наложить блины один на другой, то дорожки с указанным номером, выбранные на каждом блине, образуют своеобразный цилиндр. Иными словами, цилиндр — это совокупность дорожек, взятых с каждого блина и расположенных на заданном расстоянии от центра. Значение головки соответствует именно той головке, которая будет выполнять считывание или запись (соответственно, по значению головки определяется и блин, на котором будет происходить эта операция). Теперь область поиска сужается до конкретной дорожки на конкретном блине. После этого диск использует значение сектора, чтобы идентифицировать нужный сектор на известной дорожке. Итак, поиск завершен. Жесткому диску известно, в каком блине, на какой дорожке и в каком секторе находятся интересующие нас данные. Считывающе-записывающая головка может быть размещена на нужной дорожке нужного блина, после чего информация будет считана с требуемого сектора (или записана в него).

К счастью, современные жесткие диски не вынуждают компьютер оперировать дисками в режиме работы с цилиндрами, головками и секторами. Вместо этого такие современные диски ассоциируют уникальный *номер блока* (также называемого *физическим блоком* или *блоком устройства*) с тройной комбинацией, состоящей из цилиндра/головки/сектора. После этого современная операционная система может адресовать жесткие диски на базе таких номеров блоков. Этот процесс именуется *логической адресацией блоков* (LBA); на внутрисистемном уровне жесткий диск преобразует номер блока в соответствующий CHS-адрес¹. Это не гарантируется, однако ассоциирование блоков с CHS обычно является последовательным: как правило, логический блок n физически является смежным с логическим блоком $n+1$. Как мы вскоре увидим, такое последовательное ассоциирование очень важно.

В свою очередь, файловые системы существуют только на уровне программного обеспечения. Они оперируют собственными единицами, которые называются *логическими блоками* (иногда используется термин «*блоки файловой системы*» или совсем непрозрачный термин «*блоки*»). Размер логического блока должен без остатка делиться на размер физического блока. Иными словами, каждый логический блок файловой системы ассоциируется с одним или несколькими физическими блоками диска.

Жизненный цикл планировщика ввода-вывода

Планировщики ввода-вывода выполняют две основные операции: объединение и сортировку. В процессе *объединения* берется два и более смежных запроса ввода-вывода, которые затем комбинируются в единый запрос. Рассмотрим два запроса. Один из них должен выполнить считывание блока 5, а другой — считывание блоков 6, 7. Два этих запроса объединяются в один, в ходе которого происходит

¹ Пределы абсолютного размера такого номера блока — основной фактор, определяющий различные параметры жестких дисков на протяжении многих лет.

считывание блоков 5–7. Общий объем ввода-вывода будет таким же, как и в первом случае, но количество операций ввода-вывода уменьшится вдвое.

Сортировка (sorting) — более важная операция. В ходе сортировки находящиеся в очереди запросы на ввод-вывод упорядочиваются по возрастанию номера блока. Например, если у нас есть запросы на считывание информации из блоков 52, 109 и 7, то планировщик ввода-вывода отсортирует их в следующем порядке: 7, 52, 109. Если после этого будет выдан запрос на считывание блока 81, то он займет место между блоками 52 и 109. Соответственно, планировщик ввода-вывода будет направлять запросы на диск в порядке, в котором они стоят в очереди: 7, 52, 81 и, наконец, 109.

Таким образом минимизируется количество операций позиционирования (подвода головок) при работе с диском. Вместо потенциально бессистемного перемещения — прыжки по всему диску то вперед, то назад — головка движется плавно и линейно. Позиционирование — самая затратная операция при работе с диском, поэтому производительность ввода-вывода в результате значительно возрастает.

Помощь при считывании

Каждый запрос на считывание должен возвращать актуальные данные. Следовательно, если запрошенные данные отсутствуют в страничном кэше, то считывающий процесс должен блокироваться, пока данные не будут получены с диска. Потенциально такая операция может получиться достаточно длительной. Такое воздействие на производительность называется *задержкой чтения*.

Обычное приложение может выдать несколько запросов на ввод-вывод за краткий период времени. Каждый запрос синхронизируется отдельно, поэтому последующие запросы *зависят* от того, как выполняются предыдущие. Допустим, нам требуется прочитать все файлы в каталоге. Программа открывает первый файл, считывает его часть, дожидается данных, считывает другой фрагмент и т. д., пока не будет прочитан весь файл. Затем программа таким же образом начинает обрабатывать следующий файл. Запросы становятся сериализованными: последующий запрос не может быть выполнен ранее, чем закончится обработка текущего.

В этом отношении запросы на считывание разительно отличаются от запросов на запись, которые (в своем стандартном, несинхронизированном состоянии) не должны инициировать какого-либо дискового ввода-вывода до определенного момента в будущем. Соответственно, с точки зрения пользовательского приложения запросы на запись *текут*, не обремененные производительностью диска. Такое потоковое поведение только осложняет проблему считывания; текущие операции записи могут оттягивать на себя практически все ресурсы ядра и диска. Этот феномен известен под названием *записи, истощающей чтение*.

Если бы планировщик ввода-вывода *всегда* сортировал новые запросы в порядке их поступления, то запросы из отдаленных блоков могли бы «голодать» неопределенно долго. Вернемся к нашему примеру. Если бы новые запросы непрерывно поступали к блокам в диапазоне 50–60, то запрос к блоку 109 так и не был

бы обслужен. Задержка при чтении является важнейшим показателем, поэтому подобное поведение может исключительно пагубно сказаться на производительности системы. Таким образом, планировщики ввода-вывода применяют специальный механизм, предотвращающий зависание запросов.

Самый простой способ, позволяющий справиться с подобной проблемой (например, лифтовой алгоритм Линуса¹, применявшийся в планировщике ввода-вывода в ядре Linux 2.4) заключается в остановке процесса вставки и сортировки, если в очереди оказывается достаточно старый запрос. В данном случае мы снижаем общую производительность ради более «честного» обслуживания запросов. В случае со считыванием такой подход позволяет уменьшить длительность задержек. Проблема в том, что такая эвристика слишком все упрощает, поэтому разработчики ядра Linux 2.6 признали непрактичность лифтового алгоритма Линуса и заменили его сразу несколькими новыми планировщиками ввода-вывода.

Планировщик ввода-вывода Deadline

Планировщик ввода-вывода Deadline был создан для решения проблем, возникающих с традиционным планировщиком, применявшимся в версии ядра 2.4 и с традиционными лифтовыми алгоритмами вообще. Лифт Линуса ведет отсортированный список запросов ввода-вывода, ожидающих обработки. Раньше всех будет обслужен запрос ввода-вывода, который находится в голове очереди. Планировщик ввода-вывода Deadline также сохраняет такую очередь, но немного подгоняет процесс, организуя две дополнительные очереди: *считывающую FIFO-очередь* и *записывающую FIFO-очередь*. Элементы в каждой из этих очередей сортируются в порядке прибытия (первым пришел — первым обслужен). Считывающая FIFO-очередь, как понятно из названия, содержит только запросы на считывание. Аналогично, записывающая FIFO-очередь содержит только запросы на запись. Каждому запросу, находящемуся в FIFO-очереди, присваивается значение устаревания. Так, в считывающей FIFO-очереди запрос считается устаревшим через 500 миллисекунд. В записывающей FIFO-очереди запрос устаревает через 5 секунд.

Когда поступает новый запрос на ввод-вывод, он сортируется и вставляется в соответствующую FIFO-очередь (считывающую или записывающую) и оказывается в ее конце. Как правило, на жесткий диск поступают запросы ввода-вывода из начала стандартной очереди. Таким образом мы улучшаем общую пропускную способность, минимизируя при этом количество операций позиционирования, поскольку глобальная очередь сортируется по номеру блока (как в лифтовом алгоритме Линуса).

Однако если в голове одной из FIFO-очередей (считывающей или записывающей) оказывается объект, достигший действующего в этой очереди значения устаревания, то планировщик прекращает диспетчеризацию запросов ввода-вывода

¹ Да, в честь этого великого человека назван планировщик ввода-вывода. Планировщики ввода-вывода иногда называются лифтовыми алгоритмами, так как решаемая ими проблема напоминает обеспечение бесперебойной работы лифта.

из глобальной очереди и приступает к обслуживанию запросов из той FIFO-очереди, в которой был обнаружен устаревший объект. Обслуживается устаревший запрос из головы этой очереди, а также, на всякий случай, еще несколько запросов, следующих за ним. Планировщик ввода-вывода должен проверять и обрабатывать только запросы, которые находятся ближе к голове очереди, поскольку именно эти запросы являются самыми старыми.

Таким образом, планировщик ввода-вывода Deadline может налагать нежесткие сроки на обработку запросов ввода-вывода. Он не гарантирует, что запрос ввода-вывода обязательно будет обслужен до устаревания, но в большинстве случаев успевает их обслужить незадолго до момента устаревания. Таким образом, планировщик ввода-вывода обеспечивает достаточно высокую пропускную способность, не вызывая чрезмерно долгого голодания какого-либо отдельного запроса. Запросы на считывание устаревают быстрее, поэтому проблема истощающего считывания сводится к минимуму.

Планировщик ввода-вывода Anticipatory

Планировщик ввода-вывода Deadline хорош, но несовершенен. Вспомните наш разговор о зависимости от чтения. При использовании планировщика ввода-вывода Deadline первый запрос на считывание из серии таких запросов обслуживается в срочном порядке, в момент устаревания или непосредственно перед ним, после чего этот планировщик возвращается к обслуживанию запросов из отсортированной очереди. В целом приемлемо. Однако что делать, если приложение отправляет нам еще один запрос на считывание? Скоро и он устареет, планировщик ввода-вывода отправит его на диск. На диске нам придется совершить операцию позиционирования, чтобы вовремя обработать неожиданно поступивший запрос, а затем — еще одну операцию позиционирования, чтобы вернуться к обработке запросов из отсортированной очереди. Такие подводы головок туда и обратно могут продолжаться достаточно долго, поскольку подобное поведение характерно для многих приложений. Конечно, задержка остается минимальной, но пропускная способность оставляет желать лучшего, так как запросы на считывание продолжают поступать и для их обработки постоянно требуется подводить головки к разным местам диска. Производительность могла бы повыситься, если бы после поступившего запроса диск сделал небольшую паузу, а не возвращался сразу к обслуживанию отсортированной очереди. Если бы за время этой паузы поступил новый внеочередной запрос, то мы бы обработали его без перехода к отсортированной очереди. Однако, к сожалению, когда приложение назначит и отправит на обработку свой новый зависимый запрос на считывание, планировщик ввода-вывода уже успеет переключиться на общую очередь.

Эта проблема связана с пресловутыми зависимыми операциями считывания. Каждый новый запрос на считывание выдается только после возврата предыдущего. Однако к моменту, как приложение получит прочитанные данные, после чего будет запланирован следующий прогон этого приложения и оно выдаст новый запрос на считывание, планировщик ввода-вывода уже уйдет вперед и приступит к обслуживанию других запросов. В результате мы имеем две лишние операции

позиционирования при каждом считывании: диск позиционирует головки для считывания, обслуживает этот запрос, а потом вновь выполняет позиционирование, уходя назад. Если бы планировщик ввода-вывода мог каким-то образом узнать (предположить), что вскоре в эту же часть диска поступит новый запрос на считывание, он мог бы не выполнять двух лишних операций позиционирования, а остаться на месте и дожидаться следующего зависимого считывания. Избавившись от этих ужасных подводов головок, мы сократили бы длительность ожидания не менее чем на несколько миллисекунд.

Именно таким образом и работает планировщик ввода-вывода *Anticipatory*. Изначально он разрабатывался и как планировщик, учитывающий крайние сроки, но затем в него внедрили механизм предположений. Когда поступает запрос на считывание, планировщик ввода-вывода *Anticipatory* обслуживает его до наступления крайнего срока — как обычно. Однако в отличие от рассмотренного выше планировщика *Deadline* планировщик *Anticipatory* ненадолго задерживается на месте, ничего не делая в течение определенного довольно долгого периода — до 6 миллисекунд. Вполне вероятно, что за эти 6 миллисекунд приложение пришлет еще один запрос на считывание в ту же часть файловой системы, куда пришел первый. Если это действительно произойдет, то новый запрос будет обслужен немедленно и планировщик ввода-вывода *Anticipatory* вновь задержится здесь на некоторое время. Если же 6 миллисекунд пройдут без нового вызова, планировщик ввода-вывода *Anticipatory* «решит», что прогноз был неверным, и вернется к выполнению основной работы, которую прежде оставил (например, к обслуживанию отсортированной очереди). Даже если будет верно предугадано умеренное количество запросов, экономится огромное количество времени — по две операции позиционирования на каждый внеочередной вызов. Большинство операций считывания зависимы, поэтому подобное ожидание чаще всего оправдывает себя.

Планировщик ввода-вывода CFQ

Планировщик ввода-вывода *CFQ* работает на достижение схожих целей, но применяет иной подход¹. При применении *CFQ* каждому процессу присваивается собственная очередь, а каждая очередь получает свой квант времени. Планировщик ввода-вывода посещает каждую очередь по принципу карусели, обслуживая запросы из определенной очереди, пока не истечет ее квант времени либо пока в ней не останется ожидающих запросов. В последнем случае *CFQ*-планировщик в течение короткого периода будет простаивать — по умолчанию этот период равен 10 миллисекундам — и ожидать поступления нового запроса в эту очередь. Если ожидание себя оправдывает, то планировщик ввода-вывода избежит лишней операции позиционирования, если нет — просто перейдет к обработке следующей очереди.

¹ Далее по тексту планировщик ввода-вывода *CFQ* рассмотрен в таком виде, в котором он реализован в настоящее время. В предыдущих версиях не использовались кванты времени и эвристика предположений, но общий принцип работы был примерно таким же.

В рамках каждой процессной очереди синхронизированные запросы (например, на считывание) получают приоритет перед несинхронизированными. Таким образом, CFQ предпочитает одни запросы на считывание другим и предотвращает возникновение истощающего считывания. Каждый процесс имеет собственную очередь, поэтому планировщик ввода-вывода CFQ в равной степени удовлетворяет все запросы, но при этом обеспечивает высокую общую производительность.

Планировщик ввода-вывода CFQ хорошо работает при большинстве нагрузок и очень хорошо угадывает с выбором.

Планировщик ввода-вывода Noop

Планировщик ввода-вывода Noop — простейший из себе подобных. Он вообще не занимается сортировкой, выполняя только несложное объединение. Он используется в специализированных устройствах, не требующих собственной сортировки запросов (или выполняющих ее отдельно).

Твердотельные диски

Твердотельные диски (Solid-State Drives, SSD), в частности флэш-диски, становятся все популярнее с каждым годом. Целые классы устройств, например смартфоны и планшеты, не имеют вращательных дисковых накопителей; используется только флэш-память. Твердотельные диски (флэш-диски в том числе) тратят на позиционирование гораздо меньше времени, чем традиционные жесткие диски, так как при поиске нужного блока данных не тратится время на вращение. Механизм ссылки на информационные блоки в SSD напоминает принцип, применяемый в ОЗУ с произвольным доступом. Он может работать эффективнее, считывая большие области смежных данных одним махом, но также может обращаться и к разрозненным данным в любой части диска, не снижая производительность.

Следовательно, на таких дисках польза от сортировки запросов ввода-вывода существенно уменьшается и применение планировщиков ввода-вывода оказывается не таким актуальным. Во многих системах, работающих с твердотельными дисками, просто применяется планировщик Noop, так как он обеспечивает актуальное в данном случае объединение, но не занимается сортировкой. Если же система должна оптимизировать производительность при интерактивной работе, то она обычно использует планировщик ввода-вывода CFQ даже при оперировании твердотельными дисками.

Выбор и настройка планировщика ввода-вывода

Применяемый по умолчанию планировщик ввода-вывода можно выбрать во время загрузки, указав его в командной строке с помощью параметра ядра `iosched`. Допустимые значения — `as`, `cfq`, `deadline` и `noop`. Планировщик ввода-вывода можно выбрать и во время выполнения (отдельно для каждого устройства) с помощью

/sys/block/[device]/queue/scheduler, где device — интересующее нас блочное устройство. При считывании этого файла возвращается актуальный планировщик ввода-вывода. Записав в этот файл один из допустимых параметров, мы задаем планировщик ввода-вывода. Например, чтобы определить для устройства hda планировщик CFQ, мы напишем:

```
# echo cfq > /sys/block/hda/queue/scheduler
```

В каталоге /sys/block/[device]/queue/iosched содержатся файлы, позволяющие администратору узнавать и устанавливать настраиваемые значения, относящиеся к планировщику ввода-вывода. Для изменения любого из этих значений требуются права администратора.

Хороший разработчик пишет программы, с одинаковым успехом работающие с любой подсистемой ввода-вывода. Тем не менее знание этих подсистем определенно помогает писать оптимальный код.

Оптимизация производительности ввода-вывода

Дисковый ввод-вывод — сравнительно медленная операция по сравнению со скоростью работы других компонентов системы, но в то же время ввод-вывод — важнейшая часть современной вычислительной техники, поэтому повышение производительности ввода-вывода оказывается насущной проблемой.

Минимизация количества операций ввода-вывода (путем объединения множества мелких операций в несколько сравнительно крупных), выполнение ввода-вывода, выровненного по границам блоков, применение пользовательской буферизации (см. гл. 3) — важные инструменты системного программиста. Не менее существенны в нашем деле и такие навыки, как применение продвинутых приемов ввода-вывода (векторный и позиционный ввод-вывод — см. гл. 2), в частности асинхронного ввода-вывода. Системный программист обязательно должен ориентироваться в этих возможностях.

В наиболее трудоемких приложениях, работа которых связана с интенсивным вводом-выводом, но при этом должна быть безотказной, мы можем задействовать и дополнительные приемы, позволяющие повысить производительность. Хотя выше мы уже говорили, что ядро Linux использует сложные планировщики ввода-вывода для минимизации количества подвода головок, приложения из пользовательского пространства также могут работать в этом направлении и дополнительно повышать производительность.

Планирование ввода-вывода в пользовательском пространстве

Приложения, интенсивно выполняющие ввод-вывод и выдающие большое количество запросов на такие операции, должны до капли выжимать возможности производительности при сортировке и объединении таких запросов, ожидающих

обработки. Они должны участвовать в решении тех же задач, которые обычно относятся к компетенции планировщика ввода-вывода Linux¹.

Зачем дважды выполнять одну и ту же работу, если мы знаем, что планировщик ввода-вывода будет сортировать запросы поблочно, минимизируя количество операций позиционирования и обеспечивая плавное линейное движение головки диска? Допустим, у нас есть приложение, выдающее множество несортированных запросов ввода-вывода. В принципе, эти запросы приходят в общую очередь планировщика ввода-вывода в случайном порядке. Планировщик ввода-вывода делает свою работу, сортируя и объединяя запросы перед отправкой их на диск, но когда такие запросы начинают попадать на диск, приложение продолжает генерировать ввод-вывод и отправлять новые запросы. Таким образом, планировщик ввода-вывода может рассортировать в единицу времени лишь часть поступающих запросов.

Следовательно, если приложение генерирует множество подобных запросов — в особенности если они касаются данных, разбросанных по всему диску, — целесообразно организовать сортировку запросов еще до их отправки. Так можно обеспечить их попадание к планировщику ввода-вывода в желаемом порядке.

Однако приложение из пользовательского пространства не располагает доступом ко всей информации, доступной ядру. На низших уровнях в планировщике ввода-вывода запросы уже представлены в параметрах физических дисковых блоков. Сортировать их совсем просто. Однако в пользовательском пространстве запросы «состоят» из файлов и смещений. Приложения из пользовательского пространства должны зондировать информацию и делать информированные предположения о компоновке файловой системы.

Итак, наша цель — определить вариант перехода по диску, который позволяет выполнить заданный набор операций ввода-вывода для известных файлов, обходясь минимальным количеством подводов головок. Эта задача может решаться в пользовательском приложении несколькими способами. Сортировка может происходить по:

- полному пути;
- номеру индексного дескриптора;
- физическому блоку на диске, соответствующему заданному файлу.

При любом из этих вариантов приходится идти на компромисс. Вкратце рассмотрим все три возможности.

Сортировка по пути к файлу

Сортировка по имени пути к файлу — это простейший, но наименее эффективный способ аппроксимации побитовой сортировки. В большинстве файловых систем применяются такие алгоритмы компоновки информации, в соответствии

¹ Приемы, описанные здесь, следует применять только в приложениях, которые должны очень активно и безотказно выполнять большие объемы ввода-вывода. Если в приложении не происходит активного ввода-вывода, то сортировка таких запросов на уровне приложения — если материал для нее вообще имеется — это глупая и ненужная затея.

с которыми файлы из каждого каталога — и, следовательно, файлы дочерних каталогов, относящихся к одному и тому же родительскому, — обычно располагаются в смежных областях диска. Это свойство прослеживается тем более явно, поскольку файлы из одного каталога часто создаются примерно в одно и то же время.

Таким образом, при сортировке по имени файла происходит грубая аппроксимация физических местоположений файлов на диске. Несомненно, два файла из одного каталога с большей вероятностью окажутся на диске поблизости друг от друга, чем два файла, относящиеся к совершенно разным частям файловой системы. Недостаток этого подхода заключается в том, что он не учитывает фрагментацию диска: чем сильнее фрагментирована файловая система, тем бесполезнее сортировать ее информацию по пути файла. Даже при игнорировании фрагментации сортировка по пути лишь приблизительно отображает фактическое поблочное расположение. С другой стороны, сортировка по пути так или иначе применима во всех файловых системах. Независимо от того, каким образом в данной системе komponуются файлы, сортировка по пути гарантирует как минимум умеренную точность. Кроме того, выполнять ее очень просто.

Сортировка по индексному дескриптору

Индексные дескрипторы — это сущности UNIX, в которых содержатся метаданные, ассоциированные с отдельными файлами. Сама информация файла может занимать на диске много физических блоков, но каждому файлу соответствует всего один индексный дескриптор. В этом дескрипторе содержится следующая информация: размер файла, права доступа к нему, владелец и т. д. Подробнее мы поговорим об индексных дескрипторах в гл. 8, а пока достаточно знать два факта: с каждым файлом ассоциирован индексный дескриптор и каждому индексному дескриптору присваивается уникальный идентификационный номер.

Сортировка по индексному дескриптору в целом точнее, чем сортировка по пути к файлу, при условии, что следующее отношение:

номер индексного дескриптора файла i < номера индексного дескриптора файла j

предполагает *в целом* следующее:

физические блоки файла i расположены раньше физических блоков файла j

Именно такая ситуация складывается в UNIX-подобных файловых системах, например, ext3 и ext4. Если файловая система вообще не реализует индексные дескрипторы, то в ней может происходить что угодно, но номер индексного дескриптора (с чем бы он ни ассоциировался) — все равно хороший ориентир для аппроксимации первого порядка.

Получение номера индексного дескриптора происходит с помощью системного вызова `stat()`, также рассматриваемого в гл. 8. Если при каждом запросе ввода-вывода нам будет известен индексный дескриптор, ассоциированный с файлом, то запросы можно сортировать по номеру индексного дескриптора в порядке возрастания.

Вот простая программа, выводящая на экран индексный дескриптор заданного файла:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/*
 * get_inode – возвращает индексный дескриптор файла, ассоциированного
 * с заданным файловым дескриптором, либо возвращает -1 при ошибке
 */
int get_inode (int fd)
{
    struct stat buf;
    int ret;

    ret = fstat (fd, &buf);
    if (ret < 0) {
        perror ("fstat");
        return -1;
    }

    return buf.st_ino;
}

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

    if (argc < 2) {
        fprintf (stderr, "usage: %s <file>\n", argv[0]);
        return 1;
    }

    fd = open (argv[1], O_RDONLY);
    if (fd < 0) {
        perror ("open");
        return 1;
    }

    inode = get_inode (fd);
    printf ("%d\n", inode);

    return 0;
}
```

Вы можете без труда адаптировать функцию `get_inode()` для использования в ваших программах.

Сортировка по номеру индексного дескриптора имеет свои достоинства. Этот номер легко получить, сам процесс такой сортировки прост, кроме того, в данном случае достигается хорошая аппроксимация к физической компоновке файла. Основные недостатки связаны с тем, что фрагментация ухудшает получаемую аппроксимацию, сама аппроксимация фактически основана на догадке и в файловых системах не-UNIX получается сравнительно неточной. Тем не менее это наиболее часто используемый метод планирования запросов ввода-вывода в пользовательском пространстве.

Сортировка по физическому блоку

При создании собственного лифтового алгоритма лучше всего, конечно же, сортировать ввод-вывод по физическому блоку диска. Как было сказано выше, каждый файл подразделяется на логические блоки, являющиеся мельчайшими единицами размещения в файловой системе. Размер логического блока зависит от файловой системы; каждый логический блок ассоциирован с одним физическим блоком. Следовательно, мы можем определить количество логических блоков в файле, узнать, с какими физическими блоками они ассоциированы, и выполнить сортировку по этому параметру.

Ядро предоставляет метод для получения физического блока диска исходя из номера логического блока в файле. Это делается с помощью системного вызова `ioctl()`, который будет рассмотрен в гл. 8, с применением команды `FIBMAP`:

```
ret = ioctl (fd, FIBMAP, &block);  
if (ret < 0)  
    perror("ioctl");
```

Здесь `fd` — это дескриптор интересующего нас файла, `a block` — логический блок, физический блок которого мы хотим узнать. При успешном возврате `block` заменяется номером физического блока. Передаваемые вызову логические блоки индексируются, начиная с нуля, и указываются относительно к конкретному файлу. Таким образом, если файл состоит из восьми логических блоков, для него допустимы значения от 0 до 7.

Таким образом, определение соответствия между логическими и физическими блоками — это двухэтапный процесс. Во-первых, мы должны выяснить количество блоков в конкретном файле. Это делается с помощью системного вызова `stat()`. Во-вторых, для каждого логического блока необходимо выдать запрос `ioctl()`, чтобы найти соответствующий физический блок.

Вот простая программа, выполняющая именно такую операцию с файлом, переданным через командную строку:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <sys/ioctl.h>
```

```
#include<linux/fs.h>

/*
 * get_block – для файла, связанного с заданным fd, возвращает
 * физический блок, ассоциированный с logical_block
 */
int get_block (int fd, int logical_block)
{
    int ret;

    ret = ioctl (fd, FIBMAP, &logical_block);
    if (ret < 0) {
        perror ("ioctl");
        return -1;
    }

    return logical_block;
}

/*
 * get_nr_blocks – возвращает количество логических блоков,
 * занимаемых файлом, связанным с fd
 */
int get_nr_blocks (int fd)
{
    struct stat buf;
    int ret;

    ret = fstat (fd, &buf);
    if (ret < 0) {
        perror ("fstat");
        return -1;
    }
    return buf.st_blocks;
}

/*
 * print_blocks – для каждого логического блока, занимаемого файлом,
 * ассоциированным с fd, выдает в стандартный вывод пару значений
 * "(логический блок, физический блок)"
 */
void print_blocks(int fd)
{
    int nr_blocks, i;

    nr_blocks = get_nr_blocks (fd);
    if (nr_blocks < 0) {
        fprintf (stderr, "get_nr_blocks failed!\n");
        return;
    }

    if (nr_blocks == 0) {
```



```

        printf ("no allocated blocks\n");
        return;
    } else if (nr_blocks == 1)
        printf ("1 block\n\n");
    else
        printf ("%d blocks\n\n", nr_blocks);

    for (i = 0; i <nr_blocks; i++) {
        int phys_block;

        phys_block = get_block (fd, i);
        if (phys_block <0) {
            fprintf (stderr, "get_block failed!\n");
            return;
        }
        if (!phys_block)
            continue;

        printf ("%u, %u ", i, phys_block);
    }

    putchar ('\n');
}

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

    if (argc < 2) {
        fprintf (stderr, "usage: %s <file>\n", argv[0]);
        return 1;
    }

    fd = open (argv[1], O_RDONLY);
    if (fd < 0) {
        perror ("open");
        return 1;
    }

    print_blocks(fd);

    return 0;
}

```

Данные в файле обычно расположены непрерывно, поэтому будет как минимум непросто отсортировать запросы ввода-вывода в точном соответствии с логическими блоками. Гораздо удобнее учитывать при сортировке лишь положение первого логического блока конкретного файла. Следовательно, `get_nr_blocks()` не требуется и наше приложение может выполнять сортировку по возвращаемому значению:

```
get_block (fd, 0);
```

Недостаток FIBMAP заключается в том, что для его использования нужна возможность CAP_SYS_RAWIO — фактически привилегии администратора. Следовательно, приложения, не обладающие такими привилегиями, не могут использовать FIBMAP. Кроме того, хотя команда FIBMAP и стандартизирована, ее точная реализация зависит от конкретной файловой системы. Сравнительно распространенные системы ext2 и ext3 ее поддерживают, а экзотические, возможно, и нет. В последнем случае вызов `ioctl()` вернет `EINVAL`.

Большое достоинство этого подхода заключается в следующем: вызов возвращает *точный* физический блок диска — а именно по этому параметру мы собираемся выполнять сортировку. Даже если требуется отсортировать весь ввод-вывод единственного файла, ориентируясь на местоположение всего одного блока (планировщик ввода-вывода в ядре сортирует каждый отдельный запрос в зависимости от блока), этот подход обеспечивает упорядочение, близкое к оптимальному. Конечно, это не отменяет неудобств, связанных с необходимостью привилегий администратора.

Резюме

В трех последних главах мы рассмотрели все аспекты файлового ввода-вывода в Linux. В гл. 2 мы познакомились с основами файлового ввода-вывода в Linux (фактически с основами файлового вывода в любых системах UNIX), поговорили о системных вызовах `read()`, `write()`, `open()` и `close()` и др. В гл. 3 мы обсудили буферизацию в пользовательском пространстве и реализацию этой возможности в стандартной библиотеке C. В гл. 4 мы рассмотрели различные аспекты расширенного ввода-вывода — от сравнительно сложных, но при этом мощных системных вызовах ввода-вывода до приемов оптимизации и способов борьбы с излишними операциями позиционирования, существенно замедляющими работу.

В следующих двух главах мы поговорим о создании процессов, уничтожении их и управлении ими. Вперед!

5 Управление процессами

Как уже было сказано в гл. 1, процессы — это одна из самых фундаментальных абстракций в системах UNIX после файлов. Если рассматривать процессы как объект выполняемого кода — живой, работающей, активной программы, — они являются чем-то большим, нежели просто язык ассемблера, и состоят из данных, ресурсов, состояния и виртуального процессора.

В этой главе мы рассмотрим главные составляющие процесса, от создания до завершения. Основы не претерпели значительных изменений с первых дней существования UNIX. Именно здесь, в управлении процессами, ярче всего проявляются дальновидность и долговечность первоначальной концепции UNIX. История системы интересна и весьма незаурядна, так как в концепции UNIX создание нового процесса отделяется от загрузки нового двоичного образа. Хотя в большинстве случаев эти действия выполняются совместно, разделение позволяет экспериментировать и открывает широкие возможности для развития каждой задачи. Таким образом, эта редко встречающаяся концепция сохранилась до наших дней, в то время как большинство операционных систем предлагают один системный вызов для запуска одной программы, в UNIX требуются два: `fork` и `exec`. Однако до изучения этих системных вызовов давайте подробно рассмотрим сам процесс.

Программы, процессы и потоки

Бинарный модуль — это скомпилированный, исполняемый код, находящийся в каком-либо хранилище данных, например на диске. Следовательно, мы можем также использовать термин «*программа*»; большие и значительные бинарные модули мы можем называть *приложениями*. И `/bin/ls`, и `/usr/bin/X11` являются бинарными модулями.

Процессом является запущенная программа. Процесс включает в себя бинарный образ, загружаемый в память, и многое другое: подгрузку виртуальной памяти, ресурсы ядра, например открытые файлы, выполнение требований по безопасности (к примеру, выбор определенного пользователя), а также запуск одного или нескольких потоков. *Поток* — это одно из действий внутри процесса. Каждый поток имеет собственный виртуализированный процессор, включающий в себя стек, состояние процессора, например регистры, а также командные указатели.

Если в процессе поток только один, то процесс и является потоком. У него только один экземпляр виртуальной памяти и один виртуализированный процессор. В многопоточных процессах потоков несколько. Виртуализация памяти связана с процессом, поэтому все потоки одновременно используют одно и то же адресное пространство памяти.

Идентификатор процесса

Каждый процесс обозначается уникальным идентификатором (process ID, обычно сокращается до *pid*). *Pid* обязательно является уникальным в любой *конкретный момент времени*. Это значит, что в момент времени $t+0$ может быть только один процесс с *pid* 770 (или ни одного процесса с таким значением идентификатора), но нельзя гарантировать, что в момент времени $t+1$ не будет существовать другого процесса с тем же идентификатором *pid* 770. На практике, впрочем, ядро не обязательно быстро меняет местами идентификаторы процессов — предположение, являющееся, как вы скоро увидите, весьма небезопасным. Конечно, с точки зрения самого процесса, *его pid* остается неизменным.

Процесс *бездействия*, или *пассивный*, который выполняется ядром в отсутствие всех других процессов, имеет *pid* 0. Первый процесс, который ядро выполняет во время запуска системы, называется *процессом инициализации* и имеет *pid* 1. Обычно *init process* в Linux является программой инициализации. Мы используем термин «инициализация» для обозначения и начального процесса, запускаемого при загрузке, и специальной программы, используемой для этих целей.

Если пользователь не указывает ядру напрямую, какой процесс следует запускать (с помощью *init* в командной строке), ядро выбирает подходящий процесс инициализации самостоятельно — нечастый случай, когда политику диктует ядро. Ядро Linux перебирает четыре исполняемых модуля в следующем порядке.

1. */sbin/init* — предпочтительное и наиболее вероятное размещение процесса инициализации.
2. */etc/init* — следующее наиболее вероятное размещение процесса инициализации.
3. */bin/init* — резервное размещение процесса инициализации.
4. */bin/sh* — местонахождение оболочки Bourne, которую ядро пытается запустить, если найти процесс инициализации не удалось.

Первый из этих процессов, который будет существовать, и запустится в качестве процесса инициализации. Если не удалось запустить ни один из четырех, то ядро Linux переводит систему в состояние «паники».

После запуска процесс инициализации обрабатывает оставшуюся часть загрузки. Как правило, в нее включены инициализация системы, запуск различных сервисов и программы авторизации.

Выделение идентификатора процесса

По умолчанию ядро может выставить максимальную величину идентификатора, равную 32 768. Это объясняется необходимостью совместимости с более старыми системами UNIX, которые использовали 16-битные типы данных для идентификаторов. Администратор системы может установить большую величину через `/proc/sys/kernel/pid_max`, выделив большее пространство для `pid`, но снизив таким образом совместимость.

Идентификаторы назначаются ядром строго линейно. Если в настоящий момент наибольший имеющийся `pid` равен 17, то следующей будет назначена величина 18, даже если процесс с последним назначенным идентификатором 17 уже не выполняется во время старта нового процесса. Ядро не назначает использованные ранее идентификаторы процессов, пока не пройдет верхнее значение, то есть меньшие значения не будут устанавливаться, пока не будет достигнута величина, записанная в `/proc/sys/kernel/pid_max`. Таким образом, Linux не гарантирует уникальность идентификаторов процессов на длинные периоды, но тем не менее есть некоторая уверенность, что в течение коротких отрезков времени идентификаторы будут стабильными и уникальными.

Иерархия процессов

Процесс, запускающий другой процесс, называется *родительским*; новый процесс, таким образом, является *дочерним*. Каждый процесс запускается каким-либо другим процессом (кроме, разумеется, процессов инициализации). Таким образом, каждый дочерний процесс имеет «родителя». Эти взаимоотношения записаны в каждом идентификаторе родительского процесса (`ppid`), значение которого для дочернего процесса равно значению `pid` родительского процесса.

Каждый процесс принадлежит определенному *пользователю* и *группе*. Эти принадлежности используются для управления правами доступа к ресурсам. С точки зрения ядра пользователь и группа — это просто некие целочисленные величины. Они хранятся в файлах `/etc/passwd` и `/etc/group`, с помощью которых сопоставляются с привычными глазу пользователя UNIX именами, воспринимаемыми человеком, например имя пользователя `root` или группа `wheel` (в общем случае ядро Linux никак не взаимодействует с этими строками, оперируя с объектами посредством целочисленных величин). Каждый дочерний процесс наследует пользователя и группу, которым принадлежал родительский процесс.

Каждый процесс является также частью *группы процессов*, которая означает, по сути, его отношение к другим процессам (не следует путать группу процессов с упомянутыми выше пользователем и группой!). Дочерние процессы, как правило, принадлежат к тем же группам процессов, что и родительские. Кроме того, когда пользователь запускает конвейер (например, введя `ls | less`), все команды в конвейере становятся членами одной и той же группы процессов. Понятие группы процессов упрощает отправку сигналов или получение информации от всего конвейера, так как все дочерние процессы находятся в конвейере. С точки зрения пользователя группа процессов тесно связана с понятием *задания*.

pid_t

С точки зрения программирования идентификатор процесса обозначается типом `pid_t`, величина которого определяется в заголовочном файле `<sys/types.h>`. Конкретный тип `C` зависит от архитектуры и не определяется каким-либо стандартом `C`. В Linux, однако, для `pid_t` чаще всего используется тип данных `C int`.

Получение идентификаторов процесса и родительского процесса

Системный вызов `getpid()` возвращает идентификатор вызывающего процесса:

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getpid (void);
```

Системный вызов `getppid()` возвращает идентификатор родителя вызывающего процесса:

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getppid (void);
```

Ни один из них не может вернуть ошибку, поэтому использование данных вызовов несложно:

```
printf ("My pid=%jd\n", (intmax_t) getpid ());
printf ("Parent's pid=%jd\n", (intmax_t) getppid ());
```

Здесь мы приводим возвращенную величину к типу `intmax_t`, который является типом `C/C++`, гарантированно способным хранить в системе любое целое число со знаком. Другими словами, он равен любым другим типам для целых чисел со знаками или больше их. В сочетании с `printf()` с модификатором `(%j)` такой подход позволяет нам безопасно печатать переменные типа `integer`, назначенные `typedef`. До появления `intmax_t` не существовало приемлемого способа сделать это (если в вашей системе нет `intmax_t`, вы можете предположить, что `pid_t` принадлежит к `int`, что правдиво для большинства систем UNIX).

Запуск нового процесса

В UNIX действие загрузки в память и запуска образа программы выполняется отдельно от операции по созданию нового процесса. Один системный вызов загружает бинарную программу в память, замещая текущее содержание адресного пространства, и начинает выполнение новой программы. Это называется *выполнением* новой программы, а функциональность обеспечивается семейством вызовов `exec`.

Другой системный вызов используется для создания нового процесса, который изначально является практически копией своего родительского. Часто новый процесс немедленно приступает к выполнению новой программы. Акт создания нового процесса называется *ветвлением* и обеспечивается системным вызовом `fork()`. Два действия — сначала ветвление для создания нового процесса, а затем `exec` для открытия нового выполнения этого процесса — требуются для запуска новой программы в новом процессе. Изучим вызовы `exec` и `fork()`.

Семейство вызовов `exec`

Единой функции `exec` не существует; на одном системном вызове построено целое семейство таких функций. Сначала рассмотрим самый простой из этих вызовов, `exec1()`:

```
#include<unistd.h>
```

```
int exec1 (const char *path,  
           const char *arg,  
           ...);
```

Вызов `exec1()` замещает текущий образ процесса новым, загружая в память программу, определенную `path`. Параметр `arg` — первый аргумент этой программы. Многоточие означает переменное количество аргументов — у функции `exec1()` их количество может быть любым, дополнительные аргументы можно указывать в скобках один за другим. Список аргументов всегда завершается значением `NULL`.

Например, следующий программный код замещает выполняющуюся в настоящий момент программу с `/bin/vi`:

```
int ret;  
  
ret = exec1 ("/bin/vi", "vi", NULL);  
if (ret == -1)  
    perror ("exec1");
```

Обратите внимание: следуя соглашениям UNIX, мы передаем в качестве первого аргумента программы значение `"vi"`. Оболочка помещает последний компонент пути, то есть `"vi"`, в первый аргумент во время ветвления или запуска процессов, благодаря чему программа может проверить первый аргумент `argv[0]` для выяснения имени двоичного образа. В большинстве случаев несколько системных утилит, отображающихся пользователю под разными именами, в действительности представляют собой единую программу с жестко прописанными ссылками к различным именам. Программа использует первый аргумент, чтобы определить свое поведение.

Другой пример. Если вы хотите редактировать файл `/home/kidd/hooks.txt`, то должны запустить следующий код:

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
    perror ("execl");
```

Как правило, `execl()` не возвращает никаких значений. Успешный вызов завершается переходом к входной точке новой программы, и только что выполненный код больше не находится в процессном адресном пространстве. Если произошла ошибка, `execl()` возвращает -1 и устанавливает `errno` для обозначения проблемы. Мы рассмотрим возможные значения `errno` в этом разделе позднее.

В случае успешного выполнения вызов `execl()` изменяет не только адресное пространство и образ процесса, но и некоторые другие атрибуты процесса:

- любые ожидающие сигналы исчезают;
- любые сигналы, отлавливаемые процессом (см. гл. 10), возвращаются к своему поведению по умолчанию, поскольку обработчиков сигналов больше нет в процессном адресном пространстве;
- все блокировки памяти удаляются;
- большинство атрибутов потока возвращается к значениям по умолчанию;
- большая часть статистических данных процесса сбрасывается;
- все адресное пространство памяти, относящееся к данному процессу, включая загруженные файлы, очищается;
- все, находящееся исключительно в пользовательском пространстве, включая функциональности библиотеки C, например поведение `atexit()`, удаляется.

Некоторые свойства процесса, однако, *не* изменяются. Например, идентификатор (свой и родительский), приоритет, а также пользователь и группа остаются теми же.

Обычно при работе системных вызовов семейства `exec` наследуются и открытые файлы: у запущенных программ сохраняется полный доступ ко всем файлам, открытым в изначальном процессе, если им известны значения дескрипторов. Однако зачастую это нежелательно. Обычной практикой является закрытие файлов перед запуском `exec`, но возможно дать команду ядру делать это автоматически через `fcntl()`.

Остальная часть семейства

Кроме `execl()`, в семействе есть еще пять членов:

```
#include <unistd.h>

int execlp (const char *file,
            const char *arg,
            ...);

int execl (const char *path,
```



```

        const char *arg,
        ....
        char * const envp[]);

int execv (const char *path, char *const argv[]);

int execvp (const char *file, char *const argv[]);

int execve (const char *filename,
            char *const argv[],
            char *const envp[]);

```

Запомнить все очень просто. `l` и `v` указывают, передаются ли аргументы списком или массивом. Символ `p` указывает, что система будет искать указанный файл по полному пользовательскому пути. В командах, где используются варианты с `p`, можно указать только имя файла, если он находится в пределах пользовательского пути. Наконец, `e` обозначает, что для нового процесса создается новое окружение. Интересно, что, хотя технических ограничений для этого не существует, в семействе `exes` нет элемента, позволяющего и искать путь к файлам, и создавать новое окружение. Возможно, это объясняется тем, что варианты с `p` предназначены для использования оболочками, а процессы, исполняемые в оболочках, как правило, наследуют свое окружение от них.

В следующем фрагменте кода используется `execvp()` для выполнения `vi`, как и в предыдущем варианте, учитывая, что `vi` находится на пользовательском пути:

```

int ret;

ret = execvp ("vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
    perror ("execvp");

```

ВНИМАНИЕ

Программы установки идентификаторов группы и пользователя (`Set-group-ID` и `set-user-ID`) — это процессы, запускаемые от имени группы или пользователя, которым принадлежит их бинарный код, а не группы или пользователя, которые их вызывают. Следовательно, они не должны ссылаться на оболочку или операции, которые, в свою очередь, затрагивают оболочку. Это действие приведет к появлению уязвимого места в безопасности, так как вызывающий пользователь может установить переменные окружения и управлять оболочкой. Самая частая форма такой атаки — внедрение пути, когда атакующий указывает переменной `PATH` заставить процесс выполнить функцию `exec()` и запустить любой двоичный код по выбору злоумышленника, в результате чего последний может запустить любую программу под учетными данными `set-group-ID` и `set-user-ID`.

Элементы семейства `exes`, принимающие в качестве аргумента массив, работают точно так же, за одним исключением — вместо списка генерируется и передается массив. Использование массива позволяет определять аргументы во время выполнения программы. Как и в случае аргумента в виде списка переменной длины, массив должен заканчиваться значением `NULL`.

В следующем примере `execv()` используется для запуска `vi`, как мы уже делали ранее:

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;

ret = execv ("/bin/vi", args);
if (ret == -1)
    perror ("execvp");
```

В Linux только один элемент семейства `exec` является системным вызовом. Остальные — это оболочки системного вызова в библиотеке `C`. Системные вызовы с переменным количеством аргументов сложны в реализации, а концепция пользовательского пути существует только в пользовательском пространстве, единственным элементом, способным осуществлять функции системного вызова, является `execve()`. Прототип системного вызова идентичен пользовательскому вызову.

Коды ошибок

В случае успешной работы системные вызовы `exec` не возвращают результатов. При неудаче вызов возвращает `-1` и присваивает `errno` одно из следующих значений:

- `E2BIG` — общее количество байтов в предоставленном списке аргументов (`arg`) или окружения (`envp`) слишком велико;
- `EACCES` — процесс не имеет доступа к компонентам пути, указанным в аргументе `path`; файл `path` является некорректным; файл назначения не является исполняемым; в файловой системе, содержащей `path` или `file`, активно свойство `noexec`;
- `EFAULT` — дан недопустимый указатель;
- `EIO` — низкоуровневая ошибка ввода-вывода (это плохо);
- `EISDIR` — конечный пункт пути `path`, или интерпретатор, является каталогом;
- `ELOOP` — путь `path` содержит слишком много символьных ссылок;
- `EMFILE` — вызывающий процесс достиг предела ограничения количества открытых файлов;
- `ENFILE` — система достигла предела ограничения количества открытых файлов;
- `ENOENT` — цели `path` или `file` не существует либо отсутствует необходимая общая библиотека;
- `ENOEXEC` — цель `path` или `file` является недопустимым двоичным файлом или же предназначена для другой машинной архитектуры;
- `ENOMEM` — доступных ресурсов ядра недостаточно для выполнения другой программы;
- `ENOTDIR` — какой-либо из компонентов пути, кроме конечного, не является каталогом;
- `EPERM` — в файловой системе, в которой находятся `path` или `file`, активно свойство `nosuid`, пользователь не имеет прав `root` либо для `path` или `file` установлен бит `suid` или `sgid`;
- `ETXTBSY` — назначение `path` или `file` открыто для записи другим пользователям.

Системные вызовы fork()

Новый процесс, запускающий тот же системный образ, что и текущий, может быть создан с помощью системного вызова `fork()`:

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork (void);
```

В случае успешного обращения к `fork()` создается новый процесс, во всех отношениях идентичный вызывающему. Оба процесса выполняются от точки обращения к `fork()`, как будто ничего не происходило.

Новый процесс является дочерним по отношению к вызывающему, который, в свою очередь, называется родительским. В дочернем процессе успешный запуск `fork()` возвращает 0. В родительском `fork()` возвращает `pid` дочернего. Родительский и дочерний процессы практически идентичны, за исключением некоторых особенностей:

- `pid` дочернего процесса, конечно, назначается заново и отличается от родительского;
- родительский `pid` дочернего процесса установлен равным `pid` родительского процесса;
- ресурсная статистика дочернего процесса обнуляется;
- любые ожидающие сигналы прерываются и не наследуются дочерним процессом (см. гл. 10);
- никакие вовлеченные блокировки файлов не наследуются дочерним процессом.

В случае ошибки дочерний процесс не создается, `fork()` возвращает -1, устанавливая соответствующее значение `errno`. Вот два возможных значения `errno` и их смысл:

- `EAGAIN` — ядро не способно выделить определенные ресурсы, например новый `pid`, или достигнуто ограничение по ресурсам `RLIMIT_NPROC` (см. гл. 6);
- `ENOMEM` — недостаточно ресурсов памяти ядра, чтобы завершить запрос.

Использование очень простое:

```
pid_t pid;

pid = fork ();
if (pid > 0)
    printf ("Я родительский процесс cpid=%d!\n", pid);
else if (!pid)
    printf ("А я дочерний!\n");
else if (pid == -1)
    perror ("fork");
```

Чаще всего системный вызов `fork()` используется для создания нового процесса и последующей загрузки в него нового двоичного образа. Представим себе оболочку,

в которой пользователь запускает новое приложение или процесс начинает вспомогательную программу. Сначала процесс ответвляет новый процесс, а потом дочерний процесс создает новый двоичный образ. Сочетание `fork` и `exec` используется часто и без осложнений. В следующем примере ответвляется новый процесс, запускающий двоичный файл `/bin/windlass`:

```
pid_t pid;

pid = fork ();
if (pid == -1)
    perror ("fork");
/* дочерний ... */
if (!pid) {
    const char *args[] = { "windlass", NULL };
    int ret;

    ret = execv ("/bin/windlass", args);
    if (ret == -1) {
        perror ("execv");
        exit (EXIT_FAILURE);
    }
}
```

Родительский процесс продолжает выполняться, как и ранее, за исключением появления у него нового дочернего. Вызов `execv()` влияет только на дочерний процесс, заставляя его выполнить программу `/bin/windlass`.

Копирование при записи

В ранних версиях UNIX ветвление было очень простым, если не сказать примитивным. После вызова ядро создавало копии всех структур внутренних данных, дублировало записи таблиц страницы процесса, а затем выполняло постраничное копирование родительского адресного пространства в дочернее. К сожалению, постраничное копирование — весьма затратный по времени процесс.

В современных системах UNIX подход усовершенствован. Вместо копирования всего объема родительского адресного пространства в современных системах UNIX, в том числе Linux, используются страницы *копирования при записи* (COW).

Копирование при записи — это откладывающая стратегия оптимизации, разработанная для уменьшения нагрузки из-за дублирования процессов. Принцип весьма прост: если несколько пользователей запрашивают доступ для чтения их собственных копий ресурса, нет смысла дублировать создаваемые копии. Вместо этого каждый потребитель может получить указатель к одному и тому же ресурсу. Пока пользователи не пытаются изменить свою «копию» ресурса, сохраняется иллюзия эксклюзивного доступа к ресурсу и затрат на копирование не требуется. Если пользователь пытается редактировать свою копию ресурса, то в этот момент ресурс прозрачно дублируется и копия отправляется редактирующему пользователю. Потребитель, не видя происходящего, может изменять свою копию ресурса, пока другие продолжают просматривать оригинальную,

неизмененную версию. Отсюда и название: *копирование* происходит только *при записи*.

Основная выгода здесь заключается в том, что, если пользователь не пытается модифицировать свою копию ресурса, она и не требуется. Общее преимущество откладывающих алгоритмов — откладывание наиболее затратных действий на последний момент — работает и здесь.

В определенном примере виртуальной памяти копирование при записи реализуется по постраничному принципу. Таким образом, поскольку процесс не затрагивает все адресное пространство целиком, копия его всего и не требуется. По окончании ветвления и родительский, и дочерний процессы «думают», что каждый из них имеет свое уникальное адресное пространство, в то время как на самом деле они делят доступ к оригинальным страницам родителя — которые, в свою очередь, могут предоставлять доступ к себе другим родительским или дочерним процессам, и т. д.

Реализация на уровне ядра проста. Страницы данных, относящиеся к структуре ядра, помечены как доступные только для чтения и копируемые только при записи. Когда какой-либо процесс пытается модифицировать страницу, на ней происходит ошибка, которую затем обрабатывает ядро, создавая видимую копию страницы. В этот момент для страницы атрибут «копирование при записи» для страницы удаляется и доступ к ней больше не делится. Поскольку архитектура современных компьютеров поддерживает копирование при записи на уровне аппаратного обеспечения, в элементах, управляющих памятью (MMUs), весь процесс достаточно прост и внедряется без проблем.

Копирование при записи обладает еще одним преимуществом в процессе ветвления. Поскольку после ветвления чаще всего выполняется *ехес*, копирование родительского адресного пространства в дочернее представляет собой пустую трату времени: ведь если дочерний процесс сразу же приступает к выполнению нового двоичного образа, его адресное пространство немедленно очищается. Копирование при записи избавляет и от этого неудобства.

vfork()

До внедрения страниц, поддерживающих копирование при записи, разработчики UNIX были вынуждены реализовывать бесполезное копирование адресного пространства в течение ветвления, сразу за которым следовало выполнение *ехес*. В связи с этим разработчики BSD представили в 3.0BSD системный вызов, который называется *vfork*:

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t vfork (void);
```

Успешное исполнение *vfork()* работает так же, как и *fork()*, кроме того, что дочерний процесс должен немедленно успешно вызвать одну из функций *ехес* или завершиться, вызвав *_exit()* (будет рассмотрен в следующем разделе). Система

`vfork()` не выполняет копирования адресного пространства и таблиц страниц, относящихся к родительскому процессу, пока дочерний не завершится или не выполнит новый двоичный образ. Между этим родительский и дочерний процессы совместно используют — без семантики копирования при записи — свое адресное пространство и страницы с табличными записями. Фактически единственная работа, выполняемая `vfork()`, — дублирование внутренней структуры данных ядра. Следовательно, дочерний процесс не должен модифицировать никаких данных в памяти адресного пространства.

Системный вызов `vfork()` является архаизмом и не должен реализовываться в Linux, хотя, надо отдать должное, даже с учетом копирования при записи `vfork()` работает быстрее `fork()`, так как отпадает необходимость в самом копировании страниц¹. В любом случае появление страниц с поддержкой копирования при записи, любые аргументы в пользу альтернатив `fork()` утрачивают силу. Действительно, до появления ядра Linux 2.2.0 `vfork()` был просто оболочкой для `fork()`. Требования для `vfork()` менее строгие, чем для `fork()`, поэтому данная реализация вполне осуществима.

Строго говоря, ни одно внедрение `vfork()` не застраховано от ошибок. Представьте, что вызов `exec` завершился сбоем: родительский процесс будет находиться в заблокированном состоянии, пока дочерний процесс не найдет способа разрешить эту ситуацию или не завершится. В программах лучше использовать `fork()` напрямую.

Завершение процесса

POSIX и C89 определяют следующую стандартную функцию для завершения текущего процесса:

```
#include <stdlib.h>
void exit (int status);
```

Вызов `exit()` выполняет некоторые основные шаги перед завершением, а затем отправляет ядру команду прекратить процесс. Эта функция не может вернуть ошибку — по сути, она вообще не возвращает никаких результатов. Следовательно, нет смысла в каких-либо других инструкциях по выполнению вызова `exit()`.

Параметр `status` используется для обозначения статуса процесса завершения. Другие программы — как и пользователь оболочки — могут проверять эту величину. В частности, статус `status & 0377` возвращается родительскому процессу. Далее в этой главе мы рассмотрим, как извлекать возвращаемое значение.

Значения `EXIT_SUCCESS` и `EXIT_FAILURE` определяются в качестве способов представления успеха и неудачи. В Linux значение 0 обычно означает успех, а любое ненулевое значение, например -1 или 1, соответствует неудаче.

¹ Хотя в настоящее время `vfork()` не является частью ядра Linux 2.6, исправление, реализующее общие записи таблиц страниц с копированием при записи, было выпущено в рассылке Linux Kernel Mailing List (lkml). Без этого исправления у системного вызова `vfork()` вообще не было бы никаких преимуществ.

Следовательно, успешный выход — единственная строка:

```
exit (EXIT_SUCCESS);
```

Перед тем как прервать процесс, библиотека C выполняет подготовительные шаги в следующем порядке.

1. Вызов всех функций, зарегистрированных с `atexit()` или `on_exit()`, в порядке обратном порядку регистрации (мы еще поговорим о них позже в этой главе).
2. Сброс всех стандартных потоков ввода-вывода (см. гл. 3)
3. Удаление всех временных файлов, созданных функцией `tmpfile()`.

Эти шаги завершают всю работу, которую процесс должен проделать в пользовательском пространстве, после чего `exit()` выполняет системный вызов `_exit()`, позволяющий ядру обработать оставшуюся часть завершения процесса:

```
#include <unistd.h>
```

```
void _exit (int status);
```

Когда процесс завершается, ядро очищает все ресурсы, которые были выделены для нужд процесса и более не используются. К ним относятся выделенная память, открытые файлы, семафоры System V и др. После очистки ядро уничтожает процесс и предупреждает родительский процесс о завершении дочернего.

Приложения могут вызывать `_exit()` напрямую, но зачастую это лишено смысла: большинству приложений необходимо провести некоторую зачистку перед полным выходом, например прервать поток `stdout`. Обратите внимание, однако, что при использовании `vfork()` после ветвления нужно вызывать `_exit()`, а не `exit()`.

Стремясь к чрезмерной точности, стандарт ISO C99 добавил функцию `_Exit()`, поведение которой полностью идентично `_exit()`:

```
#include <stdlib.h>
```

```
void _Exit (int status);
```

Другие способы завершения

Классический способ прекратить работу программы — не использование явного системного вызова, а простое «достижение конечной точки» программы. Для C или C++, например, это происходит, когда результат возвращает функция `main()`. Подход «достижение конечной точки» тем не менее все равно совершает системный вызов: компилятор просто вставляет неявный системный вызов `exit()` после завершения собственного кода. Очень полезно возвращать статус выхода явно, через `exit()` или возвращая какую-либо величину из `main()`. Оболочка использует величину `exit` для определения успешного либо неуспешного исполнения команд. Помните, что успешное значение — `exit(0)` или возвращение 0 из `main()`.

Процесс также может завершиться, если ему отправлен сигнал, действие которого по умолчанию — окончание процесса. К таким сигналам относятся `SIGTERM` и `SIGKILL` (см. гл. 10).

Последний способ прервать выполнение программы — срабатывание защитных функций ядра. Ядро может прервать процесс, выполняющий недопустимые инструкции, нарушающий сегментацию, исчерпавший ресурсы памяти и т. д.

atexit()

POSIX 1003.1-2001 определяет, а Linux поддерживает библиотечный вызов `atexit()`, используемый для регистрации функций, вызываемых при завершении процесса:

```
#include <stdlib.h>
```

```
int atexit (void (*function)(void));
```

При успешном срабатывании `atexit()` регистрирует указанную функцию для запуска при нормальном завершении процесса, то есть с помощью системного вызова `exit()` или возврата результатов функцией `main()`. Если процесс запускает функцию `exes`, список зарегистрированных функций очищается (поскольку функции больше не существуют в новом адресном пространстве процесса). Если процесс прерывается сигналом, зарегистрированные функции не вызываются.

Данная функция не требует каких-либо параметров и не возвращает значений. Прототип выглядит следующим образом:

```
void my_function (void);
```

Функции вызываются в порядке, обратном регистрации: это значит, что они хранятся в стеке и последняя вошедшая функция будет вызвана первой (LIFO). Зарегистрированные функции не должны вызывать `exit()` во избежание бесконечной рекурсии. Если функция должна окончить процесс завершения раньше, необходимо вызвать `_exit()`. Такой подход не рекомендуется, потому что затем возможен сбой запуска важных завершающих функций.

Стандарт POSIX требует поддержки `atexit()` по крайней мере для `ATEXIT_MAX` зарегистрированных функций, следовательно, эта величина должна быть равна по меньшей мере 32. Точный максимум может быть определен через `sysconf()` и величину `_SC_ATEXIT_MAX`:

```
long atexit_max;
```

```
atexit_max = sysconf (_SC_ATEXIT_MAX);  
printf ("atexit_max=%ld\n", atexit_max);
```

В случае успеха `atexit()` возвращает 0, при ошибке она возвращает значение -1.

Вот простой пример:

```
#include <stdio.h>  
#include <stdlib.h>
```

```
void out (void)  
{  
    printf ("atexit() succeeded!\n");
```



```
}

int main (void)
{
    if (atexit (out))
        fprintf(stderr, "atexit() failed!\n");
    return 0;
}
```

on_exit()

SunOS 4 определяет собственный эквивалент `atexit()`, библиотека `glibc` в Linux поддерживает его:

```
#include <stdlib.h>
```

```
int on_exit (void (*function)(int, void *), void *arg);
```

Эта функция работает так же, как `atexit()`, но прототип зарегистрированной функции несколько отличается:

```
void my_function (int status, void *arg);
```

Аргумент `status` — это величина, переданная `exit()` или возвращенная `main()`. Аргумент `arg` — второй параметр, переданный `exit()`. Следует удостовериться, что данные, содержащиеся в памяти `arg`, являются допустимыми на момент вызова функции.

Последняя версия Solaris уже не поддерживает эту функцию. Вместо нее необходимо использовать стандартно компилируемую `atexit()`.

SIGCHLD

Когда процесс завершается, ядро посылает сигнал `SIGCHLD` родительскому процессу. По умолчанию этот сигнал игнорируется и родительский процесс не предпринимает каких-либо действий. Однако при необходимости процессы могут обработать данный сигнал с помощью системных вызовов `signal()` или `sigaction()`. Эти вызовы, как и остальные представители волшебного мира сигналов, описаны в гл. 10.

Сигнал `SIGCHLD` может быть сгенерирован и отправлен в любое время, так как завершение дочернего процесса не синхронно родительскому. Однако часто предку требуются сведения о завершении его потомка или даже некоторое время для ожидания события. Это возможно с помощью системных вызовов, описанных далее.

Ожидание завершенных дочерних процессов

Получение предупреждения на сигнал — это прекрасно, но многие родительские процессы ходят получить больше информации, когда завершается их дочерний процесс — например, если тот возвращает какое-либо значение.

Если бы потомок по завершении полностью исчезал, как можно было бы предположить, то получение каких-либо сведений было бы для его предка невозможно. Следовательно, первые разработчики UNIX решили, что когда дочерний процесс завершается прежде родительского, ядро должно поместить потомка в особый процессный статус. Процесс в этом состоянии известен как *зомби*. В данном состоянии существует лишь «скелет» процесса — некоторые основные структуры данных, содержащие потенциально нужные сведения. Процесс в таком состоянии ожидает запроса о своем статусе от предка (процедура, известная как *ожидание процесса-зомби*). Только после того как предок получит всю необходимую информацию о завершенном дочернем процессе, последний формально удаляется и перестает существовать даже в статусе зомби.

Ядро Linux предоставляет несколько интерфейсов для получения информации о завершенном дочернем процессе. Самый простой из них, определенный POSIX, называется `wait()`:

```
#include<sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

Вызов `wait()` возвращает `pid` завершенного дочернего процесса или `-1` в случае ошибки. Если никакого дочернего процесса не было прервано, вызов блокируется, пока потомок не завершится. Если дочерний процесс уже был завершен, вызов возвращает результаты немедленно. Следовательно, если вызвать `wait()` сразу после сообщения о завершении дочернего процесса, результат будет немедленно выдан без блокировки.

В случае ошибки возможно присвоение переменной `errno` одного из двух значений:

- `ECHILD` — вызывающий процесс не имеет дочерних;
- `EINTR` — сигнал был получен во время ожидания, в результате чего вызов вернул результат слишком рано.

Если указатель `status` не содержит значения `NULL`, там находится дополнительная информация о дочернем процессе. POSIX позволяет при реализации определение битов статуса разработчиками самостоятельно, поэтому стандарт предусматривает семейство макросов для интерпретации параметра:

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

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);

int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Любой из первых двух макросов может возвращать значение `true` (ненулевое) в зависимости от хода завершения процесса. Первый, `WIFEXITED`, возвращает `true`, если процесс завершается через вызов `_exit()`, обычным образом. Соответственно, макрос `WEXITSTATUS` предоставляет 8 бит младших разрядов и передает их `_exit()`.

Макрос `WIFSIGNALED` возвращает `true`, если прерывание процесса вызвал сигнал (подробнее сигналы рассмотрены в гл. 10). В этом случае `WTERMSIG` возвращает номер сигнала, который вызвал прерывание, а `WCOREDUMP` возвращает `true`, если процесс сбросил ядро в ответ на получение сигнала. `WCOREDUMP` не определяется POSIX, хотя многие системы UNIX, Linux в том числе, поддерживают его.

Макросы `WIFSTOPPED` и `WIFCONTINUED` возвращают `true`, если процесс был остановлен или продолжен соответственно и его можно в настоящий момент отследить с помощью системного вызова `ptrace()`. Эти условия обычно возможны только во время реализаций отладчика, хотя при использовании вместе с `waitpid()` (см. следующий раздел) они могут использоваться для контроля работы. Обычно `wait()` применяется только для обмена информацией о завершении процесса. Если `WIFSTOPPED` возвратил `true`, `WSTOPSIG` приводит номер сигнала, остановившего процесс. `WIFCONTINUED` не определяется POSIX, хотя более поздние стандарты определили его для `waitpid()`. В версии 2.6.10 ядра Linux также предоставлен макрос для `wait()`.

Рассмотрим пример программы, которая использует `wait()`, чтобы определить, что произошло с дочерним процессом:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (void)
{
    int status;
    pid_t pid;

    if (!fork ())
        return 1;

    pid = wait (&status);
    if (pid == -1)
        perror ("wait");

    printf ("pid=%d\n", pid);

    if (WIFEXITED (status))
        printf ("Нормальное завершение, статус=%d\n",
                WEXITSTATUS (status));

    if (WIFSIGNALED (status))
        printf ("Убит сигналом=%d\n",
                WTERMSIG (status),
```

```

        WCOREDUMP (status) ? " (dumped core)" : "");

    if (WIFSTOPPED (status))
        printf ("Остановлен сигналом=%d\n",
                WSTOPSIG (status));

    if (WIFCONTINUED (status))
        printf ("Продолжен\n");

    return 0;
}

```

Эта программа отвечает дочерний процесс, который немедленно завершается. После этого предок запускает системный вызов `wait()` для определения статуса потомка. Процесс печатает `pid` потомка и сведения о его завершении. Поскольку в этом случае дочерний процесс завершился возвратом результата `main()`, понятно, что на выходе мы получим примерно следующее:

```

$ ./wait
pid=8529
Нормальное завершение со статусом выхода=1

```

Если вместо обычного возврата дочернего процесса реализован вызов `abort()`¹, который отправляет процессу сигнал `SIGABRT`, то результат будет выглядеть так:

```

$ ./wait
pid=8678
Прерван сигналом=6

```

Ожидание определенного процесса

Наблюдение за поведением дочернего процесса очень важно. Часто, однако, процесс имеет нескольких потомков и, если нужен только определенный, ожидание всех нежелательно. Возможным решением были бы многократные вызовы `wait()` с постоянной проверкой возвращаемого значения, однако это весьма неудобно: что, если позже понадобится проверить статус другого завершенного процесса? Родительскому процессу пришлось бы сохранять результаты всех вызовов `wait()` на случай, если они понадобятся позднее.

Если вам известен `pid` процесса, завершения которого вы ждете, можно использовать системный вызов `waitpid()`:

```

#include <sys/types.h>
#include <sys/wait.h>

```

```
pid_t waitpid (pid_t pid, int *status, int options);
```

Вызов `waitpid()` — более мощная версия `wait()`. Его дополнительные параметры позволяют настроить его более тонко.

¹ Определяется в заголовке `<stdlib.h>`.

Параметр `pid` точно определяет, какой процесс или процессы нужно ожидать. Его значения могут попадать в четыре промежутка:

- `< -1` — ожидание любого дочернего процесса, чей ID группы процессов равен абсолютному значению этой величины; например, при `-500` ожидается любой процесс из группы процессов `500`;
- `-1` — ожидание любого дочернего процесса; поведение аналогично `wait()`;
- `0` — ожидание любого дочернего процесса, принадлежащего той же группе процессов, что и вызывающий;
- `> 0` — ожидание любого дочернего процесса, чей `pid` в точности равен указанной величине; например, при величине `500` ожидается дочерний процесс с `pid`, равным `500`.

Параметр `status` работает аналогично таковому в системном вызове `wait()` и может быть обработан с помощью макросов, описанных выше.

Параметр `options` может передавать следующие значения с помощью логического «ИЛИ» либо пустое значение:

- `WNOHANG` — не блокировать вызов, немедленно вернуть результат, если ни один подходящий процесс еще не завершился (остановился или продолжился);
- `WUNTRACED` — при его выборе устанавливается параметр `WIFSTOPPED`, даже если вызывающий процесс не отслеживает свой дочерний; это свойство помогает реализовать более общее управление заданиями, как это сделано в оболочке;
- `WCONTINUED` — если установлен, то бит `WIFCONTINUED` в возвращаемом параметре статуса будет установлен, даже если вызывающий процесс не отслеживает свой дочерний; как и в случае с `WUNTRACED`, параметр полезен для реализации оболочки.

В случае успеха `waitpid()` возвращает `pid` процесса, статус которого изменился. Если установлен `WNOHANG`, а указанный дочерний процесс (один или несколько) не изменил свой статус, `waitpid()` вернет `0`. В случае ошибки вызов возвращает `-1`, а `errno` принимает одно из следующих трех значений:

- `ECHILD` — процесс или процессы, указанные с помощью аргумента `pid`, не существуют или не являются потомками вызывающего;
- `EINTR` — параметр `WNOHANG` не был установлен, а сигнал был получен во время ожидания;
- `EINVAL` — аргумент `options` указан некорректно.

Рассмотрим пример. Программа должна получить значение, возвращаемое дочерним процессом с `pid` `1742`, причем сделать это немедленно, если дочерний процесс еще не завершился. Можно реализовать это следующим образом:

```
int status;
pid_t pid;

pid = waitpid (1742, &status, WNOHANG);

if (pid == -1)
```

```

        perror ("waitpid");
else {
    printf ("pid=%d\n", pid);

    if (WIFEXITED (status))
        printf ("Обычное завершение со статусом выхода=%d\n",
                WEXITSTATUS (status));

    if (WIFSIGNALED (status))
        printf ("Убит сигналом=%d%s\n",
                WTERMSIG (status),
                WCOREDUMP (status) ? " (дамп ядра)" : "");
}

```

Последний пример. Обратите внимание на следующее использование `wait()`:

```
wait (&status);
```

Это полностью аналогично:

```
waitpid (-1, &status, 0);
```

Еще больше гибкости при ожидании

Для приложений, которые требуют еще большей гибкости в их функциональности ожидания дочерних процессов, расширение XSI стандарта POSIX определяет, а Linux поддерживает системный вызов `waitid()`:

```

#include <sys/wait.h>

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);

```

Как и `wait()` и `waitpid()`, системный вызов `waitid()` используется для ожидания и получения информации об измененном статусе (завершение, остановка, продолжение) дочернего процесса. Он предоставляет еще больше параметров, но их использование несколько сложнее.

Аналогично `waitpid()` с помощью `waitid()` разработчик может выбрать ожидаемый процесс. Однако `waitid()` требует для этого указания не одного, а двух параметров. С помощью аргументов `idtype` и `id` определяется, какой дочерний процесс нужно ожидать (аналогично использованию одного аргумента `pid` в `waitpid()`). Значения `idtype` могут быть следующими:

- `P_PID` — ожидание дочернего процесса, `pid` которого совпадает со значением аргумента `id`;
- `P_GID` — ожидание дочернего процесса, идентификатор группы которого совпадает со значением `id`;
- `P_ALL` — ожидание всех дочерних процессов, `id` игнорируется.

Аргумент `id` принадлежит к достаточно редкому типу аргументов `id_t`, представляющему общий идентификационный номер. Его используют, если в будущем планируется ввести новое значение `idtype`, обеспечив таким образом гарантию, что в заданном типе можно будет хранить новый идентификатор (его размер достаточен для помещения любого значения `pid_t`). Разработчики Linux могут применять его аналогично типу `pid_t`, напрямую передавая значение `pid_t` или числовых констант. Педантичные программисты, однако, могут и преобразовать типы.

В параметре `options` могут быть указаны одно или несколько из следующих значений с помощью двоичного «ИЛИ»:

- `WEXITED` — вызов будет ждать дочерних процессов (определенных с помощью `id` или `idtype`), которые завершились;
- `WSTOPPED` — вызов будет ожидать дочерних процессов, которые остановили выполнение в ответ на получение сигнала;
- `WCONTINUED` — вызов будет ожидать дочерних процессов, которые продолжили выполнение в ответ на получение сигнала;
- `WNOHANG` — вызов не может быть заблокирован и вернет результаты немедленно, если ни один из указанных дочерних процессов еще не завершен (или остановлен, или продолжен);
- `WNOWAIT` — вызов не будет выводить указанный процесс из статуса зомби; этот процесс будет обработан в будущем.

В случае успеха `wiatid()` возвращает параметр `infor`, который укажет на допустимый тип `siginfo_t`. Точная структура `siginfo_t` зависит от реализации¹, но после выполнения `waitid()` заполненными остаются лишь несколько полей. Таким образом, при успешном вызове допустимые значения будут содержаться в следующих полях:

- `si_pid` — `pid` дочернего процесса;
- `si_uid` — `uid` дочернего процесса;
- `si_code` — может принимать значения `CLD_EXITED`, `CLD_KILLED`, `CLD_STOPPED` или `CLD_CONTINUED` в результате завершения дочернего процесса, окончания или продолжения его по сигналу соответственно;
- `si_signo` — устанавливается значение `SIGCHLD`;
- `si_status` — если `si_code` установился равным `CLD_EXITED`, это поле содержит код выхода дочернего процесса, и наоборот, это поле принимает значение номера сигнала, отправленного дочернему процессу и вызвавшего изменения.

В случае успеха `waitid()` возвращает 0. При ошибке `waitid()` возвращает -1, а `errno` принимает одно из следующих значений:

- `ECHLD` — процесс или процессы, указанные с помощью `id` или `idtype`, не существуют;

¹ В Linux структура `siginfo_t` очень сложна. Ее определение можно найти в файле `/usr/include/bits/siginfo.h`. Подробнее об этом будет рассказано в гл. 10.

- EINTR — WNOHANG не был установлен в options, а сигнал прервал выполнение;
- EINVAL — аргумент options или комбинация аргументов id и idtype некорректны.

Функция `waited()` предоставляет дополнительную полезную семантику, отсутствующую в `wait()` и `waitpid()`. В частности, информация, предоставляемая в структуре `siginfo_t`, может быть очень полезной. Если эти сведения не требуются, имеет смысл использовать более простые функции, которые поддерживаются большим количеством систем и, следовательно, могут быть перенесены в другие системы, не относящиеся к Linux.

На сцену выходит BSD: `wait3()` и `wait4()`

Вызов `waitpid()` является наследником System V Release 4 системы AT&T, а BSD идет собственным путем и предоставляет две другие функции, которые используются для ожидания изменения статуса дочернего процесса:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3 (int *status,
             int options,
             struct rusage *rusage);
pid_t wait4 (pid_t pid,
             int *status,
             int options,
             struct rusage *rusage);
```

Цифры 3 и 4 указывают, что эти две функции являются версиями `wait()` соответственно с тремя и четырьмя параметрами.

Функции работают схожим образом с `waitpid()`, за исключением аргумента `rusage`. Такой запуск `wait3()`:

```
pid = wait3 (status, options, NULL);
```

эквивалентен следующему вызову `waitpid()`:

```
pid = waitpid (-1, status, options);
```

в то время как данный запуск `wait4()`:

```
pid = wait4 (pid, status, options, NULL);
```

эквивалентен такому вызову `waitpid()`:

```
pid = waitpid (pid, status, options);
```

Таким образом, `wait3()` ожидает любого дочернего процесса с изменившимся статусом, а `wait4()` ожидает определенного дочернего процесса, указанного параметром `pid` и изменившего статус. Аргумент `options` работает так же, как и в `waitpid()`.

Как упоминалось выше, самое большое различие между этими вызовами и `waitpid()` — параметр `rusage`. Если его значение не `NULL`, функция заполняет указатель, прописанный в `rusage`, информацией о дочернем процессе. Эта структура предоставляет информацию об использовании ресурсов дочерним процессом:

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

struct rusage {
    struct timeval ru_utime; /* затраченное пользовательское время */
    struct timeval ru_stime; /* затраченное системное время */
    long ru_maxrss; /* максимальный размер резидентной части */
    long ru_ixrss; /* размер общей памяти */
    long ru_idrss; /* размер собственных данных */
    long ru_isrss; /* размер собственного стека */
    long ru_minflt; /* восстановления страниц */
    long ru_majflt; /* страничные прерывания */
    long ru_nswap; /* операции подкачки */
    long ru_inblock; /* блочные операции ввода */
    long ru_oublock; /* блочные операции вывода */
    long ru_msgsnd; /* отправленные сообщения */
    long ru_msgrcv; /* полученные сообщения */
    long ru_nsignals; /* полученные сигналы */
    long ru_nvcsw; /* добровольные переключения контекста */
    long ru_nivcsw; /* вынужденные переключения контекста */
};
```

Об использовании ресурсов я еще расскажу далее в этой главе.

В случае успеха эти функции возвращают `pid` процесса, изменившего статус. В случае неудачи они возвращают `-1` и устанавливают в `errno` те же значения ошибок, что описаны для `waitpid()`.

Поскольку `wait3()` и `wait4()` не определены в стандарте POSIX¹, желательно применять их, только если получение информации об использовании ресурсов чрезвычайно важно. Однако, несмотря на отсутствие стандартизации POSIX, почти все системы UNIX поддерживают эти два вызова.

Запуск и ожидание нового процесса

В стандартах ANSI C и POSIX определяется интерфейс, объединяющий запуск нового процесса и ожидание его завершения — представьте себе это как синхронное создание процессов. Если процесс запускает дочерний, только чтобы немедленно начать ожидать его завершения, лучше всего использовать следующий интерфейс:

```
#define _XOPEN_SOURCE /* если нужен WEXITSTATUS и т. д. */
#include <stdlib.h>

int system (const char *command);
```

¹ Вызов `wait3()` входил в первоначальную спецификацию Single UNIX Specification, но был впоследствии удален из нее.

Функция `system()` называется так потому, что синхронный запуск процессов называется *выходом в систему*. Принято использовать `system()` для запуска простой утилиты или сценария оболочки, когда основной целью является получение возвращаемого ими значения.

Вызов `system()` запускает команду, предоставляемую параметром `command`, включая любые дополнительные аргументы. Параметр `command` добавляется к аргументам `/bin/sh -c`. В этом смысле параметр передается всей оболочке.

В случае успеха возвращаемой величиной является статус, который вернула команда и предоставил вызов `wait()`. Таким образом, код выхода выполненной команды получается с помощью `WEXIT STATUS`. Если вызов `/bin/sh` потерпел неудачу, величина, передаваемая `WEXITSTATUS`, та же, что возвращается `exit(127)`. Поскольку вызываемая команда также может вернуть 127, безошибочного метода определения источника ошибки — командой она вызвана или оболочкой — не существует. В случае ошибки вызов возвращает значение -1.

Если параметр `command` имеет значение `NULL`, `system()` возвращает ненулевую величину, если `shell /bin/sh` доступен, и 0 в противном случае.

Во время выполнения команды `SIGCHLD` блокируется, а `SIGINT` и `SIGQUIT` игнорируются. Игнорирование `SIGINT` и `SIGQUIT` имеет различные последствия, в частности, когда `system()` вызывается внутри цикла. В этом случае нужно убедиться, что программа правильно проверяет статус выхода дочернего процесса, например:

```
do {
    int ret;

    ret = system ("pidof rudder");
    if (WIFSIGNALED (ret) &&
        (WTERMSIG (ret) == SIGINT ||
         WTERMSIG (ret) == SIGQUIT))
        break; /* или другой вариант обработки */
} while (1);
```

Реализация `system()` с использованием `fork()`, функции из семейства `exec`, и `waitpid()` — очень полезное упражнение. Вы должны попытаться выполнить его самостоятельно, поскольку для этого потребуется большая часть материала данной главы. Для завершенности приведу пример простой реализации:

```
/*
 * my_system — синхронно ответвляет дочерний процесс и ожидает команды
 * "/bin/sh -c<cmd>".
 *
 * Возвращает -1 в случае любой ошибки или код выхода запущенного процесса
 * Не блокирует и не игнорирует сигналы
 */
int my_system (const char *cmd)
{
    int status;
    pid_t pid;

    pid = fork ();
```

```
    if (pid == -1)
        return -1;
    else if (pid == 0) {
        const char *argv[4];
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = cmd;
        argv[3] = NULL;
        execv ("/bin/sh", argv);

        exit (-1);
    }

    if (waitpid (pid, &status, 0) == -1)
        return -1;
    else if (WIFEXITED (status))
        return WEXITSTATUS (status);

    return -1;
}
```

Обратите внимание, что в этом примере не блокируются и не отключаются никакие сигналы, в отличие от официальной версии `system()`. Этот подход может быть лучше или хуже, в зависимости от конкретной программы, но не блокировать по крайней мере `SIGINT` часто может быть полезно, потому что это позволяет прерывать запущенную программу так, как привычно пользователю в обычных ситуациях. Усложнить задачу можно, добавив дополнительные указатели в виде параметров, которые указывали бы на ошибки, в настоящее время неразличимые, когда их значение отлично от `NULL`. В качестве примера можно добавить `fork_failed` и `shell_failed`.

ВНИМАНИЕ

Системному вызову `system()` свойственны те же проблемы с безопасностью, что и `execp()` и `execvp()`. Нельзя запускать `system()` с установленного идентификатора группы (`set-group-ID`) или пользователя (`set-user-ID`) программы, так как злоумышленник может манипулировать настройками окружения (чаще всего `PATH`), чтобы расширить свои привилегии. Рукописные замены `system()` также уязвимы, так как используют оболочку.

Во избежание атак такого типа программы установки идентификаторов пользователя или группы должны запускать желаемый внешний бинарный модуль через `fork()` и `exec()` без использования оболочки. Отказ от запуска внешнего бинарного модуля является даже лучшим решением!

Зомби

Как было сказано выше, процесс, который прервался, но чей предок пока не ожидает его завершения, называется зомби. Процессы-зомби продолжают занимать системные ресурсы, пусть и в небольших количествах, ведь требуется поддерживать лишь «скелет» работавшего ранее процесса. Эти ресурсы нужны, чтобы родительские процессы, желающие проверить статус своих потомков, получили информацию,

относящуюся к существованию и прекращению данных процессов. Как только родительский процесс сделает это, ядро окончательно удаляет процесс и зомби отправляется на покой.

Однако каждый, кто какое-то время использовал Linux, время от времени натывается на процессы-зомби. Эти процессы, часто называемые *призраками*, — дети безответственных родителей. Если ваше приложение отвечает дочерний процесс, то оно несет ответственность за его обслуживание, даже если это заключается лишь в удалении собранной информации (кроме случаев, когда приложение кратковременно, о чем мы скоро поговорим). Иначе все дочерние процессы будут становиться призраками и существовать в списках процессов системы, что никак не украшает приложение.

Что происходит, если родительский процесс умирает раньше дочернего или если он завершается до того, как получит возможность позаботиться о своих потомках-зомби? Когда бы ни завершился процесс, ядро Linux проходит по списку его потомков и *переназначает* родительский процесс, делая их предком процесс инициализации, `pid` которого равен 1. Таким образом, ни один процесс не станет сиротой без родительского. Процесс инициализации, в свою очередь, периодически заботится обо всех своих потомках, в результате чего ни один из них не остается зомби чрезмерно долго (никаких призраков). Следовательно, если родительский процесс завершается раньше своих потомков или не обслуживает их перед завершением, дочерние процессы переходят под опеку процесса инициализации и обслуживаются им до полного завершения. Хотя подобная реализация считается хорошей практикой, данная предосторожность означает, что краткосрочные процессы могут особенно не беспокоиться об ожидании всех своих потомков.

Пользователи и группы

Как упоминалось ранее в этой главе и обсуждалось в гл. 1, процессы принадлежат определенным пользователям и группам. Идентификаторы группы и процесса — численные величины, представленные типами `Cuid_t` и `gid_t` соответственно. Связь между численными величинами и именами, удобными для восприятия человеком, — например, у пользователя `root` значение `uid` равно 0 — осуществляется в пользовательском пространстве с помощью файлов `/etc/passwd` и `/etc/group`. Ядро может работать только с численными величинами.

В системе Linux идентификаторы пользователя и группы какого-либо процесса определяют операции, доступные для выполнения данным процессом. Следовательно, процессы исполняются от имени определенных пользователей и групп. Много процессов может быть запущено только от имени пользователя `root`. Однако при разработке программного обеспечения лучше всего следовать доктрине *наименьших прав*, что означает: процесс должен работать с минимальным из возможных уровней прав. Это требование динамично: если процессу для выполнения требуются права `root` для выполнения какой-либо операции на начальном этапе существования, а после этого необходимости в расширенных правах нет, он должен как можно

скорее избавиться от прав root, поэтому большинство процессов — в частности, как раз те, которым для выполнения определенных операций требуются права доступа root, — часто манипулируют своими идентификаторами пользователя или группы.

Перед тем как мы посмотрим, как это происходит, нужно разобраться в особенностях идентификаторов пользователя и группы.

Реальные, действительные и сохраненные идентификаторы пользователя и группы

ПРИМЕЧАНИЕ

В тексте ниже обсуждаются в основном идентификаторы пользователя, но все сказанное применимо также к идентификаторам групп.

На самом деле существует не один, а четыре пользовательских идентификатора, ассоциированных с процессом: реальный, действительный, сохраненный и идентификатор файловой системы. *Реальный идентификатор пользователя* — это uid пользователя, который изначально запустил процесс. Он устанавливается по реальному идентификатору пользователя родительского процесса и не изменяется в течение работы вызова exes. Обычно при авторизации устанавливается реальный идентификатор пользователя, и все процессы пользователя продолжают работу с этим идентификатором. Пользователь с правами доступа root может менять реальный идентификатор пользователя по мере надобности, но другие пользователи не могут этого делать.

Действительный идентификатор пользователя — это идентификатор, под которым в настоящий момент выполняется процесс. Проверка доступа обычно основывается на этом значении. Изначально этот идентификатор равен реальному идентификатору пользователя, поскольку, когда процесс начинает ветвление, действительный идентификатор пользователя передается от родительского процесса к дочернему. Далее, когда процесс сталкивается с вызовом exes, действительный пользователь обычно не меняется. Однако именно во время работы вызова exes обнаруживается ключевое различие между реальным и действительным идентификаторами: процесс может изменить свой действительный пользовательский идентификатор с помощью запуска бинарного модуля `setuid (suid)`. Точнее, действительный идентификатор пользователя меняется на идентификатор пользователя, который является владельцем программы. Например, так как файл `/usr/bin/passwd` является `setuid`, а его владелец — пользователь root, когда оболочка обычного пользователя отвечает процессу, выполняющий файл, процессу присваивается действительный идентификатор пользователя root, независимо от того, какой пользователь выполняет его фактически.

Непривилегированные пользователи могут устанавливать в качестве действительного в реальный или сохраненный пользовательский идентификатор. Пользователь с правами root может присваивать действительному идентификатору любое значение.

Сохраненным идентификатором пользователя называется изначальный действительный пользовательский идентификатор. Когда процесс начинает ветвление, потомок наследует сохраненный пользовательский идентификатор родителя. Во время вызова `exes`, однако, ядро устанавливает в качестве сохраненного действительный идентификатор пользователя, таким образом сохраняя информацию о действительном идентификаторе на момент запуска `exes`. Непривилегированные пользователи не могут менять свой сохраненный идентификатор; пользователи с правами `root` могут присвоить ему значение реального идентификатора пользователя.

Для чего это нужно? Важнее всего — действительный идентификатор пользователя. Именно эта величина проверяется во время валидации доступа процесса. Реальный и сохраненный идентификаторы пользователя играют роль суррогатов или потенциальных значений идентификатора пользователя, которые могут использовать процессы, не обладающие правами `root`. Реальный идентификатор пользователя — это действительный идентификатор, принадлежащий пользователю, фактически выполняющему программу, а сохраненный — это действительный идентификатор до изменения им во время вызова `exes` двоичным файлом `suid`.

Изменение реального или сохраненного идентификатора пользователя или группы

Идентификаторы пользователя или группы устанавливаются с помощью двух системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>
```

```
int setuid (uid_t uid);
int setgid (gid_t gid);
```

Вызов `setuid()` устанавливает действительный идентификатор пользователя текущего процесса. Если текущий действительный идентификатор пользователя данного процесса равен 0 (`root`), устанавливаются также реальный и сохраненный идентификаторы. Пользователь `root` сможет передавать в качестве параметра `uid` любые значения, устанавливая всем трем значениям пользовательского идентификатора величину `uid`. Пользователь без прав `root` может изменять только величину реального или сохраненного пользовательского идентификатора. Другими словами, пользователь без прав `root` может установить действительному идентификатору пользователя лишь одно из этих значений.

В случае успеха `setuid()` возвращает 0. При ошибке вызов возвращает -1, а `errno` присваивает одно из следующих значений:

- `EAGAIN` — `uid` отличается от реального пользовательского идентификатора, и установление `uid` в качестве реального идентификатора пользователя выведет его за пределы `RLIM_NPROC` (определяющего количество процессов, которыми может владеть пользователь);

- EPERM — пользователь не обладает правами root, а uid не является ни действительным, ни сохраненным идентификатором пользователя.

Изменение действительного идентификатора пользователя или группы

Linux предоставляет две функции, утвержденные POSIX, с помощью которых можно установить действительный идентификатор пользователя или группы:

```
#include <sys/types.h>
#include <unistd.h>
```

```
int seteuid (uid_t euid);
int setegid (gid_t egid);
```

Вызов `seteuid()` устанавливает действительный идентификатор пользователя равным `euid`. Пользователь `root` может установить в качестве `euid` любую величину. Пользователи без прав `root` могут установить действительный идентификатор пользователя только равным реальному или сохраненному идентификатору пользователя. В случае успеха `setuid()` возвращает 0. При неудаче он возвращает -1 и устанавливает `errno` значение `EPERM`, указывающее, что владелец текущего процесса не имеет прав `root`, и этот `euid` не равен ни сохраненному, ни реальному пользовательскому идентификатору.

Обратите внимание, что в случае отсутствия прав `root` функции `seteuid()` и `setuid()` работают одинаково. Таким образом, стандартная практика — и хорошая идея — всегда использовать `seteuid()`, если ваш процесс не будет выполняться от имени пользователя `root`, когда применять `setuid()` выгоднее.

Все сказанное выше касается и групп, нужно только заменить `seteuid()` на `setegid()`, а `euid` на `egid`.

Изменение идентификаторов пользователя и группы согласно стилю BSD

Разработчики BSD пришли к собственным интерфейсам для изменения идентификаторов пользователя и группы. В Linux эти интерфейсы предоставляются для обеспечения совместимости:

```
#include <sys/types.h>
#include <unistd.h>
```

```
int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

Вызов `setreuid()` устанавливает реальные и действительные идентификаторы процесса равными `ruid` и `euid` соответственно. Передача величины -1 для любого из параметров не изменит соответствующий идентификатор пользователя. Процессы без прав `root` могут только устанавливать значение действительного идентификатора

пользователя реальному или сохраненному идентификатору, а реального — действительному. Если реальный пользовательский идентификатор изменился или действительный пользовательский идентификатор изменился, став величиной, не равной предыдущему реальному идентификатору, сохраненный идентификатор пользователя изменяется на новый действительный идентификатор. Во всяком случае, именно так Linux и большинство систем UNIX реагируют на подобные изменения; данный подход не освещается в стандарте POSIX.

В случае успеха `setreuid()` возвращает 0. При неудаче возвращает -1 и присваивает `errno` значение `EPERM`. Это означает, что текущий процесс не имеет прав `root`, а этот `euid` не равен ни реальному, ни сохраненному идентификатору пользователя или этот `ruid` не равен эффективному идентификатору пользователя.

Все сказанное выше касается и групп, нужно только заменить `setreuid()` на `setregid()`, `ruid` на `rgid`, а `euid` на `egid`.

Изменение идентификаторов пользователя и группы согласно стилю HP-UX

Вы можете подумать, что это уже слишком, но тем не менее HP-UX, система UNIX от Hewlett-Packard, тоже представила собственные механизмы для установки идентификаторов группы и пользователя какого-либо процесса. Linux следует за ними и предоставляет следующие интерфейсы, которые полезны, если вам требуется совместимость с HP-UX:

```
#define _GNU_SOURCE
#include <unistd.h>
```

```
int setresuid (uid_t ruid, uid_t euid, uid_t suid);
int setresgid (gid_t rgid, gid_t egid, gid_t sgid);
```

Вызов `setresuid()` устанавливает реальный, действительный и сохраненный идентификаторы пользователя равными `ruid`, `euid`, и `suid` соответственно. Указание значения -1 для любого из этих параметров оставит его неизменным.

Пользователь `root` может присвоить любое значение любому из идентификаторов пользователя. Пользователи без прав `root` могут установить любой идентификатор равным текущим реальному, действительному или сохраненному идентификатору. В случае успеха `setuid()` возвращает 0. При ошибке вызов возвращает -1 и `errno` присваивается одно из следующих значений:

- `EAGAIN` — `uid` не совпадает с реальным ID пользователя, и установка реального идентификатора равным `uid` выведет пользователя за пределы `RLIM_NPROC` (определяющего количество процессов, которыми может владеть пользователь);
- `EPERM` — пользователь не имеет прав `root`, а набор новых величин для реального, действительного или сохраненного идентификатора не совпадает ни с одним из значений реального, действительного или сохраненного пользовательских ID.

Все вышесказанное применимо и к группам, просто замените `setresuid()` на `setresgid()`, `ruid` на `rgid`, `euid` на `egid` и `suid` на `sgid`.

Действия с предпочтительными идентификаторами пользователя или группы

Процессы без прав `root` должны использовать `setuid()` для изменения своих действительных идентификаторов пользователя. Процессы с правами `root` должны применять `setuid()`, если они хотят изменить все три пользовательских идентификатора, и `seteuid()`, если нужно временно изменить только действительный идентификатор пользователя. Эти функции просты и работают в соответствии с POSIX, принимая во внимание сохраненный идентификатор пользователя.

Несмотря на предоставление дополнительных функциональностей, функции BSD и HP-UX не позволяют вносить полезные изменения, в отличие от `setuid()` и `seteuid()`.

Поддержка сохраненных пользовательских идентификаторов

Существование сохраненных идентификаторов пользователя и группы регулируется IEEE Std 1003.1-2001 (POSIX 2001), и Linux поддержала эти идентификаторы с момента появления ядра 1.1.38. В программах, написанных только для Linux, существование сохраненных идентификаторов пользователя предусмотрено изначально и они всегда будут на месте. В программах, которые были написаны для более старых систем UNIX, необходимо проверять наличие макроса `_POSIX_SAVED_IDS` и лишь затем ссылаться на сохраненный идентификатор пользователя или группы.

Если сохраненные ID пользователя или группы отсутствуют, вышеизложенное все равно действительно, просто игнорируйте правила, в которых упоминаются сохраненные идентификаторы пользователя или группы.

Получение идентификаторов пользователя и группы

Эти два системных вызова возвращают реальные идентификаторы пользователя и группы соответственно:

```
#include <unistd.h>
#include <sys/types.h>
```

```
uid_t getuid (void);
gid_t getgid (void);
```

Они не могут привести к неудаче. Аналогично эти два вызова возвращают действительные идентификаторы пользователя и группы соответственно:

```
#include <unistd.h>
#include <sys/types.h>

uid_t geteuid (void);
gid_t getegid (void);
```

Они также не могут привести к сбою.

Сессии и группы процессов

Каждый процесс является членом *группы процессов*, которая представляет собой коллекцию из одного или нескольких процессов, в общем случае связанных друг с другом с целью *управления заданиями*. Основная особенность группы процессов заключается в том, что эти сигналы могут быть отправлены всем процессам в группе: одно действие может прервать, остановить или продолжить все процессы в этой группе.

Каждая группа процессов идентифицируется с помощью *идентификатора группы процессов*, а также имеет *лидера группы процессов*. Идентификатор группы процессов равен pid лидера группы процессов. Группы существуют, пока в них остается хотя бы один член. Даже если лидер группы процессов прерывается, группа продолжает существовать.

Когда новый пользователь впервые входит в систему, процесс авторизации создает новую сессию, которая содержит единственный процесс — *оболочку авторизации* пользователя. Оболочка авторизации функционирует как *лидер сессии*. Сессией называется набор из одной или нескольких групп процессов. С помощью сессий устанавливается порядок среди действий авторизованных пользователей, а сами пользователи связываются с *управляющим терминалом* — особым устройством tty, управляющим терминальным процессом ввода-вывода для данного пользователя. Следовательно, сессии в основном связаны с оболочками. Фактически больше ничто в системе не управляет ими.

В то время как группы процессов предоставляют механизм для упрощения адресации сигналов всем членам группы, выполнения контроля задач и других функций оболочки, сессии отвечают лишь за объединение авторизации пользователей вокруг управляющих терминалов. Группы процессов в сессии делятся на единственную *приоритетную группу процессов* и *фоновые группы процессов*, которых может быть несколько или не быть вообще. Когда пользователь покидает терминал, SIGQUIT отправляется всем процессам в приоритетной группе. Когда терминал отключается от сети, SIGHUP отправляется всем процессам в приоритетной группе процессов. Когда пользователь вводит команду прерывания (обычно сочетанием клавиш Ctrl+C), всем процессам в приоритетной группе отправляется SIGINT. Таким образом сессии упрощают для оболочки управление терминалами и авторизациями.

В качестве примера представим, что пользователь авторизуется в системе и ее оболочка авторизации, bash, имеет pid 1700. Экземпляр bash пользователя становится

единственным членом и лидером новой группы процессов с идентификатором группы процессов 1700. Группа находится внутри новой сессии с идентификатором сессии 1700, а `bash` — единственный член и лидер этой сессии. Новые команды, которые пользователь запустит в оболочке, будут работать в новой группе процессов внутри сессии 1700. Одна из этих групп процессов — та, которая связана непосредственно с пользователем и находится под управлением терминала, — является *приоритетной группой процессов*. Все остальные группы — *фоновые группы процессов*.

В данной системе существует много сессий: одна для каждой пользовательской сессии авторизации и другие для процессов, не связанных с пользовательскими сессиями, такие как демоны. Демоны стараются создавать свои собственные сессии во избежание проблем из-за связи с другими сессиями, которые могут прерваться.

Каждая из этих сессий содержит одну или несколько групп процессов, а каждая группа процессов содержит как минимум один процесс. Группа процессов, содержащая более одного процесса, в общем виде реализует управление заданиями.

Команда оболочки наподобие:

```
$ cat ship-inventory.txt | grep booty | sort
```

возвращает группу процессов, включающую три процесса. Таким образом, оболочка может отправлять сигналы трем процессам одновременно. Поскольку пользователь напечатал эту команду в консоли без использования ведущего амперсанда, можно утверждать, что эта группа процессов будет приоритетной. Рисунок 5.1 иллюстрирует взаимоотношения между сессиями, группами процессов, процессами и контролирующими терминалами.

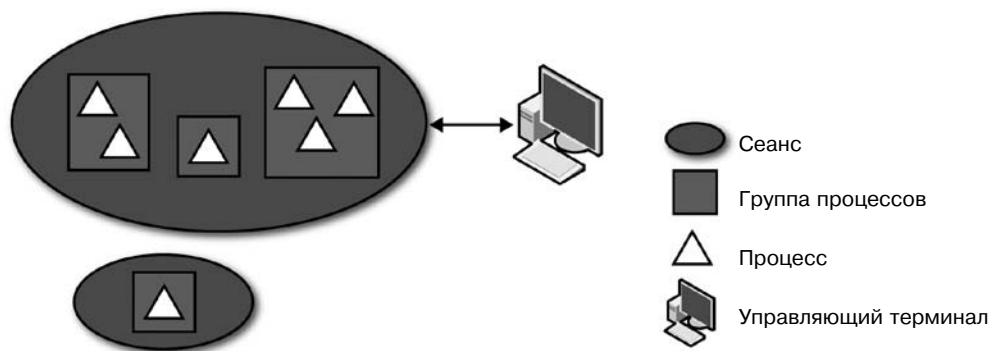


Рис. 5.1. Взаимоотношения между сессиями, группами процессов, процессами и управляющими терминалами

Linux предоставляет несколько интерфейсов для установления и получения сессии и группы процессов, связанных с данным процессом. В основном они используются для нужд оболочки, но также могут быть полезны для таких процессов, как демоны, которые предпочитают не взаимодействовать с сессиями и группами процессов.

Системные вызовы сессий

Оболочки создают новые сессии при авторизации. Они делают это с помощью специального системного вызова, который весьма упрощает создание новой сессии:

```
#include<unistd.h>

pid_tsetsid (void);
```

Вызов `setsid()` создает новую сессию, предполагая, что процесс в данный момент не является лидером группы процессов. Вызывающий процесс создает лидера сессии и единственного члена новой сессии, не имеющего контролирующего `tty`. Вызов также создает новую группу процессов внутри сессии, делая вызывающий процесс ее лидером и единственным членом. Новые идентификаторы сессии и группы процессов равны `pid` вызывающего процесса.

Другими словами, `setsid()` создает новую группу процессов внутри новой сессии и делает вызывающий процесс лидером того и другого. Это полезно для демонов, которые не хотят быть членами существующих сессий или иметь контролирующие терминалы, а также для оболочек, которые хотят создавать новую сессию для каждого пользователя после авторизации.

В случае успеха `setsid()` возвращает идентификатор вновь созданной сессии. В случае ошибки возвращается `-1`, единственно возможный код ошибки `EPERM`, который означает, что процесс является лидером группы процессов. Самый простой путь гарантировать, что какой-либо процесс не является лидером группы процессов, — разветвить его, завершить родительский процесс и заставить дочерний выполнить `setsid()`. Например:

```
pid_t pid;

pid = fork ();
if (pid == -1) {
    perror ("fork");
    return -1;
} else if (pid != 0)
    exit (EXIT_SUCCESS);

if (setsid () == -1) {
    perror ("setsid");
    return -1;
}
```

Получение текущего идентификатора сессии менее полезно, но тоже возможно:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid (pid_t pid);
```

Вызов `getsid()` возвращает идентификатор сессии процесса, определенного через параметр `pid`. Если аргумент `pid` равен 0, вызов возвращает идентификатор сессии вызывающего процесса. В случае ошибки вызов возвращает -1. Единственная возможная величина кода ошибки — `ESRCH`, означающая, что нет доступных процессов с соответствующим `pid`. Обратите внимание, что другие системы UNIX также могут устанавливать в `errno` код ошибки `EPERM`, означающий, что `pid` и вызывающий процесс не принадлежат одной сессии; Linux не возвращает эту ошибку и успешно доставляет идентификатор сессии любого процесса.

Используется `getsid()` нечасто и, как правило, в диагностических целях:

```
pid_t sid;

sid = getsid (0);
if (sid == -1)
    perror ("getsid"); /* должно быть невозможно */
else
    printf ("Мой идентификатор сеанса=%d\n", sid);
```

Системные вызовы групп процессов

Вызов `setpgid()` устанавливает идентификатор группы процессов `pgid` процессу, определенному через `pid`:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
```

```
int setpgid (pid_t pid, pid_t pgid);
```

Если `pid` равен 0, то используется текущий процесс. Если `pgid` равен 0, идентификатор процесса, определенного через `pid`, используется в качестве идентификатора группы процессов.

В случае успеха `setpgid()` возвращает 0. Успех зависит от нескольких условий:

- процесс, определенный через `pid`, должен быть вызывающим процессом или потомком вызывающего процесса, который еще не выполнил вызов `exec` и принадлежит той же сессии, что и вызывающий процесс;
- процесс, определенный с помощью `pid`, не должен быть лидером сессии;
- если `pgid` уже существует, он должен находиться в той же самой сессии, что и вызывающий процесс;
- `pgid` должен быть неотрицательным.

В случае ошибки вызов возвращает -1 и устанавливает в `errno` один из следующих кодов ошибки:

- `EACCESS` — процесс, определенный через `pid`, является потомком вызывающего процесса, который уже вызвал `exec`;
- `EINVAL` — `pgid` меньше 0;
- `EPERM` — процесс, определенный через `pid`, является лидером сессии или находится в другой сессии по отношению к вызывающему процессу; кроме того,

может быть предпринята попытка переместить процесс в группу процессов, принадлежащей другой сессии;

- ESRCH — pid не принадлежит текущему процессу либо потомку текущего процесса или не равен 0.

Как и с сессиями, получение идентификатора группы процессов, в которую входит определенный процесс, менее полезно, но возможно:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
```

```
pid_t getpgid (pid_t pid);
```

Вызов `getpgid()` возвращает идентификатор группы процессов для процесса, определенного через `pid`. Если `pid` равен 0, возвращается идентификатор группы процессов для текущего процесса. В случае ошибки он возвращает -1 и устанавливает `errno` равным ESRCH, единственному возможному коду ошибки, означающему, что использована недопустимая величина `pid`.

Как и `getsid()`, `getpgid()` используется обычно в диагностических целях:

```
pid_t pgid;
pgid = getpgid (0);

if (pgid == -1)
    perror ("getpgid"); /* не должно быть возможно */
else
    printf ("Идентификатор группы процессов равен=%d\n", pgid);
```

Устаревшие функции для группы процессов

Linux поддерживает два устаревших интерфейса из BSD для управления или получения идентификатора группы процессов. Поскольку они менее полезны, чем представленные выше системные вызовы, новые программы должны их использовать только в случае наличия жестких требований о переносимости. Вызов `setpgrp()` может быть использован для установления идентификатора группы процессов.

```
#include <unistd.h>
```

```
int setpgrp (void);
```

Следующий вызов:

```
if (setpgrp () == -1)
    perror ("setpgrp");
```

идентичен такому:

```
if (setpgid (0,0) == -1)
    perror ("setpgid");
```

Обе попытки назначить текущий процесс группе процессов с тем же номером, что и `pid` текущего процесса, возвращают 0 в случае успеха и -1 при неудаче. Все коды ошибки `setpgid()` возможны и для `setpgrp()`, кроме ESRCH.

Аналогично вызов `getpgrp()` может быть использован, чтобы получить идентификатор группы процессов:

```
#include <unistd.h>
```

```
pid_t getpgrp (void);
```

Вызов:

```
pid_t pgid = getpgrp ();
```

идентичен:

```
pid_t pgid = getpgid (0);
```

Оба возвращают идентификатор группы вызывающего процесса. Функция `getpgid()` не может привести к ошибке.

Демоны

Демоном называется процесс, который запущен в фоновом режиме и не привязан ни к какому управляющему терминалу. Демоны обычно запускаются во время загрузки с правами `root` или другими специфическими пользовательскими правами (например, `apache` или `postfix`) и выполняют задачи системного уровня. Принято называть демоны именами, оканчивающимися на букву `d` (например, `crond` или `sshd`), но это не обязательно.

ПРИМЕЧАНИЕ

Название происходит от демона Максвелла, персонажа мыслительного эксперимента, поставленного в 1867 году физиком Джеймсом Максвеллом (James Maxwell). Демоны — сверхъестественные существа в греческой мифологии, существующие где-то в пространстве между богами и людьми, обладающие большим могуществом и тайными знаниями. В отличие от демонов, описанных в христианстве, древнегреческие демоны не всегда злые. В действительности демоны из мифологии часто помогали богам в делах, которые жители Олимпа находили недостойными себя, — как и демоны в UNIX, выполняющие задачи, которых избегают пользователи.

Две обязательные особенности демона таковы: он должен запускаться как потомок процесса `init` и не должен быть связан с терминалом.

В общем случае программа выполняет следующие действия, прежде чем становится демоном.

1. Вызов `fork()`. Это создает новый процесс, который станет демоном.
2. Родительский процесс выполняет вызов `exit()`. Это гарантирует, что оригинальный родитель («дедушка» демона) не должен будет обслуживать своего потомка, что родительский процесс демона более не существует, а демон является лидером группы процессов (последнее требование обязательно для следующего шага).
3. Вызов `setsid()`, создающий для демона новую группу процессов и сессию; в обеих из них он является лидером. Это также гарантирует, что процесс не имеет

связанных с ним контролирующих терминалов (так как процесс только что создал новую сессию и не будет назначать терминал).

4. Изменение рабочего каталога на корневой через `chdir()`. Это делается потому, что унаследованный рабочий каталог может быть где угодно в файловой системе. Демоны обычно выполняются в течение всего времени работы системы, и какой-то случайный каталог держать постоянно открытым не очень хорошо. Таким образом мы предотвращаем размонтирование администратором файловой системы, содержащей этот каталог.
5. Закрытие всех файловых дескрипторов. Нам не нужно наследовать открытые файловые дескрипторы и оставлять их открытыми.
6. Открытие файловых дескрипторов 0, 1 и 2 (стандартный ввод, стандартный вывод и стандартный вывод ошибки) и перенаправление их к `/dev/null`.

Вот пример программы, «демонизирующей» себя согласно этим правилам:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void)
{
    pid_t pid;
    int i;

    /* создание нового процесса */
    pid = fork ();
    if (pid == -1)
        return -1;
    else if (pid != 0)
        exit (EXIT_SUCCESS);

    /* создание нового сеанса и группы процессов */
    if (setsid () == -1)
        return -1;

    /* установка в качестве рабочего каталога корневого каталога */
    if (chdir ("/") == -1)
        return -1;

    /* закрытие всех открытых файлов */
    /* NR_OPEN это слишком, но это работает */
    for (i = 0; i < NR_OPEN; i++)
        close (i);

    /* перенаправление дескрипторов файла 0,1,2 в /dev/null */
```



```
open ("/dev/null", O_RDWR);    /* stdin */
dup (0);                       /* stdout */
dup (0);                       /* stderr */

/* всякие действия демона... */

return 0;
}
```

Большинство систем UNIX предоставляют функцию `daemon()` в своих библиотеках C, автоматизируя эти шаги и сводя их к простому:

```
#include <unistd.h>
```

```
int daemon (int nochdir, int noclose);
```

Если `nochdir` не равен нулю, демон не изменяет свою рабочую директорию на `root`. Если `noclose` не равен нулю, демон не закрывает все открытые дескрипторы файлов. Эти параметры полезны, если родительский процесс уже позаботился об этих аспектах процедуры создания демона. Обычно, однако, всем параметрам все же передается 0.

В случае успеха вызов возвращает 0. При неудаче возвращается -1 и `errno` присваивается соответствующий код ошибки из `fork()` или `setsid()`.

Резюме

В этой главе мы рассмотрели фундаментальные понятия управления процессами в UNIX, от создания до уничтожения. В следующей главе мы поговорим о более сложных интерфейсах управления процессами, включая интерфейсы для изменения подхода к планированию процессов.

6 Расширенное управление процессами

В гл. 5 мы рассмотрели, что такое процесс и какие части системы им затрагиваются, в том числе изучили системные вызовы, используемые для создания процессов, управления или их завершения. Эта глава продолжает данную тему: мы начнем с обсуждения планировщика процессов Linux и его алгоритмов планирования, а затем перейдем к изучению сложных интерфейсов управления процессами. Системные вызовы управляют ходом планирования процесса. Они влияют на действия планировщика, направленные на выполнение задач приложения или достижение целей пользователя.

Планирование процессов

Планировщик процессов — это подсистема ядра, которая разделяет ограниченное значение ресурса времени процессора между системными процессами. Иными словами, планировщик процессов (или просто *планировщик*) — это компонент ядра, обеспечивающий выбор процесса, который будет выполняться следующим. Принимая решение, какие процессы могут быть запущены и когда, планировщик отвечает за максимизацию использования процессора одновременно с обеспечением видимости, что несколько процессов выполняются одновременно, не мешая друг другу.

В этой главе мы будем много говорить о *работоспособных процессах*. Работоспособным процесс можно назвать, если он не заблокирован; *заблокированным процессом* является «спящий», ожидающий ввода-вывода из ядра. Процессы, взаимодействующие с пользователями, считывающие или записывающие большие файлы либо реагирующие на события в сети, скорее всего, будут проводить довольно много времени в заблокированном состоянии, ожидая ресурсов, позволяющих им снова начать работу; следовательно, в это время они не являются работоспособными. Если имеется только один работоспособный процесс, задача планировщика весьма проста — его запуск. Она, однако, усложняется, когда работоспособных процессов становится больше. В такой ситуации некоторые процессы будут запущены, а другие — ожидать своей очереди. Решение о том, какие процессы запустить, когда и на какой период времени, — основная ответственность планировщика процессов.

Операционная система на машине с одним процессором является *многозадачной*, если она способна чередовать выполнение нескольких процессов, создавая видимость, что в каждый момент времени работает более одного процесса. В компьютерах с несколькими процессорами многозадачная операционная система обеспечивает параллельную работу процессов в действительности, на разных процессорах. Немногозадачная операционная система, например DOS, в каждый момент времени может запускать только одно приложение.

Многозадачные операционные системы бывают двух видов — *кооперативные* и *приоритетные*. Linux предпочла последний вид многозадачности, при котором планировщик решает, когда прекратить выполнение одного процесса и возобновить другой. Действие по приостановке одного процесса для возобновления другого называется *переключением*. Время, в течение которого процесс может выполняться, прежде чем планировщик его прервет, известно как *квант времени* процесса, так как планировщик выделяет процессу определенный «квант» времени процессора.

При кооперативном типе многозадачности, напротив, процесс не останавливается, пока самостоятельно не примет решение сделать это. Добровольное прерывание выполнения процесса им самим называется *уступкой*. В идеальном случае процессы «уступали» бы часто, но операционная система в любом случае не может влиять на их поведение. Плохо написанная или поврежденная программа может выполняться достаточно долго, чтобы нарушить видимость многозадачности, или даже работать неопределенное время, влияя на работу всей системы. Из-за этого существенного недостатка большинство операционных систем предпочитают приоритетный вид многозадачности; Linux не исключение.

Планировщик процессов Linux менялся со временем. В настоящий момент планировщик, доступный, начиная с версии ядра Linux 2.6.23, называется CFS (Completely Fair Scheduler). Название происходит от внедрения в планировщик принципа равноправной очереди, планирующего алгоритма, который старается обеспечить равный доступ к ресурсам всем конкурирующим потребителям. CFS резко отличается от других планировщиков процессов в UNIX, включая своего предшественника, планировщик ввода-вывода.

Кванты времени

Квант времени, который планировщик выделяет каждому процессу, является очень важной величиной с точки зрения общего поведения и производительности системы. Если квант слишком велик, процессы должны ждать долгое время между периодами выполнения, минимизируя видимость одновременной работы. Пользователей могут раздражать заметные задержки. И наоборот, если квант времени слишком мал, значительная доля системного времени тратится на переключение между разными приложениями, из-за чего выгода сосредоточенности на времени пропадает.

Следовательно, определить идеальный временной квант непросто. Одни системы выделяют процессам большие временные кванты, надеясь на максимизацию пропускной способности системы и увеличение общей производительности. Другие

системы дают процессам очень маленькие кванты, надеясь обеспечить отличную интерактивную производительность системы. Как вы вскоре увидите, CFS отвечает на вопрос об идеальном размере кванта достаточно оригинальным образом — отменой временных квантов.

Процессы ввода-вывода против ограниченных процессором

Процессы, которые последовательно расходуют все доступные им временные кванты, считаются *ограниченными процессором*. Они активно тратят ресурсы времени и обрабатывают все, что выделит им планировщик. Самый простой пример — бесконечный цикл:

```
// 100 % ограничено процессором
while (1)
    ;
```

В качестве более жизненных примеров можно привести научные вычисления, математические расчеты, загрузку и обработку изображений.

С другой стороны, процессы, которые проводят большую часть времени в заблокированном состоянии, ожидая каких-то событий, а затем выполняются снова, называются *ограниченными вводом-выводом*. Они часто прерываются, ожидая файлов или событий ввода-вывода в сети, заблокированные вводом с клавиатуры или ожиданием движений мыши. Примерами таких процессов являются файловые утилиты, основная задача которых — отправка ядру системного вызова на выполнение ввода-вывода, такие как `cp` или `mv`, а также большинство приложений с графическим пользовательским интерфейсом, которые проводят основную часть времени, ожидая действий пользователя.

Процессы, ограниченные ресурсами процессора или вводом-выводом, отличаются наиболее выгодными для них действиями планировщика. Приложения, ограниченные процессором, требуют максимально больших квантов времени для оптимизации доли попадания в кэш (за счет сосредоточенности процесса во времени) и максимально быстрого выполнения задачи. В противоположность им процессы, ограниченные вводом-выводом, не требуют больших квантов времени, так как выполняются достаточно быстро, а затем отправляют системный запрос ввода-вывода или блокируются, ожидая ресурсов ядра. Однако этим процессам внимание планировщика необходимо, так как скорость возобновления работы после блокировки и отправка новых запросов ввода-вывода напрямую влияет на эффективность работы аппаратного обеспечения системы. Кроме того, если приложение ждет ответа пользователя, то чем выше оно стоит в планировщике, тем быстрее будет его выполнение с точки зрения пользователя.

Удовлетворение нужд процессов обоих видов — непростая задача. В действительности в большинстве приложений имеются процессы, ограниченные и вводом-выводом, и ресурсами процессора. Хороший пример — программы кодирования/декодирования аудио и видео, не допускающие категоризации. Большинство игр

также включают в себя оба вида процессов. Не всегда возможно определить тип того или иного приложения, а кроме того, в разные моменты времени один и тот же процесс может вести себя по-разному.

Приоритетное планирование

В традиционном планировании процессов UNIX всем запущенным процессам назначаются определенные кванты времени. Когда процесс исчерпывает свой квант, ядро прерывает его и запускает следующий. Если в системе больше нет доступных к запуску процессов, ядро берет набор процессов с закончившимися квантами, назначает им новые кванты и приступает к их запуску в порядке очереди. Таким образом, процедура повторяется, последовательно вводя и выводя процессы из списка работающих, по мере того как они создаются или прерываются, блокируются вводом-выводом или «просыпаются». Это позволяет всем процессам по очереди запускаться, даже если в системе присутствуют процессы более высокого приоритета. Процессам с более низким приоритетом просто приходится подождать, пока высокоприоритетные исчерпают свой временной квант или заблокируются. Данный принцип приводит к важному, хоть и неявному правилу планирования UNIX: все процессы должны постепенно выполняться.

Completely Fair Scheduler

Completely Fair Scheduler (CFS) заметно отличается от традиционных планировщиков процессов UNIX. В большинстве систем UNIX, включая Linux перед презентацией CFS, в процедуре планирования было два основных фактора: приоритетность и временной квант. Как было сказано выше, процессам назначается временной квант, представляющий собой фрагмент ресурсов, выделенный данному процессу. Процессы могут выполняться в течение назначенного им временного кванта. Аналогично процессам назначаются приоритеты. Алгоритм очень прост и успешно работал для ранних систем UNIX с разделением времени. Однако в системах, в которых требуется хорошая интерактивная производительность и надежность, например на современных компьютерах и мобильных устройствах, ситуация несколько менее радужная.

CFS представляет значительно отличающийся алгоритм, называемый *беспристрастным планированием*, в котором квантов времени как единиц контроля доступа к процессору уже нет. Вместо них CFS назначает каждому процессу *долю* процессорного времени. Алгоритм очень прост: CFS запускается, назначая N процессам каждому по $1/N$ времени процессора. Затем эти доли уточняются, «взвешивая» каждый процесс по его точной величине. Процессы с нулевой точной величиной по умолчанию имеют вес, равный единице, следовательно, их пропорция не меняется. Процессы с меньшей точной величиной (высший приоритет) получают больший вес, увеличивая свою долю ресурсов процессора, в то время как процессы

с большей точной величиной (низший приоритет) получают меньший вес, уменьшая свой расход ресурсов процессора.

Таким образом CFS получает взвешенную пропорцию времени процессора, назначенную каждому процессу. Чтобы определить точное время выполнения каждого процесса, CFS требуется поделить эти пропорции на фиксированные периоды. Такой период называется *целевой задержкой*, так как он представляет собой запланированную задержку в работе системы. Чтобы понять, что такое целевая задержка, давайте представим себе, что она равна 20 миллисекундам и у нас имеется два выполняемых процесса равного приоритета. Таким образом, каждый процесс имеет один и тот же вес и ему назначена одна и та же доля процессора, 10 миллисекунд. CFS запустит один процесс на 10 миллисекунд, затем другой на 10 миллисекунд и т. д. Если в системе имеется пять выполняемых процессов, CFS будет запускать каждый из них на 4 миллисекунды.

Однако что если у нас, скажем, 200 процессов? С целевой задержкой, равной 20 миллисекунд, CFS будет запускать каждый процесс всего на 100 микросекунд. Из-за потерь времени на переключение от одного процесса к другому, известных как *задержки переключения*, и снизившегося временного ресурса общая пропускная способность системы значительно уменьшится. Чтобы разрешить эту ситуацию, CFS представляет другой ключевой фактор — минимальную детализацию.

Минимальная детализация — нижний предел длины промежутка времени для запуска любого процесса. Все процессы, независимо от полученной ими доли ресурсов процессора, будут запускаться самое меньшее на величину промежутка минимальной детализации (или пока не заблокируются). Это позволяет убедиться, что задержки переключения не занимают неприемлемо большой доли системного времени за счет сохранения величины целевой задержки. Таким образом, с помощью минимальной детализации и обеспечивается беспристрастность. С обычными величинами целевой задержки и минимальной детализации при разумном количестве запущенных процессов минимальная детализация не требуется, ибо беспристрастность и величина целевой задержки достигаются и так.

Назначая доли ресурсов процессора вместо фиксированных временных квантов, CFS способен обеспечить справедливость: каждый процесс получает свою *честную долю* ресурсов процессора. Более того, CFS способен обеспечить конфигурируемую и планируемую величину задержки, поскольку *целевая задержка* — величина, устанавливаемая пользователем. В традиционных планировщиках UNIX процессы запускаются в течение фиксированных промежутков времени, известных априори, но целевая задержка (как часто они запускаются) неизвестна. В CFS процессы запускаются согласно назначенным пропорциям, с заранее известной величиной целевой задержки, но временной квант становится величиной динамической, зависящей от количества запущенных процессов в системе. Это совершенно другой подход к обработке планирования процессов, решающий множество проблем, связанных с интерактивными процессами и процессами ввода-вывода, по сравнению с традиционными планировщиками процессов.

Высвобождение ресурсов процессора

Хотя Linux — многозадачная операционная система приоритетного типа, в ней имеется системный вызов, позволяющий принудительно прервать выполнение процессов и заставить планировщик выбрать новый процесс для исполнения:

```
#include<sched.h>
```

```
int sched_yield(void);
```

Вызов `sched_yield()` приведет к остановке выполняющегося в настоящее время процесса, после чего планировщик выбирает новый процесс для запуска таким же образом, как если бы ядро само прекратило выполняющийся процесс и запустило новый. Необходимо отметить, что, если нет других выполняющихся процессов, как часто и бывает, прерванный процесс будет тут же запущен снова. Из-за этой неопределенности в сочетании с общим принципом, что всегда есть лучший выбор, данный системный вызов используется нечасто.

В случае успеха вызов возвращает 0; при неудаче — -1 и отправляет `errno` в соответствующем коде ошибки. В Linux, а также, скорее всего, в большинстве других UNIX-систем, `sched_yield()` всегда выполняется успешно, возвращая 0. Внимательный программист может проверить возвращаемую величину, например:

```
if (sched_yield ())  
    perror ("sched_yield");
```

Правомерное использование. На практике существует не так много вариантов правомерного использования `sched_yield()` в действительно многозадачной системе с вытеснением наподобие Linux. Ядро отлично справляется с принятием оптимальных и наиболее выгодных решений по планированию самостоятельно — действительно, ведь ядро лучше, чем отдельное приложение, подготовлено к задачам очередности вытеснения. Вот почему разработчики операционных систем отказываются от кооперативной многозадачности в пользу вытесняющей.

Почему же тогда в POSIX вообще существует такой системный вызов, как «перделка расписания»? Ответ заключен в приложениях, которым приходится ожидать событий вовне, которые могут быть вызваны действиями пользователя, аппаратного обеспечения или другими событиями. В частности, если один процесс должен ожидать другого, самое простое решение — «просто уступи ресурсы процессора, пока другой процесс не завершится». В качестве примера можно привести реализацию простого потребителя в паре «потребитель — производитель»:

```
/* потребитель... */  
do {  
  
    while (producer_not_ready ())  
        sched_yield ();  
    process_data ();  
} while (!time_to_quit ());
```

К счастью, программисты UNIX нечасто создают код, подобный этому. Обычно работа программ UNIX управляется событиями и использует какой-либо механизм блокировки (например, конвейер) между потребителем и производителем, а не `sched_yield()`. В данном случае потребитель получает с конвейера информацию, блокируя себя, если нужные данные пока недоступны. Производитель в это же время отправляет данные в конвейер по мере появления. Таким образом, пользовательское пространство не влияет на координацию процесса, которой занимается ядро, способное оптимизировать ситуацию: приостанавливать процессы в нужный момент и возобновлять их по мере необходимости. В целом программы UNIX должны стремиться к работе, управляемой происходящими событиями, которые основываются на дескрипторах файла блокировки.

До недавнего времени только одна ситуация не позволяла избежать использования `sched_yield()`: блокировка потока в пользовательском пространстве. Когда поток пытался перехватить блокировку, удерживаемую другим потоком, новый поток должен был уступить системные ресурсы, пока блокировка не становилась доступной для него. В отсутствие блокировок пользовательского пространства ядром этот подход был самым простым и наиболее эффективным. К счастью, реализация потоков в современной Linux (NPTL) представляет собой оптимальное решение с использованием *фьютексов*, которые предоставляют поддержку ядра для эффективных блокировок в пользовательском пространстве.

Еще один пример использования `sched_yield()` — «любезная игра»: программы, сильно нагружающие процессор, периодически могут вызывать `sched_yield()`, пытаясь минимизировать свое воздействие на систему. Эти благие намерения имеют, однако, два недостатка. Во-первых, ядро способно скорее принимать стратегические решения о планировании, чем обрабатывать отдельные процессы, и, следовательно, нести ответственность за корректную работу системы должен скорее планировщик, а не процесс. Во-вторых, уменьшение нагрузки, создаваемой приложениями, и предоставление другим приложениям доступа к ресурсам процессора — это ответственность пользователя, а не самих приложений. Пользователь может формулировать свои представления о приоритетах производительности приложения с помощью команды оболочки `nice`, о которой мы поговорим далее в этой главе.

Приоритеты процессов

ПРИМЕЧАНИЕ

Обсуждение в этом разделе относится к обычным процессам, а не к процессам реального времени. Последние требуют других критериев по планированию и системы отдельных приоритетов. Мы будем обсуждать процессы реального времени далее в этой главе.

Linux не планирует процессы как попало. Каждому процессу назначается *приоритет*, который влияет на то, как долго тот будет работать: напомним, что доля ресурсов процессора, назначаемая процессу, взвешивается согласно его значению любезности. Исторически эти приоритеты называются в Linux *значениями любезности*,

так как их основная идея — «быть любезными» по отношению к другим процессам путем следования процессной приоритетности, позволяя другим процессам потреблять больше системного процессорного времени.

Пределы значений любезности — от -20 до 19 включительно, а значение по умолчанию — 0. Может быть не совсем понятно, почему чем ниже значение любезности процесса, тем выше его приоритет и больше квант времени. Увеличение значения любезности процесса «оказывает любезность» остальной части системы. Численная перестановка тоже может выглядеть запутанно. Когда мы говорим, что процесс имеет «высокий приоритет», то подразумеваем, что он может быть запущен на более долгое время, чем процессы с низшим приоритетом, но этот процесс будет иметь более низкое значение любезности.

nice()

Linux предоставляет несколько вызовов для получения и установки значения любезности процесса. Самый простой из них — `nice()`:

```
#include <unistd.h>
```

```
int nice (int inc);
```

Успешная работа `nice()` увеличивает значение любезности процесса на значение `inc` и возвращает обновленную величину. Только процесс с характеристикой `CAP_SYS_NICE` (которым фактически владеет пользователь `root`) может установить отрицательную величину `inc`, уменьшая его значение любезности и, таким образом, повышая приоритет процесса. Следовательно, процессы без прав `root` могут только снижать свои приоритеты (увеличивая значения любезности).

В случае ошибки `nice()` возвращает -1. Однако из-за того, что `nice()` возвращает новое значение любезности, -1 может быть возвращено и в случае успеха. Чтобы различать успешное срабатывание и сбой, можно обнулить `errno` перед запуском, а затем проверить ее значение. Например:

```
int ret;
```

```
errno = 0;
```

```
ret = nice (10); /* увеличение значения любезности на 10 */
```

```
if (ret == -1 &&errno != 0)
```

```
    perror ("nice");
```

```
else
```

```
    printf ("значение любезности теперь равно %d\n", ret);
```

Linux возвращает только один код ошибки, `EPERM`, означающий, что вызывающий процесс пытался увеличить свой приоритет, но не имеет характеристики `CAP_SYS_NICE`. Другие системы могут также вернуть `EINVAL`, когда `inc` пытается установить значение любезности, выходящее за пределы разрешенных значений, но Linux этого не делает. Вместо этого Linux самостоятельно снижает или повышает значение `inc`, чтобы значение любезности попало в допустимый диапазон.

Отправка `inc`, равного 0, — простой способ получить текущее значение любезности:

```
printf ("текущее значение любезности равно %d\n", nice (0));
```

Часто процесс хочет установить абсолютное значение любезности вместо относительного изменения. Это может быть сделано с помощью кода наподобие следующего:

```
int ret, val;

/* получение текущего значения любезности */
val = nice (0);

/* нам нужно значение любезности, равное 10 */
val = 10 - val;
errno = 0;
ret = nice (val);
if (ret == -1 &&errno != 0)
    perror ("nice");
else
    printf ("текущее значение любезности равно %d\n", ret);
```

getpriority() и setpriority()

Предпочтительным решением является применение системных вызовов `getpriority()` и `setpriority()`, которые предоставляют больше возможностей для управления и контроля, но сложнее в использовании:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

Эти вызовы воздействуют на процесс, группу процессов или пользователей, что определено с помощью `which` и `who`. Значение `which` должно быть `PRIO_PROCESS`, `PRIO_PGRP` или `PRIO_USER`, в то время как `who` указывает идентификатор процесса, группы процессов или идентификатор пользователя соответственно. Если значение `who` равно 0, то вызов работает с текущими идентификатором процесса, группы процессов или идентификатором пользователя соответственно.

Вызов `getpriority()` возвращает наибольший приоритет (наименьшую численную величину значения любезности) каждого из указанных процессов. Вызов `setpriority()` устанавливает значение приоритета каждого из указанных процессов, равное `prio`. Как и `nice()`, только процесс со свойством `CAP_SYS_NICE` может увеличить приоритет процесса (снизить численное значение любезности). Следовательно, только процесс с этим свойством может увеличить или уменьшить приоритет процесса, не принадлежащего вызывающему пользователю.

Как и `nice()`, `getpriority()` возвращает `-1` в случае ошибки. Это может быть и возвращенный результат при успешном срабатывании, поэтому программист должен очистить `errno` перед вызовом, если нужно обработать условия ошибки. Вызов `setpriority()` не приводит к таким проблемам, он всегда возвращает `0` в случае успеха и `-1` — в случае ошибки.

Следующий код возвращает текущую величину приоритета процесса:

```
int ret;

ret = getpriority (PRIO_PROCESS, 0);
printf ("значение любезности равно %d\n", ret);
```

А данный код устанавливает приоритеты всех процессов текущей группы процессов равными `10`:

```
int ret;

ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
    perror ("setpriority");
```

В случае ошибки обе функции устанавливают одно из следующих значений `errno`:

- `EACCESS` — процесс пытался увеличить приоритет указанного процесса, но не обладает свойством `CAP_SYS_NICE` (касается только `setpriority()`);
- `EINVAL` — величина, указанная в `which`, не является `PRIO_PROCESS`, `PRIO_PGRP` или `PRIO_USER`;
- `EPERM` — действительный идентификатор пользователя указанного процесса не совпадает с действительным идентификатором пользователя запущенного процесса, а запущенный процесс не обладает `CAP_SYS_NICE` (касается только `setpriority()`);
- `ESRCH` — не найден ни один процесс, удовлетворяющий условиям, указанным в `which` и `who`.

Приоритеты ввода-вывода

В дополнение к приоритетам планирования Linux позволяет процессам указывать приоритеты ввода-вывода. Эти значения влияют на относительные приоритеты запросов ввода-вывода процессов. Планировщик ядра, ответственный за ввод-вывод (о котором мы говорили в гл. 4), обслуживает запросы, поступающие от процессов, по мере снижения приоритетов ввода-вывода последних.

По умолчанию планировщики ввода-вывода используют значение любезности процесса для определения приоритетов ввода-вывода. Следовательно, установка значения любезности автоматически меняет приоритет ввода-вывода. Однако ядро Linux дополнительно предоставляет два системных вызова для экстренной установки и получения приоритетов ввода-вывода независимо от значения любезности:

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

К сожалению, `glibc` не предоставляет в пользовательском пространстве интерфейсов для этих системных вызовов. Без поддержки `glibc` их применение весьма затруднительно. Кроме того, когда (и если) появится поддержка `glibc`, интерфейсы могут отличаться от системных вызовов. Пока же поддержки нет, существует два способа влиять на приоритеты ввода-вывода процессов: через значение любезности или с помощью утилиты `ionice`, являющейся частью пакета `util-linux`¹.

Не все планировщики ввода-вывода поддерживают приоритеты ввода-вывода. Планировщик `Completely Fair Queuing (CFQ)` поддерживает их; другие планировщики — в настоящее время нет. Если имеющийся планировщик ввода-вывода не поддерживает приоритеты ввода-вывода, они игнорируются без каких-либо предупреждений.

Привязка процессов к процессору

Linux поддерживает работу нескольких процессоров в одной системе. Не считая процесса загрузки, большая часть работы по поддержке нескольких процессоров лежит на планировщике процессов. В мультипроцессорной машине планировщик процессов должен решить, какой процесс должен быть запущен на каждом процессоре (CPU).

Из-за такого распределения ответственности появляются две проблемы: планировщик должен работать на полной загрузке всех системных процессоров, так как невыгодно, чтобы один CPU оставался в бездействии, пока процесс ждет запуска. Однако когда процесс оказался запланированным на одном из CPU, планировщик должен стараться запланировать его на тот же процессор в будущем. Это удобно, так как *миграция* процессов с одного процессора на другой весьма затратна.

Наибольшие затраты связаны с *эффектами кэширования* при миграции. Принцип разработки современных SMP систем таков, что кэши, связанные с каждым процессором, независимы друг от друга и самостоятельны. Таким образом, данные в кэше одного процессора совсем иные, чем другого. Если процесс переносится на новый CPU и переписывает новые данные в память, данные в кэше прежнего CPU устаревают. Если продолжать их использовать, можно повредить часть данных. Чтобы не допустить этого, кэши делают данные, хранящиеся в каждом из них, *недействительными* для других кэшей. Следовательно, какой-либо набор данных может находиться строго в кэше только одного процессора в каждый момент времени (предполагая, что данные кэшируются в принципе). Когда процесс перемещается с одного процессора на другой, получается два взаимосвязанных источника затрат: кэшированные данные, более недействительные для перемещенного процесса, и данные в оригинальном кэше процесса, которые должны быть объявлены недействительными. По этой причине планировщики процессов стараются держать процесс на одном процессоре как можно дольше.

¹ Пакет `util-linux` доступен на `kernel.org`. Он лицензирован GNU General Public License v2.

Две цели планировщика процессов потенциально взаимоисключающие. Если один процессор имеет значительно большую процессную нагрузку, чем другой, — или, что еще хуже, один процессор полностью занят, а другой свободен, — есть смысл перепланировать некоторые процессы на менее занятый процессор. Решение о том, когда произвести перепланировку, зависит от степени несбалансированности и называется *распределением нагрузки*. Это очень важно для производительности машин SMP.

Привязка процессов к процессору — это вероятность, что процесс будет постоянно запланирован на один и тот же процессор. Термин «*мягкая привязка*» означает естественное для планировщика стремление продолжать выполнение процесса на одном и том же процессоре. Как я уже сказал, это важно. Планировщик Linux старается запланировать одни и те же процессы на одни и те же процессоры насколько возможно долго, перемещая процессы с одного CPU на другой только в крайнем случае, когда дисбаланс нагрузки очень велик. Это позволяет планировщику минимизировать эффекты кэширования, возникающие при миграции, при одновременной гарантии, что все процессоры в системе загружены примерно одинаково.

Иногда, однако, пользователь или приложение хочет принудительно связать процесс с каким-либо процессором. Чаще всего это обусловлено высокой чувствительностью процесса к кэшированию, из-за чего процессу лучше выполняться на одном и том же процессоре. Привязывая процесс к определенному процессору и поддерживая эту связь на уровне ядра, мы обеспечиваем *жесткую привязку*.

sched_getaffinity() и sched_setaffinity(). Процессы наследуют привязки CPU от своих предков, и по умолчанию процессы могут быть запущены на любом CPU. Linux предоставляет два системных вызова для получения и установки жесткой привязки процесса:

```
#define _GNU_SOURCE

#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize,
                      const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                      cpu_set_t *set);
```

Вызов `sched_getaffinity()` получает привязку к CPU процесса с идентификатором `pid` и сохраняет ее в специальном типе `cpu_set_t` type, куда можно получить доступ через специальный макрос. Если `pid` равен 0, вызов возвращает привязку текущего процесса. Параметр `setsize` определяет размер типа `cpu_set_t`, который

может быть использован `glibc` для обеспечения совместимости при изменении размера типа, возможного в будущем. В случае успеха `sched_getaffinity()` возвращает 0; в случае неудачи он возвращает -1 и устанавливает переменную `errno`. Вот пример:

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
    perror ("sched_getaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
    int cpu;

    cpu = CPU_ISSET (i, &set);
    printf ("cpu=%i is %s\n", i,
           cpu ? "set" : "unset");
}
```

Перед выполнением мы используем `CPU_ZERO` для обнуления всех битов в наборе `set`. Затем мы пробегаемся по набору от 0 до `CPU_SETSIZE`. Обратите внимание, что `CPU_SETSIZE` — это вовсе не размер набора: *никогда* не передавайте это значение в качестве параметра `setsize`. Данная реализация представляет каждый процессор как один бит, поэтому `CPU_SETSIZE` намного больше, чем `sizeof(cpu_set_t)`. Мы используем `CPU_ISSETN` для проверки, привязан ли данный процессор системы, `i`, к нашему процессу. Макрос возвращает 0, если процессор не привязан, и ненулевую величину, если это так.

Устанавливаются только процессоры, физически находящиеся в системе. Таким образом, запуск этого кода в системе с двумя процессорами даст следующий результат:

```
cpu=0 is set
cpu=1 is set
cpu=2 is unset
cpu=3 is unset
...
cpu=1023 is unset
```

Как и получается на выходе, `CPU_SETSIZE` (для которого нумерация начинается с нуля) в настоящее время равен 1024.

Нас интересуют только процессоры 0 и 1, так как они единственные физические процессоры в системе. Например, мы хотим удостовериться, что наш процесс выполняется только на процессоре 0, но никогда — на процессоре 1. Вот такой код сделает это:

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set); /* очистить набор процессоров */
```

```

CPU_SET (0, &set); /* разрешить процессор 0 */
CPU_CLR (1, &set); /* запретить процессор 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
    perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
    int cpu;

    cpu = CPU_ISSET (i, &set);
    printf ("cpu=%i is %s\n", i,
        cpu ? "set" : "unset");
}

```

Как обычно, мы начинаем с обнуления набора с помощью CPU_ZERO. Затем мы устанавливаем процессор 0 посредством CPU_SET и отбрасываем (очищаем) процессор 1 с помощью CPU_CLR. Операция CPU_CLR, по сути дела, здесь не нужна, так как мы только что обнулили весь набор, но все же выполним ее для законченности.

Запуск в той же двухпроцессорной системе отличается от приведенного выше:

```

cpu=0 is set
cpu=1 is unset
cpu=2 is unset
...
cpu=1023 is unset

```

Мы видим, что процессор 1 не установлен. Этот процесс будет выполняться только на процессоре 1, независимо ни от чего.

Возможны четыре значения errno:

- EFAULT — приведенный указатель находится вне адресного пространства процесса или является некорректным;
- EINVAL — в этом случае ни один процессор, физически находящийся в системе, не доступен в set (только для sched_setaffinity()) или setsize() меньше, чем размер внутренней структуры данных ядра, которая представляет набор процессоров;
- EPERM — процесс, указанный с помощью pid, не принадлежит текущему пользователю, а процесс не владеет CAP_SYS_NICE;
- ESRCH — ни одного процесса с указанным pid не найдено.

Системы реального времени

В мире компьютерных технологий термин *«реальное время»* часто становится источником изрядной путаницы. Систему можно отнести к системам реального времени, если она соблюдает ограничения на *временные характеристики функционирования* — минимальные и обязательные периоды времени между воздействием и реакцией. Знакомый всем пример системы реального времени — антиблокировочная

тормозная система (ABS), которой оснащены почти все современные автомобили. В этой системе при нажатии педали тормоза компьютер регулирует давление, применяя максимальное давление и отпуская тормоз много раз в секунду. Это предотвращает блокирование колес, которое снижает эффективность торможения и даже может привести к неконтролируемому заносу автомобиля. Таким образом, ограничения на временные характеристики функционирования в такой системе означают, насколько быстро система должна реагировать на заблокированное колесо и применять давление торможения.

Самые современные операционные системы, включая Linux, обеспечивают некий уровень поддержки реального времени.

Мягкие и жесткие системы реального времени

Системы реального времени бывают двух видов: мягкие и жесткие. *Жесткая система реального времени* требует абсолютно точного соблюдения функциональных временных ограничений. Превышение временного ограничения считается серьезным сбоем и является серьезной неполадкой. *Мягкая система реального времени*, с другой стороны, не считает превышение временного ограничения критической ошибкой.

Применение жесткой системы реального времени обнаружить легко: примерами могут служить антиблокировочные тормозные системы, военные системы вооружения, медицинская техника, устройства обработки сигналов. Мягкие системы реального времени выделить не так просто. Самый привычный представитель этой группы — приложение для обработки видео: если ограничения превышены, пользователь заметит ухудшение качества, но пропуск нескольких кадров не критичен.

На многие другие приложения накладываются временные ограничения, несоблюдение которых приводит к ухудшению впечатлений пользователя от работы с программой. На ум приходят мультимедийные приложения, игры и сетевые программы. А что насчет текстового редактора? Если программа не может быстро среагировать на нажатие клавиши, ничего хорошего сказать о ней нельзя и пользователь имеет полное право раздражаться и злиться. Является ли это примером приложения мягкой системы реального времени? Пожалуй, когда разработчики писали приложение, они понимали, что система должна своевременно реагировать на нажатие клавиши. Однако может ли это считаться временным ограничением? Четкую границу между мягкой и жесткой системами реального времени провести невозможно.

Считается, что система реального времени работает быстро, однако это не всегда так. На самом деле при том же самом аппаратном обеспечении система реального времени будет скорее медленней, чем *система модельного времени*, хотя бы из-за увеличения нагрузки вследствие необходимости поддержания процессов реального времени. Аналогично различие между мягкой и жесткой системами реального времени не связано с размерами временных ограничений по функциональности. Ядерный реактор перегреется, если система аварийной быстрой остановки реактора не опустит стержни, регулирующие мощность реактора, в течение нескольких секунд после обнаружения избыточного потока нейтронов. Это жесткая система

реального времени с большим (по сравнению с привычными в компьютерном мире) функциональным временным пределом. И наоборот, видеопроигрыватель может пропустить кадр или допустить задержку звука, если приложение не может заполнить буфер воспроизведения в течение 100 миллисекунд. Это мягкая система реального времени с небольшим функциональным временным ограничением.

Задержка, колебание и временное ограничение

Задержка означает период от момента воздействия до начала выполнения реакции. Если она меньше либо равна функциональному временному ограничению, система работает правильно. Во многих жестких системах реального времени функциональное временное ограничение и задержка равны — система обрабатывает воздействия через определенные интервалы, в точные моменты времени. В мягких системах реального времени точное необходимое время реакции может быть меньше, вследствие чего и задержка наследует некоторую определенность — требуется лишь, чтобы время реакции уложилось в функциональное временное ограничение.

Достаточно часто задержку измерить тяжело, так как расчет требует известного времени момента воздействия. Способность же зафиксировать временную отметку воздействия, однако, ухудшает способность реагировать на него. Таким образом, большинство усилий по измерению задержки направлено не на фиксацию ее величины, а на измерение отклонений по времени реакции. Эти отклонения между временами успешной реакции называются *колебаниями*, а не задержками.

Например, допустим, что воздействие происходит каждые 10 миллисекунд. Чтобы измерить производительность системы, мы должны фиксировать время ответных действий системы и убедиться, что они также происходят каждые 10 миллисекунд. Отклонения от этой величины не являются задержками, это и есть колебания. Мы измеряли отклонения времени успешной реакции от требуемой величины. Если неизвестно, когда произошло воздействие, мы не можем узнать абсолютную величину разницы во времени между воздействием и реакцией. Даже зная, что воздействия происходят каждые 10 миллисекунд, мы не знаем, когда произошло *первое* из них. Невероятно, но многие попытки измерить задержку спотыкаются именно на этом и приводят в качестве результата колебание, а не задержку. На самом деле колебание — очень полезный показатель, его измерение тоже может принести пользу. Тем не менее следует называть вещи своими именами.

Жесткие системы реального времени часто имеют небольшое колебание, так как они дают реакцию после — а не *в течение* — точного количества времени. Такие системы стремятся к нулевому колебанию, а также задержке, равной функциональному времени. Если задержка превышает временное ограничение, в системе происходит сбой.

Мягким системам реального времени свойственны большие колебания. В этих системах время ответа полностью укладывается в функциональное ограничение — часто наступая раньше, но иногда и нет. Колебания, таким образом, часто вполне могут заменить задержку в качестве показателя производительности.

Поддержка реального времени в Linux

Linux обеспечивает поддержку приложений реального времени через семейство системных вызовов, определенных IEEE Std 1003.1b-1993 (обычно сокращается до POSIX 1993 или POSIX.1b).

С технической точки зрения стандарт POSIX не указывает, является ли предоставленная поддержка реального времени жесткой или мягкой. Фактически все, что указано в стандарте, — это описание нескольких политик планирования с поддержкой приоритетов. Какой тип временных ограничений будет реализован в системе в соответствии с такой политикой, выбирают сами разработчики.

В течение многих лет ядро Linux обеспечивало все лучшую поддержку реального времени, добиваясь постоянного уменьшения задержек и более устойчивых колебаний без ущерба для системной производительности. Это объясняется тем, что уменьшение задержки помогает в работе многим классам приложений, например связанным с **Рабочим столом** или с процессами ввода-вывода (не только приложениям реального времени). Усовершенствования можно приписать также вниманию Linux к встроенным системам и системам реального времени.

Однако многие из этих встроенных модификаций и модификаций реального времени, произведенных в ядре Linux, существуют только в определенных версиях Linux вне основного официального пути развития ядра. Некоторые из этих модификаций предоставляют дальнейшие уменьшения задержки и даже жесткие подходы реального времени. В следующих разделах обсуждаются только официальные интерфейсы ядра и основной путь развития. К счастью, большинство модификаций, связанных с системами реального времени, продолжают использовать интерфейсы POSIX. Значит, последующее обсуждение применимо и к модифицированным системам.

Политики планирования и приоритеты в Linux

Поведение планировщика Linux по отношению к процессам зависит от *политики планирования* процесса, известной также как *класс планирования*. В дополнение к обычной политике по умолчанию Linux предоставляет две политики планирования процессов реального времени. Предпроцессорный макрос из заголовка `<sched.h>` представляет каждую из них: `SCHED_FIFO`, `SCHED_RR` и `SCHED_OTHER`.

У каждого процесса есть *статический приоритет*, независимый от значения любезности. Для обычных приложений этот приоритет всегда равен 0. Для процессов реального времени он варьируется до 1 до 99 включительно. Планировщик Linux всегда выбирает для запуска процесс с самым большим приоритетом (то есть процесс, численная величина приоритета которого наибольшая). Если процесс выполняется со статическим приоритетом 50, а в это же время становится работоспособным процесс с приоритетом 51, то планировщик приостанавливает работающий процесс и переключается на появившийся с приоритетом 51. И наоборот, если выполняется процесс с приоритетом 50, а появляется другой с приоритетом 49, то планировщик не запускает последний, пока процесс с приоритетом 50 не забло-

кируется (то есть приостановит выполнение). Обычные процессы имеют приоритет, равный 0, любой процесс реального времени, готовый к запуску, всегда переключает обычный процесс и запустится.

Политика FIFO, или «первый вошел — первый вышел»

Класс *FIFO* (first in, first out — «первый вошел — первый вышел») — очень простая политика реального времени без использования квантов времени. Процесс класса FIFO будет выполняться, пока не появится другой готовый к работе процесс с более высоким приоритетом. Класс FIFO представлен макросом `SCHED_FIFO`.

Понятие квантов времени в этой политике отсутствует, поэтому ее правила очень просты.

- Работоспособный процесс FIFO-класса будет всегда работать, если его приоритет самый высокий в системе. В частности, как только процесс класса FIFO становится готовым к запуску, он немедленно перекрывает обычный процесс и начинает выполняться.
- Процесс FIFO-класса будет продолжать выполняться, пока не заблокируется, не выполнит `sched_yield()` или не станет работоспособным другой процесс с более высоким приоритетом.
- Когда процесс FIFO-класса блокируется, планировщик убирает его из списка работоспособных. Как только он снова становится работоспособным, он помещается в конец списка процессов с таким же приоритетом. Таким образом, он не запустится снова, пока не закончат работу другие процессы с высшим или *равным* приоритетом.
- Когда процесс FIFO-класса вызывает `sched_yield()`, планировщик передвигает его в конец списка процессов с таким же приоритетом. Это означает, что он не запустится снова, пока не закончат работу другие процессы с тем же приоритетом. Если вызывающий процесс является единственным процессом с такой величиной приоритета, `sched_yield()` не окажет никакого эффекта.
- Когда процесс с высшим приоритетом замещает процесс FIFO-класса, последний остается на той же позиции в списке процессов согласно своему приоритету. Таким образом, как только процесс высшего приоритета прерывает выполнение, замещенный процесс FIFO может продолжить работу.
- Как только процесс присоединяется к FIFO-классу или статический приоритет процесса меняется, он помещается в голову списка процессов согласно своей величине приоритета. Следовательно, процесс FIFO-класса с вновь определенным приоритетом может заместить выполняющийся процесс с тем же приоритетом.

Фактически можно сказать, что процессы FIFO класса выполняются столько времени, сколько хотят, если они являются процессами с самым высоким приоритетом в системе. Самые интересные правила описывают взаимодействия FIFO-процессов равного приоритета.

Политика RR, или карусели

Класс *RR* (round-robin — «карусель») идентичен *FIFO* за исключением того, что он включает дополнительные правила для процессов, имеющих равные приоритеты. Этот класс представляется макросом `SCHED_RR`.

Планировщик назначает квант времени каждому процессу *RR*-класса. Когда процессы класса *RR* исчерпают свой квант, планировщик передвигает их в конец списка процессов того же приоритета. Таким образом, *RR*-процессы данного приоритета планируются как бы в виде карусели. Если существует только один процесс данного приоритета, *RR*-класс идентичен *FIFO*. В этом случае, когда квант исчерпан, процесс просто немедленно продолжает выполнение.

Можно считать процессы *RR*-класса идентичными процессам класса *FIFO*, кроме того, что первые дополнительно прерывают выполнение, исчерпав свой квант времени, после чего передвигаются в конец списка работоспособных процессов с таким же приоритетом.

Выбирать из `SCHED_FIFO` и `SCHED_RR` следует, исключительно исходя из желаемого механизма взаимодействия процессов равного приоритета. Кванты времени *RR*-процессов релевантны только для процессов одинакового приоритета. Процессы класса *FIFO* будут продолжать выполняться; процессы *RR*-класса будут планироваться отдельно для каждой величины приоритета. В обоих случаях процессы с более низким приоритетом не будут выполняться, пока существует процесс с более высоким приоритетом.

Обычная политика

Макрос `SCHED_OTHER` представляет стандартную политику планирования, класс без поддержки реального времени, используемый по умолчанию. Все процессы обычного класса имеют статический приоритет, равный 0. Следовательно, все работоспособные процессы класса *FIFO* или *RR* будут замещать работоспособные процессы обычного класса.

Планировщик использует значение любезности, обсуждавшееся выше, чтобы расставить приоритеты процессов внутри обычного класса. Значение любезности не влияет на статический приоритет, который остается равным 0.

Пакетная политика планирования

Макрос `SCHED_BATCH` представляет *пакетную, или пассивную, политику планирования*. Этот подход в какой-то степени противоположен политикам реального времени: процессы этого класса запускаются, только когда других работоспособных процессов в системе нет, даже если прочие процессы уже исчерпали свои временные кванты. Это отличается от поведения процессов с наибольшими значениями любезности (то есть с наименьшими приоритетами): они запускаются, как только процессы с более высокими приоритетами исчерпали свои временные кванты.

Установка политики планирования Linux

Процессы могут управлять политикой планирования Linux через `sched_getscheduler()` и `sched_setscheduler()`:

```
#include <sched.h>

struct sched_param {
    /* ... */
    int sched_priority;
    /* ... */
};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                       int policy,
                       const struct sched_param *sp);
```

Успешный вызов `sched_getscheduler()` возвращает политику планирования процесса, определенного через `pid`. Если `pid` равен 0, вызов возвращает политику планирования вызывающего процесса. Величина целого типа, указанная в `<sched.h>`, представляет политику планирования: «первый вошел — первый вышел» обозначена константой `SCHED_FIFO`, политика карусели — `SCHED_RR`, а обычная политика — `SCHED_OTHER`. В случае ошибки вызов возвращает -1 (политика планирования никогда не может быть представлена этим числом), и `errno` заполняется соответствующим образом.

Использование очень простое:

```
int policy;

/* получаем нашу политику планирования*/
policy = sched_getscheduler (0);

switch (policy) {
case SCHED_OTHER:
    printf ("Обычная политика\n");
    break;
case SCHED_RR:
    printf ("Политика карусели\n");
    break;
case SCHED_FIFO:
    printf("Политика FIFO\n");
    break;
case -1:
    perror ("sched_getscheduler");
    break;
default:
    fprintf(stderr, "Неизвестная политика!\n");
}
```

Вызов `sched_setscheduler()` устанавливает `policy` в качестве политики планирования процесса, определенного `pid`. Любые параметры, связанные с `policy`, устанавливаются через `sp`. Если `pid` равен 0, политика планирования и параметры устанавливаются для вызывающего процесса. В случае успеха вызов возвращает 0. При неудаче вызов возвращает -1 и `errno` заполняется соответствующим образом.

Допустимые поля внутри структуры `sched_param` зависят от того, какие политики планирования поддерживаются операционной системой. Для `SCHED_RR` и `SCHED_FIFO` требуется одно поле, где указывается статический приоритет. `SCHED_OTHER` не использует полей параметров, но политики планирования, которые будут поддерживаться в будущем, могут потребовать использования новых полей. Таким образом, в переносимых и разрешенных программах не следует делать предположений относительно компоновки структуры.

Установить политику планирования процесса и параметры очень просто:

```
struct sched_param sp = { .sched_priority = 1 };
int ret;

ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1) {
    perror ("sched_setscheduler");
    return 1;
}
```

Данный фрагмент кода устанавливает политику карусели со статическим приоритетом 1 в качестве политики планирования вызывающего процесса. Мы предполагаем, что 1 — это допустимая величина приоритета, хотя технически так будет не всегда. В следующем разделе мы выясним, как определить допустимые пределы приоритетов для данной политики.

Установка политики планирования, кроме `SCHED_OTHER`, требует наличия характеристики `CAP_SYS_NICE`. Следовательно, пользователь, обладающий правами `root`, как правило, запускает процессы реального времени. Начиная с версии ядра 2.6.12 пределы ресурсов `RLIMIT_RTPRIO` позволяют пользователям без прав `root` устанавливать политики реального времени до определенной величины приоритета.

В случае ошибки возможны следующие значения `errno`:

- `EFAULT` — указатель `sp` определяет недопустимый или недоступный фрагмент памяти;
- `EINVAL` — политика планирования, определенная в `policy`, недопустима, или величина, указанная в `sp`, не имеет смысла для данной политики (касается только `sched_setscheduler()`);
- `EPERM` — вызывающий процесс не имеет необходимых характеристик;
- `ESRCH` — отсутствует работающий процесс с указанной величиной `pid`.

Установка параметров планирования

Интерфейсы `sched_getparam()` и `sched_setparam()`, определенные POSIX, используются для получения и установки параметров, связанных с уже выставленной политикой планирования:

```
#include <sched.h>

struct sched_param {
```

```

    /* ... */
    int sched_priority;
    /* ... */
};

int sched_getparam (pid_t pid, struct sched_param *sp);

int sched_setparam (pid_t pid, const struct sched_param *sp);

```

Интерфейс `sched_getscheduler()` возвращает только политику планирования без связанных с ней параметров. Вызов `sched_getparam()` возвращает через `sp` параметры планирования процесса с идентификатором `pid`:

```

struct sched_param sp;
int ret;

ret = sched_getparam (0, &sp);
if (ret == -1) {
    perror ("sched_getparam");
    return 1;
}
printf ("Приоритет равен %d\n", sp.sched_priority);

```

Если `pid` равен 0, возвращаются параметры вызывающего процесса. В случае успеха вызов возвращает 0. При неудаче возвращается -1 и `errno` заполняется соответствующим образом.

Поскольку `sched_setscheduler()` также устанавливает некоторые сопредельные параметры планирования, `sched_setparam()` полезен только для изменения параметров в дальнейшем:

```

struct sched_param sp;
int ret;

sp.sched_priority = 1;
ret = sched_setparam (0, &sp);
if (ret == -1) {
    perror ("sched_setparam");
    return 1;
}

```

В случае успеха параметры планирования процесса, определенного через `pid`, устанавливаются согласно указанному в `sp` и вызов возвращает 0. При неудаче вызов возвращает -1, а `errno` заполняется соответствующим образом.

Если мы запустим два вышеупомянутых фрагмента кода по порядку, то получим на выходе следующее:

```
Приоритет равен 1
```

В примере снова предполагается, что 1 — допустимое значение приоритета. Это так, но в реальных приложениях следует провести дополнительную проверку. Сейчас мы узнаем, как определить допустимые пределы величины приоритетов.

Коды ошибок

В случае ошибки возможны четыре значения `errno`:

- `EFAULT` — указатель `sp` определяет недопустимую или недоступную область памяти;
- `EINVAL` — величина, указанная в `sp`, не имеет смысла для данной политики (касается только `sched_getparam()`);
- `EPERM` — вызывающий процесс не имеет необходимых характеристик;
- `ESRCH` — отсутствует работающий процесс с указанной величиной `pid`.

Определение диапазона допустимых значений приоритета

В предыдущих примерах в системных вызовах жестко прописывались значения приоритета. POSIX не гарантирует, что какие-то приоритеты планирования непременно будут существовать в данной системе, кроме указания, что между минимальной и максимальной величинами должно быть как минимум 32 приоритета. Как упоминалось в предыдущем разделе, Linux реализует пределы от 1 до 99 включительно для двух политик планирования реального времени. В хорошо написанной переносимой программе обычно имеется собственный диапазон приоритетов, который накладывается на системный. Получается, что, если вы хотите запустить процессы на четырех разных уровнях приоритета реального времени, вы динамически определяете диапазон приоритетов и выбираете четыре величины.

Linux предоставляет два системных вызова для получения диапазона допустимых значений. Один возвращает минимальное значение, другой — максимальное:

```
#include <sched.h>
```

```
int sched_get_priority_min (int policy);
```

```
int sched_get_priority_max (int policy);
```

В случае успеха вызов `sched_get_priority_min()` возвращает минимальное, а `sched_get_priority_max()` — максимальное допустимое значение приоритета, связанное с политикой планирования, определенной в `policy`. При неудаче оба вызова возвращают `-1`. Единственная возможная ошибка — когда значение `policy` недопустимо; `errno` принимает значение `EINVAL`.

Использование вызовов простое:

```
int min, max;
```

```
min = sched_get_priority_min (SCHED_RR);
if (min == -1) {
    perror ("sched_get_priority_min");
    return 1;
}
```

```
max = sched_get_priority_max (SCHED_RR);
if (max == -1) {
```



```

        perror ("sched_get_priority_max");
        return 1;
    }

    printf ("Диапазон приоритетов дл SCHED_RR равен %d - %d\n", min, max);

```

В стандартной системе Linux данный фрагмент кода выдаст следующий результат:

Диапазон приоритетов для SCHED_RR равен 1 - 99

Как было сказано выше, чем больше численная величина приоритета, тем выше приоритет. Чтобы присвоить процессу высший приоритет в его политике планирования, можно проделать следующее:

```

/*
 * set_highest_priority – устанавливает для процесса, определенного pid,
 * максимально допустимое значение приоритета планирования.
 * позволенное его политикой планирования. Если pid равен 0, устанавливается
 * приоритет текущего процесса.
 *
 * Возвращает 0 в случае успеха
 */
int set_highest_priority (pid_t pid)
{
    struct sched_param sp;
    int policy, max, ret;

    policy = sched_getscheduler (pid);
    if (policy == -1)
        return -1;

    max = sched_get_priority_max (policy);
    if (max == -1)
        return -1;

    memset (&sp, 0, sizeof (struct sched_param));
    sp.sched_priority = max;
    ret = sched_setparam (pid, &sp);

    return ret;
}

```

Обычно программы извлекают минимальную и максимальную величину в системе, а затем используют приращения, равные 1 (то есть $\text{max}-1$, $\text{max}-2$ и т. д.), чтобы назначить желаемые приоритеты.

sched_rr_get_interval()

Как уже обсуждалось ранее, процессы SCHED_RR ведут себя так же, как и SCHED_FIFO, кроме того, что планировщик назначает этим процессам кванты времени. Когда

процесс SCHED_RR исчерпывает свой квант времени, планировщик перемещает этот процесс в конец списка запуска процессов с тем же самым приоритетом. Таким образом, все процессы SCHED_RR с одинаковым приоритетом выполняются по очереди наподобие вращения карусели. Процессы с более высоким приоритетом (а для SCHED_FIFO — процессы с равным или более высоким приоритетом) всегда будут замещать выполняющийся SCHED_RR процесс независимо от того, потрачен его квант времени или нет.

POSIX определяет интерфейс для получения размера кванта времени данного процесса:

```
#include <sched.h>
```

```
struct timespec {
    time_t tv_sec; /* секунды */
    long tv_nsec; /* наносекунды */
};
```

```
int sched_rr_get_interval (pid_t pid, struct timespec *tp);
```

В случае успеха вызов с ужасным названием sched_rr_get_interval() сохраняет в структуре timespec, указанной tp, продолжительность кванта времени процесса, определенного через pid, и возвращает 0. При неудаче вызов возвращает -1, а errno присваивается соответствующее значение.

В соответствии с POSIX эта функция требуется только для работы с процессами SCHED_RR. В Linux, однако, с ее помощью можно получить длину временного кванта любого процесса. Совместимые приложения могут предполагать, что данная функция работает только с процессами типа карусели; однако программы, специфичные для Linux, могут использовать этот вызов в случае необходимости. Вот пример:

```
struct timespec tp;
int ret;

/* получение длины временного кванта текущей задачи */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1) {
    perror ("sched_rr_get_interval");
    return 1;
}

/* преобразование секунд и наносекунд в миллисекунды */
printf("Наш временной квант равен %.21fmilliseconds\n",
      (tp.tv_sec * 1000.0f) + (tp.tv_nsec / 1000000.0f));
```

Если процесс работает в классе FIFO, и tv_sec, и tv_nsec равны 0, что означает бесконечность.

В случае ошибки возможны следующие значения errno:

- EFAULT — указатель tp определяет недопустимую или недоступную область памяти;

- EINTR — величина `pid` имеет недопустимое значение (например, отрицательное);
- ESRCH — отсутствует работающий процесс с указанной величиной `pid` (хотя она и валидна).

Предосторожности при работе с процессами реального времени

Из-за особенностей природы процессов реального времени разработчикам необходимо соблюдать осторожность, разрабатывая и отлаживая такие программы. Если этого не сделать, программа может выйти из-под контроля. Любой привязанный к процессору цикл в программе реального времени — то есть любой фрагмент кода, который не блокируется, — будет продолжать выполняться до бесконечности, пока не появится процесс с более высоким приоритетом.

Следовательно, разработка программ реального времени требует внимательно-сти. Такие программы безраздельно властвуют в компьютере и легко могут повредить систему. Вот несколько советов и рекомендаций.

- Помните, что любые привязанные к процессору циклы будут выполняться до завершения без перерывов, если в системе нет процесса с более высоким приоритетом. Если цикл бесконечный, система может перестать отвечать на запросы.
- Процессы реального времени выполняются ценой работы всего остального в системе, поэтому их разработке необходимо уделять особое внимание. Позаботьтесь, чтобы остальная система не испытывала недостатка процессорного времени.
- Будьте очень осторожны с активным ожиданием. Если процесс реального времени активно ожидает ресурсов, занятых процессом с более низким приоритетом, он всегда будет находиться в состоянии активного ожидания.
- При разработке процесса реального времени необходимо держать терминал открытым, чтобы он выполнялся как процесс реального времени с более высоким приоритетом, чем у процесса, который вы разрабатываете. При необходимости терминал сможет среагировать на ваши действия и прекратить вышедший из-под контроля процесс реального времени (если терминал находится в пассивном состоянии, ожидая ввода с клавиатуры, он не оказывает на работу процессов реального времени никакого воздействия).
- Утилита `chrt`, являющаяся частью набора инструментов `util-linux`, упрощает получение и изменение атрибутов реального времени других процессов. С ее помощью можно без труда запускать любые программы с планированием реального времени, например упомянутый выше терминал, или менять приоритеты реального времени у существующих приложений.

Детерминизм

Для процессов реального времени очень важен детерминизм. В программировании реального времени действие является *детерминированным*, если при одних и тех

же исходных данных выдается один и тот же результат за одно и то же время. Современные компьютеры очень далеки от детерминизма: множество уровней кэша (с непредсказуемыми попаданиями и промахами), несколько процессоров, разбивка на страницы, подкачка и многозадачность делают любые оценки длительности того или иного действия невозможными. Словом, мы достигли точки, когда почти каждое действие (кроме доступа к жесткому диску) выполняется «невероятно быстро», но одновременно очень сложно точно определить, сколько именно времени займет та или иная операция.

Приложения реального времени часто стараются ограничить непредсказуемость в целом и, в частности, задержки в худшем случае. Далее мы рассмотрим два метода, которые можно применить для этого.

Запись данных и блокировка памяти на случай сбоя

Представим себе картину: срабатывает сигнал аппаратуры специального монитора приближения межконтинентальных баллистических ракет и драйвер устройства немедленно начинает копирование данных с оборудования в ядро. Драйвер обнаруживает, что процесс находится в пассивном состоянии, так как он заблокирован в отсутствие данных, которые должен получить от ядра. Драйвер дает ядру команду активизировать процесс. Ядро, заметив, что этот процесс запущен с политикой планирования реального времени и высоким приоритетом, тут же прекращает работу текущих процессов и перезагружается, чтобы немедленно запустить процесс реального времени. Планировщик запускает процесс реального времени и переключает контекст в адресное пространство этого процесса. Процесс наконец запущен. Все эти действия заняли около 0,3 миллисекунды, что неплохо в условиях максимально допустимой задержки в 1 миллисекунду.

Теперь, в пользовательском пространстве, процесс реального времени замечает приближающуюся ракету и начинает обрабатывать ее траекторию. Когда параметры баллистики вычислены, процесс реального времени активизирует развертывание системы противоракетной обороны. Всего 0,1 миллисекунды было затрачено — достаточно быстро, чтобы получить ответ системы противоракетной обороны и сохранить множество жизней. Однако — о нет! — код системы противоракетной обороны был переправлен на диск. Происходит ошибка обращения к несуществующей странице, процессор переключается снова в режим ядра, которое иницирует ввод-вывод с жестким диском, чтобы восстановить данные. Планировщик переводит процесс в пассивный режим до момента, когда ошибка отсутствия страницы будет исправлена. На все это потрачено несколько секунд. Слишком долго.

Ясно, что разбивка на страницы и подкачка представляют собой крайне недетерминированный подход, который может нарушить весь порядок работы процессов реального времени. Чтобы предотвратить катастрофу, подобную описанной выше, приложения реального времени часто «блокируют» или «жестко прописывают» все страницы из своего адресного пространства в физическую память, сохраняют их там и не позволяют системе снова перебросить их на диск. После того как страницы были заперты в памяти, ядро больше не сможет перебросить их снова

на диск. Таким образом, никакие операции со страницами не приведут к ошибкам доступа. Большинство приложений реального времени запирают часть или все свои страницы в физической памяти.

Linux предоставляет интерфейсы и для предаварийной записи, и для блокировки данных. В гл. 4 были рассмотрены интерфейсы для предаварийной записи, а в гл. 9 мы обсудим интерфейсы для блокировки данных в физической памяти.

Процессы реального времени и привязка к процессору

Вторым камнем преткновения для процессов реального времени является многозадачность. Хотя ядро Linux работает на основе вытесняющей многозадачности, его планировщик не всегда способен немедленно перепланировать один процесс в пользу другого. Иногда запущенный в настоящее время процесс выполняется внутри критической области ядра, и планировщик не может перекрыть его, пока он не покинет эту область. Если процесс, ожидающий запуска, является процессом реального времени, данная задержка может быть неприемлемой и временной предел функционирования очень быстро окажется нарушенным.

Таким образом, многозадачность добавляет в систему недетерминированность, такую же непредсказуемую, как и разбивка на страницы. Решение в отношении мультизадачности то же — избегать ее. Конечно, маловероятно, что удастся избавиться от всех остальных процессов в системе. Если бы это было возможно в вашем окружении, то вам, вероятно, вовсе не понадобилась бы Linux — простой пользовательской операционной системы было бы достаточно. Если же, однако, в вашей системе много процессоров, вы можете выделить один или несколько из них для вашего процесса (процессов) реального времени. В результате этого вам удастся оградить ваши процессы реального времени от мультизадачности.

Мы обсудили системные вызовы для управления привязкой процессов к процессору ранее в этой главе. Возможная оптимизация для приложений реального времени состоит в том, что один процессор выделяется для каждого процесса реального времени и позволяет остальным процессам разделить между собой ресурсы оставшихся процессоров.

Самый простой путь достигнуть этого — модифицировать программу `init` в Linux, `SysVinit`¹, перед тем как она начнет процесс загрузки, примерно следующим образом:

```
cpu_set_t set;
int ret;

CPU_ZERO(&set); /* очистить все процессоры */
ret = sched_getaffinity(0, sizeof(cpu_set_t), &set);
if (ret == -1) {
    perror("sched_getaffinity");
    return 1;
}
```

¹ Исходный код `SysVinit` можно найти на FTP-сервере <ftp://ftp.cistron.nl/pub/people/miquels/sysvinit>. Он лицензирован согласно GNU General Public License v2.

```

}

CPU_CLR (1, &set); /* запретить использование процессора 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
    perror ("sched_setaffinity");
    return 1;
}

```

Этот фрагмент кода собирает имеющийся набор разрешенных в программе `init` процессоров, которые должны быть ею включены. После этого один из процессоров (с номером 1) удаляется из набора, а список разрешенных процессоров обновляется.

Набор разрешенных процессоров наследуется от родительского процесса к дочернему, а `init` является верховным родительским процессом для всех, поэтому все процессы в системе запускаются с использованием этого набора разрешенных процессоров. Следовательно, ни один процесс не будет запущен на процессоре 1.

Затем модернизируем наш процесс реального времени так, чтобы он работал только на процессоре 1:

```

cpu_set_t set;
int ret;

CPU_ZERO (&set); /* очистить набор процессоров */
CPU_SET (1, &set); /* разрешить процессор 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
    perror ("sched_setaffinity");
    return 1;
}

```

В результате процесс реального времени запустится только на процессоре 1, а все другие процессы — на оставшихся.

Лимиты ресурсов

Ядро Linux накладывает на процессы ряд *лимитов ресурсов*. Они достаточно жестко ограничивают количество ресурсов ядра, которые может использовать процесс, — это, например, количество открытых файлов, страниц памяти, ожидающих сигналов, и т. д. Лимиты строго соблюдаются. Ядро не допускает никаких действий, в результате которых они будут нарушены. Например, если открытие файла приведет к тому, что у процесса окажется больше открытых файлов, чем позволено, вызов `open()` вернет ошибку¹.

¹ В этом случае вызов присвоит переменной `errno` значение `EMFILE`, указывающее, что процесс достиг ограничения ресурсов, определяющего максимальное количество открытых файлов. Системный вызов `open()` был описан в гл. 2.

Linux предоставляет два системных вызова для управления лимитами ресурсов. Оба интерфейса стандартизированы POSIX, но Linux поддерживает несколько лимитов в дополнение к прописанным в стандарте. Пределы можно проверить с помощью `getrlimit()` и установить посредством `setrlimit()`:

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit {
    rlim_t rlim_cur; /* мягкий лимит */
    rlim_t rlim_max; /* жесткий лимит */
};

int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim);
```

Целочисленные постоянные, такие как `RLIMIT_CPU`, представляют ресурсы. Структура `rlimit` представляет имеющиеся пределы. Структура определяет два ограничения: *мягкий лимит* и *жесткий лимит*. Ядро соблюдает мягкие лимиты ресурсов для процессов, но сам процесс может свободно менять свой мягкий лимит, присваивая ему любую величину от 0 до значения жесткого лимита. Процесс, не имеющий свойства `CAP_SYS_RESOURCE` (то есть любой процесс без прав `root`), может только снизить свой жесткий лимит. Непривилегированный процесс не может поднять свой жесткий лимит даже до предыдущей более высокой величины; уменьшение жесткого лимита необратимо. Привилегированный процесс может дать жесткому лимиту любое допустимое значение.

Представленные в результате пределы будут зависеть от конкретного ресурса. Если `resource` имеет значение `RLIMIT_FSIZE`, например, то предел представляет собой максимальный размер файла в байтах, который может создать процесс. В данном случае, если `rlim_cur` равен 1024, процесс не может создать или увеличить размер файла более чем до 1 Кбайт.

Все ресурсные лимиты имеют два особых значения: 0 и бесконечность. Первое из них вообще исключает возможность использования данного ресурса. Например, если `RLIMIT_CORE` равен 0, то ядро вообще не сможет создать файл ядра. Соответственно, последнее ликвидирует любые ограничения на использование данного ресурса. Бесконечность обозначается ядром с помощью присваивания `RLIM_INFINITY` значения -1 (что может вызвать некоторую путаницу, так как -1 — это еще и значение, возвращаемое при ошибке). Если `RLIMIT_CORE` равно бесконечности, ядро может создавать файлы ядра любого размера. Функция `getrlimit()` помещает текущие мягкий и жесткий лимиты для ресурса, определенного как `resource`, в структуру, определенную как `rlim`. В случае успеха вызов возвращает 0. При неудаче возвращается -1 и переменная `errno` принимает соответствующее значение.

Соответственно, функция `setrlimit()` устанавливает мягкий и жесткий лимиты, связанные с `resource`, равными величинам, на которые указывает `rlim`. В случае успеха вызов возвращает 0 и ядро обновляет пределы ресурсов, как было указано.

В случае неудачи вызов возвращает -1 и переменная `errno` принимает соответствующее значение.

Linux в настоящее время предоставляет 16 ресурсных лимитов.

- `RLIMIT_AS` — ограничивает максимальный размер адресного пространства процесса в байтах. Любые попытки выйти за пределы этого адресного пространства (через вызовы `mmap()`, `brk()` и др.) будут безуспешными и возвратят ошибку `ENOMEM`. Если стек процесса, увеличиваясь, выходит за пределы этого лимита, ядро посылает процессу сигнал `SIGSEGV`. Как правило, этот лимит равен `RLIM_INFINITY`.
- `RLIMIT_CORE` — определяет максимальный размер файлов ядра в байтах. Если он не равен 0, файлы ядра, размер которых превышает лимит, сжимаются до его значения. Если максимальный размер равен 0, процесс не может создавать файлы ядра.
- `RLIMIT_CPU` — определяет максимальное процессорное время, которое может использовать процесс, в секундах. Если процесс выполняется дольше указанного лимита, ядро посылает сигнал `SIGXCPU`, который процессы могут получить и обработать. Портативные программы при получении этого сигнала завершают работу, так как в стандарте POSIX нет указаний, какие действия должны предприниматься дальше. Некоторые системы могут прерывать процесс, если он продолжает выполняться. Linux не прекращает работу процесса, но продолжает отправлять ему сигналы `SIGXCPU` каждую секунду. Однако, как только процесс достигает жесткого лимита, он получает `SIGKILL` и прерывается.
- `RLIMIT_DATA` — контролирует максимальный размер сегмента данных и кучи процесса в байтах. Попытки выйти за пределы сегмента данных и превысить данный лимит путем вызова `brk()` ни к чему не приведут и вернут `ENOMEM`.
- `RLIMIT_FSIZE` — определяет максимальный размер в байтах файла, который может создать процесс. Если процесс увеличивает размер файла с превышением данного размера, ядро посылает процессу сигнал `SIGXFSZ`. По умолчанию этот сигнал завершает работу процесса. Однако процесс может перехватить и обработать этот сигнал, в результате чего системный вызов завершается ошибкой и возвращает `EFBIG`.
- `RLIMIT_LOCKS` — контролирует максимальное количество блокировок файлов, которые может держать процесс (подробнее блокировки файлов будут рассмотрены в гл. 8). Как только этот лимит достигнут, дальнейшие попытки установить дополнительные файловые блокировки будут безуспешны и вернут `ENOLCK`. В ядре Linux 2.4.25, однако, эта функциональность удалена. В текущих версиях ядра этот лимит технически можно установить, но он ни на что не влияет.
- `RLIMIT_MEMLOCK` — указывает максимальное количество байтов памяти, которое процесс без свойства `CAP_SYS_IPC` (по сути, процесс без прав `root`) может заблокировать в памяти через вызовы `mlock()`, `mlockall()` или `shmctl()`. Если этот предел достигнут, вызовы будут безуспешны и возвратят `EPERM`. На практике реальный лимит округляется в меньшую сторону до целого числа страниц. Процессы, обладающие `CAP_SYS_IPC`, могут блокировать любое число страниц в памяти, на них этот лимит не распространяется. До появления версии ядра 2.6.9

этот предел означал максимальное количество байтов, которое процесс с `CAP_SYS_IPC` может запереть в памяти, а непривилегированные процессы не могли запирать никаких страниц вообще. Этот лимит не является частью POSIX; он был представлен в BSD.

- `RLIMIT_MSGQUEUE` — указывает максимальное количество байтов, которое пользователь может выделить для серии сообщений POSIX. Если вновь созданная серия сообщений нарушает данное ограничение, `mq_open()` выдает ошибку и возвращает `ENOMEM`. Этот лимит не является частью POSIX; он был добавлен в версию ядра 2.6.8 и специфичен для Linux.
- `RLIMIT_NICE` — обозначает максимальное значение, до которого процесс может снизить свое значение любезности (то есть повысить приоритет). Как было сказано ранее в этой главе, обычно процессы могут только повышать свое значение любезности (снижать приоритет). Этот лимит позволяет администратору определить максимальный уровень (минимальное значение любезности), до которого процессы могут повысить свой приоритет. Возможны отрицательные значения любезности, поэтому ядро интерпретирует значение как `20 - rlim_cur`. Таким образом, если установлен лимит, равный 40, то процессу разрешается понизить свое значение любезности до значения, равного -20 (это и будет означать максимальный для него приоритет). Впервые данный лимит появился в версии ядра 2.6.12.
- `RLIMIT_NOFILE` — определяет значение, на единицу превышающее максимальное число файловых дескрипторов, которые процесс может держать открытыми. Попытки выйти за этот предел будут безуспешными, и соответствующий системный вызов вернет `EMFILE`. Этот лимит также можно указать под именем `RLIMIT_OFIL`, которое используется в BSD.
- `RLIMIT_NPROC` — определяет максимальное количество процессов, которое может выполняться в каждый момент времени от имени одного пользователя. Попытки выйти за этот предел будут безуспешными, а системный вызов `fork()` вернет ошибку `EAGAIN`. Этот лимит не входит в POSIX и впервые был представлен в BSD.
- `RLIMIT_RSS` — означает максимальное количество страниц памяти, которое процессор может выделить (известно как RSS (resident set size — «размер резидентной части»)). Только ранние версии ядра 2.4 поддерживали этот лимит. Текущие версии позволяют его устанавливать, но не поддерживают. Этот лимит не входит в POSIX и был представлен в BSD.
- `LIMIT_RTIME` — определяет лимит процессорного времени в микросекундах, которое может использовать процесс реального времени без блокирующего системного вызова. Как только процесс выполняет блокирующий системный вызов, процессное время сбрасывается до 0. Это предотвращает выход процесса реального времени из-под контроля и повреждение системы. Лимит был добавлен в ядре Linux версии 2.6.25 и является специфичным для этой операционной системы.
- `LIMIT_RTPRIO` — указывает максимальный уровень приоритета процесса реального времени без свойства `CAP_SYS_NICE` (то есть процесса без прав `root`), который тот

может запросить. Обычно непривилегированные процессы не могут запросить никакой класс планирования реального времени. Лимит был добавлен в ядре Linux версии 2.6.12 и является специфичным для этой операционной системы.

- `LIMIT_SIGPENDING` — указывает максимальное количество сигналов (стандартных и реального времени), которые могут находиться в очереди для данного пользователя. Попытки поместить в очередь дополнительные сигналы будут безуспешны, и системные вызовы, такие как `sigqueue()`, вернут `EAGAIN`. Обратите внимание, что всегда можно, независимо от этого лимита, поместить в очередь еще один экземпляр неожиданного сигнала. Таким образом, система всегда может получить сигнал `SIGKILL` или `SIGTERM`. Этот лимит не входит в POSIX, он специфичен для Linux.
- `RLIMIT_STACK` — определяет максимальный размер стека процесса в байтах. При превышении данного лимита отправляется сигнал `SIGSEGV`.

Ядро управляет лимитами ресурсов отдельно для каждого процесса. Дочерний процесс наследует свои лимиты от родителя во время ветвления; лимиты поддерживаются и для вызова `exec`.

Лимиты по умолчанию

Лимиты по умолчанию, доступные процессу, зависят от трех переменных: начального мягкого лимита, начального жесткого лимита и администратора вашей системы. Ядро предписывает мягкий и жесткий лимиты; они перечислены в табл. 6.1. Ядро устанавливает эти лимиты для процесса `init`, и, поскольку все дочерние процессы наследуют свои лимиты от родительских, все последующие процессы наследуют мягкие и жесткие лимиты `init`.

Таблица 6.1. Мягкий и жесткий лимиты ресурсов по умолчанию

Лимит ресурса	Мягкий лимит	Жесткий лимит
<code>RLIMIT_AS</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_CORE</code>	0	<code>RLIM_INFINITY</code>
<code>RLIMIT_CPU</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_DATA</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_FSIZE</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_LOCKS</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_MEMLOCK</code>	8 страниц	8 страниц
<code>RLIMIT_MSGQUEUE</code>	800 Кбайт	800 Кбайт
<code>RLIMIT_NICE</code>	0	0
<code>RLIMIT_NOFILE</code>	1024	1024
<code>RLIMIT_NPROC</code>	0 (означает отсутствие ограничений)	0 (означает отсутствие ограничений)
<code>RLIMIT_RSS</code>	<code>RLIM_INFINITY</code>	<code>RLIM_INFINITY</code>
<code>RLIMIT_RT�RIO</code>	0	0
<code>RLIMIT_SIGPENDING</code>	0	0
<code>RLIMIT_STACK</code>	8 Мбайт	<code>RLIM_INFINITY</code>

Эти лимиты могут быть изменены двумя путями.

- Любой процесс может свободно повысить мягкий лимит от 0 до значения жесткого лимита или снизить жесткий. Дочерние процессы наследуют во время ветвления обновленные величины лимитов.
- Привилегированные процессы могут устанавливать любое значение жесткому лимиту. Дочерние процессы наследуют во время ветвления обновленные величины лимитов.

Маловероятно, что процесс `root` в обычной линии преемственности процессов будет менять какие-либо жесткие лимиты. Следовательно, изменения лимитов будут происходить скорее по первому варианту. Действительно, реальные лимиты какого-либо процесса обычно устанавливаются пользовательской оболочкой, которой системный администратор настраивает способность передавать различные лимиты. В оболочке `Boogie-again (bash)`, например, администратор может проделать это с помощью команды `ulimit`. Обратите внимание, что администратор не обязательно будет уменьшать лимиты; он может также увеличивать мягкие лимиты до значения жестких, придавая пользователю более разумные значения пределов по умолчанию. Это часто делается с помощью `RLIMIT_STACK`, который устанавливается в значение `RLIM_INFINITY` во многих системах.

Установка и проверка лимитов

Вооружившись знаниями о различных лимитах ресурсов, рассмотрим проверку и установку лимитов. Получить величину ресурсного лимита очень просто:

```
struct rlimit rlim;
int ret;

/* получение лимитов на размеры файлов ядра */
ret = getrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
    perror ("getrlimit");
    return 1;
}

printf ("Лимиты RLIMIT_CORE: мягкий=%ld жесткий=%ld\n",
        rlim.rlim_cur, rlim.rlim_max);
```

Компиляция этого фрагмента кода в большую программу и запуск приведут к следующему результату:

Лимиты RLIMIT_CORE: мягкий=0 жесткий=-1.

Как видите, мягкий лимит равен 0, а жесткий лимит — бесконечности (-1 означает `RLIM_INFINITY`). Таким образом, мы можем установить новую величину мягкого лимита любого размера. Следующий пример устанавливает максимальный размер файлов ядра 32 Мбайт:

```
struct rlimit rlim;
```

```
int ret;

rlim.rlim_cur = 32 * 1024 * 1024; /* 32 Мбайт */
rlim.rlim_max = RLIM_INFINITY; /* не менять */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
    perror ("setrlimit");
    return 1;
}
```

Коды ошибок

В случае ошибки возможны три значения `errno`:

- `EFAULT` — область памяти, на которую указывает `rlim`, недопустима или недоступна;
- `EINVAL` — величина, определенная через `resource`, недопустима или `rlim.rlim_cur` больше, чем `rlim.rlim_max` (касается только `setrlimit()`);
- `EPERM` — вызывающий процесс не имеет прав `CAP_SYS_RESOURCE`, но пытается повысить жесткий лимит.

На этом мы закончим изучение работы процессов и перейдем к поточности.

7 Поточность

Поточность — это создание и управление множеством исполняемых элементов внутри одного процесса. Поток является единственным источником программных ошибок за счет внедрения конкурентности данных и клинчей (взаимных блокировок). По теме поточности можно написать — и уже написаны — целые книги. Эти работы обычно сосредотачиваются на множестве интерфейсов, в частности на библиотеках потоков. Мы рассмотрим основы поточности API в Linux, чтобы разобраться, как поточность вписывается в системный инструментарий программиста; когда нужно использовать потоки и, что важнее, когда не нужно; какие шаблоны проектирования могут помочь спланировать и построить поточные приложения; что такое конкурентность данных и как ее можно предотвратить.

Бинарные модули, процессы и потоки

Бинарные модули — это программы, находящиеся в пассивном состоянии в хранилище и скомпилированные в формате, приемлемом для данной операционной системы и машинной архитектуры, готовые к запуску, но пока бездействующие. *Процессы* — это абстракции операционной системы, представляющие собой данные модули в действии: загруженный двоичный код, виртуальная память, ресурсы ядра, такие как открытые файлы, связанный пользователь и т. д. *Поток* — это элемент выполнения внутри процесса: виртуальный процессор, стек или статус программы. Иными словами, процессы — это выполняющиеся бинарные модули, и потоки являются наименьшими исполняемыми элементами, предусмотренными планировщиком операционной системы.

Процесс *содержит* один или несколько потоков. Если процесс содержит только один поток, то в процессе находится лишь один исполняемый элемент и только одна задача выполняется в единицу времени. Такие процессы можно назвать *однопоточными*. Это классические процессы UNIX. Если процесс содержит более одного потока, значит одновременно выполняется несколько действий. Такие процессы называются *многопоточными*.

Современные операционные системы представляют в пользовательском пространстве две основные виртуальные абстракции — виртуальную память и виртуальный процессор. Совместно они создают для каждого выполняющегося процесса

иллюзию, что он один пользуется всеми ресурсами машины. Виртуальная память предоставляет каждому процессу уникальный вид памяти, который якобы соответствует оперативной памяти или хранилищу на диске (это достигается с помощью разбивки на страницы). Оперативная память системы может в действительности содержать данные сотни разных процессов, запущенных одновременно, но каждому из них кажется, что вся память принадлежит только ему. Виртуальный процессор позволяет каждому процессу работать так, словно он один в системе, а операционная система скрывает, что несколько процессов работают одновременно на (возможно) нескольких процессорах.

Виртуальная память связывается с процессом, а не с потоком. Таким образом, каждый процесс обладает уникальным видением памяти, но все потоки в данном процессе *разделяют* эту память. И наоборот, виртуальный процессор связан именно с потоками, а не с процессами. Каждый поток — индивидуально запланированное действие, позволяющее каждому процессу выполнять более одного действия в каждый момент времени. Многие программисты сочетают две иллюзии виртуальной памяти и виртуального процессора, но потоки требуют разделять их. У потоков, как и у процессов, должна быть иллюзия, что процессор (или несколько) целиком принадлежит им. Потоки, в отличие от процессов, не нуждаются в иллюзии, что им принадлежит вся память, — все потоки внутри процесса совместно используют имеющееся в их распоряжении адресное пространство.

Многопоточность

В чем суть потоков? Процессы обычно нужны потому, что они являются абстракцией запущенной программы. Однако зачем разделять элементы выполнения и вводить понятие потоков? Вот шесть основных преимуществ многопоточности.

- *Программная абстракция.* Разделение работы и назначение каждому дивизиону исполняемого элемента (потока) — естественный подход при решении многих задач. В шаблоны проектирования, которые используют этот подход, входят «один поток на одно соединение» и «пул потоков». Программисты обычно считают эти шаблоны полезными и интуитивно понятными. Некоторые, однако, рассматривают эти потоки как антишаблоны. Неподражаемый Алан Кокс (Alan Cox) резюмировал этот подход цитатой «потоки для людей, которые не могут проектировать конечные машины». Таким образом, в теории нет программистских задач, которые разрешимы с помощью потоков и не решаемы с помощью конечных автоматов.
- *Параллелизм.* В машинах с несколькими процессорами потоки обеспечивают эффективный способ достижения настоящего параллелизма. Каждый поток получает собственный виртуальный процессор, где его планирование не зависит ни от чего, поэтому несколько потоков могут выполняться на нескольких процессорах в одно и то же время, улучшая пропускную способность системы. Таким образом, максима «потоки для людей, которые не могут проектировать конечные автоматы» не применяется к достижению параллелизма, так как потоков не может быть больше, чем процессоров.

- *Улучшение реагирования.* Даже на однопроцессорной машине многопоточность может ускорить реагирование процессов. При однопоточном процессе длительно выполняемая операция может помешать приложению среагировать на какое-то действие пользователя, которому может показаться, что приложение «зависло». При многопоточности такие операции могут быть делегированы рабочим потокам, позволив хотя бы одному потоку оставаться способным реагировать на действия пользователя и выполнять операции ввода-вывода.
- *Блокировка ввода-вывода.* Это связано с предыдущим пунктом. В отсутствие потоков блокирование ввода-вывода останавливает весь процесс. Это может пагубно влиять как на общую пропускную способность, так и на время задержки. В многопоточном процессе индивидуальные потоки могут заблокироваться, ожидая ввода-вывода, пока другие потоки продолжают работу. Асинхронность и неблокирование ввода-вывода могут быть альтернативными решениями этой задачи.
- *Переключение контекста.* Затраты на переключение от одного потока к другому внутри того же самого процесса значительно меньше, чем перемещение контекста от одного процесса к другому.
- *Экономия памяти.* Потоки обеспечивают эффективный способ разделения памяти, одновременно позволяя выполнение нескольких элементов. Таким образом они являются альтернативой нескольким процессам.

По вышеперечисленным причинам поточность довольно распространена в различных операционных системах и их приложениях. В некоторых системах, таких как Android, почти каждый процесс действует с помощью нескольких потоков. 10–15 лет назад утверждение «потоки нужны людям, которые не могут программировать конечные автоматы» было правдой, поскольку большинство преимуществ поточности можно было реализовать с помощью других средств, таких как неблокирование ввода-вывода, и, да, конечных автоматов. Сегодня количество процессоров даже на небольших машинах — мобильные устройства сейчас также имеют по несколько процессоров — и технологии (многоядерная и одновременная многопоточность) создают необходимость в поточности для максимального увеличения пропускной способности в системном программировании. В наше время невозможно представить себе высокопроизводительный веб-сервис, работающий без нескольких потоков на нескольких ядрах.

Переключение контекста: процессы или потоки

Одно из преимуществ производительности происходит от уменьшения затрат на переключение от потока к потоку внутри одного и того же процесса (*внутрипроцессорное переключение*). В любой системе ресурсные издержки внутрипроцессорного переключения меньше, чем переключения между процессами; первое в любом случае входит во второе. Эта разница в затратах особенно велика в системах, отличающихся от Linux, где процессы являются абстракциями, поэтому во многих системах потоки называются легковесными процессами.

В Linux затраты на переключение между процессами невелики, но стоимость переключения внутри процесса близка к нулю: по сути, она равна входу и выходу из ядра. Процессы незатратны, но потоки все равно экономнее.

Машинная архитектура приводит к большим издержкам на переключение, которые не касаются процессов, так как они переключаются из одного виртуального адресного пространства в другое. На x86, например, буфер ассоциативной трансляции (TLB), являющийся отображением кэша физической оперативной памяти, должен быть очищен от виртуального адресного пространства. При определенных рабочих нагрузках потери TLB весьма значительно ухудшают системную производительность. Самый экстремальный пример наблюдается на машинах ARM, где необходимо очищать содержимое процессорного кэша! Потоков не касаются эти затраты, поскольку переключение от потока к потоку не требует очищения виртуального адресного пространства.

Издержки многопоточности

Несмотря на все свои преимущества, многопоточность все-таки сопряжена с некоторыми издержками. В частности, одна из самых страшных, самых коварных ошибок в истории программирования была связана именно с поточностью. Разработка, написание, понимание и — самое поучительное — отладка многопоточных программ значительно сложнее, чем однопоточных процессов.

Источник опасности потоков непосредственно заключается в смысле их существования: несколько виртуальных процессоров существуют при только одном экземпляре виртуальной памяти. Иными словами, при многопоточном процессе выполняется одновременно несколько действий (*параллелизм*), разделяющих одну и ту же область памяти. Таким образом, неизбежно, что потоки в процессе делят между собой ресурсы — скажем, им нужно считать или записать одни и те же данные. Понимание, как работает ваша программа, таким образом, переходит от простого изложения последовательности выполнения инструкций к концептуализации нескольких потоков, выполняемых независимо друг от друга, с непредсказуемым выбором времени выполнения и инструкций, необходимых для правильной работы. Сбой в *синхронизации* может перевести к перекрытию вывода, некорректному выполнению и прекращению работы программы. Понимание и отладка многопоточных программ довольно сложна, поэтому крайне важно, чтобы ваша поточная модель и стратегия синхронизации были частью системного ди-зайна с самого начала.

Альтернативы многопоточности

В зависимости от ваших целей, связанных с многопоточностью, возможны альтернативы. Например, выгоды в задержке исполнения и ввода-вывода также достижимы с помощью мультиплексного ввода-вывода (см. разд. «Мультиплексный ввод-вывод» гл. 2), неблокируемого ввода-вывода (см. подразд. «Неблокирующее

считывание» разд. «Считывание с помощью `read()`» гл. 2) и асинхронного ввода-вывода (см. разд. «Синхронизированные, синхронные и асинхронные операции» гл. 4). Эти техники позволяют процессам осуществлять операции ввода-вывода, не блокируя процесс. Если же вашей целью является истинный параллелизм, то N процессов могут использовать процессор так же, как N потоков, с учетом увеличения потребления ресурсов и затрат на переключение. И наоборот, если ваша цель — экономия памяти, Linux обеспечивает средства для разделения памяти, работающие в более ограниченной манере, чем потоки.

Современные системные программисты, впрочем, не находят эти альтернативы адекватными. Асинхронный ввод-вывод, например, часто просто выводит из себя. И даже если вы можете скомпенсировать затраты на несколько процессов за счет разделения памяти и других ресурсов, затраты на переключение никуда не денутся. Таким образом, потоки широко используются не только в системном программировании, но и вообще в стеках: от ядра до приложений с графическим интерфейсом. По причине все большего распространения многоядерности использование потоков будет только учащаться.

Поточные модели

Существует несколько подходов к реализации поточности в системе, с различной степенью функциональности, обеспечиваемой ядром и пользовательским пространством. Самая простая модель заключается в том, что ядро обеспечивает свою встроенную поддержку потоков, и каждый из них напрямую отправляет в пользовательское пространство свою информацию. Такая модель называется *поточностью 1:1*, поскольку в ней присутствует соотношение 1:1 между тем, что предоставляет ядро, и тем, что получает пользователь. Эта модель также известна как *поточность на уровне ядра*, поскольку ядро является основной системной поточной модели.

Поточность в Linux, которую мы скоро обсудим в подразд. «Реализация поточности в Linux», является поточностью 1:1. Ядро Linux реализует потоки просто как процессы, разделяющие между собой ресурсы. Поточная библиотека создает новый поток через системный вызов `clone()`, и возвращаемый «процесс» напрямую управляется как концепт потока в пользовательском пространстве. Таким образом в Linux то, что является потоком в пользовательском пространстве, в целом является тем же самым и с точки зрения ядра.

Поточность на уровне пользователя

Полностью противоположная модель — *поточность $N:1$* , также называемая *поточностью на уровне пользователя*. В отличие от поточности на уровне ядра, в этой модели пользовательское пространство — основа системной поддержки поточности, поскольку именно оно реализует концепт потока. Процесс с N потоками приведет к одному процессу в ядре — вот почему модель называется $N:1$. Эта модель (почти) не требует поддержки ядра, но для нее нужно достаточно много кода

в пользовательском пространстве, включая планировщик в пользовательском пространстве для планирования потоков и механизм для отлавливания и обработки ввода-вывода без блокирования. Преимущество потоков на уровне пользователя заключается в том, что переключение контекста почти не требует ресурсов, поскольку приложение само может решать, какой поток нужно запустить и когда без участия ядра. Недостатком можно назвать то, что только один элемент в ядре поддерживает N потоков, из-за чего модель не может использовать N процессоров и, следовательно, истинный параллелизм. На современном аппаратном обеспечении это весьма значительный недостаток, особенно учитывая, что выгода от сниженной стоимости переключения контекста — ключевая величина в Linux, которая обеспечивает низкие затраты.

Хотя существуют библиотеки поточности уровня пользователя для Linux, большинство библиотек обеспечивают поточность 1:1, которую мы и обсудим далее в этой главе.

Комбинированная поточность

Что, если скомбинировать поточность уровней пользовательского пространства и ядра? Возможно ли достичь истинного параллелизма, как при модели 1:1, в сочетании с незатратным переключением контекста, как в модели $N:1$? Да, это возможно, если вы не боитесь некоторого усложнения. *Поточность $N:M$* , известная также как *комбинированная поточность*, позволит взять лучшее от обоих подходов: ядро обеспечивает встроенную поддержку потоков, в то время как пользовательское пространство реализует пользовательские потоки. Пользовательское пространство, возможно, при участии ядра, затем решает, как соотнести N пользовательских потоков с M ядерных, где $N \geq M$.

Эти подходы различаются в реализации, но обычная стратегия — не поддерживать большинство пользовательских потоков потоком на уровне ядра. Процесс может содержать тысячи пользовательских потоков, но только небольшое количество ядерных с небольшим количеством функций процессора (как минимум с одним потоком для каждого процессора, обеспечивающим полную загрузку системы) и блокировкой ввода-вывода. Как вы можете себе представить, эта модель значительно более сложна для внедрения. С учетом небольших затрат на переключение контекста в Linux большинство системных разработчиков не считают применение этого подхода оправданным и для Linux остается наиболее популярной модель 1:1.

ПРИМЕЧАНИЕ

Планировщик активации — решение, которое обеспечивает поддержку пользовательских потоков ядром, позволяя более производительную реализацию поточности $N:M$. Он зародился как академическая разработка в Университете Вашингтона и был впоследствии адаптирован для FreeBSD и NetBSD, став основой реализации поточности в них обеих. Планировщик активации позволяет пользовательскому пространству контролировать и понимать планирование процессов в ядре, что делает комбинированную модель более эффективной и исправляет несколько неполадок, которые возникают реализации без задействования ядра.

FreeBSD и NetBSD отказались от планировщика активации, отдав предпочтение простой поточности 1:1. Вы можете рассматривать это как отказ от сложности модели $N:M$ и ответ на распространение архитектуры x86, которая обеспечивает относительно эффективное переключение контекста.

Сопрограммы и фиберы

Сопрограммы и *фиберы* обеспечивают элементы выполнения еще более легкие, чем потоки (первые берут свое название от конструкции языка программирования, а вторые — от системной конструкции). И то и другое, как и пользовательские потоки, является явлением пользовательского пространства, но для их планирования и выполнения почти не требуется поддержка пользовательского пространства. Вместо этого они планируются совместно, требуя только четкого перехода от одного к другому. Сопрограммы и фиберы лишь немного отличаются от подпрограмм (обычные понятия C/C++). В принципе, вы можете рассматривать сопрограммы как частный случай подпрограмм. Сопрограммы и фиберы в большей степени нужны для контроля выполнения программы, чем для обеспечения параллелизма.

По природе своей Linux не поддерживает сопрограммы и фиберы, опять-таки потому, что в нем и так обеспечивается быстрое переключение контекста, устраняя необходимость в сложных конструкциях для значительного повышения производительности поточности ядра. Язык программирования Go обеспечивает поддержку сопрограммно-подобных конструкций в Linux на уровне языка, они называются Go-программами. Сопрограммы позволяют использовать различные программистские парадигмы и модели ввода-вывода, на которые стоит обратить внимание, хотя они и выходят за рамки данной книги.

Шаблоны поточности

Первый и самый важный шаг в построении поточного приложения — выбор поточного шаблона, который одновременно будет являться процессной моделью и моделью ввода-вывода для вашего приложения. Существует множество абстракций и деталей реализации, которые нужно учесть, но два основных шаблона программирования, из которых вам надо выбрать один, — это *поток на соединение* и *поток, управляемый событием*.

Поток на соединение

Поток на соединение — это шаблон программирования, в котором одному элементу выполнения назначается один поток, и этот поток назначается не более чем одному рабочему элементу на протяжении всей работы. Рабочим элементом мы можем назвать все, на что можно разложить работу вашего приложения: запрос, соединение и т. п. В дальнейшем обсуждении мы будем говорить «соединение», так как это наиболее общий термин при рассмотрении данного шаблона.

Другой способ описать этот шаблон — это «запуск до завершения». Поток подхватывает соединение или запрос и обрабатывает его до конца, после чего поток может обработать новый запрос заново. Таким образом, очень интересной становится реализация ввода-вывода, где и находятся основные различия между этим шаблоном и потоком, управляемым событием. При потоке на соединение блокирование ввода-вывода — в самом деле, любого ввода-вывода — допустимо, так как

соединение «владеет» потоком. Блокировка потока может затормозиться, только если соединение вызывает блок. Таким образом, шаблон потока на соединение использует ядро для обработки планирования работы и управления вводом-выводом.

В этом шаблоне количество потоков относится к деталям реализации. До сих пор мы обсуждали шаблон «поток на соединение», полагая, что всегда существует поток на каждый рабочий элемент. Это может быть правдой, но большинство реализаций стараются ограничивать количество потоков, которые могут создать. Когда количество действующих соединений (и, соответственно, количество потоков) достигает лимита, соединения либо становятся в очередь, либо запрещаются до тех пор, пока количество действующих соединений не станет ниже этого предела.

Обратите внимание, что мы не говорили о требованиях поточности относительно этой модели, но, заменив «поток» на «процесс», мы будем описывать UNIX-сервер старой школы. Стандартная для Apache модель ветвления, например, основана на этом шаблоне. Это также типичный шаблон для ввода-вывода в Java, хотя предпочтения меняются.

Поток, управляемый событием

Принцип событийного шаблона противоположен шаблону потока на соединение. Представьте себе веб-сервер. С точки зрения вычислительной мощности современное оборудование способно обработать одновременно значительное количество запросов. При шаблоне потока на соединение это означает огромное количество потоков. Потоки вызывают определенные затраты, в первую очередь ресурсов ядра и стека пользовательского пространства. Эти постоянные издержки накладывают ограничения масштабирования по количеству потоков в данном процессе, особенно в 32-битной системе (аргументы против потока на соединение в отношении 64-битной системы несколько менее серьезны, но все же очевидно, что управление событием по-прежнему считается лучшим выбором даже для 64-битной системы). Системы могут иметь вычислительные ресурсы для обработки нескольких тысяч работающих соединений, но испытывать ограничения масштабирования на запуск большого количества параллельных потоков.

В поисках альтернативы системные разработчики заметили, что большинство потоков много времени проводят в ожидании: чтения файлов, возвращения результатов базами данных, отработки удаленных процедур. Действительно, вспомните обсуждение многопоточности: используя больше потоков, чем имеется процессоров у вас в системе, вы не получите преимуществ параллелизма. Вместо этого данный способ использования потоков отражает программистскую абстракцию, простоту программирования, которые могут быть достигнуты с помощью менее формального вида модели управления.

Эти соображения и привели к появлению *событийно управляемой поточности*. Поскольку такая большая часть рабочей нагрузки потоков через соединение является простым ожиданием, отделим ожидание от потоков. Вместо этого сделаем все вводы-выводы асинхронными (см. разд. «Синхронизированные, синхронные

и асинхронные операции» гл. 4) и используем мультиплексный ввод-вывод (см. разд. «Мультиплексный ввод-вывод» гл. 2) для управления процессом на сервере. В этой модели обработка запросов преобразована в серию асинхронных запросов ввода-вывода и связанных с ними обратных вызовов. Последние могут ожидать через мультиплексный ввод-вывод; если процесс поступает именно так, он называется *циклом событий*. Когда запросы ввода-вывода возвращаются, цикл событий снова передает обратный вызов ожидающему потоку.

Как и в шаблоне потока на соединение, здесь ничего не говорится о необходимости поточности в событийном шаблоне. Действительно, событийный цикл просто может заканчиваться, когда однопоточный процесс завершил работу и выполнил обратный вызов. Потоки нужны лишь для обеспечения истинного параллелизма. В этой модели нет смысла иметь больше потоков, чем процессоров.

Популярность шаблонов то растет, то падает, но событийно управляемый шаблон в целом пользуется большим успехом при разработке многопоточного сервера. Несколько популярных альтернатив Apache, например разработанные в последние несколько лет, являются событийно управляемыми. При разработке вашей поточной системы лучше и вам отдать предпочтение событийно управляемому шаблону: асинхронный ввод-вывод, обратные вызовы, событийный цикл и небольшое количество потоков — по одному на каждый процессор.

Конкурентность, параллелизм и гонки

Из-за потоков появляется два взаимосвязанных, но различных явления: конкурентность и параллелизм. Их нельзя назвать ни преимуществами, ни недостатками, так как оба они влияют как на затраты поточности, так и на извлекаемые из нее выгоды. *Конкурентностью* называется способность двух и более потоков выполняться в перекрывающиеся периоды времени. *Параллелизм* — способность выполнять два и более потока одновременно. Конкурентность может происходить и без параллелизма: в качестве примера можно привести многозадачность в однопроцессорной системе. Параллелизм (иногда подчеркивается — *истинный параллелизм*) — особая форма конкурентности, требующая нескольких процессоров (или единственного процессора, способного обеспечить несколько механизмов исполнения, такого как GPU). При конкурентности несколько потоков могут обеспечивать прогресс, но не обязательно одновременно. При параллелизме потоки выполняются параллельно в буквальном смысле, позволяя многопоточным программам использовать несколько процессоров.

Конкурентность — это шаблон программирования, способ подхода к решению проблемы, параллелизм — функциональность аппаратного обеспечения, достижимая через конкурентность. Оба явления полезны.

Условия гонки. Именно конкурентность вызывает большинство сложностей в поточности. При одновременной работе потоки могут выполняться в непредсказуемом порядке друг относительно друга. В какие-то моменты это прекрасно. Однако что, если потокам придется делить между собой ресурсы? Доступ даже к такой

простой вещи, как слово в памяти, становится «гонкой», и поведение программы может быть разным в зависимости от того, какой поток оказался «расторопнее».

Формально *условиями гонки* называется ситуация, когда несинхронизированный доступ к общему ресурсу для двух и более потоков приводит к ошибочному поведению программы¹. Общим ресурсом может быть что угодно: аппаратное обеспечение системы, ресурсы ядра, данные в памяти. Последнее — наиболее часто встречающаяся форма, известная под названием *гонки за данными*. Окно, в котором может произойти такая гонка, — область кода, которая должна быть синхронизирована, — называется *критической областью*. Гонки устраняются путем синхронизации кода в критических областях. Перед тем как погрузиться в методы синхронизации, рассмотрим несколько примеров условий гонок.

Гонки в реальности. Представим себе банкомат (АТМ, также может называться АВМ). Использовать его очень просто: приходите, вставляете свою карточку, вводите ПИН-код и необходимое количество денег. После этого вы получаете наличные. В процессе банку нужно проверить, имеется ли необходимая сумма на вашем счету, и, если так, вычесть сумму снятия. Алгоритм получится примерно следующий.

1. Имеется ли на счету как минимум X единиц валюты?
2. Если да, уменьшить размер счета на величину X и выдать пользователю X денежных единиц.
3. Если нет, выдать сообщение об ошибке.

Код на C будет выглядеть приблизительно так:

```
int withdraw (struct account *account, int amount)
{
    const int balance = account->balance;
    if (balance < amount)
        return -1;
    account->balance = balance - amount;

    disburse_money (amount);

    return 0;
}
```

Если возможно конкурентное выполнение, то здесь созданы условия для серьезной гонки. Представьте, что банк выполняет эту операцию дважды, параллельно. Например, пользователь снимает деньги одновременно с тем, как банк обрабатывает онлайн оплату счетов или взымает какой-либо регулярный сбор. Что, если снятие средств произойдет одновременно, перед тем как баланс счета обновится?

¹ Доброкачественная гонка возникает, когда несинхронизированный доступ ведет к неожиданному, но не ошибочному поведению приложения. Иногда программисты предпочитают не синхронизировать доступ к общим данным, не являющимся критическими (например, статистические счетчики), в погоне за производительностью. Я же рекомендую всегда синхронизировать общие данные.

Обе операции могут осуществиться, даже если баланса счета недостаточно, чтобы проделать и то и другое. Например, если на счету было \$500 и одновременно пришли два запроса на снятие \$200 и \$400, они оба могут быть удовлетворены, хотя на счете окажется отрицательная величина $-\$100$, чего явно не желал допустить программист, написавший код.

Фактически в этой функции есть вторая гонка. Рассмотрим механизм обновления баланса в структуре `account`. Две операции по снятию могут «соревноваться» между собой за обновление баланса. Используя пример выше, мы можем остаться либо с \$300, либо с \$100 на счету. Мало того, что банк произведет снятие средств, которого не должно было произойти, но еще и подарит счастливому обладателю карты лишние \$400.

На самом деле практически каждая строка этой функции находится внутри критической области. Чтобы банк не прогорел, нужно синхронизировать доступ к функции `withdraw()`, обеспечив гарантию, что, даже если будут выполняться конкурентно два и более потока, сама функция будет выполняться как один атомарный элемент: банк открывает баланс счета, проверяет доступные средства, а затем изменяет их количество для каждой транзакции индивидуально.

Перед обсуждением решения, как банк может это проделать, рассмотрим пример, который проиллюстрирует, как именно создаются условия гонки. Пример с банком слишком высокоуровневый: на самом деле не было нужды даже рассматривать этот пример кода. Несложно понять, что, если позволить конкурентное кредитование и дебетование счетов, математика будет не в пользу банка. Однако условия для гонки могут появиться и на самых высоких уровнях.

Посмотрите на эту простую строку кода C:

```
x++; // x целое число
```

Это постинкрементный оператор, и все мы знаем, как он работает: берет текущую величину `x`, увеличивает ее на единицу, а затем сохраняет новое значение, обновленный `x`, снова в ту же величину. Компиляция этого в машинный код зависит, конечно, от архитектуры, но мы можем представить себе приблизительно следующее:

```
load x into register
add 1 to register
store register in x
```

Да, даже `x++` — «гоночная» операция. Представьте два потока, выполняющих `x++` конкурентно для `x = 5`. Вот желаемый результат.

Шаг	Поток 1	Поток 2
1	load x into register (5)	
2	add 1 to register (6)	
3	store register in x (6)	
4		load x into register (6)
5		add 1 to register (7)
6		store register in x (7)

Этот результат тоже подходит.

Шаг	Поток 1	Поток 2
1		load x into register (5)
2		add 1 to register (6)
3		store register in x (6)
4	load x into register (6)	
5	add 1 to register (7)	
6	store register in x (7)	

Нам повезет, если получится именно это, но ничто не защищает от такого.

Шаг	Поток 1	Поток 2
1	load x into register (5)	
2	add 1 to register (6)	
3		load x into register (6)
4	store register in x (6)	
5		add 1 to register (6)
6		store register in x (6)

Множество других комбинаций может привести именно к таким непредсказуемым результатам. Данные примеры иллюстрируют конкурентность, а не параллелизм. При параллелизме потоки могут выполняться одновременно, добавляя еще больше сочетаний для путаницы.

Шаг	Поток 1	Поток 2
1	load x into register (5)	load x into register (6)
2	add 1 to register (6)	add 1 to register (6)
3	store register in x (6)	store register in x (6)

Теперь вы поняли картину. Даже что-то совсем простое, вроде приращения величины, всего одна строка на С или С++, чревато появлением условий для гонки, как только появляется несколько потоков, выполняющихся одновременно. Не обязательно даже наличие параллелизма. Однопроцессорная машина может — и, скорее всего, будет — испытывать гонки. Условия гонок относятся к источникам самого большого недовольства программистов, а также программных ошибок. Посмотрим, как можно их избежать.

Синхронизация

Основным источником появления гонок являются такие критические области, как окна, в которых предусматривается недопустимость одновременного выполнения потоков при корректной работе программы. Чтобы предупредить возникновение условий гонки, программист должен синхронизировать доступ к этому окну, обеспечив *взаимоисключающий* доступ к критической области.

В теории информатики считается, что операция (или набор операций) является *атомарной*, если она неделима, не может смешиваться с выполнением других операций. Для остальной части системы атомарная операция (или операции) выглядит происходящей *мгновенно*. В этом-то и проблема критических областей: они не являются неделимыми, они не выполняются мгновенно, следовательно, они не атомарны.

Мьютексы

Существует множество способов сделать критические области атомарными: от единичных инструкций до больших блоков кода. Самая распространенная техника — *замок*, то есть механизм, обеспечивающий взаимное исключение доступа к критической области, делая ее атомарной. Запирающие обеспечивают взаимную блокировку, поэтому они известны в Pthreads (и повсеместно) как *мьютексы* (mutexes — буквально: «взаимоисключающие»)¹.

Замок работает аналогично своему тезке из реального мира: представьте себе, что критической областью является комната. Без замка люди (потoki) могут беспрепятственно входить в комнату и выходить оттуда, когда им захочется. В частности, там может быть более одного человека одновременно. Затем мы врезаем в дверь замок и запираем его. У замка имеется только один ключ. Когда человек (поток) подходит к комнате, он находит ключ висящим снаружи около двери. Человек отпирает дверь ключом, входит и запирает дверь изнутри. Теперь никто больше не может войти, и тот, кто находится внутри, может спокойно заняться своими делами в этой комнате без опасений, что его побеспокоят. Никто больше не сможет занять комнату параллельно; это взаимоисключающий ресурс. Закончив свои дела в комнате, человек открывает дверь и выходит, оставляя ключ снаружи. Следующий человек может зайти, запереть дверь изнутри, после чего процесс повторяется.

Замок в контексте поточности работает точно так же. Программист определяет замок и убеждается, что в критическую область невозможно попасть, минуя его. При реализации замка следует убедиться, что только один поток может «владеть» замком единолично. Если он становится необходимым другому потоку, последний должен ждать освобождения замка перед тем, как продолжить работу. По окончании работы с критической областью вы освобождаете замок, позволяя ожидающему потоку (если он есть) завладеть замком в свою очередь.

Вернемся к примеру с банком, приведенному выше. Посмотрим, как мьютекс мог бы предупредить катастрофические (во всяком случае, для банка) условия гонки. Настоящие действия с мьютексами мы обсудим позже (см. разд. «Мьютексы Р-потокa» этой главы), а сейчас представим, что у нас есть функции `lock()` и `unlock()` для запирающего и отпирающего мьютекса соответственно.

```
int withdraw (struct account *account, int amount)
{
    lock ();
```

¹ Известны также как бинарные семафорные механизмы.

```
const int balance = account->balance;
if (balance < amount) {
    unlock ();
    return -1;
}
account->balance = balance - amount;
unlock ();

disburse_money (amount);

return 0;
}
```

Мы запираем только часть функционала, где может произойти гонка: считывание баланса счета, проверку наличия необходимых средств и обновление баланса. Программа «уверена» в корректности транзакции и размере обновленного баланса счета, поэтому она может отпереть замок и, таким образом, распределить средства без необходимости взаимного исключения. Чем меньшей вы сделаете критическую область, тем лучше, ибо записание предотвращает конкурентность и, соответственно, уменьшает преимущества поточности.

Таким образом, ничего сверхъестественного в замках нет. Физически взаимного исключения нет. Замки действуют по принципу *джентльменского соглашения*. Все потоки должны получить в нужных местах права на необходимые замки. Помните, что ничто не мешает потоку получить несанкционированный доступ к замку, кроме добросовестного программирования.

ПРИМЕЧАНИЕ

Один из самых важных шаблонов многопоточного программирования — записание данных, а не кода. Хотя выше мы обсуждали ситуацию, когда условия гонки создаются в критической области, хороший программист никогда не рассматривает код как объект записания и никогда не скажет что-то вроде «этот замок защищает вот эту функцию». Вместо этого хороший программист связывает замки с данными. Данные защищаются замком, и доступ к ним означает удержание замка.

В чем же состоит разница? Когда вы привязываете замок к коду, запирающие семантики понять несколько труднее. С течением времени отношения между замком и данными становятся все более неясными, и программисты могут реализовать новое использование данных без соответствующего замка. Привязывая замок непосредственно к данным, вы делаете эти отношения намного четче.

Взаимные блокировки

Жестокая ирония поточности заключается в том, что, намереваясь избавиться от каких-то проблем, мы находим решения, которые вызывают новые вопросы. Поточность нужна для обеспечения конкурентности, но конкурентность создает условия гонки. Борясь с ними, мы вводим в действие мьютексы, но и они представляют собой новый источник программных ошибок — взаимные блокировки.

Взаимная блокировка — это ситуация, когда два потока ожидают окончания работы друг друга и, таким образом, ни один из них не может закончиться. При наличии мьютексов взаимная блокировка происходит, когда двум потокам нужны разные мьютексы, каждым из которых владеет другой. Худший вариант — один поток

заблокирован в ожидании мьютекса, которым он уже владеет. Отладка взаимных блокировок может быть довольно замысловатой, поскольку программа не должна внезапно прекращать работу. Вместо этого она просто останавливает выполнение каких-то задач, так как потоки ожидают дня, который никогда не наступит.

Роковая ошибка многопоточности на Марсе

Можно привести множество реальных примеров бед, к которым привела поточность, но один из самых интересных — это Mars Pathfinder, который в 1977 году успешно прибыл на планету Марс, но его климатические и геологические исследования были нарушены частыми перезагрузками системы.

Работа марсохода обеспечивалась высокопоточным встроенным ядром реального времени (не Linux). Ядро обеспечивало преимущественное планирование потоков. Как и в Linux, потоки реального времени имели приоритеты: поток с более высоким приоритетом всегда запускался перед потоком с более низким приоритетом. Среди множества различных потоков оказалось три, которые и привели к сбою: поток с низким приоритетом, обеспечивавший сбор метеорологических данных, поток со средним приоритетом, отвечавший за связь с Землей, и поток с высоким приоритетом, управляющий хранилищем всего вездехода. Как вы уже убедились ранее в этой главе, изучая тему «Условия гонки», синхронизация очень важна для предотвращения условий гонки, таким образом, потоки управляют конкурентностью через мьютексы. Примечательно, что мьютекс синхронизировал метеорологический поток с низким приоритетом (который генерировал данные) перед потоком хранилища высокого приоритета (который управлял этими данными).

Метеорологический поток запускался нечасто, реагируя на различные датчики космического аппарата. После запуска поток должен был завладеть мьютексом, записать метеорологические данные в подсистему хранилища, а затем освободить мьютекс. Поток хранилища запускался несколько чаще, реагируя на системные события. Перед управлением подсистемой хранилища он тоже должен был завладеть мьютексом. Если тот оказывался занят, поток ожидал, когда метеорологический поток освободит мьютекс.

Все шло хорошо, пока случайно связной поток не активировался и не запустился в тот же момент, когда метеорологический поток удерживал мьютекс, а поток хранилища ожидал его. Связной поток имел более высокий приоритет, чем метеорологический, поэтому первый был запущен вместо последнего. К сожалению, связной поток имел очень долговременную задачу — ведь Марс очень далеко! Таким образом, для обеспечения выполнения задачи связного потока метеорологический не запустился. Так и было задумано при проектировании, так диктовали существующие приоритеты. Однако метеорологический поток удерживал ресурс (мьютекс), который был нужен потоку хранилища. Следовательно, поток с низким приоритетом (связной) запустился вместо потока с более высоким приоритетом (хранилища), хоть и не напрямую. Система должна была заметить, что поток хранилища приостановил выполнение своей задачи, определить, что, вероятно, произошел какой-то сбой, и выполнить перезагрузку. Это и есть классический пример типа ошибок, называемого *инверсией приоритетов*.

Техника исправления известна как *наследование приоритетов* и заключается в том, что процесс, удерживающий ресурс, наследует самый высокий из приоритетов процессов, ожидающих этого ресурса. В данном случае поток с самым низким приоритетом — метеорологический — должен был унаследовать высокий приоритет потока хранилища на время, которое он владел мьютексом. Это предотвратило бы вытеснение метеорологического потока связным, обеспечив быстрое освобождение мьютекса и планирование работы потока хранилища. Если вы не находите эту историю поучительной, ваша стезя — однопоточное программирование!

Профилактика взаимных блокировок. Предотвращение взаимных блокировок очень важно, и единственный надежный и безопасный способ — с самого начала позаботиться о разработке замков в вашей многопоточной программе. Очень важно связывать мьютексы с данными, а не с кодом, и обеспечить четкую иерархию данных (и этих мьютексов). Например, самая простая форма взаимных блокировок известна как *взаимная блокировка АББА*, или *клинч*. Это происходит, когда один поток заведует мьютексом А, за которым следует мьютекс Б, а другой поток в то же время заведует мьютексом Б, за которым следует А (отсюда АББА). При правильном распределении по времени оба потока могут успешно воспользоваться первым мьютексом: поток 1 удерживает А, а поток 2 удерживает Б; затем они хотят перейти к использованию следующего мьютекса, но обнаруживают, что он занят другим потоком, и каждый из них блокируется в ожидании освобождения мьютекса. Каждый поток, удерживающий мьютекс, также ожидает мьютекса, поэтому ни один из них так никогда и не освободится и потоки оказываются заблокированными намертво.

Поможет избежать этой ситуации простое правило: завладеть мьютексом Б должно быть возможно только после А. По мере того как сложность вашей программы и, следовательно, ее синхронизации возрастает, ввести эти правила становится все труднее. Начните заранее и разрабатывайте сразу начисто.

Р-потоки

Ядро Linux обеспечивает только базовые примитивы для обеспечения поточности, например системный вызов `clone()`. Основная часть любой поточной библиотеки находится в пользовательском пространстве. Многие большие проекты по разработке ПО определяют собственную поточную библиотеку: например, Android, Apache, GNOME и Mozilla обеспечивают собственные библиотеки; кроме того, языки программирования, такие как C++11 и Java, обеспечивают стандартную библиотечную поддержку для потоков. Тем не менее POSIX регламентирует поточную библиотеку с IEEE Std 1003.1c-1995, также известную как POSIX 1995 или POSIX.1c. Разработчики называют этот стандарт *POSIX-потоками*, или, для краткости, *Р-потоками*. Р-потоки остаются лидирующим решением в отношении поточности для C и C++ в системах UNIX.

Реализация поточности в Linux

Р-поток как стандарт — это всего лишь набор слов на странице. В Linux реализация этого стандарта осуществляется с помощью `glibc`, библиотеки C в Linux. Со временем `glibc` обеспечила две различные реализации Р-потоков: `LinuxThreads` и `NPTL`.

LinuxThreads — это оригинальная реализация Р-потоков, обеспечивающая поточность 1:1. Она впервые была включена в `glibc` версии 2.0, хотя и прежде была доступна как внешняя библиотека. `LinuxThreads` была разработана для ядра, которое обеспечивало достаточно слабую поддержку поточности: кроме системного вызова `clone()`, создававшего новый поток, `LinuxThreads` реализовала поточность POSIX, используя существующие интерфейсы UNIX. Например, `LinuxThreads` обрабатывает связь «поток-на-поток» с использованием сигналов (см. гл. 10). Из-за отсутствия поддержки Р-потоков со стороны ядра в Linux реализации `LinuxThreads` требовалось наличие управляющего действиями «менеджерского» потока, который плохо справлялся с большим количеством потоков и не вполне соответствовал стандарту POSIX.

Native POSIX Thread Library (NPTL) заменила `LinuxThreads` и остается стандартной реализацией Р-потоков в Linux. Она была представлена в Linux 2.6 и `glibc` 2.3. Как и `LinuxThreads`, `NPTL` обеспечивает поточность 1:1, основывающуюся на системном вызове `clone()` и модели ядра, где потоки рассматриваются как любые другие процессы, за исключением способности потоков разделять между собой некоторые ресурсы. В отличие от `LinuxThreads`, `NPTL` извлекает выгоду из дополнительных интерфейсов ядра, новых для версии 2.6, включающих системный вызов `futex()` для синхронизации потоков, системный вызов `exit_group()` для прекращения всех потоков в процессе и поддержку ядра для локального поточного хранилища (TLS). Таким образом, в `NPTL` решена проблема `LinuxThreads` с несоответствием и значительно улучшено поточное масштабирование, благодаря чему возможно создание тысяч потоков в единичном процессе без какого-либо замедления.

ПРИМЕЧАНИЕ

Конкурентом и ранней альтернативой `NPTL` были Next Generation POSIX Threads (поток следующего поколения POSIX), `NGPT`. Как и `NPTL`, `NGPT` пытались избежать ограничений `LinuxThreads` и улучшить масштабирование. В отличие от `NPTL` и `LinuxThreads`, `NGPT` внедрили поточность типа N:M. Однако, как это часто бывает в случае Linux, более простое решение победило и `NGPT` остались всего лишь частью истории.

Хотя из систем, основанных на `LinuxThreads`, должно быть, уже сыплется песок, они все еще встречаются. `NPTL` — значительный рывок вперед по сравнению с `LinuxThreads`, поэтому хорошенько подумайте о переводе таких систем на `NPTL` (если вам вообще нужна другая причина, чтобы не использовать такую древность) или, если это невозможно, об использовании только однопоточных программ.

API для работы с Р-потоками

API для работы с Р-потоками включает все необходимое — хотя и на довольно низком уровне — для написания многопоточной программы. Он достаточно велик — предоставляет около 100 интерфейсов. Из-за большого размера и некоторых

других неудобств данный API не лишен недоброжелателей. Тем не менее это ядро поточных библиотек в системах UNIX, и желательно изучить его, даже если вы используете другие решения для поточности, так как большинство из них построено на Р-потоках.

API для работы с Р-потоками определяется в `<pthread.h>`. Каждая функция в API снабжена префиксом `pthread_`. Например, функция для создания потока называется `pthread_create()` (скоро мы рассмотрим ее в разд. «Создание потоков» данной главы). Функции Р-потоков могут быть разбиты на две большие группы:

- *управление потоками* — функции для создания, уничтожения, соединения и отсоединения потоков (мы рассмотрим их далее в этой главе);
- *синхронизация* — функции для управления синхронизацией потоков, включающие мьютексы, условные переменные и барьеры (в данной главе мы изучили мьютексы).

Связывание Р-потоков

Хотя Р-потоки обеспечиваются `glibc`, они находятся в отдельной библиотеке `libpthread`, требующей явной привязки. При `gcc` это обеспечивается автоматически с помощью флага `-pthread`, который гарантирует, что с вашим исполняемым модулем связана нужная библиотека:

```
gcc -Wall -Werror -pthread beard.c -o beard
```

Если вы написали и привязали бинарный модуль с множественными вызовами `gcc`, вам лучше установить `-pthread` им всем: флаг также влияет на препроцессор, устанавливая определенный препроцессор для управления безопасностью потока.

Создание потоков

При первом запуске вашей программы и выполнении функции `main()` программа является однопоточной. Действительно, за исключением обеспечения компилятором некоторых параметров безопасности потока и наличия привязки к библиотеке Р-потоков, ваш процесс ничем не отличается от любого другого. Из этого изначального потока, который иногда называют *потоком по умолчанию* или *главным потоком*, вы можете создать один или несколько дополнительных, чтобы запустить многопоточность.

Р-потоки обеспечивают одну функцию для определения и запуска нового потока, которая называется `pthread_create()`:

```
#include <pthread.h>
```

```
int pthread_create (pthread_t *thread,  
                   const pthread_attr_t *attr,  
                   void *(*start_routine) (void *),  
                   void *arg);
```

При успешном выполнении создается новый поток и начинает выполнять функцию, представленную в `start_routine` с единственным аргументом `arg`. Функция сохранит идентификатор потока, использованный для представления нового потока, в `pthread_t`, указанного с помощью `thread`, если он не равен `NULL` (мы обсудим идентификаторы потоков в следующем разделе).

Объект `pthread_attr_t`, определяемый через `attr`, используется для изменения *атрибутов потока*, присваиваемых по умолчанию, для вновь создаваемого потока. Большинство вызовов `pthread_create()` отправляет в качестве аргумента `NULL`, присваивая атрибуты по умолчанию. Атрибуты потока позволяют программам изменять множество свойств потока, таких как размер стека, параметры планирования, начальный отдельный статус. Полное обсуждение атрибутов потока выходит за рамки этой главы, но *man-страницы P-потоков* — хороший ресурс.

Функция `start_routine` должна иметь запись следующего вида:

```
void * start_thread (void *arg);
```

Таким образом поток начинает существование с исполнения функции, которая принимает указатель `void` как единственный аргумент, а затем возвращает его как свою возвращаемую величину. Аналогично `fork()` новый поток наследует большинство атрибутов, совместимость и статус от своего родителя. Однако в отличие от `fork()` потоки *разделяют* ресурсы со своим родителем вместо получения копии. Самым важным разделяемым ресурсом является, конечно же, адресное пространство процесса, но потоки также делят (вместо получения копии) обработчики сигналов и открытые файлы.

Код, использующий эту функцию, должен отправлять `-pthread` в `gss`. Это относится ко всем функциям P-потоков, далее я не буду это специально указывать.

В случае ошибки `pthread_create()` возвращает ненулевой код ошибки напрямую (без использования `errno`) и содержимое потока `thread` не определено. Ошибки могут включать в себя:

- `EAGAIN` — вызывающему процессу существенно не хватает ресурсов для создания нового потока; обычно это вызвано тем, что процесс достиг предела количества потоков для каждого пользователя или для всей системы;
- `EINVAL` — объект `pthread_attr_t`, указанный через `attr`, включает недопустимые атрибуты;
- `EPERM` — вызывающий процесс не имеет полномочий для установки некоторых атрибутов объекта `pthread_attr_t`, указанного через `attr`.

Пример использования:

```
pthread_t tread;
int ret;

ret = pthread_create (&thread, NULL, start_routine, NULL);
if (!ret) {
    errno = ret;
    perror("pthread_create");
    return -1;
}
```

```
}
```

```
/* Новый поток создан и параллельно выполняет start_routine... */
```

Мы рассмотрим полноценный пример, как только разберем еще несколько техник.

Идентификаторы потоков

Идентификаторы потоков (TID) для потоков являются аналогами идентификаторов процессов (PID). В то время как PID назначаются ядром Linux, TID назначаются всего лишь библиотекой Р-потоков¹. Этот тип представлен `pthread_t`, и POSIX не требует, чтобы он был арифметическим. Как мы уже знаем, TID нового потока определяется с помощью аргумента `thread` при успешном вызове `pthread_create()`. Поток может получить свой TID при запуске с помощью функции `pthread_self()`:

```
#include <pthread.h>
```

```
pthread_t pthread_self (void);
```

Использовать функцию очень просто, так как она всегда работает успешно:

```
const pthread_t me = pthread_self ();
```

Сравнение идентификаторов потоков. Стандарт Р-потоков не требует, чтобы `pthread_t` был арифметического типа, поэтому нет гарантии, что оператор равенства будет работать. Следовательно, чтобы сравнить идентификаторы потоков, библиотеке Р-потоков нужен специальный интерфейс:

```
#include <pthread.h>
```

```
int pthread_equal (pthread_t t1, pthread_t t2);
```

Если приведенные идентификаторы потоков равны, `pthread_equal()` возвращает ненулевую величину. Если они не равны, возвращается 0; ошибка не может произойти. Вот простой пример:

```
int ret;
ret = pthread_equal(thing1, thing2);
if (ret != 0)
    printf("The TIDs are equal!\n");
else
    printf("The TIDs are unequal!\n");
```

¹ Для ядра Linux потоки — это только процессы, которые могут совместно использовать различные ресурсы, поэтому ядро взаимодействует с каждым потоком с помощью уникального PID, как и с любым другим процессом. Программы пользовательского пространства могут получить этот PID через системный вызов `gettid()`, но эта величина может быть полезной в очень редких случаях. Программисты должны использовать концепцию идентификаторов Р-потоков для взаимодействия с этими потоками.

Завершение потоков

Естественной противоположностью созданию потоков является их завершение. Завершение потоков очень похоже на завершение процессов, за исключением того, что, когда поток завершается, остальные потоки в процессе продолжают выполняться. В некоторых поточных шаблонах, таких как *поток на соединение* (см. подразд. «Поток на соединение» разд. «Шаблоны поточности» данной главы), потоки часто создаются и уничтожаются.

Потоки могут прерываться при определенных обстоятельствах, которые имеют аналоги в завершении процессов:

- если поток возвращается из стартовой процедуры, он прерывается; это аналог «выхода за пределы» в `main()`;
- если поток вызывает функцию `pthread_exit()` (будет рассмотрена далее), он завершается; это аналог вызова `exit()`;
- если поток отменяется другим потоком через функцию `pthread_cancel()`, он завершается; это аналог отправки сигнала `SIGKILL` через `kill()`.

В этих трех примерах завершается только поток, на который направлено действие. Все потоки в процессе завершаются, останавливая таким образом сам процесс, при следующих обстоятельствах:

- процесс возвращается из своей функции `main()`;
- процесс завершается через `exit()`;
- процесс выполняет новый двоичный образ через `execve()`.

Сигналы могут убить процесс или отдельный поток в зависимости от того, как они направлены. Р-потоки делают обработку сигналов несколько сложнее, поэтому лучше минимизировать использование сигналов в многопоточных программах (полное описание обработки сигналов приведено в гл. 10).

Самозавершение

Самый простой путь для потока, чтобы завершить самого себя, — это «выход за пределы» своей начальной процедуры. Однако часто вам будет нужно завершить поток где-то в глубине стека вызова функции, далеко от вашей стартовой процедуры. Для таких случаев в Р-потоках имеется вызов `pthread_exit()`, поточный эквивалент `exit()`:

```
#include <pthread.h>
```

```
void pthread_exit (void *retval);
```

По выполнении вызывающий поток завершается; `retval` обеспечивается для каждого потока, ожидающего завершения (см. разд. «Присоединение и отсоединение потоков» этой главы), аналогично `exit()`. Ошибка не может произойти.

Использование:

```
/* Прощай, жестокий мир! */  
pthread_exit (NULL);
```

Завершение других потоков

P-потоки вызывают завершение других потоков через их отмену. Это обеспечивает функция `pthread_cancel()`:

```
#include <pthread.h>  
  
int pthread_cancel (pthread_t thread);
```

Успешный вызов `pthread_cancel()` посылает запрос на отмену потоку, представленному через идентификатор потока `thread`. Может ли поток быть отменен и когда, зависит от его *состояния отмены* и *типа отмены* соответственно. В случае успеха `pthread_cancel()` возвращает 0. Обратите внимание, что успех в данном случае означает лишь успешную обработку запроса на отмену. В действительности же завершение происходит асинхронно. В случае ошибки `pthread_cancel()` возвращает `ESRCH`, означающее, что значение `thread` недопустимо.

Условия, при которых поток может быть отменен, не так просты. Состояние отмены потока может быть *доступно* или *недоступно*. По умолчанию оно является доступным для новых потоков. С другой стороны, тип отмены указывает, когда происходит отмена. Потоки могут изменять свое состояние через `pthread_setcancelstate()`:

```
#include <pthread.h>  
  
int pthread_setcancelstate (int state, int *oldstate);
```

В случае успеха состояние отмены вызывающего потока устанавливается на `state`, а предыдущее состояние сохраняется в `oldstate`¹. Значением `state` может быть `PTHREAD_CANCEL_ENABLE` или `PTHREAD_CANCEL_DISABLE` для разрешения или запрещения отмены соответственно.

В случае ошибки `pthread_setcancelstate()` возвращает `EINVAL`, что означает недопустимое значение `state`.

Тип отмены потока может быть *асинхронным* или *отложенным*; по умолчанию обычно установлен последний. С асинхронным типом отмены поток может быть убит в любой точке после получения команды на отмену. С отложенным типом поток может быть убит только в специальных *точках отмены*, которые являются функциями P-потоков или библиотеки C и представляют собой безопасные моменты, в которых вызывающий поток может быть прерван. Асинхронная отмена может быть полезна лишь в определенных ситуациях, так как она может оставить процесс в неопределенном состоянии. Например, что, если отменяемый поток был где-то в середине критической области? Чтобы программа вела себя корректно,

¹ Linux позволяет сохранение `NULL` в `oldstate`, но POSIX не допускает этого. Программы, обеспечивающие совместимость между ОС, должны всегда помещать здесь соответствующий указатель, даже если он не потребуется в дальнейшем.

асинхронная отмена должна использоваться только потоками, для которых не предусмотрено совместное использование каких-либо ресурсов и возможен вызов только сигнально-безопасных функций (см. разд. «Реентерабельность» гл. 10). Потоки могут изменить свой тип через `pthread_setcanceltype()`:

```
#include <pthread.h>
```

```
int pthread_setcanceltype (int type, int *oldtype);
```

В случае успеха статус отмены вызывающего потока устанавливается в `type`, а старый тип сохраняется в `oldtype`¹. Значением `type` может быть `PTHREAD_CANCEL_ASYNCHRONOUS` или `PTHREAD_CANCEL_DEFERRED` для установки асинхронной или отложенной отмены соответственно.

В случае ошибки `pthread_setcanceltype()` возвращает `EINVAL`, что означает недопустимое значение `type`.

Рассмотрим пример, когда один поток должен завершить другой. Сначала поток, который должен завершиться, разрешает свою отмену и устанавливает ее тип как отложенный (эти значения установлены по умолчанию, так что в данном случае это просто пример):

```
int unused;  
int ret;
```

```
ret = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &unused);  
if (ret) {  
    errno = ret;  
    perror ("pthread_setcancelstate");  
    return -1;  
}
```

```
ret = pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &unused);  
if (ret) {  
    errno = ret;  
    perror ("pthread_setcanceltype");  
    return -1;  
}
```

Затем другой поток посылает команду на завершение:

```
int ret;  
  
/* 'thread' в данном случае означает идентификатор завершаемого потока */  
ret = pthread_cancel (thread);  
if (ret) {  
    errno = ret;  
    perror ("pthread_cancel");  
    return -1;  
}
```

¹ Как и в случае `pthread_setcancelstate()`, для обеспечения совместимости не следует отправлять `NULL` в `oldtype`, хотя Linux и допускает это.

Присоединение и отсоединение потоков

Учитывая, что потоки достаточно просто создаются и уничтожаются, должен быть и способ их синхронизировать вместо завершения других потоков — эквивалент `wait()` для поточности. Действительно, он существует. Это присоединение потоков.

Присоединение потоков

Присоединение позволяет одному из потоков заблокироваться в ожидании завершения другого:

```
#include <pthread.h>
```

```
int pthread_join (pthread_t thread, void **retval);
```

После успешного выполнения вызывающий поток блокируется до тех пор, пока поток, указанный как `thread`, не завершится (если `thread` уже завершен, `pthread_join()` возвращается немедленно). Как только `thread` завершается, вызывающий поток активизируется и, если `retval` не равен `NULL`, получает возвращаемое значение завершенного процесса, переданное `pthread_exit()` или возвращенное от его стартовой процедуры. После этого можно сказать, что потоки *присоединились* друг к другу. Присоединение всегда позволяет потокам синхронизировать свое выполнение по отношению к периоду существования других потоков. Все потоки в Р-потоках являются равноправными; каждый поток может присоединяться к любому другому. Один поток может присоединяться ко многим (фактически, как мы скоро увидим, чаще всего один главный поток ожидает других потоков, которые сам и создал), но только один поток может попытаться присоединиться к определенному другому, несколько потоков не должны стараться присоединиться к какому-либо одному.

В случае ошибки `pthread_join()` возвращает один из следующих ненулевых кодов ошибок:

- `EDEADLK` — произошла взаимная блокировка — `thread` уже ожидает присоединения к вызывающему потоку или *сам* является вызывающим потоком;
- `EINVAL` — невозможно присоединить поток, определенный через `thread` (см. следующий раздел);
- `ESRCH` — значение `thread` недопустимо.

Пример использования:

```
int ret;
```

```
/* присоединяем к 'thread' и больше не заботимся о возвращаемой величине */  
ret = pthread_join (thread, NULL);  
if (ret) {  
    errno = ret;  
    perror ("pthread_join");  
    return -1;  
}
```

Отсоединение потоков

По умолчанию потоки создаются *способными к присоединению*. Однако они могут и *отсоединяться*, но в этом случае они станут в дальнейшем неприсоединяемыми. Поскольку до присоединения потоки потребляют какие-либо системные ресурсы, как делают это и процессы, пока их предки вызывают `wait()`, потоки, которые вы не планируете присоединять, должны быть отсоединены.

```
#include <pthread.h>
```

```
int pthread_detach (pthread_t thread);
```

В случае успеха `pthread_detach()` отсоединяет поток, указанный как `thread`, и возвращает 0. Результаты не определены, если вы вызываете `pthread_detach()` относительно потока, который уже отсоединен. В случае ошибки функция возвращает значение `ESRCH`, означающее, что значение `thread` недопустимо.

Для каждого потока в процессе необходимо вызвать `pthread_join()` или `pthread_detach()`, чтобы системные ресурсы могли высвободиться после завершения потока (конечно, после того как завершается весь процесс, все поточные ресурсы высвобождаются, но присоединение или отсоединение всех процессов в явной форме остается хорошей практикой).

Пример поточности

Следующий пример полной программы соединит все интерфейсы, рассмотренные выше. Программа создает два потока (всего их будет три), начиная оба в одной и той же стартовой процедуре `start_thread()`. Поведение потоков в стартовой процедуре отличается аргументами. Затем оба потока присоединяются друг к другу; если бы этого не произошло, главный поток мог бы завершиться до остальных, прервав весь процесс.

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <pthread.h>
```

```
void * start_thread (void *message)
{
    printf ("%s\n", (const char *) message);
    return message;
}
```

```
int main (void)
{
    pthread_t thing1, thing2;
    const char *message1 = "Thing 1";
    const char *message2 = "Thing 2";
```

```
/* Создаются два потока, каждый со своим сообщением */
```

```

pthread_create (&thing1, NULL, start_thread, (void *) message1);
pthread_create (&thing2, NULL, start_thread, (void *) message2);
/*
 * Ожидание завершения потоков. Если мы не присоединим их здесь,
 * то рискуем уничтожить главный поток до того,
 * как остальные потоки завершатся.
 */
pthread_join (thing1, NULL);
pthread_join (thing2, NULL);

return 0;
}

```

Вот и вся программа. Если вы сохраните ее как `example.c`, то можете скомпилировать ее с помощью такой команды:

```
gcc -Wall -O2 -pthread example.c -o example
```

Затем запустить следующим образом:

```
./example
```

На выходе вы получите:

```
Thing 1
```

```
Thing 2
```

Или, возможно:

```
Thing 2
```

```
Thing 1
```

Однако ненужной информации никогда не будет. Почему? Потому что функция `printf()` безопасна для поточности.

Мьютексы P-потоков

Из подразд. «Мьютексы» разд. «Синхронизация» этой главы вы уже знаете, что основным методом предотвращения взаимных блокировок является мьютекс. При всей их мощности и важности мьютексы достаточно просты в использовании.

Инициализация мьютексов

Мьютексы представляются объектом `pthread_mutex_t`. Как и большинство объектов в API P-потоков, это подразумевает их непрозрачную структуру, обеспечивающую разнообразие интерфейсов мьютексов. Хотя вы можете создавать мьютексы динамически, в большинстве случаев их использование статично:

```

/* определим и запустим мьютекс с именем 'mutex' */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

```

Этот фрагмент кода определяет и инициализирует мьютекс под названием `mutex`. Это все, что мы должны сделать, чтобы начать его использовать.

Запирание мьютексов

Запирание, называемое также *завладением*, P-поточного мьютекса обеспечивается с помощью функции `pthread_mutex_lock()`:

```
#include <pthread.h>
```

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Успешный вызов `pthread_mutex_lock()` заблокирует вызывающий поток, пока мьютекс, указанный как `mutex`, не станет доступным. После этого вызывающий поток активируется и эта функция вернет 0. Если мьютекс доступен в момент вызова, функция вернет значение немедленно.

В случае ошибки функция возвращает один из следующих ненулевых кодов ошибки:

- EDEADLK — вызывающий поток уже владеет запрашиваемым мьютексом; этот код ошибки не обязательно возвратится по умолчанию; попытка завладеть уже имеющимся мьютексом может привести к взаимной блокировке (см. подразд. «Взаимные блокировки» разд. «Синхронизация» этой главы);
- EINVAL — значение `mutex` недопустимо.

Вызывающие потоки, как правило, не проверяют возвращаемую величину, поскольку хорошо написанный код не должен генерировать каких-либо ошибок во время выполнения. Вот пример использования:

```
pthread_mutex_lock (&mutex);
```

Отпирание мьютексов

Противоположностью запиранию является *отпирание*, или *высвобождение*, мьютексов.

```
#include <pthread.h>
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Успешный вызов `pthread_mutex_unlock()` высвобождает мьютекс, указанный как `mutex`, и возвращает 0. Вызов не блокируется; мьютекс освобождается немедленно.

В случае ошибки функция возвращает ненулевой код ошибки, включающий:

- EINVAL — значение `mutex` недопустимо;
- EPERM — вызывающий процесс не владеет мьютексом, указанным как `mutex`; этот код ошибки не гарантируется; попытка высвободить мьютекс, которым вы не владеете, является ошибкой.

Как и с запиранием, пользователи обычно не проверяют возвращаемую величину:

```
pthread_mutex_unlock (&mutex);
```

Локальные замки

RAII — один из самых мощных шаблонов программирования в C++. RAII эффективно работает с распределением и высвобождением ресурсов, связывая срок службы ресурса со сроком существования объекта в данной области. RAII был создан для работы с очисткой ресурсов после того, как произошло исключение, поэтому он — самый мощный как способ управления ресурсами. Например, RAII позволяет создать такой объект, как «локальный файл», где файл открывается, когда объект создан, и автоматически закрывается, когда объект покидает заданную область. Аналогично мы можем создать и «локальный замок», который позволяет создать мьютекс и автоматически освободить его после выхода из заданной области:

```
class ScopedMutex {
public:
    ScopedMutex (pthread_mutex_t& mutex)
        :mutex_ (mutex)
    {
        pthread_mutex_lock (&mutex_);
    }

    ~ScopedMutex ()
    {
        pthread_mutex_unlock (&mutex_);
    }
private:
    pthread_mutex_t& mutex_;
};
```

Чтобы использовать его, просто вызовите `ScopedMutex m(mutex)`. Замок автоматически разблокируется, когда `m` покидает заданную область. Это делает разворачивание функций и обработку ошибок более надежными и освобождает от использования отчетности `goto`.

Пример использования мьютексов

Рассмотрим простой фрагмент кода, в котором показано использование мьютексов для обеспечения синхронизации. Вспомним пример с банком из разд. «Конкурентность, параллелизм и гонки» текущей главы. Наш воображаемый банк находится в угрожающих условиях гонки, что может иметь нежелательные последствия. Вот как мы можем исправить это с использованием P-поточных мьютексов:

```
static pthread_mutex_t the_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int withdraw (struct account *account, int amount)
{
    pthread_mutex_lock (&the_mutex);
    const int balance = account->balance;
    if (balance < amount) {
        pthread_mutex_unlock (&the_mutex);
```



```

        return -1;
    }
    account->balance = balance - amount;
    pthread_mutex_unlock (&the_mutex);

    disburse_money (amount);

    return 0;
}

```

В этом примере используется `pthread_mutex_lock()` для завладения мьютексом, а затем `pthread_mutex_unlock()` для освобождения его в конечном итоге. Это позволяет избежать условий гонки, но создает для банка новую проблему: в каждый момент времени только один клиент может снять деньги! Здесь-то и находится самое узкое место; а для «слишком-больших-чтобы-обанкротиться» банков это настоящий провал.

Поэтому в большинстве случаев при использовании замков стараются избегать *глобальных блокировок*, а вместо этого связывают замки с конкретными структурами данных. Это называется *детализированной блокировкой*. Такой подход может усложнить ваши блокирующие семантики, в частности, при избегании взаимной блокировки, но он является ключевым моментом при увеличении количества ядер на современных машинах.

В нашем примере вместо того, чтобы определять глобальную блокировку `the_mutex`, мы определим мьютекс внутри структуры `account`, выделяя каждому счету собственную блокировку. Это сработает, так как данные внутри критической области — только структура `account`. Запирая только сам счет в момент списания, мы даем банку возможность параллельно выполнять операции других клиентов.

```

int withdraw (struct account *account, int amount)
{
    pthread_mutex_lock (&account->mutex);
    const int balance = account->balance;
    if (balance < amount) {
        pthread_mutex_unlock (&account->mutex);
        return -1;
    }
    account->balance = balance - amount;
    pthread_mutex_unlock (&account->mutex);

    disburse_money (amount);

    return 0;
}

```

Дальнейшее изучение

В одну главу невозможно вместить всю информацию о потоковых API поточности в POSIX, который, если быть вежливыми, является полнофункциональной и мощной

библиотекой с мириадами интерфейсов для изучения (если отбросить корректность в сторону, можно дополнить, что поточность в POSIX очень усложнена и громоздка). Множество крупномасштабных системных приложений определяют собственные поточные интерфейсы; например, такие механизмы, как поточные пулы и рабочие очереди, являются более подходящими уровнями абстракции для системного программного обеспечения, чем предоставляемые POSIX. В этом случае основы поточности, описанные в данной главе, могут послужить отправной точкой для ваших собственных решений.

Если вы хотите глубже изучить тему Р-поток, рекомендую для дальнейшего чтения приложение Б. Соответствующие справочные страницы будут особенно полезны.

8 Управление файлами и каталогами

В гл. 2–4 мы рассмотрели несколько подходов к организации ввода-вывода файлов. В данной главе мы снова поговорим о файлах, на этот раз концентрируясь не на чтении или записи файлов, а на управлении ими и их метаданными.

Файлы и их метаданные

Как уже рассказывалось в гл. 1, каждый файл представляется структурой `inode` — индексного дескриптора, которой присваивается уникальная в данной файловой системе численная величина, которая называется *номером индексного дескриптора*. Индексный дескриптор — и физический объект, находящийся на диске файловой системы UNIX, и концептуальная сущность, представляемая структурой данных ядра Linux. Индексный дескриптор хранит *метаданные*, связанные с файлом, такие как права доступа к файлу, время последнего доступа, владелец, группа, размер, а также размещение данных файла¹.

Получить номер индексного дескриптора можно с помощью команды `ls` и флага `-li`:

```
$ ls -li
1689459 Kconfig      1689461 main.c      1680144 process.c  1689464 swsusp.c
1680137 Makefile     1680141 pm.c            1680145 smp.c        1680149 user.c
1680138 console.c    1689462 power.h        1689463 snapshot.c
1689460 disk.c        1680143 poweroff.c    1680147 swap.c
```

Этот вывод показывает, что, например, `disk.c` имеет номер индексного дескриптора 1689460. В этой конкретной файловой системе ни один другой файл не может иметь такого же номера. О другой файловой системе, однако, мы не можем утверждать этого.

¹ Интересно, что единственной вещью, не входящей в `inode`, является имя файла! Оно хранится в таблице директориев.

Семейство stat

UNIX предоставляет семейство функций для получения метаданных файла:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Каждая из этих функций возвращает информацию о файле. Функция `stat()` предоставляет информацию о файле, определенном через `path`, в то время как `fstat()` возвращает информацию о файле, представленном файловым дескриптором `fd`. Функция `lstat()` идентична `stat()`, за исключением того, что в случае передачи ей символической ссылки `lstat()` возвращает информацию о самой ссылке, а не о конечном файле.

Каждая из этих функций хранит информацию в структуре `stat`, которая предоставляется пользователям. Структура `stat` определена в `<bits/stat.h>`, включенном в `<sys/stat.h>`:

```
struct stat {
    dev_t st_dev;           /* идентификатор устройства, на котором хранится файл */
    ino_t st_ino;           /* номер индексного дескриптора */
    mode_t st_mode;         /* разрешения */
    nlink_t st_nlink;       /* количество жестких ссылок */
    uid_t st_uid;           /* пользовательский идентификатор владельца */
    gid_t st_gid;           /* групповой идентификатор владельца */
    dev_t st_rdev;          /* идентификатор устройства (для специальных файлов) */
    off_t st_size;          /* общий размер в байтах */
    blksize_t st_blksize;   /* размер блока для ввода-вывода в файловой системе */
    blkcnt_t st_blocks;     /* количество выделенных блоков */
    time_t st_atime;        /* время последнего доступа */
    time_t st_mtime;        /* время последней модификации */
    time_t st_ctime;        /* время последнего изменения метаданных */
};
```

Рассмотрим поля подробнее.

- Поле `st_dev` описывает узел устройства, на котором хранится файл (позже мы поговорим об узлах устройств). Если файл не поддерживается устройством, а находится, например, на ресурсе NFS, эта величина равна 0.
- Поле `st_ino` field предоставляет номер индексного дескриптора файла.
- Поле `st_mode` предоставляет байты режима файла, которые описывают тип файла (например, это файл или директория), а также разрешения доступа (например, доступен всем только для чтения). О байтах режима и разрешениях мы говорили в гл. 1 и 2.

- Поле `st_nlink` приводит количество жестких ссылок, связанных с файлом. Каждый файл в файловой системе имеет хотя бы одну жесткую ссылку.
- Поле `st_uid` приводит идентификатор пользователя, который владеет файлом.
- Поле `st_gid` приводит идентификатор группы, которая владеет файлом.
- Если файл является узлом устройства, то поле `st_rdev` описывает устройство, которое представляет этот файл.
- Поле `st_size` представляет размер файла в байтах.
- Поле `st_blksize` описывает предпочитаемый объем блока, достаточный для ввода-вывода файла. Эта величина (или кратное ей значение) — оптимальный размер блока для ввода-вывода с пользовательской буферизацией (см. гл. 3).
- Поле `st_blocks` предоставляет количество блоков в файловой системе, занимаемых файлом. Эта величина, умноженная на размер блока, будет всегда меньше, чем значение, предоставленное `st_size`, если у файла есть дыры (то есть если файл разреженный).
- Поле `st_atime` содержит *время последнего доступа к файлу*. Это самое позднее время, когда к файлу предоставлялся доступ (например, через `read()` или `execle()`).
- Поле `st_mtime` содержит *время последней модификации файла*, то есть время, когда в файл была сделана последняя запись.
- Поле `st_ctime` содержит *время последнего изменения файла*. Поле содержит время, когда метаданные файла (например, его владелец или права доступа) менялись в последний раз. Его часто путают со временем создания файла, которое не сохраняется в Linux и других UNIX-подобных системах.

В случае успеха все три вызова возвращают 0 и сохраняют метаданные файла в структуру, предоставленную `stat`. При ошибке они возвращают -1 и присваивают `errno` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути `path` (касается только `stat()` и `lstat()`);
- `EBADF` — значение `fd` является недопустимым (касается только `fstat()`);
- `EFAULT` — указатели `path` или `buf` являются некорректными;
- `ELoop` — путь содержит слишком много символических ссылок (касается только `stat()` и `lstat()`);
- `ENAMETOOLONG` — `path` слишком велик (касается только `stat()` и `lstat()`);
- `ENOENT` — один из компонентов `path` не существует (касается только `stat()` и `lstat()`);
- `ENOMEM` — недостаточно памяти для выполнения запроса;
- `ENOTDIR` — один из компонентов `path` не является директорией (касается только `stat()` и `lstat()`).

Следующая программа использует `stat()` для получения размера файла, указанного в командной строке:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    struct stat sb;
    int ret;

    if (argc < 2) {
        fprintf (stderr,
                "usage: %s <file>\n", argv[0]);
        return 1;
    }

    ret = stat (argv[1], &sb);
    if (ret) {
        perror ("stat");
        return 1;
    }

    printf ("%s is %ld bytes\n",
            argv[1], sb.st_size);
    return 0;
}
```

Запуск программы относительного ее собственного файла приводит к следующему результату:

```
$ ./stat stat.c
stat.c is 392 bytes
```

Эта программа, в свою очередь, выводит тип файла (такого как символическая ссылка или блокирующий узел устройства), указанного как первый аргумент программы:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    struct stat sb;
    int ret;

    if (argc < 2) {
        fprintf (stderr,
```

```

        "usage: %s <file>\n", argv[0]);
    return 1;
}

ret = stat (argv[1], &sb);
if (ret) {
    perror ("stat");
    return 1;
}

printf ("Тип файла: ");
switch (sb.st_mode & S_IFMT) {
case S_IFBLK:
    printf("блокирующий узел устройства\n");
    break;
case S_IFCHR:
    printf("текстовый идентификатор устройства\n");
    break;
case S_IFDIR:
    printf("каталог\n");
    break;
case S_IFIFO:
    printf("FIFO\n");
    break;
case S_IFLNK:
    printf("символическая ссылка\n");
    break;
case S_IFREG:
    printf("обычный файл\n");
    break;
case S_IFSOCK:
    printf("программный интерфейс\n");
    break;
default:
    printf("неизвестный\n");
    break;
}
return 0;
}

```

Наконец, вот этот фрагмент кода использует `fstat()`, чтобы проверить, находится ли уже открытый файл на физическом (или, напротив, на сетевом) устройстве:

```

/*
 * is_on_physical_device – возвращает положительное целое число,
 * если файл с дескриптором fd находится на физическом устройстве.
 * 0 если файл находится на нефизическом или виртуальном устройстве
 * (например, связанном ресурсе NFS)
 * и -1 в случае ошибки.
 */

```

```
int is_on_physical_device (int fd)
{
    struct stat sb;
    int ret;

    ret = fstat (fd, &sb);
    if (ret) {
        perror ("fstat");
        return -1;
    }
    return gnu_dev_major (sb.st_dev);
}
```

Разрешения

С помощью вызовов `stat` можно получить значения разрешений для данного файла, но для их изменения используются два других системных вызова:

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

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

Оба вызова `chmod()` и `fchmod()` устанавливают `mode` в качестве значения системных разрешений. В вызове `chmod()` относительный или абсолютный путь к файлу указывается в переменной `path`. В `fchmod()` файл определяется файловым дескриптором `fd`.

Допустимые величины `mode`, представленные непрозрачным целочисленным типом `mode_t`, те же, что возвращаются полем `ds_mode` в структуре `stat`. Значения принадлежат к типу простых целочисленных, однако их значения специфичны для каждой реализации UNIX. Следовательно, в POSIX определяется набор постоянных, представляющий все разнообразие переменных (подробно об этом было рассказано в подразд. «Права доступа новых файлов» разд. «Открытие файлов» гл. 2). Эти постоянные могут быть объединены между собой с помощью двоичного «ИЛИ», чтобы сформировать допустимые значения `mode`. Например, `(S_IRUSR | S_IRGRP)` устанавливает значение, разрешающее чтение файла его владельцу и группе.

Для изменения разрешений файла действительный идентификатор процесса, вызывающего `chmod()` или `fchmod()`, должен соответствовать владельцу файла или процесс должен обладать свойством `CAP_FOWNER`.

В случае успеха оба вызова возвращают 0, при неудаче возвращают -1, а `errno` присваивается одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути `path` (касается только `chmod()`);
- `EBADF` — значение `fd` является недопустимым (касается только `fchmod()`);
- `EFAULT` — указатель `path` является некорректным (касается только `chmod()`);

- EIO — в системе произошла внутренняя ошибка ввода-вывода; это очень плохое значение ошибки, которое может указывать на повреждение диска или файловой системы;
- ELOOP — путь содержит слишком много символических ссылок (касается только `chmod()`);
- ENAMETOOLONG — `path` слишком велик (касается только `chmod()`);
- ENOENT — путь `path` не существует (касается только `chmod()`);
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов `path` не является каталогом (касается только `chmod()`);
- EPERM — действительный идентификатор вызывающего процесса не соответствует владельцу файла, а процесс не имеет свойства `CAP_FOWNER`;
- EROFS — файл находится в файловой системе, доступной только для чтения.

Данный фрагмент кода устанавливает файлу `map.png` разрешения на считывание и запись для владельца:

```
int ret;

/*
 * Установка файлу 'map.png' в текущем каталоге разрешений
 * на считывание и запись для владельца файла. Аналогично команде
 * 'chmod 600 ./map.png'.
 */
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
    perror ("chmod");
```

Данный фрагмент делает то же, предполагая, что `fd` представляет открытый файл `map.png`:

```
int ret;

/*
 * Установка для файла с дескриптором fd разрешений на чтение
 * и запись для владельца файла.
 */
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
    perror ("fchmod");
```

Оба вызова, `chmod()` и `fchmod()`, доступны во всех современных системах UNIX. Стандарт POSIX требует реализации первого, а второй считает необязательным.

Владение

В структуре `stat` поля `st_uid` и `st_gid` представляют владельца и группу файла соответственно. Три системных вызова позволяют пользователю изменить эти два значения:

```
#include <sys/types.h>
#include <unistd.h>

int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

Вызовы `chown()` и `lchown()` устанавливают владение для файла, указанного с помощью пути `path`. Они работают одинаково для всех файлов за исключением символических ссылок: первый вызов следует по символической ссылке и изменяет владение целевого файла, а не самой ссылки; `lchown()` не переходит по ссылке, а вместо этого изменяет владение самого файла символической ссылки. Наконец, `fchown()` устанавливает владение файла, указанного с помощью файлового дескриптора `fd`.

В случае успеха все три системных вызова делают владельцем файла `owner`, группой файла — `group` и возвращают 0. Если параметры `owner` или `group` равны -1, эта величина не устанавливается. Только процесс со свойством `CAP_CHOWN` (обычно это процесс с правами `root`) может менять владельца файла. Владелец файла может изменить группу файла на любую другую, членом которой является пользователь; процессы со свойством `CAP_CHOWN` могут изменять группу файла без ограничений.

В случае неудачи вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути `path` (касается только `chown()` и `lchown()`);
- `EBADF` — значение `fd` является недопустимым (касается только `fchown()`);
- `EFAULT` — указатель `path` является некорректным (касается только `chown()` и `lchown()`);
- `EIO` — в системе произошла внутренняя ошибка ввода-вывода (очень плохо);
- `ELOOP` — путь содержит слишком много символических ссылок (касается только `chown()` и `lchown()`);
- `ENAMETOOLONG` — `path` слишком велик (касается только `chown()` и `lchown()`);
- `ENOENT` — путь `path` не существует;
- `ENOMEM` — недостаточно памяти для выполнения запроса;
- `ENOTDIR` — один из компонентов `path` не является каталогом (касается только `chown()` и `lchown()`);
- `EPERM` — вызывающий процесс не имеет прав, необходимых для изменения владельцев группы так, как запрошено;
- `EPOFS` — файл находится в системе, доступной только для чтения.

Данный фрагмент кода изменяет группу файла `manifest.txt` в текущем рабочем каталоге на группу `officers`. Чтобы проделать это, вызывающий пользователь должен обладать свойством `CAP_CHOWN` либо быть пользователем `kidd` и состоять в группе `officers`.

```

struct group *gr;
int ret;
/*
 * getgrnam() возвращает информацию о группе.
 * принимая в качестве параметра ее имя.
 */
gr = getgrnam ("officers");
if (!gr) {
    /* скорее всего, недопустимая группа */
    perror ("getgrnam");
    return 1;
}

/* изменяем группу файла manifest.txt's на 'officers' */
ret = chown("manifest.txt", -1, gr->gr_gid);
if (ret)
    perror ("chown");

```

Перед исполнением вызова файл принадлежит группе crew:

```

$ ls -l
-rw-r--r-- 1 kidd crew 13274 May 23 09:20 manifest.txt

```

После исполнения файл принадлежит только группе officers:

```

$ ls -l
-rw-r--r-- 1 kidd officers 13274 May 23 09:20 manifest.txt

```

Владелец файла, kidd, не изменился, так как в коде для параметра uid было передано значение -1.

Эта функция устанавливает файлу, определенному fd, пользователя-владельца и группу-владельца root:

```

/*
 * make_root_owner — изменяет владельца и группу файла,
 * определенного 'fd', на root. Возвращает 0 в случае успеха
 * и -1 в случае неудачи.
 */
int make_root_owner (int fd)
{
    int ret;

    /* 0 является идентификатором и группы, и пользователя для root */
    ret = fchown (fd, 0, 0);
    if (ret)
        perror ("fchown");
    return ret;
}

```

Вызывающий процесс должен обладать свойством CAP_CHOWN. В отношении характеристик это обычно означает, что он должен принадлежать root.

Расширенные атрибуты

Расширенные атрибуты представляют собой механизм для создания постоянных связей между файлами и парами «ключ — значение». В этой главе мы уже обсуждали разные виды метаданных «ключ — значение», связанных с файлами: размер, владелец, момент последней модификации файла и т. д. Расширенные атрибуты позволяют существующей файловой системе поддерживать новые функции, не предусмотренные изначально, например обязательный контроль доступа в целях обеспечения безопасности. Особенно интересно в расширенных атрибутах то, что приложения из пользовательского пространства могут самостоятельно создавать, считывать и записывать пары «ключ — значение».

Расширенные атрибуты *не связаны с файловой системой* в том смысле, что эти приложения используют стандартный интерфейс для управления ими; интерфейс не является специфичным для какой-либо файловой системы. Приложения, таким образом, могут использовать расширенные атрибуты, не заботясь, в какой файловой системе находятся файлы или как эта система хранит свои ключи и значения. Однако все-таки внедрение расширенных атрибутов значительно зависит от файловой системы. Разные файловые системы хранят расширенные атрибуты различными способами, но ядро скрывает эти различия, абстрактно представляя их с помощью системы интерфейсов расширенных атрибутов.

Файловая система `ext4`, например, хранит расширенные атрибуты файлов в пустом пространстве внутри индексного дескриптора файла¹. Эта функциональность делает считывание расширенных атрибутов значительно быстрее. Поскольку блок файловой системы, содержащий индексный дескриптор, считывается с диска в память, как только приложение обращается к файлу, то и расширенные атрибуты «автоматически» отправляются в память и могут быть доступны без каких-либо дополнительных затрат.

Другие файловые системы, например `FAT` и `minixfs`, не поддерживают расширенные атрибуты вообще. Эти системы возвращают `ENOTSUP`, если к файлам направляется действие, связанное с расширенными атрибутами.

Ключи и значения

Уникальный *ключ* идентифицирует каждый расширенный атрибут. Ключи должны соответствовать кодировке UTF-8. Каждый ключ обладает структурой пространство_имен.атрибут. Каждый ключ должен быть полностью уточнен; это значит, что он должен начинаться с указания допустимого пространства имен, после которого следует точка. Пример допустимого названия ключа — `user.mime_type`; этот ключ находится в пользовательском пространстве `user` и обладает атрибутом с названием `mime_type`.

¹ Конечно, пока в индексном дескрипторе остается пространство. После этого файловая система `ext4` начинает записывать расширенные атрибуты в свои дополнительные блоки.

Старые и новые способы хранить типы MIME в файловой системе

Файловые менеджеры с графическим интерфейсом наподобие GNOME подходят по-разному к файлам различных типов: они предлагают уникальные значки, разное поведение по умолчанию при щелчке на файле, специальные списки доступных для выполнения операций и т. д. Чтобы обеспечить все это, файловый менеджер должен знать формат каждого файла. Для определения формата системы наподобие Windows просто смотрят на расширение файла. Системы UNIX, однако, из соображений как безопасности, так и сохранения традиций стараются исследовать каждый файл и интерпретировать его тип. Этот процесс называется исследованием типа MIME.

Некоторые файловые менеджеры генерируют такую информацию на лету, то есть при любой необходимости, другие — лишь однажды, затем кэшируя ее. Предпочитающие кэширование обычно помещают информацию в собственную базу данных. Файловый менеджер должен постоянно держать эту базу синхронизированной с файлами, которые могут измениться, не уведомляя менеджера базы, поэтому лучше просто отказаться от использования базы данных и держать такие метаданные в расширенных атрибутах: их проще поддерживать, быстрее считывать, а также они доступны из любого приложения.

Ключ может быть *определенным* или *неопределенным*. Если ключ является определенным, его значение может быть пустым или непустым. Таким образом, есть разница между определенным ключом, которому не назначено значение, и неопределенным. Как вы скоро узнаете, это значит, что требуется специальный интерфейс для удаления ключей, а просто назначить им пустое значение недостаточно.

Величина, ассоциированная с ключом, если она непустая, может быть любым произвольным массивом байтов. Поскольку эта величина не всегда представляет собой текстовую строку, не обязательно завершать ее нулем, хотя это может быть разумно, если вы решите сохранить в качестве значения ключа строку C. Считывая атрибут, ядро предоставляет размер; записывая атрибут, вы должны предоставить размер самостоятельно.

Linux не ограничивает количество ключей, их длину, размер значения или общее пространство, которое может быть занято всеми ключами или значениями, связанными с файлом. Файловые системы, однако, имеют некоторые технические пределы. Как правило, они ограничивают общий размер всех ключей и значений, связанных с данным файлом.

В ext3, например, все расширенные атрибуты для данного файла должны укладываться в пустое пространство внутри индексного дескриптора файла и занимать в файловой системе не более одного блока (более старые версии ext3 имели ограничение до одного блока в файловой системе без дополнительного хранилища внутри индексного дескриптора). На практике это эквивалентно ограничению приблизительно 1–8 Кбайт на файл в зависимости от размера блоков в файловой системе. В XFS, наоборот, практически не имеется ограничений. Впрочем, даже

в ext3 лимиты, как правило, не представляют проблемы, поскольку большинство ключей и значений — короткие текстовые строки. Однако забывать о них все равно нельзя — дважды подумайте перед тем, как сохранить всю историю управления версиями проекта в расширенных атрибутах файла!

Пространства имен расширенных атрибутов

Пространства имен, связанные с расширенными атрибутами, представляют собой нечто большее, чем просто организационные средства. Ядро реализует различные политики доступа в зависимости от вида пространства имен.

Linux в настоящее время определяет четыре пространства имен для расширенных атрибутов и может добавить новые в будущем. Существующие четыре — следующие.

- `system` — это пространство используется для реализации функциональностей ядра, использующих расширенные атрибуты, таких как списки управления доступом (ACLs). Примером расширенного атрибута в этом пространстве имен может быть `system.posix_acl_access`. Могут ли пользователи считывать или записывать информацию в эти атрибуты, зависит от используемого модуля безопасности. Предполагайте худший вариант — никакие пользователи (включая `root`) не могут даже читать эти атрибуты.
- `security` — данное пространство используется для внедрения модулей безопасности, например SELinux. Как и в предыдущем пространстве, возможность для пользователей считывать или записывать информацию в эти атрибуты зависит от используемого модуля безопасности. По умолчанию все процессы могут считывать свои атрибуты, но только процесс с `CAP_SYS_ADMIN` может записывать в них данные.
- `trusted` — это пространство имен хранит закрытую для доступа информацию в пользовательском пространстве. Только процесс со свойством `CAP_SYS_ADMIN` может считывать и записывать эти атрибуты.
- `user` — данное пространство имен — стандартное для использования большинством обычных процессов. Ядро управляет доступом к этому пространству через обычные биты разрешений для файлов. Чтобы считать значение существующего ключа, процесс должен иметь право чтения данного файла. Чтобы создать новый ключ или записать значение в существующий, процесс должен иметь право записи в данный файл. Вы можете назначить расширенные атрибуты в пользовательском пространстве имен только обычным файлам, но не символическим ссылкам или файлам устройств. Разрабатывая приложение для пользовательского пространства, использующее расширенные атрибуты, скорее всего, вы выберете именно это пространство.

Действия с расширенными атрибутами

POSIX определяет четыре действия, которые приложение может проделать с расширенными атрибутами данного файла:

- для указанного файла возвращается список всех ключей расширенных атрибутов, назначенных файлу;
- для заданных файла и ключа возвращаются соответствующие величины;
- для известных файла, ключа и значения можно назначить это значение известному ключу;
- для указанных файла и ключа можно удалить расширенный атрибут из файла. Для каждого из этих действий POSIX предлагает три системных вызова:
- версию, которая работает с указанным именем файла; если путь — символическая ссылка, то действие производится над объектом ссылки (обычное поведение);
- версию, работающую с указанным путем к файлу; если путь ведет к символической ссылке, действие производится над ней (стандартный l-вариант системного вызова);
- версию, работающую с файловым дескриптором (стандартный f-вариант).

В разделах далее мы рассмотрим все 12 комбинаций.

Получение расширенного атрибута

Самое простое действие — получение значения расширенного атрибута из файла по известному ключу:

```
#include <sys/types.h>
#include <attr/xattr.h>
```

```
ssize_t getxattr (const char *path, const char *key,
                  void *value, size_t size);
ssize_t lgetxattr (const char *path, const char *key,
                   void *value, size_t size);
ssize_t fgetxattr (int fd, const char *key,
                  void *value, size_t size);
```

Успешный вызов `getxattr()` хранит расширенный атрибут с именем `key` из файла `path` в предоставленном пользователем буфере `value`, который имеет длину `size` байт. Он возвращает значение величины.

Если размер `size` равен 0, то вызов возвращает размер величины без ее сохранения в буфере `value`. Таким образом, установка 0 позволяет приложениям определить точный размер буфера, в котором хранится значение ключа. Передавая этот размер, приложения могут затем передавать или менять размер буфера.

`lgetxattr()` ведет себя аналогично `getxattr()` за исключением случая, когда `path` — символическая ссылка. Тогда вызов возвращает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

`fgetxattr()` работает с файловым дескриптором `fd`; в остальном он ведет себя аналогично `getxattr()`.

В случае ошибки все три вызова возвращают -1 и устанавливают `errno` одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок-компонентов пути path (касается только `getattr()` и `lgetattr()`);
- EBADF — дескриптор некорректен (касается только `fgetattr()`);
- EFAULT — указатели path, key, value являются недопустимыми;
- ELOOP — путь path содержит слишком много символических ссылок (касается только `getattr()` и `lgetattr()`);
- ENAMETOOLONG — путь path слишком велик (касается только `getattr()` и `lgetattr()`);
- ENOATTR — атрибута key не существует или процесс не имеет доступа к нему;
- ENOENT — какого-либо компонента пути не существует (касается только `getattr()` и `lgetattr()`);
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только `getattr()` и `lgetattr()`);
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты;
- ERANGE — размер size слишком мал, чтобы сохранить величину key; как было сказано выше, вызов может быть повторен с size, равным 0; возвращаемая величина будет означать требуемый размер буфера, и величина value может быть изменена соответственно.

Установка расширенного атрибута

Следующие три системных вызова устанавливают указанный расширенный атрибут:

```
#include <sys/types.h>
#include <attr/xattr.h>

int setattr (const char *path, const char *key,
             const void *value, size_t size, int flags);
int lsetattr (const char *path, const char *key,
             const void *value, size_t size, int flags);
int fsetattr (int fd, const char *key,
             const void *value, size_t size, int flags);
```

Успешный вызов `setattr()` устанавливает расширенный атрибут key файлу path равным value, имеющего size байт длины. Поле flags управляет поведением вызова. Если в flags прописано `XATTR_CREATE` (создание расширенного атрибута) и расширенный атрибут уже существует, вызов приведет к ошибке. Если в flags прописано `XATTR_REPLACE` (замещение расширенного атрибута), а расширенных атрибутов еще нет, вызов приведет к ошибке. По умолчанию, если в flags указан 0, позволяют и создание, и замена расширенных атрибутов. Независимо от значения flags, никакие ключи, кроме указанных в key, не затрагиваются.

Вызов `lsetxattr()` работает аналогично `setxattr()`, за исключением случая, когда `path` — символическая ссылка. Тогда вызов устанавливает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

`Fsetxattr()` работает с файловым дескриптором `fd`; в остальном он ведет себя аналогично `getxattr()`.

В случае успеха все три системных вызова возвращают 0; при неудаче вызовы возвращают -1 и присваивают `errno` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет прав доступа для поиска в одной из папок-компонентов пути `path` (касается только `setxattr()` и `lsetxattr()`);
- `EBADF` — дескриптор некорректен (касается только `fgetxattr()`);
- `EDQUOT` — предельный размер квоты не позволяет использовать пространство, необходимое для выполнения запрошенной операции;
- `EEXIST` — значение `flags` установлено `XATTR_CREATE`, но у данного файла уже существует ключ `key`;
- `EFAULT` — указатели `path`, `key`, `value` являются недопустимыми;
- `ELoop` — путь `path` содержит слишком много символических ссылок (касается только `setxattr()` и `lsetxattr()`);
- `ENAMETOOLONG` — путь `path` слишком велик (касается только `setxattr()` и `lsetxattr()`);
- `ENOATTR` — значение `flags` установлено `XATTR_REPLACE`, но у данного файла не существует ключа `key`;
- `ENOENT` — какого-либо компонента пути не существует (касается только `setxattr()` и `lsetxattr()`);
- `ENOMEM` — недостаточно доступной памяти для выполнения запроса;
- `ENOSPC` — недостаточно пространства в файловой системе для хранения расширенного атрибута;
- `ENOTDIR` — один из компонентов `path` не является каталогом (касается только `setxattr()` и `lsetxattr()`);
- `ENOTSUP` — файловая система, в которой находится путь `path` с дескриптором `fd`, не поддерживает расширенные атрибуты.

Перечисление расширенных атрибутов файла

Следующие три системных вызова выводят весь набор ключей расширенных атрибутов, назначенных данному файлу:

```
#include <sys/types.h>
#include <attr/xattr.h>
```

```
ssize_t listxattr (const char *path,
```

```

        char *list, size_t size);
ssize_t llistxattr (const char *path,
        char *list, size_t size);
ssize_t flistxattr (int fd,
        char *list, size_t size);

```

Успешный вызов `listxattr()` возвращает список ключей расширенных атрибутов, связанных с файлом, определенным через `path`. Список хранится в буфере `list`, размер которого равен `size` байт. Системный вызов возвращает фактический размер списка в байтах.

Каждый ключ расширенного атрибута, возвращенного в списке `list`, заканчивается нулевым символом, поэтому список выглядит примерно так:

```
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"
```

Таким образом, хотя каждый ключ представляет собой традиционную, оканчивающуюся нулем строку C, чтобы пройти по списку ключей, нужно знать его длину (которая доступна из возвращаемой вызовом величины). Чтобы определить размер буфера, необходимый для размещения списка, следует вызвать одну из списочных функций с параметром `size`, равным 0. Это заставит функцию вернуть актуальную длину полного списка ключей. Как и с `getxattr()`, приложения могут использовать эту функциональность для размещения или изменения размера буфера для передачи `value`.

`Llistxattr()` ведет себя аналогично `listxattr()`, за исключением случая, когда `path` — символическая ссылка. Тогда вызов обрабатывает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

`Flistxattr()` работает с файловым дескриптором `fd`; в остальном он ведет себя аналогично `listxattr()`.

В случае успеха все три вызова возвращают -1 и присваивают `errno` один из следующих кодов ошибки:

- `EACCES` — вызывающий процесс не имеет прав доступа для поиска в одной из папок-компонентов пути `path` (касается только `listxattr()` и `llistxattr()`);
- `EBADF` — дескриптор имеет недопустимое значение (касается только `flistxattr()`);
- `EFAULT` — указатели `path` или `list` недопустимы;
- `ELOOP` — путь `path` содержит слишком много символических ссылок (касается только `listxattr()` и `llistxattr()`);
- `ENAMETOOLONG` — путь `path` слишком велик (касается только `listxattr()` и `llistxattr()`);
- `ENOENT` — какого-либо компонента пути не существует (касается только `listxattr()` и `llistxattr()`);
- `ENOMEM` — недостаточно памяти для выполнения запроса;
- `ENOTDIR` — один из компонентов `path` не является каталогом (касается только `listxattr()` и `llistxattr()`);

- ENOTSUP — файловая система, в которой находится путь `path` с дескриптором `fd`, не поддерживает расширенные атрибуты;
- ERANGE — `size` не равен 0, но недостаточно велик, чтобы вместить полный список ключей; приложение может повторить вызов с `size`, равным 0, чтобы выяснить размер списка, а затем изменить размер `value` и повторить системный вызов.

Удаление расширенного атрибута

Наконец, следующие три системных вызова удаляют указанный ключ из указанного файла:

```
#include <sys/types.h>
#include <attr/xattr.h>
```

```
int removexattr (const char *path, const char *key);
int lremovexattr (const char *path, const char *key);
int fremovexattr (int fd, const char *key);
```

Успешный вызов `removexattr()` удаляет расширенный атрибут `key` из файла `path`. Вспомните, что есть разница между неопределенным ключом и определенным ключом с пустым (нулевой длины) значением.

Вызов `lremovexattr()` ведет себя аналогично `removexattr()`, за исключением случая, когда `path` — символическая ссылка. Тогда вызов удаляет ключ атрибута самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

`Fsetxattr()` работает с файловым дескриптором `fd`; в остальном он ведет себя аналогично `getxattr()`.

В случае успеха все три системных вызова возвращают 0; при неудаче вызовы возвращают -1 и присваивают `errno` одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок-компонентов пути `path` (касается только `removexattr()` и `lremovexattr()`);
- EBADF — дескриптор имеет недопустимое значение (касается только `fremovexattr()`);
- EFAULT — указатели `path` или `key` являются недопустимыми;
- ELOOP — путь `path` содержит слишком много символических ссылок (касается только `removexattr()` и `lremovexattr()`);
- ENAMETOOLONG — путь `path` слишком велик (касается только `removexattr()` и `lremovexattr()`);
- ENOATTR — для данного файла `key` не существует;
- ENOENT — какой-либо компонент пути не существует (касается только `removexattr()` и `lremovexattr()`);

- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только `removexattr()` и `lremovexattr()`);
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты.

Каталоги

Концепция *каталогов* в UNIX очень проста: они содержат список названий файлов, с каждым из которых связан номер индексного дескриптора. Каждое имя называется *записью в каталоге*, а каждая связь имени и номера индексного дескриптора именуется *ссылкой*. Содержимое директории, которое пользователь видит в результате выполнения команды `ls`, является списком всех имен файлов в этом каталоге. Когда пользователь открывает файл в данном каталоге, ядро ищет имя файла в списке данной директории, чтобы определить соответствующий номер индексного дескриптора. Затем ядро передает этот номер в файловую систему, которая использует его, чтобы определить физическое размещение файла на устройстве.

Каталоги могут также содержать другие каталоги. *Подкаталогом* называется каталог, находящийся внутри другого каталога. Согласно этому определению все каталоги являются подкаталогами каких-либо *родительских каталогов*, за исключением каталога в корне дерева файловой системы, `/`. Этот каталог так и называется *корневым каталогом* (не следует путать его с домашним каталогом пользователя `root`, то есть `/root`).

Путь к файлу состоит из имени файла вместе с одним или несколькими его родительскими каталогами. *Абсолютным путем к файлу* называется путь, который начинается с корневого каталога, например `/usr/bin/sextant`. *Относительным путем* называется путь, который не начинается с корневого каталога, к примеру `bin/sextant`. Чтобы использовать такой путь, операционная система должна знать каталог, к которому он относится. Текущий рабочий каталог (обсуждаемый в следующем разделе) используется в качестве стартовой точки.

Названия файла и папки могут содержать любые символы, кроме `/`, разделяющих каталоги в записи пути к файлу, и пустого значения (`null`), завершающего путь к файлу. Тем не менее принято ограничивать использование символов в записях пути: как правило, используются только допустимые печатные символы из текущих языковых настроек или даже только символы ASCII. Однако поскольку ни ядро, ни библиотеки C не предписывают такую практику, разработчики приложений самостоятельно принимают решение о допустимости использования тех или иных символов.

В старых системах UNIX длина имени файла ограничивалась 14 символами. Сегодня все современные системы UNIX позволяют использовать для каждого файла по меньшей мере 255 байт. (Обратите внимание: речь идет не о 255 символах, а о 255 байтах. Многобайтовые символы, разумеется, займут больше одного байта

из этих 255.) Многие файловые системы под Linux позволяют использование даже более длинных имен файлов¹.

Каждая папка содержит две особые папки: `.` и `..` (они называются «точка» и «точка-точка»). Точка — ссылка на саму папку. Точка-точка — ссылка на родительский каталог для данного каталога. Например, `/home/kidd/gold/..` — то же самое, что `/home/kidd`. Для каталога `root` каталоги точка и точка-точка ссылаются сами на себя: `/`, `/.` и `/..` — это одна и та же папка. Технически говоря, таким образом, `root` тоже является подкаталогом — в данном случае самого себя.

Текущий рабочий каталог

Каждый процесс имеет текущий каталог, который наследует непосредственно от своего родительского процесса. Этот каталог называется *текущим рабочим каталогом* процесса. Текущая рабочая директория является начальной точкой, из которой ядро прокладывает относительные пути к файлам. Например, если текущая рабочая директория процесса `/home/blackbeard`, то, когда процесс пытается открыть `parrot.jpg`, ядро будет стараться открыть файл `/home/blackbeard/parrot.jpg`. Однако если процесс попытается открыть `/usr/bin/mast`, то ядро в самом деле откроет `/usr/bin/mast`. Текущий рабочий каталог никак не влияет на абсолютные пути к файлам (то есть пути, начинающиеся с косой черты).

Процесс может получить и изменить свою рабочую папку.

Получение текущего рабочего каталога

Предпочтительный метод получения текущего рабочего каталога — системный вызов `getcwd()`, регламентированный в POSIX:

```
#include <unistd.h>
```

```
char * getcwd (char *buf, size_t size);
```

Успешный вызов `getcwd()` копирует текущий рабочий каталог как абсолютный путь в буфер, указанный как `buf` и имеющий длину `size` байт, и возвращает указатель к `buf`. В случае ошибки вызов возвращает `NULL` и присваивает `errno` одно из следующих значений:

- `EFAULT` — указатель `buf` имеет недопустимое значение;
- `EINVAL` — `size` равен 0, но `buf` не равен `NULL`;
- `ENOENT` — текущий рабочий каталог более не действителен; это может случиться, если текущий рабочий каталог был удален;

¹ Конечно, в старых файловых системах, которые поддерживает Linux для обеспечения обратной совместимости, например FAT, до сих пор действуют собственные ограничения. В случае FAT имя файла может включать только восемь символов, за которыми следуют точка и еще три символа. Нужно отметить, что принудительное использование точки в качестве специального символа в файловой системе — не самое мудрое решение.

- ERANGE — size слишком мал, чтобы текущий рабочий каталог был сохранен в buf; приложение должно выделить больший размер буфера и попробовать снова.

Пример использования getcwd():

```
char cwd[BUF_LEN];

if (!getcwd (cwd, BUF_LEN)) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

POSIX регламентирует, что поведение getcwd() не определено, если buf равен NULL. Библиотека C в Linux в данном случае выделит буфер длиной size байт и сохранит там текущий рабочий каталог. Если size равен 0, библиотека C выделит буфер достаточного размера, чтобы сохранить текущий рабочий каталог. Затем приложение должно очистить буфер через free(), окончив работу с ним. Это поведение уникально для Linux, поэтому нельзя полагаться на его надежную работу в условиях совместимости. Тем не менее эта функция делает использование очень простым. Пример:

```
char *cwd;

cwd = getcwd (NULL, 0);
if (!cwd) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);

free (cwd);
```

Библиотека C Linux также предоставляет функцию get_current_dir_name(), которая ведет себя аналогично getcwd() при получении NULL в буфере и size, равном 0:

```
#define _GNU_SOURCE
#include <unistd.h>

char * get_current_dir_name (void);
```

Таким образом, следующий фрагмент кода делает то же, что и предыдущий:

```
char *cwd;

cwd = get_current_dir_name ();
if (!cwd) {
    perror ("get_current_dir_name");
    exit (EXIT_FAILURE);
}
```

```
}  
  
printf ("cwd = %s\n", cwd);  
  
free (cwd);
```

В старых системах BSD пользовался популярностью вызов `getwd()`, который Linux поддерживает для обеспечения совместимости:

```
#define _XOPEN_SOURCE_EXTENDED /* или _BSD_SOURCE */  
#include <unistd.h>
```

```
char * getwd (char *buf);
```

Вызов `getwd()` копирует текущий рабочий каталог в буфер `buf`, который должен быть не менее `PATH_MAX` длиной. Вызов возвращает `buf` в случае успеха и `NULL` в случае ошибки. Например:

```
char cwd[PATH_MAX];  
  
if (!getwd (cwd)) {  
    perror ("getwd");  
    exit (EXIT_FAILURE);  
}  
  
printf ("cwd = %s\n", cwd);
```

По причинам как совместимости, так и безопасности в приложениях лучше не использовать `getwd()` — предпочтительнее `getcwd()`.

Изменение текущего рабочего каталога

Когда пользователь впервые авторизуется в системе, процесс авторизации устанавливает домашний каталог в качестве текущего рабочего, как указано в `/etc/passwd`. Иногда, однако, процессу необходимо изменить свой текущий рабочий каталог. Например, оболочка может захотеть сделать это, когда пользователь выполняет команду `cd`.

В Linux есть два системных вызова для изменения текущего рабочего каталога: один, который устанавливает путь к каталогу, и другой, который прописывает файловый дескриптор, представляющий открытый каталог.

```
#include <unistd.h>
```

```
int chdir (const char *path);  
int fchdir (int fd);
```

Вызов `chdir()` изменяет текущий рабочий каталог согласно пути, указанному в `path`, который может быть абсолютным или относительным. Аналогично вызов `fchdir()` изменяет текущий рабочий каталог согласно пути, указанному через файловый дескриптор `fd`, который должен быть открыт для этого каталога. В случае успеха оба вызова возвращают 0, при неудаче возвращается -1.

При ошибке `chdir()` присваивает `errno()` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет полномочий на поиск в одном или нескольких каталогах, являющихся компонентами пути `path`;
- `EFAULT` — указатель `path` является недопустимым;
- `EIO` — произошла внутренняя ошибка ввода-вывода;
- `ELOOP` — ядро обнаружило в пути `path` слишком много символических ссылок;
- `ENAMETOOLONG` — путь `path` слишком велик;
- `ENOENT` — каталог, указанный через `path`, не существует;
- `ENOMEM` — недостаточно памяти для выполнения запроса;
- `ENOTDIR` — один или несколько компонентов пути `path` не являются каталогом.

Вызов `fchdir()` устанавливает `errno` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет полномочий на поиск в каталоге, указанном через `fd` (например, бит исполнения не установлен); это может случиться, если каталог верхнего уровня доступен для чтения, но не для исполнения; вызов `open()` будет успешным, но `fchdir()` — нет;
- `EBADF` — `fd` не является дескриптором открытого файла.

В зависимости от файловой системы другие значения ошибок также возможны для этого вызова.

Данные системные вызовы могут воздействовать только на процессы, запущенные в настоящее время. В UNIX нет механизма для изменения текущего рабочего каталога для другого процесса. Таким образом, команда `cd` в оболочках не может быть отдельным процессом (как большинство команд), который выполняет `chdir()` на первом аргументе в командной строке, а затем завершается. Вместо этого `cd` должна быть специальной встроенной командой, которая заставляет оболочку самостоятельно вызывать `chdir()`, изменяя собственный текущий каталог.

Чаще всего `getswd()` используется для сохранения рабочего каталога, чтобы процесс мог позднее в него вернуться. Например:

```
char *swd;

int ret;

/* Сохраняется текущий рабочий каталог */
swd = getcwd (NULL, 0);
if (!swd) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
}

/* Переход в другой рабочий каталог */
ret = chdir (some_other_dir);
if (ret) {
    perror ("chdir");
}
```



```
        exit (EXIT_FAILURE);
    }

    /* Выполнение какой-то работы в новом каталоге... */

    /* Возврат в сохраненный каталог */
    ret = chdir (swd);
    if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
    }

    free (swd);
```

Однако лучше сначала открыть вызовом `open()` текущий каталог, а затем возвращаться в него с помощью `fchdir()`. Это быстрее, так как ядро не сохраняет в памяти полный путь к текущему рабочему каталогу; хранится только структура `inode`. Следовательно, когда пользователь вызывает `getcwd()`, ядро должно сгенерировать путь к файлу, проходя через структуру каталога. И наоборот, открытие текущего рабочего каталога менее затратно, так как в этом случае у ядра уже есть его `inode` и путь к файлу, воспринимаемый человеком, не требуется. В следующем фрагменте кода использован именно этот подход:

```
int swd_fd;

swd_fd = open (".", O_RDONLY);
if (swd_fd == -1) {
    perror ("open");
    exit (EXIT_FAILURE);
}

/* Переход в другой каталог */
ret = chdir (some_other_dir);
if (ret) {
    perror ("chdir");
    exit (EXIT_FAILURE);
}

/* Выполнение какой-то работы в новом каталоге... */

/* Возврат в сохраненный каталог */
ret = fchdir (swd_fd);
if (ret) {
    perror ("fchdir");
    exit (EXIT_FAILURE);
}

/* Закрытие дескриптора fd каталога */
ret = close (swd_fd);
if (ret) {
```

```
    perror("close");  
    exit(EXIT_FAILURE);  
}
```

Именно так оболочка выполняет кэширование предыдущего каталога (например, для `cd` — в `bash`).

Процессы, которым не приходится заботиться о своем текущем рабочем каталоге, например демоны, обычно устанавливают в качестве значения корневой каталог с помощью вызова `chdir("/")`. Приложение, которое взаимодействует с пользователем и его данными, например текстовый редактор, чаще всего устанавливает в качестве своего текущего рабочего каталога домашний каталог пользователя или специальный каталог для документов. Понятие текущего рабочего каталога существует только в контексте относительных путей к файлам, текущий рабочий каталог наиболее полезен для утилит командной строки, которые вызывает пользователь из оболочки.

Создание каталогов

Для создания каталогов Linux предоставляет только один системный вызов, регламентированный POSIX:

```
#include <sys/stat.h>  
#include <sys/types.h>  
  
int mkdir (const char *path, mode_t mode);
```

Успешный вызов `mkdir()` создает путь к каталогу `path`, который может быть относительным или абсолютным с битами разрешения `mode` (как указано в текущем значении `umask`), и возвращает 0.

Текущее значение `umask` изменяет обычным способом аргумент `mode`, а также любые биты данного режима, уникальные в данной системе. В Linux — биты разрешения вновь создаваемого каталога (`mode & ~umask & 01777`). Другими словами, параметр `umask` для данного процесса определяет значения, которые не могут быть переопределены с помощью вызова `mkdir()`. Если для нового родительского каталога текущего каталога установлен общий групповой идентификатор битов (`set group ID`, или `sgid`) или файловая система смонтирована с семантикой групп BSD, то новый каталог наследует групповую принадлежность родителя. В ином случае новый каталог получит действительный идентификатор группы процесса.

При ошибке `mkdir()` возвращает -1 и присваивает `errno` одно из следующих значений:

- `EACCES` — текущий процесс не имеет прав записи в родительском каталоге или прав поиска в одном из каталогов-компонентов пути `path`;
- `EEXIST` — путь `path` уже существует (причем не обязательно в виде каталога);
- `EFAULT` — указатель `path` является некорректным;
- `ELOOP` — ядро обнаружило в пути `path` слишком много символических ссылок;
- `ENAMETOOLONG` — путь `path` слишком велик;

- ENOENT — какой-либо компонент пути `path` не существует или является символической ссылкой, ведущей к несуществующему объекту;
- ENOMEM — недостаточно памяти ядра для выполнения запроса;
- ENOSPC — на устройстве, где находится `path`, недостаточно свободного пространства либо для данного пользователя превышена квота места на диске;
- ENOTDIR — один или несколько компонентов пути `path` не являются каталогом;
- EPERM — файловая система, в которой находится `path`, не поддерживает создание каталогов;
- EROFS — файловая система, в которой находится `path`, монтирована с доступом только для чтения.

Удаление каталогов

В противоположность `mkdir()` регламентированный POSIX вызов `rmdir()` удаляет каталог из иерархии файловой системы:

```
#include <unistd.h>
```

```
int rmdir (const char *path);
```

В случае успеха `rmdir()` удаляет `path` из файловой системы и возвращает 0. Каталог, указанный через `path`, должен быть пустым, за исключением каталогов точка и точка-точка. Не существует системного вызова, реализующего аналог рекурсивного удаления, наподобие `rm -r`. Можно выполнить подобную операцию вручную, продвигаясь в глубь файловой системы и удаляя все файлы и папки, начиная с листьев и переходя к корню файловой системы; на каждой стадии может быть использован `rmdir()` для удаления каталога сразу после того, как были удалены все находившиеся в нем файлы.

В случае ошибки `rmdir()` возвращает -1 и присваивает `errno` одно из следующих значений:

- EACCES — текущий процесс не имеет прав записи в родительском каталоге или прав поиска в одном из каталогов-компонентов пути `path`;
- EBUSY — `path` в настоящее время используется системой и не может быть удален; в Linux это может случиться, только если `path` является точкой сборки для корневого каталога (хотя корневые каталоги не должны быть точками сборки благодаря `chroot(!)`);
- EFAULT — указатель `path` является недопустимым;
- EINVAL — финальным компонентом пути `path` является каталог точка;
- ELOOP — ядро обнаружило в пути `path` слишком много символических ссылок;
- ENAMETOOLONG — путь `path` слишком велик;
- ENOENT — какой-либо компонент пути `path` не существует или является символической ссылкой, ведущей к несуществующему объекту;

- ENOMEM — недостаточно памяти ядра для выполнения запроса;
- ENOTDIR — один или несколько компонентов пути path не являются каталогом;
- ENOTEMPTY — каталог, указанный через path, содержит иные элементы, чем каталоги точка или точка-точка;
- EPERM — для каталога, являющегося предком каталога path, установлен бит закрепления в памяти, но действительный идентификатор пользователя процесса не совпадает ни с идентификатором пользователя родительского процесса, ни с идентификатором каталога path, а процесс не имеет свойства CAP_FOWNER; или файловая система, где находится path, не допускает удаления каталогов;
- EROFS — файловая система, в которой находится path, монтирована с доступом только для чтения.

Использование очень просто:

```
int ret;
```

```
/* удаление каталога /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
    perror ("rmdir");
```

Чтение содержимого каталога

POSIX определяет семейство функций для чтения содержимого каталогов, то есть для получения списка файлов, которые относятся к данной директории. Эти функции полезны, если вы реализуете `ls` или графический интерфейс диалога сохранения, выполнение операций со всеми файлами каталога или поиск среди файлов каталога тех, которые соответствуют определенному шаблону.

Чтобы начать чтение содержимого, вы должны создать *поток каталога*, который представлен объектом DIR:

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR * opendir (const char *name);
```

Успешный вызов `opendir()` создает поток каталога, представляющий каталог, определенный через `name`.

Поток каталога является, по сути, просто файловым дескриптором, представляющим открытый каталог, некоторые метаданные и буфер для сохранения содержимого. Следовательно, возможно получить файловый дескриптор для соответствующего каталога внутри данного потока каталога:

```
#define _BSD_SOURCE /* или _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>
```

```
int dirfd (DIR *dir);
```

Успешный вызов `dirfd()` возвращает файловый дескриптор, соответствующий потоку каталога `dir`. В случае ошибки вызов возвращает `-1`. Поскольку функция потока каталога использует данный дескриптор при выполнении своей задачи, программы не должны обращаться к вызовам, изменяющим положение файлов. `Dirfd()` является расширением BSD и не стандартизирован POSIX; программисты, стремящиеся к полному соответствию POSIX, должны избегать его.

Чтение из потока каталога

После того как с помощью вызова `opendir()` создан поток каталога, программа может читать находящиеся в нем записи. Чтобы сделать это, используйте `readdir()`, который возвращает записи одну за другой из указанного объекта `DIR`:

```
#include <sys/types.h>
#include <dirent.h>
```

```
struct dirent * readdir (DIR *dir);
```

Успешный вызов `readdir()` возвращает следующую запись в каталоге, представленную через `dir`. Структура `dirent` представляет запись в каталоге. Она определена в заголовочном файле `<dirent.h>` в Linux следующим образом:

```
struct dirent {
    ino_t d_ino; /* номер inode */
    off_t d_off; /* переход к следующей записи dirent */
    unsigned short d_reclen; /* длина данной записи */
    unsigned char d_type; /* тип файла */
    char d_name[256]; /* имя файла */
};
```

POSIX требует только наличия поля `d_name`, которое представляет собой имя файла в данном каталоге. Прочие поля необязательные или являются уникальными для Linux. Приложения, требующие совместимости с другими системами или строгого соответствия POSIX, должны иметь доступ только к `d_name`.

Приложения последовательно вызывают `readdir()`, получая файлы из каталога один за другим, пока записи в каталоге не закончатся либо обнаружение новых файлов не прекратится. В первом случае `readdir()` возвращает значение `NULL`.

В случае ошибки `readdir()` также возвращает `NULL`. Чтобы отличить ошибку от окончания списка файлов, перед каждым вызовом `readdir()` следует присвоить переменной `errno` значение 0, а затем проверять значения и возвращаемой величины, и `errno`. Единственное значение `errno`, которое может установить `readdir()`, — это `EBADF`, указывающее, что значение `dir` недопустимо. Однако многие приложения не обрабатывают ошибки, и в них предполагается, что возвращение `NULL` означает лишь окончание файлов в каталоге.

Заккрытие потока каталога

Чтобы закрыть поток каталога, открытый с помощью `opendir()`, используйте `closedir()`:

```
#include <sys/types.h>
#include <dirent.h>
```

```
int closedir (DIR *dir);
```

Успешный вызов `closedir()` закрывает поток каталога, обозначенного с помощью `dir`, включая соответствующий файловый дескриптор, и возвращает 0. В случае ошибки функция возвращает -1 и устанавливает `errno` в значение `EBADF`, единственный возможный код ошибки, означающий, что `dir` не является открытым потоком каталога.

Следующий фрагмент кода представляет функцию `find_file_in_dir()`, которая использует `readdir()` для поиска данного имени файла в указанном каталоге. Если файл действительно находится в каталоге, функция возвращает 0. В ином случае она возвращает ненулевую величину:

```
/*
 * find_file_in_dir – ищет в каталоге 'path' файл
 * под названием 'file'.
 *
 * Возвращает 0, если 'file' существует в 'path', и ненулевую
 * величину во всех иных случаях
 */
int find_file_in_dir (const char *path, const char *file)
{
    struct dirent *entry;
    int ret = 1;
    DIR *dir;

    dir = opendir (path);

    errno = 0;
    while ((entry = readdir (dir)) != NULL) {
        if (strcmp(entry->d_name, file) == 0) {
            ret = 0;
            break;
        }
    }

    if (errno && !entry)
        perror ("readdir");

    closedir (dir);
    return ret;
}
```

Системные вызовы для чтения содержимого каталога

Рассмотренные выше функции для чтения содержимого каталога регламентированы POSIX и предоставляются библиотекой C. Внутри системы эти функции

используют один из двух системных вызовов — `readdir()` или `getdents()`, которые мы рассмотрим для полноты обсуждения:

```
#include <unistd.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <errno.h>
/*
 * Не определены в пользовательском пространстве:
 * для доступа необходим макрос _syscall3(). */
*/
int readdir (unsigned int fd,
             struct dirent *dirp,
             unsigned int count);

int getdents (unsigned int fd,
             struct dirent *dirp,
             unsigned int count);
```

Вам не нужно использовать эти системные вызовы. Они сложны в применении и ухудшают совместимость. Вместо этого приложения в пользовательском пространстве должны использовать системные вызовы `opendir()`, `readdir()` и `closedir()`, предоставляемые библиотекой C.

Ссылки

Вспомните: обсуждая каталоги, мы называли связь между именем и номером `inode` в каталоге *ссылкой*. Исходя из этого простого определения ссылка представляет собой всего лишь имя в списке (каталоге), которое указывает на `inode`, и, таким образом, может существовать несколько ссылок на один и тот же `inode`. Действительно, на один `inode` может ссылаться, скажем, как `/etc/customs`, так и `/var/run/ledger`.

Здесь и в самом деле кроется небольшой подвох. Ссылки привязаны к номерам `inode`, а номера `inode` уникальны в каждой файловой системе, следовательно, оба файла, `/etc/customs` и `/var/run/ledger`, должны принадлежать к одной и той же файловой системе. В пределах одной файловой системы к одному и тому же файлу может относиться много ссылок. Единственное ограничение касается размера целочисленного типа данных, используемого для хранения ссылок. Среди всех различных ссылок ни одна не может считаться «основной» или «оригинальной»: статус всех ссылок одинаков и все они указывают на один файл.

Такие типы ссылок мы называем *жесткими ссылками*. Файлы могут не иметь ни одной, иметь одну или много ссылок. Большинство файлов имеют количество ссылок, равное 1. Это значит, что на них ссылается единственная запись в каталоге, но некоторые файлы имеют две ссылки или даже больше. Файлы, количество ссылок на которые равно 0, не имеют соответствующих записей каталога в системе.

Когда количество ссылок на файл равно 0, файл помечается как свободный, а его дисковые блоки становятся свободными для использования¹. Такой файл, однако, остается в файловой системе, если он открыт у какого-либо процесса. После того как все процессы закрыли этот файл, он удаляется.

Ядро Linux реализует этот подход, используя счетчик ссылок и счетчик использований. *Счетчик использований* — это общее количество экземпляров, где открыт данный файл. Файл не удаляется из файловой системы, пока количество как ссылок, так и использований не станет равным 0.

Другой тип ссылки, *символическая ссылка*, является не маршрутизатором файловой системы, а более высокоуровневым указателем, который интерпретируется во время выполнения. Такие ссылки могут охватывать разные файловые системы — мы поговорим о них позже.

Жесткие ссылки

Системный вызов `link()`, один из первоначальных системных вызовов в UNIX, а сейчас стандартизированный и в POSIX, создает новую ссылку на существующий файл:

```
#include <unistd.h>
```

```
int link (const char *oldpath, const char *newpath);
```

Успешный вызов `link()` создает новую ссылку по пути `newpath` для существующего файла `oldpath`, а затем возвращает 0. По выполнении и `oldpath`, и `newpath` ссылаются на один и тот же файл — отныне фактически нельзя сказать, который из этих путей является «исходным».

В случае сбоя вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов-компонентов `oldpath` или не имеет прав записи в одном из каталогов, составляющих `newpath`;
- EEXIST — `newpath` уже существует — `link()` не может переписать существующую запись каталога;
- EFAULT — указатель `oldpath` или `newpath` недопустим;
- EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
- ELOOP — при разрешении пути `oldpath` или `newpath` было обнаружено слишком много символических ссылок;

¹ Поиск файлов с количеством ссылок, равным 0, но с блоками, помеченными как занятые, — основная задача `fscck`, утилиты контроля над файловой системой. Подобное условие может возникать, когда файл удаляется, но остается открытым, и в системе происходит аварийный сбой до закрытия файла. Ядро не имеет возможности пометить блоки файловой системы как свободные для использования, из-за чего и возникает несоответствие. Протоколирование файловой системы устраняет этот тип ошибок.

- EMLINK — inode, на который указывает oldpath, уже достиг максимально допустимого количества ссылок на себя;
- ENAMETOOLONG — путь oldpath или newpath слишком велик;
- ENOENT — один из компонентов oldpath или newpath не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOSPC — на устройстве, где находится newpath, недостаточно памяти для новой записи каталога;
- ENOTDIR — один из компонентов oldpath или newpath не является каталогом;
- EPERM — файловая система, где находится newpath, не позволяет создание новых жестких ссылок или oldpath является каталогом;
- EROFS — newpath находится в файловой системе, доступной только для чтения;
- EXDEV — oldpath и newpath не принадлежат к одной и той же монтированной файловой системе (Linux позволяет монтирование одной и той же файловой системы в нескольких точках, но даже в этом случае жесткие ссылки не могут создаваться между точками монтирования).

Следующий пример создает новую запись в каталоге — pirate, которая ссылается на тот же самый inode (и, следовательно, на тот же файл), что и существующий файл privateer; оба находятся в /home/kidd:

```
int ret;

/*
 * создание новой записи каталога,
 * '/home/kidd/privateer', которая указывает на
 * тот же самый inode, что и '/home/kidd/pirate'
 */
ret = link ("/home/kidd/privateer", /home/kidd/pirate");
if (ret)
    perror ("link");
```

Символические ссылки

Символические ссылки, еще известные как *симссылки* и *мягкие ссылки*, сходны с жесткими ссылками тем, что они также указывают на файлы в файловой системе. Однако символические ссылки представляют собой не просто дополнительную запись каталога, но и специальный тип файла. Этот специальный файл содержит путь к *другому* файлу, называемому *целью* символической ссылки. Во время выполнения ядро на лету заменяет путь к файлу символической ссылки путем к цели символической ссылки (кроме случая, когда в программе используются различные 1-версии системных вызовов, например 1-stat, которые работают с самой ссылкой, а не с ее целью). Таким образом, жесткие ссылки на один и тот же файл неотличимы друг от друга, а различия между символической ссылкой и ее целевым файлом очевидны.

Символическая ссылка может быть относительной или абсолютной. Она может также содержать специальный каталог точка, рассмотренный выше, относящийся

к каталогу, в котором он расположен, или каталог точка-точка, относящийся к каталогу верхнего уровня. Такие виды относительных символических ссылок широко применяются и часто весьма полезны.

Мягкие ссылки, в отличие от жестких, могут охватывать целые файловые системы. На самом деле они могут указывать куда угодно! Символические ссылки могут указывать на файл, который существует (обычное явление) или не существует. Последний тип ссылки называется *повисшей символической ссылкой*. Иногда повисшие ссылки нежелательны — например, когда цель этой ссылки была удалена, но осталась сама ссылка, — но временами они необходимы. Символическая ссылка даже может указывать на другую символическую ссылку. Таким образом можно создавать петли. Системные вызовы, имеющие дело с символическими ссылками, проверяют петли до максимально достижимой глубины. Если эта глубина превышена, они возвращают ELOOP.

Системный вызов для создания символических ссылок очень похож на своего двоюродного брата, предназначенного для жестких:

```
#include <unistd.h>
```

```
int symlink (const char *oldpath, const char *newpath);
```

Успешный вызов `symlink()` создает символическую ссылку `newpath`, указывающую на цель `oldpath`, а затем возвращает 0.

В случае ошибки `symlink()` возвращает -1, а затем присваивает `errno` одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов-компонентов `oldpath` или не имеет прав записи в одном из каталогов, составляющих `newpath`;
- EEXIST — `newpath` уже существует — `symlink()` не может переписать существующую запись каталога;
- EFAULT — указатель `oldpath` или `newpath` недействителен;
- EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
- ELOOP — при разрешении пути `oldpath` или `newpath` было обнаружено слишком много символических ссылок;
- EMLINK — `inode`, на который указывает `oldpath`, уже достиг максимально допустимого количества ссылок на себя;
- ENAMETOOLONG — путь `oldpath` или `newpath` слишком велик;
- ENOENT — один из компонентов `oldpath` или `newpath` не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOSPC — на устройстве, где находится `newpath`, недостаточно памяти для новой записи каталога;
- ENOTDIR — один из компонентов `oldpath` или `newpath` не является каталогом;

- EPERM — файловая система, где находится newpath, не позволяет создание новых символических ссылок;
- EROFS — newpath находится в файловой системе, доступной только для чтения.

Вот этот фрагмент кода — то же самое, что предыдущий пример, но он создает `/home/kidd/pirate` как символическую ссылку (в отличие от жесткой) на `/home/kidd/privateer`:

```
int ret;

/*
 * создание символической ссылки,
 * '/home/kidd/privateer', которая
 * указывает на '/home/kidd/pirate'
 */
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate");
if (ret)
    perror ("symlink");
```

Удаление ссылки

Ссылку можно как создать, так и удалить, разорвав путь в файловой системе. Эту задачу может выполнить единственный вызов, `unlink()`:

```
#include <unistd.h>
```

```
int unlink (const char *pathname);
```

Успешный вызов `unlink()` удаляет `pathname` из файловой системы и возвращает 0. Если этот путь был последней ссылкой на файл, файл удаляется из системы. Если, однако, файл открыт в каком-либо процессе, ядро не будет удалять файл из файловой системы, пока процесс не закроет этот файл. Если же этот файл не открыт ни в одном процессе, он удаляется.

Если `pathname` ведет к символической ссылке, удаляется сама ссылка, а не ее цель.

Если `pathname` ссылается на другой тип специального файла, например устройство, конвейер FIFO или сокет, этот файл удаляется из файловой системы, но процессы, у которых он открыт, могут продолжать его использовать.

В случае ошибки `unlink()` возвращает -1 и присваивает `errno` один из следующих кодов ошибки:

- EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов-компонентов `pathname` или не имеет прав записи в родительском каталоге `pathname`;
- EFAULT — указатель `pathname` имеет недопустимое значение;
- EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
- EISDIR — `pathname` ссылается на каталог;
- ELOOP — при прохождении `pathname` было обнаружено слишком много символических ссылок;

- ENAMETOOLONG — путь `pathname` слишком велик;
- ENOENT — один из компонентов `pathname` не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов `pathname` не является каталогом;
- EPERM — файловая система не позволяет удаление ссылок;
- EROFS — `pathname` находится в файловой системе, доступной только для чтения.

Вызов `unlink()` не удаляет каталоги. Для этого приложения должны использовать `rmdir()`, который мы обсуждали ранее (см. разд. «Удаление каталогов»).

Чтобы упростить удаление файлов любого типа, язык C предоставляет функцию `remove()`:

```
#include <stdio.h>
```

```
int remove (const char *path);
```

Успешный вызов `remove()` удаляет `path` из файловой системы и возвращает 0. Если `path` — это файл, `remove()` вызывает `unlink()`; если же это каталог, `remove()` вызывает `rmdir()`.

В случае ошибки `remove()` возвращает -1 и присваивает `errno` значение, соответствующее одному из возвращаемых `unlink()` и `rmdir()`.

Копирование и перемещение файлов

Две самые простые и распространенные операции с файлами — это их копирование и перемещение, обычно выполняемые с помощью команд `cp` и `mv`. На уровне файловой системы *копирование* означает дублирование содержимого файла в другое место и создание нового пути к файлу. Это отлично от создания новой жесткой ссылки на файл, так как изменение одного файла не влияет на другой: теперь существуют две различные копии файла, которым соответствуют две (по меньшей мере) разные записи каталога. *Перемещение*, напротив, является актом переименования записи каталога, под которой размещается файл. Это действие не приводит к созданию новой копии.

Копирование

Наверное, это вас удивит, но в Linux нет системного или библиотечного вызова для выполнения копирования файлов или каталогов. Вместо этого утилиты наподобие `cp` или файловые менеджеры GNOME выполняют эти операции вручную.

Для копирования файла `src` в файл под названием `dst` необходимо выполнить следующие действия.

1. Открыть файл `src`.
2. Открыть файл `dst` — создать его, если он не существует, или сократить до нулевой длины, если он есть.

3. Считать фрагмент содержимого `src` в память.
4. Записать этот фрагмент в `dst`.
5. Продолжать, пока все содержимое `src` не будет считано и переписано в `dst`.
6. Закрывать `src`.
7. Закрывать `dst`.

При копировании каталога отдельный каталог и его подкаталоги создаются с помощью `mkdir()`; каждый файл внутри них затем копируется индивидуально.

Перемещение

В отличие от копирования файлов, UNIX предоставляет системный вызов для перемещения. Стандарт ANSI C представляет вызов для файлов, а POSIX регламентирует вызов и для файлов, и для каталогов:

```
#include <stdio.h>
```

```
int rename (const char *oldpath, const char *newpath);
```

Успешный вызов `rename()` переименовывает путь `oldpath` в `newpath`. Содержимое файла и `inode` остаются теми же самыми. И `oldpath`, и `newpath` должны остаться в той же файловой системе¹; если это не так, вызов вернет ошибку. Утилиты наподобие `mv` обрабатывают этот случай копированием файла и удалением ссылки.

При успехе `rename()` возвращает 0, и путь к файлу `oldpath` меняется на `newpath`. В случае неудачи вызов возвращает -1, никак не влияя ни на `oldpath`, ни на `newpath`, и присваивает `errno` одно из следующих значений:

- `EACCES` — вызывающий процесс не имеет прав записи в родительских каталогах `oldpath` или `newpath`, прав поиска в одном из компонентов `oldpath` или `newpath` или прав записи для `oldpath`, если это каталог; последний случай может представлять проблему, так как `rename()` должен обновить `..` в `oldpath`, если это каталог;
- `EBUSY` — `oldpath` или `newpath` являются точкой монтирования;
- `EFAULT` — указатель `oldpath` или `newpath` является недопустимым;
- `EINVAL` — `newpath` находится внутри `oldpath`; таким образом, переименование одного в другой сделает `oldpath` подкаталогом самого себя;
- `EISDIR` — `newpath` существует и является каталогом, а `oldpath` не является;
- `ELoop` — при разрешении пути `oldpath` или `newpath` было обнаружено слишком много символических ссылок;
- `EMLINK` — `oldpath` уже достиг максимально допустимого количества ссылок на себя или `oldpath` является каталогом, а `newpath` уже достиг максимально допустимого количества ссылок на себя;

¹ Хотя Linux позволяет вам подмонтировать устройство в нескольких точках структуры каталогов, невозможно передвинуть файл с одной из точек монтирования на другую, несмотря на то что фактически обе точки находятся на одном устройстве.

- ENAMETOOLONG — пути `oldpath` или `newpath` слишком велики;
- ENOENT — один из компонентов `oldpath` или `newpath` не существует или является повисшей символической ссылкой;
- ENOMEM — недостаточно памяти ядра для выполнения запроса;
- ENOSPC — на устройстве, где находится `newpath`, недостаточно пространства для выполнения запроса;
- ENOTDIR — один из компонентов (за исключением потенциально последнего компонента) `oldpath` или `newpath` не является каталогом или `oldpath` — это каталог, а `newpath` уже существует и не является каталогом;
- ENOTEMPTY — `newpath` является каталогом и непустой;
- EPERM — по крайней мере один из указанных путей существует, для родительского каталога установлен бит закрепления в памяти, действительный идентификатор пользователя вызывающего процесса не совпадает ни с идентификатором пользователя файла, ни с идентификатором пользователя родительского процесса, а процесс не имеет привилегированных прав;
- EROFS — файловая система доступна только для чтения;
- EXDEV — `oldpath` и `newpath` находятся в разных файловых системах.

В табл. 8.1 приведены результаты перемещения различных типов файлов.

Таблица 8.1. Результаты перемещения различных типов файлов

Исходный объект является	Цель — файл	Цель — каталог	Цель — ссылка	Цель не существует
Файлом	Целевой файл заменяется на исходный	Ошибка EISDIR	Файл переименовывается, цель переписывается	Файл переименовывается
Каталогом	Ошибка ENOTDIR	Если целевой каталог пуст, то исходный файл переименовывается названием целевого файла; в противном случае возвращается ошибка ENOTEMPTY	Каталог переименовывается, цель переписывается	Каталог переименовывается
Ссылкой	Ссылка переименовывается, цель переписывается	Ошибка EISDIR	Ссылка переименовывается, цель переписывается	Ссылка переименовывается
Не существует	Ошибка ENOENT	Ошибка ENOENT	Ошибка ENOENT	Ошибка ENOENT

Во всех случаях, независимо от типов исходного и целевого объектов, если они находятся в разных файловых системах, вызов вернет ошибку EXDEV.

Узлы устройств

Узлами устройств называются специальные файлы, позволяющие приложениям взаимодействовать с драйверами устройств. Когда приложение выполняет обычный в UNIX ввод-вывод — открытие, закрытие, чтение, запись и т. д. — на узле устройства, ядро не обрабатывает эти запросы как обычный ввод-вывод файлов. Вместо этого ядро отправляет эти запросы драйверу устройства. Драйвер обрабатывает операцию ввода-вывода и возвращает пользователю результат. Узлы устройства обеспечивают абстрагирование устройств, благодаря чему от приложения не требуется знания специфики устройства или владения специальными интерфейсами. На самом деле узлы устройств — стандартный механизм для обеспечения доступа к аппаратному обеспечению в системах UNIX. Сетевые устройства — исключение; в истории UNIX встречались мнения об ошибочности такого исключения. В любом случае есть некоторая элегантность и красота в манипулировании всем машинным оборудованием с использованием вызовов `read()`, `write()` и `mmap()`.

Как же ядро определяет драйвер устройства, которому следует передать запрос? Каждому узлу устройства назначаются две числовые величины, называемые *старшим номером* и *младшим номером*. Эти номера связаны со специальным драйвером устройства, открытым в ядре. Если узел устройства не имеет старших и младших номеров, соответствующих драйверу устройства в ядре, что иногда может произойти по определенным причинам, то запрос `open()` на этом устройстве вернет `-1` и присвоит `errno` код ошибки `ENODEV`. Мы говорим, что этот узел устройства представляет несуществующее устройство.

Специальные узлы устройств

Несколько узлов устройств представлены во всех системах Linux. Эти узлы являются частью среды разработки Linux, и их наличие считается частью Linux ABI.

У *фиктивного устройства* старший номер равен 1, а младший равен 3. Оно находится в `/dev/null`. Файлом устройства должен владеть пользователь `root`, но он доступен для чтения и записи всем пользователям. Ядро бесшумно удаляет все запросы на запись в устройство. Все запросы на чтение файла возвращают EOF (end-of-file, то есть окончание файла).

Нулевое устройство находится в `/dev/zero` и имеет старший номер 1, а младший — 5. Аналогично фиктивному устройству ядро бесшумно отклоняет запросы на запись к нулевому устройству. Чтение с устройства возвращает бесконечный поток пустых байтов.

Полное устройство со старшим номером 1 и младшим 7 находится в `dev/full`. Как и в случае с нулевым, запросы на чтение возвращают нулевые символы (`\0`). Запросы на запись всегда возвращают ошибку `ENOSPC`, означающую, что вызываемое устройство переполнено.

Эти устройства созданы с различными целями. Они полезны для проверки, как приложение обрабатывает сложные и проблемные случаи — например, заполнение

файловой системы. Нулевое и фиктивное устройства игнорируют запись, поэтому они также предоставляют способ обойти нежелательный ввод-вывод, не создающий дополнительной нагрузки.

Генератор случайных чисел

Генераторы случайных чисел ядра находятся в `/dev/random` и `/dev/urandom`. Их старшие номера равны 1, а младшие — 8 и 9 соответственно.

Генератор случайных чисел ядра собирает шум с драйверов устройств и других источников, а ядро собирает его и необратимо хеширует. Полученный результат сохраняется в *пуле энтропии*. Ядро всегда сохраняет приблизительную оценку количества битов энтропии в пуле.

Чтение `/dev/random` возвращает энтропию из этого пула. Результаты можно использовать в качестве начальных чисел в генераторах случайных чисел для генерирования ключей и других задач, которые требуют криптографически надежной энтропии.

Теоретически злоумышленник, который способен получить достаточно данных из энтропического пула и успешно взломать необратимый хеш, мог бы получить сведения о состоянии остальной части энтропического пула. Хотя такая атака в настоящее время является лишь теоретической возможностью — пока не известно ни об одном подобном случае, — ядро учитывает эту вероятность и уменьшает оценку объема энтропии при каждом запросе на запись. Если оценка достигает нуля, считывание блокируется, пока система не сгенерирует дополнительную энтропию и оценка объема энтропии не достигнет значения, достаточного для удовлетворения запроса на чтение.

Устройство `/dev/urandom` не обладает таким свойством; чтение с него завершается успешно, даже если оценка количества энтропии, сделанная ядром, недостаточна для завершения запроса. Даже самые защищенные приложения — например, генератор ключей для защиты обмена данными в GNU Privacy Guard — требуют высокой криптографической надежности энтропии, поэтому большинство приложений должны использовать `/dev/urandom`, а не `/dev/random`. Операция чтения из последнего может заблокироваться на весьма продолжительное время, если не будет активности ввода-вывода, способной заполнить энтропический пул ядра. Эта ситуация нередко случается на бездисковых серверах с удаленной настройкой.

Внеполосное взаимодействие

Файловая модель UNIX впечатляет. Выполняя лишь простейшие операции считывания и записи, UNIX абстрагирует практически любые мыслимые действия, которые может потребоваться совершить над объектом. Однако иногда программисту требуется обмениваться информацией с файлом за пределами основного потока данных. Например, рассмотрим устройство, подключенное к последовательному порту. При считывании с него одновременно будет происходить считывание с оборудования, расположенного на удаленном конце последовательного

порта. Как процесс считает один из специальных сигналов диагностического вывода последовательного порта, например сигнал DTR («терминал данных готов»)? С другой стороны, как процесс установит четность порта?

На все эти вопросы помогает ответить системный вызов `ioctl().ioctl()` — сокращение от I/O control («управление вводом-выводом»). Данный вызов обеспечивает *внеполосный обмен данными*:

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

```
int ioctl (int fd, int request, ...);
```

Этот системный вызов требует два параметра:

- `fd` — файловый дескриптор для файла;
- `request` — специальное значение кода запроса; определяется заранее и является согласованным между ядром и процессом; указывает, какую операцию нужно осуществить над файлом, на который указывает дескриптор `fd`.

Кроме того, данный вызов может получать один или несколько нетипизированных необязательных параметров (как правило, это беззнаковые целочисленные значения или указатели) для передачи ядру.

Следующая программа использует запрос `CDROMEJECT` для вывода дискового лотка из CD-ROM. Этот запрос указывается пользователем в командной строке программы в качестве первого аргумента. Соответственно, эта программа функционально подобна стандартной команде `eject`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include <stdio.h>
```

```
int main (int argc, char *argv[])
{
    fd, ret;

    if (argc < 2) {
        fprintf (stderr,
                "usage: %s <device to eject>\n",
                argv[0]);
        return 1;
    }

    /*
     * Открывает CD-ROM только для чтения. O_NONBLOCK
     * сообщает ядру, что мы хотим открыть устройство,
     * даже если в дисковом отсуствует диск.
     */
    fd = open (argv[1], O_RDONLY | O_NONBLOCK);
```

```
if (fd < 0) {
    perror ("open");
    return 1;
}

/* Отправляет команду eject устройству CD-ROM. */
ret = ioctl (fd, CDROMEJECT, 0);
if (ret) {
    perror ("ioctl");
    return 1;
}

ret = close (fd);
if (ret) {
    perror ("close");
    return 1;
}

return 0;
}
```

Запрос `CDROMEJECT` — это возможность драйвера устройств CD-ROM для Linux. Когда ядро получает вызов `ioctl()`, оно находит файловую систему (при работе с реальными файлами) или устройство (при работе с узлами устройств), ответственные за файловый дескриптор. Затем по запросу эта информация передается для обработки. В данном случае драйвер устройства CD-ROM получает запрос и физически выдвигает лоток дисковод.

Далее в этой главе мы рассмотрим вызов `ioctl()`, использующий необязательный параметр для возврата информации запрашивающему процессу.

Отслеживание файловых событий

В Linux предоставляется интерфейс `inotify` для отслеживания операций, осуществляемых с файлами. К таким операциям относятся, в частности, перемещение, считывание, запись и удаление файлов. Предположим, вы программируете графический файловый менеджер, похожий на GNOME. Если файл копируется в каталог, в то время как содержимое файлового менеджера отображается на экране, то представление каталога в диспетчере файлов становится неверным.

Возможное решение этой проблемы — постоянно считывать содержимое каталога, обнаруживать изменения и обновлять информацию на экране. Такой вариант связан с периодическим увеличением издержек и довольно неаккуратен. Хуже того, постоянно могут возникать условия гонки между удалением или записью файла в каталоге и моментом, в который диспетчер будет заново считывать каталог.

Пользуясь вызовом `inotify`, ядро может *проталкивать* событие в приложение именно в тот момент, когда оно произойдет. Как только файл будет удален, ядро

может уведомить об этом диспетчер файлов. Последний, в свою очередь, может немедленно удалить этот файл из графического представления каталога.

Файловые события важны и для работы многих других приложений. Таковы, например, утилита для резервного копирования или инструмент индексации данных. Функция `inotify` позволяет таким программам работать в реальном времени: в тот самый момент, когда файл создается, удаляется или в него записывается информация, эти инструменты могут обновлять архив резервных копий или индекс данных.

`inotify` заменяет `dnotify`, более ранний механизм мониторинга файлов, который имел неудобный интерфейс, работавший на основе сигналов. В приложениях всегда следует предпочитать `inotify` механизму `dnotify`. Возможность `inotify` впервые появилась в версии ядра 2.6.13, она отличается гибкостью и простотой использования, так как работает с теми самыми операциями, которые программа осуществляет с обычными файлами. В этой книге мы поговорим только об `inotify`.

Инициализация `inotify`

Прежде чем процесс сможет использовать `inotify`, этот механизм необходимо инициализировать. Системный вызов `inotify_init()` выполняет такую инициализацию и возвращает файловый дескриптор, соответствующий инициализированному экземпляру:

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

```
int inotify_init1 (int flags);
```

Параметр `flags` обычно равен нулю, но может быть и побитовым «ИЛИ» следующих флагов:

- `IN_CLOEXEC` — задает для нового файлового дескриптора закрытие после исполнения;
- `IN_NONBLOCK` — задает для нового файлового дескриптора значение `O_NONBLOCK`.

При ошибке вызов `inotify_init1()` возвращает `-1` и присваивает `errno` одно из следующих значений:

- `EMFILE` — достигнут пользовательский лимит на максимально допустимое количество экземпляров `inotify`;
- `ENFILE` — достигнут системный лимит на максимально допустимое количество файловых дескрипторов;
- `ENOMEM` — недостаточно памяти для выполнения запроса.

Инициализируем `inotify`, чтобы можно было им пользоваться:

```
int fd;
```

```
fd = inotify_init1 (0);
if (fd == -1) {
    perror ("inotify_init1");
    exit (EXIT_FAILURE);
}
```

Стражи

После того как процесс инициализирует `inotify`, он устанавливает *стражи*. Страж, которому соответствует *дескриптор стража*, — это стандартизированный путь UNIX, ассоциированный с *маской стража*. Она сообщает ядру, в каких событиях заинтересован процесс — например, считывании, записи или и в том и в другом.

`Inotify` позволяет отслеживать как файлы, так и каталоги. При отслеживании каталога `inotify` сообщает о событиях, которые происходят в самом каталоге и во всех файлах, находящихся в данном каталоге (но не в файлах из подкаталогов данного каталога — работа стража не является рекурсивной).

Добавление нового стража

Системный вызов `inotify_add_watch()` добавляет страж для события или событий, описанных в маске пути к файлу или каталогу. Страж добавляется для экземпляра `inotify`, которому соответствует файловый дескриптор `fd`:

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

```
int inotify_add_watch (int fd,  
                     const char *path,  
                     uint32_t mask);
```

В случае успеха вызов возвращает дескриптор нового стража. При ошибке `inotify_add_watch()` возвращает `-1` и присваивает `errno` одно из следующих значений:

- `EACCES` — доступ на чтение к файлу, на который указывает путь `path`, не разрешен; вызывающий процесс должен иметь право доступа для чтения файла, чтобы добавить к этому файлу страж;
- `EBADF` — файловый дескриптор `fd` не соответствует допустимому экземпляру `inotify`;
- `EFAULT` — указатель `path` является недопустимым;
- `EINVAL` — маска стража, `mask`, не содержит допустимых событий;
- `ENOMEM` — недостаточно памяти для выполнения запроса;
- `ENOSPC` — достигнут пользовательский лимит, соответствующий максимальному количеству стражей для `inotify`.

Маски стражей

Маска стража является побитовым «ИЛИ» из одного или нескольких событий `inotify`, определяемых в файле `<inotify.h>`:

- `IN_ACCESS` — в файле было выполнено считывание;
- `IN_MODIFY` — в файле была выполнена запись;
- `IN_ATTRIB` — метаданные файла (например, его владелец, права доступа или расширенные атрибуты) были изменены;
- `IN_CLOSE_WRITE` — файл был закрыт после того, как был открыт для записи;

- `IN_CLOSE_NOWRITE` — файл был закрыт, но до этого не был открыт для записи;
- `IN_OPEN` — файл был открыт;
- `IN_MOVED_FROM` — файл был перемещен из отслеживаемого каталога;
- `IN_MOVED_TO` — файл был перемещен в отслеживаемый каталог;
- `IN_CREATE` — файл был создан в отслеживаемом каталоге;
- `IN_DELETE` — файл был удален в отслеживаемом каталоге;
- `IN_DELETE_SELF` — был удален сам отслеживаемый объект;
- `IN_MOVE_SELF` — был перемещен сам отслеживаемый объект.

Определяются также следующие события, но в перечисленных ниже случаях в одно значение объединяются два и более события:

- `IN_ALL_EVENTS` — все допустимые события;
- `IN_CLOSE` — все события, связанные с закрытием (в настоящее время `IN_CLOSE_WRITE` и `IN_CLOSE_NOWRITE`);
- `IN_MOVE` — все события, связанные с перемещением (в настоящее время `IN_MOVED_FROM` и `IN_MOVED_TO`).

Теперь рассмотрим, как добавить новый страж к имеющемуся экземпляру `inotify`:

```
int wd;
```

```
wd = inotify_add_watch (fd, "/etc", IN_ACCESS | IN_MODIFY);
if (wd == -1) {
    perror ("inotify_add_watch");
    exit (EXIT_FAILURE);
}
```

В этом примере добавляется страж для всех операций считывания и записи в каталоге `/etc`. При считывании или записи любого файла в `/etc` `inotify` отправляет событие файловому дескриптору `inotify`, `fd`, указывая дескриптор стража `wd`. Рассмотрим, как `inotify` представляет эти события.

События `inotify`

Структура `inotify_event`, определяемая в файле `<inotify.h>`, представляет события `inotify`:

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

```
struct inotify_event {
    int wd;           /* дескриптор стража */
    uint32_t mask;    /* маска событий */
    uint32_t cookie;  /* уникальное значение cookie */
    uint32_t len;     /* размер поля 'name' */
    char name[];      /* имя, завершаемое нулем */
};
```

`wd` указывает дескриптор стража, полученный от `inotify_add_watch()`, а `mask` представляет события. Если `wd` указывает каталог, а в файле из этого каталога произошло одно из отслеживаемых событий, то `name` указывает имя относительно данного каталога. В таком случае `len` не равно нулю. Обратите внимание: `len` не равно длине строки `name`; `name` может содержать один ведущий нуль или более. Эти символы используются в качестве заливки и обеспечивают правильное выравнивание последующей структуры `inotify_event`. Следовательно, необходимо использовать `len`, а не `strlen()` при вычислении отступа для следующей структуры `inotify_event` в массиве.

Массивы с нулевой длиной

`name` — это пример массива с нулевой длиной. Массивы с нулевой длиной, также называемые гибкими массивами, — это возможность языка C99, обеспечивающая создание массивов переменной длины. В одном практическом отношении эта возможность обладает огромным потенциалом: с ее помощью можно встраивать массивы переменной длины в структуры. Можно считать эти массивы указателями, причем все содержимое находится на месте самого указателя.

Рассмотрим пример с `inotify`. Очевидный способ возврата имени файла в этой структуре — посредством поля имени, например `name[512]`. Однако не существует максимальной длины файла, которая соблюдалась бы во всех файловых системах. Любое значение будет ограничивать полезность `inotify`. Более того, файловые имена, как правило, очень невелики, поэтому крупный буфер хранил бы вместе с такими именами множество мусора. Подобная ситуация достаточно распространена. Классический способ ее решения — сделать `name` указателем, динамически выделить где-нибудь буфер и указать на него в `name`. Однако с системным вызовом такой подход работать не будет. Массив с нулевой длиной является идеальным решением.

Например, если дескриптор `wd` соответствует стражу `/home/kidd` и имеет маску `IN_ACCESS`, а из файла `/home/kidd/canon` происходит считывание, то `name` будет равно `canon`, а `len` — не менее 6. Напротив, если бы мы отслеживали `/home/kidd/canon` непосредственно, с этой же маской, то `len` было бы равно 0, а `name` имело бы нулевую длину — вы бы его просто не трогали.

`cookie` используется для связывания двух родственных, но несвязанных событий. Мы поговорим об этом элементе в следующем разделе.

Считывание событий `inotify`

Получать события `inotify` несложно: вы просто считываете данные из файлового дескриптора, связанного с экземпляром `inotify`. Этот интерфейс обладает возможностью, называемой *заглатыванием*. Заглатывание позволяет выполнять множество операций считывания за один запрос. Количество событий ограничено лишь размером буфера, предоставленного для `read()`. Поле `name` имеет переменную длину, поэтому данный способ считывания событий `inotify` — самый распространенный.

В предыдущем примере мы создали экземпляр `inotify` и добавили к нему страж. Теперь считаем ожидающие события:

```
char buf[BUF_LEN] __attribute__((aligned(4)));
ssize_t len, i = 0;

/* считываем события в объеме BUF_LEN байт */
len = read (fd, buf, BUF_LEN);

/* обрабатываем в цикле все считанные события, пока не останется ни одного */
while (i < len) {
    struct inotify_event *event =
        (struct inotify_event *) &buf[i];
    printf ("wd=%d mask=%d cookie=%d len=%d dir=%s\n",
        event->wd, event->mask,
        event->cookie, event->len,
        (event->mask & IN_ISDIR) ? "yes" : "no");

    /* при наличии имени выводим его на экран */
    if (event->len)
        printf ("name=%s\n", event->name);

    /* обновляем индекс, чтобы он указывал на начало следующего события*/
    i += sizeof (struct inotify_event) + event->len;
}
```

Файловый дескриптор `inotify` действует как обычный файл, поэтому программы могут наблюдать за ним с помощью вызовов `select()`, `poll()` и `epoll()`. Таким образом, процессы могут мультиплексировать события `inotify` с другим файловым вводом-выводом от конкретного потока.

Расширенные события `inotify`

Кроме стандартных событий, `inotify` может генерировать и другие события.

- `IN_IGNORED` — страж, соответствующий дескриптору `wd`, был удален. Это может произойти потому, что пользователь вручную удалил страж, или потому, что отслеживаемый объект перестал существовать. Мы обсудим это событие в следующем разделе.
- `IN_ISDIR` — обрабатываемый объект является каталогом (если это событие не установлено, обрабатываемый объект является файлом).
- `IN_Q_OVERFLOW` — переполнение очереди объектов `inotify`. Ядро ограничивает размер очереди `inotify` во избежание неограниченного потребления памяти ядра. Как только количество ожидающих обработки событий достигает величины на единицу меньшей, чем максимум, ядро генерирует это событие и ставит его в хвост очереди. Больше никаких событий не генерируется, пока не произойдет считывание из очереди и ее размер не уменьшится.
- `IN_UNMOUNT` — устройство, на котором находится отслеживаемый объект, было отключено от системы (размонтировано), поэтому объект стал недоступен. В таком случае ядро удаляет страж и генерирует событие `IN_IGNORED`.

Эти события может генерировать любой страж. Пользователю не приходится задавать их явно.

Программисты должны работать с `mask` как с битовой маской ожидающих событий. Следовательно, не проверяйте события с помощью прямых тестов равенства:

```
/* НЕ ДЕЛАЙТЕ так! */
```

```
if (event->mask == IN_MODIFY)
    printf ("В файл были записаны данные !\n");
else if (event->mask == IN_Q_OVERFLOW)
    printf ("Очередь переполнена!\n");
```

Вместо этого следует выполнять побитовые тесты:

```
if (event->mask & IN_ACCESS)
    printf ("Из файла были считаны данные!\n");
if (event->mask & IN_UNMOUNTED)
    printf ("Отсоединено устройство, которому принадлежал файл!\n");
if (event->mask & IN_ISDIR)
    printf ("Файл является каталогом!\n");
```

Связывание событий перемещения

Каждое из событий `IN_MOVED_FROM` и `IN_MOVED_TO` представляет только часть перемещения: первое соответствует удалению с указанного места, а второе — прибытию на новое место. Следовательно, чтобы эта информация была действительно полезна для системы, выполняющей интеллектуальное отслеживание перемещений файлов по системе (допустим, речь идет об индексаторе, который не должен заново индексировать перемещенные файлы), процессы должны иметь возможность связывать друг с другом два события перемещения.

Рассмотрим поле `cookie` в структуре `inotify_event`.

Если поле `cookie` является ненулевым, оно содержит уникальное значение, связывающее два события. Допустим, у нас есть процесс, отслеживающий `/bin` и `/sbin`. При этом `/bin` имеет дескриптор стража 7, а `/sbin` — дескриптор стража 8. Если файл `/bin/compass` перемещается в `/sbin/compass`, ядро сгенерирует два события `inotify`.

Первое событие будет иметь `wd`, равный 7, `mask`, равную `IN_MOVED_FROM`, и имя `compass`. Второе событие будет иметь `wd`, равный 8, `mask`, равную `IN_MOVED_TO`, и имя `compass`. У обоих событий `cookie` будет иметь одно и то же значение — скажем, 12.

Если файл переименовывается, то ядро все равно генерирует два события. Значение `wd` у этих событий одинаковое.

Обратите внимание: если файл перемещается в каталог или из каталога, который не отслеживается, то процесс не получит одно из двух событий (соответствующее). Сама программа должна заметить, что второе событие с соответствующим `cookie` так и не придет.

Расширенные события отслеживания

При создании нового стража можно добавлять к `mask` одно или несколько следующих значений, управляющих поведением стража.

- `IN_DONT_FOLLOW` — если установлено это значение, а цель `path` или любого из его компонентов является символьной ссылкой, то переход по этой ссылке не происходит и функция `inotify_add_watch()` не выполняется.
- `IN_MASK_ADD` — как правило, если вы вызываете `inotify_add_watch()` для файла, у которого уже есть действующий страж, маска стража обновляется и отражает новую предоставленную маску. Если у `mask` установлен этот флаг, сообщаемые события добавляются к имеющейся маске.
- `IN_ONESHOT` — если установлено это значение, ядро автоматически удаляет страж после генерирования первого события для заданного объекта. Фактически страж получается «одноразовым».
- `IN_ONLYDIR` — при таком значении страж добавляется к объекту лишь при условии, что этот объект является каталогом. Если `path` представляет файл, а не каталог, то `inotify_add_watch()` не срабатывает.

Например, следующий фрагмент кода добавляет страж к `/etc/init.d`, если `init.d` является каталогом, причем ни `/etc`, ни `/etc/init.d` не является символьной ссылкой:

```
int wd;

/*
 * Отслеживаем '/etc/init.d' на предмет перемещения, но только если это
 * каталог и ни одна часть его пути не является символической ссылкой
 */
wd = inotify_add_watch (fd,
                       "/etc/init.d",
                       IN_MOVE_SELF |
                       IN_ONLYDIR |
                       IN_DONT_FOLLOW);

if (wd == -1)
    perror ("inotify_add_watch");
```

Удаление стража inotify

Как показано в этом примере, можно удалить страж с экземпляра `inotify` с помощью системного вызова `inotify_rm_watch()`:

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

Успешный вызов `inotify_rm_watch()` удаляет страж, представленный дескриптором `wd`, от экземпляра `inotify`, представленного файловым дескриптором `fd`, и возвращает 0.

Например:

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
    perror ("inotify_rm_watch");
```

При ошибке этот системный вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- `EBADF` — `fd` не является допустимым экземпляром `inotify`;
- `EINVAL` — `wd` не является допустимым дескриптором стража, сопровождающего заданный экземпляр `inotify`.

При удалении стража ядро генерирует событие `IN_IGNORED`. Ядро отправляет такое событие не только при удалении вручную, но и когда уничтожение стража является побочным эффектом другой операции. Например, если отслеживаемый файл удаляется, все связанные с ним стражи также удаляются. Во всех подобных случаях ядро отправляет `IN_IGNORED`. Такое поведение позволяет приложениям объединить все операции по удалению стражей в одном месте, а именно в обработчике событий `IN_IGNORED`. Такая возможность полезна для комплексных приложений, управляющих сложными структурами данных, лежащими в основе каждого стража `inotify`. Такова, например, поисковая инфраструктура `Beagle`, применяемая на рабочем столе `GNOME`.

Получение размера очереди событий

Размер очереди ожидающих событий можно получить с помощью системного вызова `ioctl` на дескрипторе файла экземпляра `inotify`. Первый аргумент запроса получает размер очереди в байтах в виде беззнакового целого числа:

```
unsigned int queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len);
if (ret < 0)
    perror ("ioctl");
else
    printf ("Ожидает в очереди %u байтов\n", queue_len);
```

Обратите внимание: запрос возвращает размер очереди в байтах, а не количество событий, присутствующих в очереди. Программа может оценить количество событий по количеству байтов, используя известный размер структуры `inotify_event` (получаемый посредством вызова `sizeof()`) и сделав предположение насчет среднего размера поля `name`. Однако еще полезнее, что на основе

количества ожидающих байтов процесс может точно определять размер фрагмента для считывания.

Константа `FIONREAD` определяется в заголовке `<sys/ioctl.h>`.

Уничтожение экземпляра `inotify`

Для уничтожения экземпляра `inotify`, а также всех ассоциированных с ним стражей достаточно закрыть дескриптор экземпляра файла:

```
int ret;

/* 'fd' был получен с помощью inotify_init() */
ret = close (fd);
if (fd == -1)
    perror ("close");
```

Разумеется, как и с любым другим файловым дескриптором, по завершении процесса ядро автоматически закрывает этот дескриптор и очищает все ресурсы.

9 Управление памятью

Память — это один из простейших и в то же время фундаментальных ресурсов, доступных для использования процессами. В этой главе мы поговорим об управлении памятью — как ее выделять, манипулировать ею, а затем высвобождать.

Термин «*выделять*», который обычно обозначает получение памяти, немного неточен. Он ассоциируется с распределением скудного ресурса, спрос на который превышает предложение. Честно говоря, многие пользователи действительно желают получить как можно больше памяти. В современных системах основные сложности работы с памятью связаны не с тем, чтобы удовлетворить потребности всех нуждающихся, располагая небольшим объемом памяти, а с тем, чтобы правильно использовать выделенную дозу памяти и отслеживать процесс такого использования.

В этой главе мы рассмотрим все возможные способы выделения памяти в различных областях программы, разберем достоинства и недостатки всех методов. Мы также изучим некоторые способы установки содержимого произвольно выбранных областей памяти и оперирования этим содержимым. Мы научимся блокировать некоторый объем памяти, чтобы он не покидал пределов RAM (оперативной памяти) и ваша программа продолжала работать. При этом программе не придется дожидаться, пока ядро скопирует данные из области подкачки.

Адресное пространство процесса

Linux, как и все современные операционные системы, виртуализирует имеющуюся в распоряжении физическую память. Процессы не обращаются непосредственно к физической памяти. Вместо этого ядро связывает каждый процесс с собственным уникальным *виртуальным адресным пространством* данного процесса. Это адресное пространство является *линейным*. Его адреса начинаются с нуля, непрерывно увеличиваясь вплоть до заданного максимального значения. Кроме того, адресное пространство является *плоским*; оно существует в единой области, непосредственно доступной и не требующей сегментирования.

Страницы и их подкачка

Память состоит из битов, причем восемь бит (как правило) составляют один байт. Байты складываются в слова, а слова — в *страницы*. При управлении памятью

наиболее важной из этих концепций является именно страница. Страница — это наименьшая адресуемая сущность в памяти, которой может управлять блок управления памятью (MMU), поэтому можно сказать, что страницы «нарезаются» из адресного пространства процесса. *Размер страницы* зависит от применяемой машинной архитектуры. Наиболее распространенными размерами являются 4 Кбайт (в 32-битных системах) и 8 Кбайт (в 64-битных системах)¹.

В 32-битном адресном пространстве содержится около миллиона страниц размером по 4 Кбайт каждая. В 64-битном адресном пространстве на несколько порядков больше страниц по 8 Кбайт. Процесс не обязательно будет обращаться ко всем этим страницам; конкретная страница может ничему не соответствовать. Страницы могут быть *валидными* или *невалидными*. Валидная страница ассоциирована с имеющейся страницей данных, которая располагается либо в физической памяти (RAM), либо на вторичном носителе (например, в разделе подкачки или в файле на диске). Невалидная страница ни с чем не ассоциирована и представляет собой неиспользуемый, невыделенный фрагмент адресного пространства. При обращении к невалидной странице происходит нарушение сегментации.

Если валидная страница ассоциирована с данными, расположенными на дополнительном носителе, то процесс не может получить доступ к этой странице, пока информация не будет перенесена в физическую память. Когда процесс пытается обратиться к такой странице, блок управления памятью генерирует *ошибку страницы*. Затем в работу вмешивается ядро, прозрачно *подкачивая* данные со вторичного носителя в физическую память. Объем виртуальной памяти обычно значительно превышает объем физической памяти, поэтому ядро может выгрузить данные из памяти, освободив таким образом пространство для подкачки. *Выгрузка* — это процесс переноса данных из физической памяти на вторичный носитель. Чтобы минимизировать количество последующих операций подкачки, ядро пытается выгрузить из памяти данные, которые с наименьшей вероятностью будут использоваться в ближайшем будущем.

Совместное использование и копирование при записи. Множественные страницы виртуальной памяти могут быть ассоциированы с единственной физической страницей, даже если они относятся к разным процессам, каждый из которых имеет собственное адресное пространство. Таким образом, различные виртуальные адресные пространства могут *совместно использовать* (разделять) данные из физической памяти. Например, в любой момент вполне вероятно, что многие процессы системы совместно используют стандартную библиотеку C. При использовании разделяемой памяти каждый из этих процессов может отображать эту библиотеку на свое виртуальное адресное пространство, но в физической памяти при этом должна присутствовать лишь одна копия библиотеки. Более наглядный пример: два процесса могут одновременно отображать в память большую базу

¹ В некоторых системах поддерживаются страницы разных размеров, поэтому размер страницы не является частью двоичного интерфейса приложений (ABI). Приложения должны программно получать размер страницы во время выполнения. Мы уже обсуждали эту тему в гл. 4 и вернемся к ней далее.

данных. В то время как нужная база данных будет присутствовать в виртуальном адресном пространстве каждого из этих процессов, в оперативной памяти будет находиться всего одна копия этой базы данных.

Разделяемые данные могут быть доступны только для чтения, записи или предоставляться для одного и другого. Когда процесс записывает информацию на совместно используемую страницу, допускающую такую запись, может произойти одна из двух вещей. В простом случае ядро разрешает запись, после чего все процессы, использующие эту страницу, могут видеть результат записи. Как правило, если мы разрешаем множественным процессам считывать одну и ту же страницу или записывать на нее информацию, требуется обеспечить некоторую степень координации и синхронизации между этими процессами, но на уровне ядра запись «просто работает» и все процессы, разделяющие данные, немедленно видят все происходящие изменения.

Во втором случае блок управления памятью может перехватить операцию записи и выдать исключение. В ответ ядро прозрачно создает новую копию страницы для записывающего процесса и позволяет процессу продолжать запись информации уже на новой странице. Такой подход называется *копированием при записи (COW)*¹. Фактически процессы получают к разделяемым данным доступ для чтения, благодаря чему экономится место. Однако если процесс пытается записать информацию на разделяемую страницу, то на лету получает уникальную копию этой страницы. Таким образом, ядро всегда может работать с учетом наличия собственной копии страницы у каждого записывающего процесса. Копирование при записи происходит постранично, поэтому такая техника фактически позволяет разделять огромный файл между многими процессами и отдельные процессы будут получать уникальные физические копии только страниц, на которые они сами записывают информацию.

Области памяти

Ядро распределяет страницы по блокам, которые имеют набор тех или иных общих свойств — например, прав доступа. Эти блоки именуются *отображениями, сегментами* или *областями памяти*. Некоторые области памяти присутствуют в любом процессе.

- *Текстовый сегмент* содержит программный код процесса, строковые литералы, константные значения переменных и другие данные, предназначенные только для чтения. В Linux этот сегмент обозначается доступным только для чтения и отображается прямо из объектного файла (это может быть исполняемый файл программы или библиотека).
- *Стек* содержит стек исполнения процесса. Стек может динамически расширяться или сжиматься по мере увеличения или уменьшения глубины стека. В стеке исполнения содержатся локальные переменные и возвращаемые данные функций. В многопоточном процессе каждому потоку соответствует собственный стек.

¹ Как вы помните из гл. 5, вызов `fork()` использует копирование при записи для дублирования и совместного использования родительского адресного пространства с потомком.

- *Сегмент данных*, или *куча*, содержит динамическую память процесса. Этот сегмент доступен для записи, может расширяться или сжиматься. Вызов `malloc()` (будет рассмотрен в следующем разделе) может удовлетворять запросы памяти из этого сегмента.
- *Сегмент bss*¹ содержит неинициализированные глобальные переменные. В этих переменных находятся специальные значения (как правило, только нули) в соответствии со стандартом C.

Linux оптимизирует эти переменные двумя способами. Во-первых, поскольку *bss*-сегмент предназначен для неинициализированных данных, компоновщик (`ld`) не сохраняет специальные значения в объектном файле, поэтому размер двоичного файла уменьшается. Во-вторых, когда этот сегмент загружается в память, ядро просто отображает его по принципу копирования при записи на страницу нулей, фактически устанавливая переменные в значения, которые заданы для них по умолчанию.

ПРИМЕЧАНИЕ

В большинстве адресных пространств содержится несколько отображенных файлов, к которым относятся, например, сам исполняемый файл программы, библиотека C и другие разделяемые библиотеки, а также файлы с данными. Взгляните на файл `/proc/self/maps` или вывод программы `rtpar`, чтобы получить представление об отображаемых файлах в процессе.

В этой главе будут рассмотрены интерфейсы, предоставляемые Linux для получения и возвращения памяти, создания и уничтожения новых отображений, а также обеспечения всех промежуточных этапов работы.

Выделение динамической памяти

Память также предоставляется в форме автоматических и статических переменных, но основа любой системы управления памятью — это выделение, использование и, наконец, возврат *динамической памяти*. Динамическая память возвращается во время выполнения, а не во время компиляции, причем в размере, который может быть неизвестен вплоть до момента выделения. Разработчик прибегает к использованию динамической памяти, когда объем требуемой памяти или длительность ее использования может варьироваться, причем точные значения размера и длительности становятся известны только после запуска программы. Например, вы хотите сохранить в памяти содержимое файла или пользовательского ввода, полученного через клавиатуру. Размер файла точно не известен, а пользователь может ввести достаточно много информации, поэтому размер буфера будет варьироваться и вам потребуется его динамически увеличивать по мере считывания все большего количества данных.

В языке C нет переменной, основой для которой является динамическая память. Так, в C отсутствует механизм получения структуры `struct pirate_ship`, существующей

¹ Это название сложилось исторически; аббревиатура расшифровывается `block started by symbol` — «блок, начинающийся с символа».

в динамической памяти. Вместо этого С предоставляет возможность выделения динамической памяти, достаточной для содержания структуры `pirate_ship`. После этого программист будет взаимодействовать с памятью с помощью указателя, в данном случае `struct pirate_ship*`.

Классический интерфейс С для получения динамической памяти называется `malloc()`:

```
#include <stdlib.h>

void * malloc (size_t size);
```

При успешном вызове `malloc()` выделяет `size` байт памяти и возвращает указатель в начальную точку только что выделенной области. Содержимое памяти не определено; не рассчитывайте, что память будет заполнена нулями. При ошибке `malloc()` возвращает `NULL`, а `errno` устанавливается значение `ENOMEM`.

Использовать `malloc()` достаточно просто, как в этом примере, в котором мы выделяем фиксированное количество байтов:

```
char *p;

/* дайте мне 2 Кбайт! */

p = malloc (2048);
if (!p)
    perror ("malloc");
```

Или в этом, где выделяется структура:

```
struct treasure_map *map;

/*
 * выделяем достаточное количество памяти для содержания структуры
 * treasure_map
 * и указываем на нее с помощью 'map'
 */
map = malloc (sizeof (struct treasure_map));
if (!map)
    perror ("malloc");
```

Язык С автоматически повышает указатели на `void` при любых типах назначения. Следовательно, в этих примерах мы обходимся без приведения типа возвращаемого значения `malloc()` к типу `lvalue`, используемому при назначениях. Однако в языке С++ не выполняется автоматического повышения указателя на `void`, поэтому программисты, имеющие дело с С++, должны приводить тип возвращаемого значения `malloc()` следующим образом:

```
char *name;

/* выделяем 512 байт */
name = (char *) malloc (512);
if (!name)
    perror ("malloc");
```


Некоторые программисты предпочитают приводить к `void` тип любого результата, получаемого от функции, возвращающей указатель, — в том числе результат `malloc()`. Я не рекомендую так поступать, поскольку в данном случае мы скроем ошибку, если по какой-то причине значение функции изменится и уже не будет возвращать указатель на `void`. Более того, такое приведение скроем ошибку и в случае, если функция была неверно объявлена. Первый из этих ошибочных случаев маловероятен с функцией `malloc()`, а второй вполне может произойти.

ПРИМЕЧАНИЕ

Необъявленные функции по умолчанию возвращают `int`. Превращение целого числа в указатель не является автоматическим и генерирует предупреждение. Если в данном случае будет выполнено приведение типа, такое предупреждение не появится.

Функция `malloc()` может возвращать `NULL`, поэтому разработчик просто обязан *всегда* проверять это условие и обрабатывать ошибки при их наличии. Многие программы определяют и используют обертку `malloc()`, выводящую сообщение об ошибке и завершающие программу, если `malloc()` возвратит `NULL`. По традиции разработчики называют эту распространенную обертку `xmalloc()`:

```
/* похоже на malloc(), но при ошибке завершается */
void * xmalloc (size_t size)
{
    void *p;
    p = malloc (size);
    if (!p) {
        perror ("xmalloc");
        exit (EXIT_FAILURE);
    }

    return p;
}
```

Выделение массивов

Динамическое выделение памяти также может быть достаточно сложным, если указанный размер `size` сам по себе является динамическим. Примером такой работы является динамическое выделение массивов, где размер элемента массива может быть фиксированным, а количество выделяемых элементов переменное. Для упрощения данного сценария в библиотеке C предоставляется функция `calloc()`:

```
#include <stdlib.h>
```

```
void * calloc (size_t nr, size_t size);
```

При успешном вызове `calloc()` возвращается указатель на блок памяти, подходящий для содержания массива из `nr` элементов, каждый из которых имеет размер `size` байт. Следовательно, объем памяти, запрашиваемый при этих двух вызовах, идентичен (каждый вызов может вернуть больше памяти, чем первоначально запрашивал, но не меньше):

```
int *x, *y;

x = malloc (50 * sizeof (int));
if (!x) {
    perror ("malloc");
    return -1;
}

y = calloc (50, sizeof (int));
if (!y) {
    perror ("calloc");
    return -1;
}
```

Однако их поведение не является идентичным. В отличие от `malloc()`, не дающего никаких гарантий относительно содержимого выделенной памяти, `calloc()` заполняет нулями все байты в возвращаемом фрагменте памяти. Таким образом, каждый из 50 элементов, содержащихся в массиве целых чисел `y`, имеет значение 0, а содержимое элементов в `x` остается неопределенным. Если программа не собирается сразу же установить все 50 значений, то программист должен сам гарантировать, что элементы в массиве не будут заполнены мусором. Обратите внимание: двоичный ноль может быть не равен нулю с плавающей точкой!

ПРИМЕЧАНИЕ

Мне не удалось найти оригинальной документации, в которой описывалась бы этимология названия `calloc()`. Историки UNIX спорят о происхождении этого названия: возможно, «с» означает `count` («количество»), так как функция принимает количество элементов массива? Может быть, эта буква означает `clear` («очистить»), так как функция заполняет память нулями? Выбирайте сторону — дебаты идут ожесточенные.

В поисках истины я обратился к Брайану Кернигану (Brian Kernighan), одному из отцов-основателей Linux, и спросил, не помнит ли он происхождения этого названия. Брайан оговорился, что исходную функцию писал не он, но ему кажется, что «с» означает «очистка». Вероятно, это самое авторитетное мнение, на которое можно сослаться.

Часто пользователи стараются «обнулить» динамическую память, даже если не работают с массивами. Далее в этой главе мы рассмотрим функцию `memset()`, предоставляющую интерфейс для установки в заданное значение каждого байта в выбранном фрагменте памяти. Однако если мы приказываем `calloc()` выполнить заполнение нулями, то работа пойдет быстрее, так как ядро может предоставить память, уже заполненную нулями.

При ошибке `calloc()`, как и `malloc()`, возвращает `NULL` и устанавливает `errno` значение `ENOMEM`.

Почему стандартизирующие органы так и не определили отдельную от `calloc()` функцию «выделить и заполнить нулями»? Это очень странно. Правда, разработчик может с легкостью определить для этого собственный интерфейс:

```
/* действует так же, как и malloc(), но память заполняется нулями */
void *malloc0 (size_t size)
{
```

```
        return calloc (1, size);  
    }
```

Мы можем с удобством скомбинировать `malloc0()` с рассмотренной выше `xmalloc()`:

```
/* похожа на malloc(), но заполняет память нулями и завершается при ошибке */  
void * xmalloc0 (size_t size)  
{  
    void *p;  
  
    p = calloc (1, size);  
    if (!p) {  
        perror ("xmalloc0");  
        exit (EXIT_FAILURE);  
    }  
  
    return p;  
}
```

Изменение размера выделенных областей

Язык С предоставляет интерфейс для изменения размера (увеличения или уменьшения) уже имеющихся выделенных областей:

```
#include <stdlib.h>
```

```
void * realloc (void *ptr, size_t size);
```

При успешном вызове `realloc()` эта функция изменяет размер области памяти, на которую направлен указатель `ptr`, задавая для нее новое значение, равное `size` байт. Она возвращает указатель на заново отмеренную область памяти, причем этот указатель может быть как равен `ptr`, так и иметь другое значение. В случае увеличения области памяти `realloc()` может оказаться не в состоянии увеличить исходный фрагмент до запрашиваемого размера на том же месте, где этот фрагмент сейчас находится. В таком случае функция может выделить новую область памяти размером `size` байт, скопировать старую область в новую, а старую после этого высвободить. При любой подобной операции содержимое области памяти сохраняется либо полностью, либо в размере, равном объему новой выделенной области. Поскольку операции `realloc()` связаны с копированием, при увеличении области памяти они могут быть довольно затратными.

Если размер `size` равен нулю, то эффект равносителен вызову `free()` применительно к `ptr`.

Если `ptr` равен `NULL`, результат операции аналогичен «свежему» использованию `malloc()`. Если указатель `ptr` не равен нулю, то он обязательно должен быть возвращен в предыдущем вызове, направленном к `malloc()`, `calloc()` или `realloc()`.

При ошибке `realloc()` возвращает `NULL` и присваивает `errno` значение `ENOMEM`. Состояние области памяти, на которую указывает `ptr`, остается неизменным.

Рассмотрим пример с уменьшением области памяти. Сначала воспользуемся `calloc()` и выделим достаточно памяти, чтобы содержать в ней двухэлементный массив структур `map`:

```
struct map *p;

/* выделяем память для двух структур map */
p = calloc (2, sizeof (struct map));
if (!p) {
    perror ("calloc");
    return -1;
}

/* используем p[0] и p[1]... */
```

Теперь предположим, что одно сокровище мы уже нашли и вторая карта (`map`) нам больше не требуется, поэтому мы изменим занятую область памяти и отдадим ее половину, отведенную под карты, обратно в распоряжение системы. Такая операция не всегда целесообразна, но она полезна, если структура `map` очень велика, а оставшуюся карту мы планируем хранить еще достаточно долго:

```
struct map *r;

/* теперь нам нужна память для хранения только одной карты */
r = realloc (p, sizeof (struct map));
if (!r) {
    /* обратите внимание: значение 'p' по-прежнему остается допустимым! */
    perror ("realloc");
    return -1;
}

/* используем 'r'... */

free (r);
```

В данном примере `p[0]` сохраняется после вызова `realloc()`. Какие бы данные там ни находились, они остаются доступны. Если вызов вернется с ошибкой, `p` останется нетронутым, а значит, действующим. Мы можем продолжать им пользоваться, а по окончании работы — высвободить. Напротив, если вызов завершится успешно, мы игнорируем `p`, а вместо него используем `r`. Теперь мы отвечаем за высвобождение `r` после окончания работы.

Освобождение динамической памяти

В отличие от автоматически выделяемых областей, которые собираются системой без нашего участия, как только распадается стек, динамически выделенные области остаются неотъемлемыми частями адресного пространства процесса, пока не будут высвобождены вручную, поэтому программист отвечает за возврат динамически выделяемой памяти системе. Разумеется, все области памяти — выделенные

как статическим, так и динамическим образом, — автоматически высвобождаются, как только завершается весь процесс.

Память, выделенная с помощью `malloc()`, `calloc()` или `realloc()`, должна быть возвращена системе, если эта память больше не используется. Это делается с помощью `free()`:

```
#include <stdlib.h>
```

```
void free (void *ptr);
```

Вызов `free()` освобождает память, на которую указывает `ptr`. Параметр `ptr` должен предварительно быть возвращен `malloc()`, `calloc()` или `realloc()`. Это означает, что с помощью `free()` вы не можете высвобождать произвольные фрагменты памяти — например, половину области памяти, — передав указатель на середину выделенного блока. Такое действие приведет к появлению неопределенной памяти, что проявится в виде сбоя.

Указатель `ptr` может быть равен `NULL`, в случае чего `free()` бесшумно возвращает значение, поэтому распространенная практика проверки `ptr` на `NULL` перед вызовом `free()` не имеет смысла.

Рассмотрим пример:

```
void print_chars (int n, char c)
{
    int i;

    for (i = 0; i < n; i++) {
        char *s;
        int j;

        /*
        * Выделяем и заполняем нулями массив элементов i+2
        * состоящий из символов. Обратите внимание: 'sizeof (char)'
        * всегда равно 1.
        */
        s = calloc (i + 2, 1);
        if (!s) {
            perror ("calloc");
            break;
        }

        for (j = 0; j < i + 1; j++)
            s[j] = c;

        printf ("%s\n", s);

        /* Все сделано. Можно вернуть память. */
        free (s);
    }
}
```

В этом примере мы выделяем n массивов из символов `char`, содержащих последовательно возрастающие количества элементов, начиная от двух (2 байт) до $n + 1$ элементов ($n + 1$ байт). Затем, обрабатывая каждый массив, цикл записывает символ `c` в каждый байт, кроме последнего (оставляя 0, уже находящийся в последнем байте). Массив выводится на экран как строка, после чего динамически выделенная память высвобождается.

При вызове `print_chars()` с n , равным 5, и `c`, установленным в `X`, получаем следующее:

```
X
XX
XXX
XXXX
XXXXX
```

Разумеется, есть и более эффективные способы реализации этой функции. Однако суть в том, что мы можем динамически выделять и высвобождать память, даже если размер и количество выделяемых областей становятся известны только во время выполнения.

ПРИМЕЧАНИЕ

Системы UNIX — в частности, SunOS и SCO — содержат вариант `free()`, называемый `cfree()`. В разных системах эта функция может либо работать аналогично `free()`, либо получать три параметра, действуя идентично `calloc()`. В Linux `free()` может обрабатывать память, полученную от любого из рассмотренных выше механизмов выделения. `Cfree()` не следует использовать практически ни в каких случаях, кроме как для обеспечения обратной совместимости. В Linux `cfree()` равнозначна `free()`.

Рассмотрим, каковы будут последствия, если в этом примере не вызвать `free()`. Программа так и не вернет память в систему. Хуже того, она потеряет свою единственную ссылку на эту память — указатель `s`, в результате чего и сама больше не сможет обратиться к памяти. Такой тип программной ошибки называется *утечкой памяти*. Утечки и другие подобные ошибки, связанные с динамической памятью, являются одними из самых распространенных и, к сожалению, относятся к наиболее серьезным неполадкам при программировании на C. В языке C вся ответственность за управление памятью лежит на плечах программиста, поэтому он должен очень внимательно следить за операциями выделения памяти.

Еще один распространенный подводный камень при программировании на C — это так называемое *использование освобожденной памяти*. Такая уязвимость возникает, если программа обращается к блоку памяти, который уже был освобожден. Как только для блока памяти вызвана функция `free()`, программа ни в коем случае не должна больше обращаться к содержимому этого блока. Программист должен проявлять особое внимание, отслеживая *повисшие указатели*: ненулевые, которые тем не менее указывают на недействительный блок памяти. Отличный инструмент, помогающий находить в программе ошибки, связанные с неверным использованием памяти, называется Valgrind.

Выравнивание

Выравнивание данных — это способ расположения данных в памяти. Об адресе A в памяти говорят, что он выровнен по n байт, если n является степенью 2, а A кратно n . Процессоры, подсистемы памяти и другие системные компоненты предъявляют специфические требования к выравниванию данных. Например, большинство процессоров оперируют машинными словами и могут обратиться к памяти, лишь если она выровнена по размеру слова. Аналогично, как уже упоминалось выше, единицы управления памятью работают только с адресами, выровненными по размеру страниц.

Если переменная расположена в памяти по адресу, который кратен ее размеру, то она называется *естественно выровненной*. Например, 32-битная переменная является естественно выровненной, если она расположена в памяти по адресу, кратному 4, — иными словами, если два самых нижних бита этого адреса равны 0. Соответственно, тип размером $2n$ байт должен иметь адрес, в котором n наименьших значимых битов имеют нулевые значения.

Правила выравнивания обусловлены аппаратным обеспечением, поэтому отличаются от системы к системе. Некоторые машинные архитектуры предъявляют очень строгие требования к выравниванию данных. В других системах требования более гибкие. Некоторые системы генерируют улавливаемые ошибки. После этого ядро может выбрать нужное действие: либо завершить процесс-нарушитель, либо (что более вероятно) позволить вручную выполнить невыровненный доступ. Как правило, такой доступ состоит из множества более мелких выровненных операций. В результате снижается производительность, кроме того, приходится жертвовать атомарностью, но хотя бы удастся не завершать процесс. При написании переносимого кода программист должен очень внимательно следить, чтобы не нарушались требования, связанные с выравниванием.

Выделение выровненной памяти

В большинстве случаев компилятор и библиотека `C` прозрачно обрабатывают проблемы, связанные с выравниванием. Стандарт POSIX предписывает, что память, возвращаемая `malloc()`, `calloc()` и `realloc()`, должна быть правильно выровнена и пригодна для использования с любыми стандартными типами `C`. В Linux такие функции всегда возвращают память, выровненную по 8-байтной границе в 32-битных системах и по 16-байтной границе в 64-битных.

Иногда программисту нужно, чтобы динамическая память была выровнена по более широкой границе, например по странице. Мотивы для этого могут быть разными, но наиболее распространенная причина — необходимость верно выравнивать блоки, используемые при непосредственном блочном вводе-выводе или другом взаимодействии между аппаратными и программными компонентами. Для этой цели в POSIX 1003.1d предоставляется функция `posix_memalign()`:

```
/* одна или другая — достаточно любой */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE
```

```
#include <stdlib.h>
```

```
int posix_memalign (void **memptr,
                   size_t alignment,
                   size_t size);
```

Успешный вызов `posix_memalign()` выделяет `size` байт динамической памяти, обеспечивая выравнивание этой памяти по адресу, кратному `alignment`. Параметр `alignment` должен быть степенью 2, а также быть кратным размеру указателя `void`. Адрес выделенной памяти помещается в `memptr`, и вызов возвращает 0.

При ошибке память не выделяется, `memptr` остается неопределенным, а вызов возвращает один из следующих кодов ошибок:

- `EINVAL` — параметр `alignment` не является степенью 2 или не кратен размеру указателя `void`;
- `ENOMEM` — недостаточно памяти, чтобы выделить запрошенный объем.

Обратите внимание: `errno` в данном случае не устанавливается — функция возвращает эти ошибки напрямую.

Память, полученная с помощью `posix_memalign()`, высвобождается посредством `free()`. Использовать эту функцию очень просто:

```
char *buf;
int ret;

/* выделяем 1 Кбайт, выровненный по 256-байтной границе */
ret = posix_memalign (&buf, 256, 1024);
if (ret) {
    fprintf (stderr, "posix_memalign: %s\n",
            strerror (ret));
    return -1;
}

/* используем 'buf'... */

free (buf);
```

До того как в POSIX была определена функция `posix_memalign()`, операционные системы BSD и SunOS предоставляли соответственно следующие интерфейсы:

```
#include <malloc.h>
```

```
void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

Функция `valloc()` аналогична `malloc()`, за одним исключением: вся выделяемая память выравнивается по размеру страниц. Как вы помните из гл. 4, применяемый в системе размер страницы легко получить с помощью `getpagesize()`.

Функция `memalign()` напоминает `posix_memalign`. Она выравнивает память по границе, равной `boundary` байт, причем это значение должно быть степенью 2.

В данном примере обе операции выделения возвращают блок памяти, достаточный для содержания структуры `ship`, выровненной по размерам страниц:

```
struct ship *pirate, *hms;

pirate = valloc (sizeof (struct ship));
if (!pirate) {
    perror ("valloc");
    return -1;
}

hms = memalign (getpagesize (), sizeof (struct ship));
if (!hms) {
    perror ("memalign");
    free (pirate);
    return -1;
}

/* используем 'pirate' и 'hms'... */

free (hms);
free (pirate);
```

В Linux память, полученная с помощью любой из этих функций, высвобождается посредством `free()`. В других UNIX-системах ситуация может быть иной. Некоторые из них предоставляют механизм для безопасного высвобождения памяти, выделенной с применением этих функций. Если в вашей программе важно обеспечить переносимость, то у вас может просто не остаться никакой возможности высвободить память, выделенную через такие интерфейсы!

Специалисты, работающие с Linux, должны использовать эти функции только для обеспечения переносимости на старые системы; `posix_memalign()` является предпочтительным и стандартизированным вариантом. Все три этих интерфейса необходимы, только если требуется выравнивание с более высокими границами, чем предоставляются в `malloc()`.

Другие проблемы, связанные с выравниванием

Подобные проблемы не ограничиваются естественным выравниванием стандартных типов и выделением динамической памяти. Например, с нестандартными и сложными типами связаны более нетривиальные требования, чем со стандартными. Кроме того, проблемы выравнивания вдвойне важны, если мы присваиваем значения между указателями различных типов и используем при этом приведение типов.

Нестандартные и сложные типы данных предъявляют к выравниванию памяти такие требования, которые не сводятся к естественному выравниванию. Вот четыре полезных правила работы с ними.

- При работе со структурой ее следует выравнивать по наибольшему из типов, входящих в ее состав. Если самым крупным типом в структуре является 32-битное

целое число, выровненное по 4-байтной границе, то структура также должна быть выровнена как минимум по 4-байтной границе.

- При работе со структурами возникает необходимость заполнения, чтобы гарантировать правильное выравнивание каждого отдельного типа в соответствии с собственными требованиями данного типа. Таким образом, если оказывается, что за `char` (с вероятным выравниванием в 1 байт) следует `int` (с вероятным выравниванием в 4 байт), то компилятор вставит между двумя типами три байта-заполнителя, чтобы обеспечить выравнивание `int` по 4-байтной границе. Иногда программисты упорядочивают члены структуры — например, по уменьшению размера, — чтобы минимизировать пространство, которое впустую расходуется на заполнители. У `gcc` есть параметр `-Wpadded`, который может в этом помочь. Дело в том, что при его использовании генерируется предупреждение всякий раз, когда компилятор вставляет неявное заполнение.
- Требование выравнивания объединений: выравнивание происходит по самому крупному типу, входящему в объединение.
- Требование выравнивания массива: массив выравнивается по базовому типу. Соответственно, массивы не предъявляют к выравниванию никаких требований, кроме предъявляемых их типами. В результате происходит естественное выравнивание всех членов массива.

Компилятор прозрачно обрабатывает большинство требований, предъявляемых к выравниванию; для выявления потенциальных проблем приходится приложить некоторые усилия. Нередко доводится сталкиваться с проблемами выравнивания при работе с указателями и приведении.

При обращении к данным через указатель, преобразованный из блока с меньшим выравниванием в блок с большим, возможна ситуация, в которой процессор будет загружать информацию, чье выравнивание не соответствует большей границе. Например, в следующем фрагменте кода при присваивании с переменной `badnews` делается попытка прочитать `c` как `unsigned long`:

```
char greeting[] = "Ahoy Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

Значение `unsigned long` естественно выравнивается по 4- или 8-байтной границе; вполне вероятно, что с не будет выравниваться по такой же границе. Следовательно, при загрузке `c` в ходе приведения типов происходит нарушение выравнивания. В зависимости от машинной архитектуры последствия этого могут быть различными: от небольшого снижения производительности до аварийного завершения программы. В машинных архитектурах, позволяющих обнаруживать нарушения выравнивания, но «не умеющих» их правильно обрабатывать, ядро посылает процессу-нарушителю сигнал `SIGBUS`, завершающий этот процесс. Подробнее мы поговорим о сигналах в гл. 10.

Подобные примеры более распространены, чем можно было бы подумать. Случаи, встречающиеся на практике, выглядят не так топорно, но в то же время менее очевидны.

Строгое перекрытие

В примере с приведением типов также нарушается правило строгого перекрытия — как раз тот аспект С и С++, который не всеми понимается правильно. Строгое перекрытие — это требование, предъявляемое к объекту. При строгом перекрытии обращение к объекту допускается только по точному типу данного объекта, квалифицированной (например, `const` или `volatile`) версии точного типа, знаковой (или беззнаковой) версии точного типа, через структуру или объединение, в числе членов которой/которого содержится точный тип, либо через указатель `char`. Например, распространенный способ доступа к `uint32_t` через два указателя `uint16_t` нарушает правило строгого перекрытия.

Вот краткое изложение сути этого правила: разыменование приведения указателя от одного типа переменной к другому типу обычно нарушает правило строгого перекрытия. Если вы когда-либо сталкивались с предупреждением gcc «dereferencing type-punned pointer will break strict-aliasing rules» («доступ по указателю с приведением типа нарушает правила перекрытия объектов в памяти»), то строгое перекрытие у вас не выполняется. Строгое перекрытие уже давно входит в состав С++, но в С это правило было стандартизировано только в С99. Как понятно из сообщения об ошибке, gcc стимулирует вас придерживаться строгого перекрытия; следуя этому правилу, вы сможете генерировать оптимальный код.

Для самых любознательных: все тонкости этого правила изложены в разделе 6.5 стандарта ISO C99.

Управление сегментом данных

Исторически в системах UNIX предоставлялись интерфейсы для непосредственного управления сегментом данных. Тем не менее в большинстве программ эта возможность практически не находила применения, так как `malloc()` и другие способы выделения памяти значительно проще в использовании, а также мощнее. Я опишу подобные интерфейсы для полноты картины, а также для немногочисленных читателей, которые собираются реализовать собственный механизм выделения памяти на основе работы с кучей:

```
#include <unistd.h>
```

```
int brk (void *end);  
void * sbrk (intptr_t increment);
```

Названия этих функций происходят из старых UNIX-систем, в которых куча и стек еще находились в одном сегменте памяти. Выделение динамической памяти в куче выполнялось вверх, начиная с нижней части сегмента; стек рос вниз из верхней части сегмента, по направлению к куче. Граница, разделявшая области стека и кучи, называлась *остановом* или *точкой останова*. В современных системах, в которых сегмент данных находится в собственном отображении в памяти, мы по-прежнему именуем конечный адрес отображения точкой останова.

Вызов `brk()` устанавливает точку останова (конец сегмента данных), задавая адрес, указанный в `end`. В случае успеха функция возвращает 0. При ошибке она возвращает -1 и устанавливает `errno` значение `ENOMEM`.

Вызов `sbrk()` приращивает конец сегмента данных на `increment` байт, причем дельта может быть как положительной, так и отрицательной. `Sbrk()` возвращает пересмотренную точку останова. Соответственно, приращение, равное 0, дает нам актуальную точку останова:

```
printf("Текущая точка останова – %p\n", sbrk(0));
```

Ни в стандарте C, ни в POSIX намеренно не определяется ни одна из этих функций. Однако практически во всех UNIX-системах поддерживается как минимум одна из них, а то и две. Переносимые программы должны работать только со стандартизированными интерфейсами.

Анонимные отображения в памяти

При выделении памяти в `glibc` используется комбинация сегмента данных и отображений в памяти. Классический метод реализации `malloc()` — разделение сегмента данных на последовательность фрагментов, размер которых является степенью 2; после этого запросы на выделение памяти удовлетворяются путем возврата фрагмента, который максимально соответствует запрошенному размеру. Высвободить память не составляет труда: достаточно просто пометить фрагмент как свободный. Если смежные фрагменты свободны, их можно объединить в один более крупный фрагмент. Если верхняя часть кучи совершенно свободна, то система может понизить точку останова с помощью `brk()`, уменьшив таким образом кучу и вернув память ядру.

Этот алгоритм называется *схемой дружеского выделения памяти*. К его достоинствам относятся высокая скорость работы и простота, а основным недостатком является возникновение двух типов фрагментации. *Внутренняя фрагментация* происходит, когда для удовлетворения запроса выделяется больше памяти, чем было запрошено. В результате доступная память расходуется неэффективно. *Внешняя фрагментация* возникает, когда в наличии есть достаточно свободной памяти для удовлетворения запроса, но эта свободная память разбита на два и более несмежных фрагмента. В таком случае память также расходуется неэффективно (так как может использоваться сравнительно крупный блок, не слишком подходящий в данном случае). Кроме того, возрастает количество ошибок при выделении памяти (если в наличии не оказывается альтернативного блока).

Более того, при такой схеме одна операция выделения памяти может «зацеплять» другую, что не позволяет библиотеке C возвращать высвободившуюся память в ядро. Предположим, было выделено два блока памяти: А и В. Блок А вплотную прилегает к точке останова, а блок В находится непосредственно под А. Даже если программа освободит блок В, библиотека C не сможет откорректировать точку останова, пока таким же образом не будет освобожден фрагмент А. Следовательно,

долгоживущая выделенная область может удерживать в памяти и другие, уже пустые области.

Данная ситуация не всегда является проблемой, поскольку библиотеки C не обязаны возвращать память в систему. Как правило, куча не уменьшается после каждого высвобождения. Вместо этого реализация `malloc()` придерживает память до следующего выделения. Только если размер кучи значительно превышает объем выделенной памяти, `malloc()` действительно сужает сегмент данных, но выделение большого фрагмента может препятствовать такому сужению.

Следовательно, при крупных выделениях памяти `glibc` не использует кучу. Вместо этого она создает *анонимное отображение в памяти*, с помощью которого удовлетворяет запрос на выделение. Анонимные отображения в памяти подобны файловым отображениям, рассмотренным в гл. 4, не считая того, что в их основе не лежит какой-либо файл — поэтому они и называются анонимными. На самом деле анонимное отображение в памяти — это просто большой блок памяти, заполненный нулями и готовый к использованию. Можете считать его новоиспеченной кучей, которая будет использоваться в рамках одного выделения в памяти. Такие отображения располагаются вне кучи, поэтому не приводят к дополнительной фрагментации сегмента данных.

При выделении памяти методом анонимных отображений мы имеем следующие преимущества.

- Отсутствуют проблемы, связанные с фрагментацией. Когда программа больше не нуждается в анонимном отображении в памяти, оно удаляется и память немедленно возвращается в систему.
- Можно изменять размер анонимных отображений в памяти, корректировать права доступа к ним, они могут получать извещения, как и обычные отображения (см. гл. 4).
- Каждое выделение существует в отдельном отображении в памяти. Нет необходимости управлять глобальной кучей.

При использовании анонимных отображений памяти вместо работы с кучей мы также сталкиваемся с двумя недостатками.

- Размер каждого отображения в памяти делится без остатка на размер страницы, применяемой в системе, поэтому если выделенное пространство не является целочисленным кратным размера страницы, то будет образовываться неиспользуемое «потерянное» пространство. Оно представляет тем большую проблему, чем меньше размер отдельных выделений, где такие «хвосты» получают относительно крупными по сравнению с полезными областями.
- Создание нового отображения в памяти сопряжено с большим количеством издержек, чем удовлетворение запросов на выделение из кучи, поскольку куча может работать вообще без взаимодействия с ядром. Чем меньше размер выделяемых областей, тем серьезнее издержки.

Учитывая все «за» и «против», функция `malloc()` из `glibc` использует сегмент данных для удовлетворения небольших выделений, а в случае с большими прибегает

к анонимным отображениям в памяти. Этот порог является нежестким (см. разд. «Расширенное выделение памяти» далее) и может изменяться от одного релиза glibc к другому. В настоящее время он равен 128 Кбайт: выделения памяти, не превышающие этого объема, делаются из кучи, а при более крупных применяются анонимные отображения в памяти.

Создание анонимных отображений в памяти

Может быть, при каком-то выделении вы хотите принудительно использовать отображение в памяти, а не кучу; возможно, вы пишете собственную систему выделения памяти. В этих случаях потребуется вручную создавать отображения в памяти. Так или иначе, в Linux это не составляет труда. Как вы помните из гл. 4, системный вызов `mmap()` создает отображение в памяти, а системный вызов `munmap()` его уничтожает.

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

```
void * mmap (void *start,  
             size_t length,  
             int prot,  
             int flags,  
             int fd,  
             off_t offset);
```

```
int munmap (void *start, size_t length);
```

Кстати, создать анонимное отображение в памяти проще, чем отображение, в основе которого лежит файл, поскольку в первом случае не приходится ни открывать файл, ни управлять им. Основное отличие заключается в присутствии специального флага, указывающего, что отображение является анонимным.

Рассмотрим пример:

```
void *p;
```

```
p = mmap (NULL,                      /* неважно где */  
          512 * 1024,                /* 512 Кбайт */  
          PROT_READ | PROT_WRITE,    /* чтение/запись */  
          MAP_ANONYMOUS | MAP_PRIVATE, /* анонимное, приватное */  
          -1,                        /* fd (игнорируется) */  
          0);                        /* смещение (игнорируется) */
```

```
if (p == MAP_FAILED)  
    perror ("mmap");
```

```
else
```

```
    /* 'p' указывает на 512 Кбайт анонимной памяти... */
```

При большинстве анонимных отображений параметры `mmap()` соответствуют этому примеру, за исключением, разумеется, передачи произвольного размера (в байтах), определяемого программистом. Другие параметры обычно таковы.

- Первый параметр, `start`, устанавливается в значение `NULL`, указывая, что анонимное отображение может начинаться в любой точке памяти, выбранной ядром. Здесь можно задать и ненулевое значение, если оно выровнено по границам страниц, но это ограничивает переносимость. Обычно для программы не имеет значения, где именно в памяти находится отображение.
- Параметр `prot` обычно одновременно задает биты `PROT_READ` и `PROT_WRITE`, допуская как считывание отображения, так и запись в него. С другой стороны, выполнение кода из анонимного отображения обычно нежелательно, так как это возможный вектор для атаки злоумышленников.
- Параметр `flags` задает флаг `MAP_ANONYMOUS`, делая данное отображение анонимным, а также флаг `MAP_PRIVATE`, делая отображение приватным.
- Параметры `fd` и `offset` игнорируются, если задан флаг `MAP_ANONYMOUS`. Однако в некоторых старых системах предполагается, что в данном случае `fd` должен иметь значение `-1`, поэтому такое значение целесообразно задавать для обеспечения переносимости.

Память, полученная с помощью анонимного отображения, выглядит как память, взятая из кучи. Одно из достоинств выделения памяти именно из анонимных отображений заключается в том, что мы получаем готовые страницы, уже заполненные нулями, причем без каких-либо издержек, так как ядро отображает анонимные страницы приложения на страницы в памяти посредством копирования при записи. Следовательно, не приходится применять вызов `memset()` к возвращенной памяти. В этом и заключается одно из достоинств использования `calloc()` по сравнению с `malloc()`, за которым следует `memset()`: `glibc` известно, что анонимные отображения уже заполнены нулями и что `calloc()`, удовлетворяемый из отображения, не требует явного заполнения такого типа.

Системный вызов `munmap()` высвобождает анонимное отображение, возвращая выделенную память ядру:

```
int ret;
```

```
/* с 'p' все готово, отдаем отображение в 512 Кбайт обратно */  
ret = munmap(p, 512 * 1024);  
if (ret)  
    perror("munmap");
```

ПРИМЕЧАНИЕ

Подробнее об использовании `mmap()`, `mmapr()` и отображений в целом было рассказано в гл. 4.

Отображение /dev/zero

В других UNIX-системах, в частности BSD, флаг `MAP_ANONYMOUS` отсутствует. Вместо этого в них применяется схожее решение, связанное с отображением особого файла устройства `/dev/zero`. Этот файл устройства обеспечивает такую же семантику, как и анонимная память. Отображение содержит скопированные при записи страницы, которые заранее заполнены нулями. Соответственно, с практической точки зрения оно не отличается от анонимной памяти.

В Linux всегда предоставлялось устройство `/dev/zero`, а также возможность его отображения и получения памяти, заполненной нулями. На самом деле до появления флага `MAP_ANONYMOUS` программисты Linux пользовались именно таким подходом. Для обеспечения обратной совместимости со сравнительно старыми версиями Linux либо переносимости на другие UNIX-системы разработчик по-прежнему может использовать `/dev/zero` вместо создания анонимного отображения. Синтаксис не отличается от операции отображения любого другого файла:

```
void *p;
int fd;

/* открываем /dev/zero для чтения и записи */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0) {
    perror ("open");
    return -1;
}

/* отображаем [0, размер страницы) из /dev/zero */
p = mmap (NULL,                /* неважно где */
          getpagesize (),       /* отображаем одну страницу */
          PROT_READ | PROT_WRITE, /* отображаем для чтения/записи */
          MAP_PRIVATE,          /* приватное отображение */
          fd,                   /* отображаем /dev/zero */
          0);                   /* без смещения */

if (p == MAP_FAILED) {
    perror ("mmap");
    if (close (fd))
        perror ("close");
    return -1;
}

/* закрываем /dev/zero, он нам больше не нужен */
if (close (fd))
    perror ("close");

/* 'p' указывает на одну страницу в памяти, используем ее... */
```

Разумеется, память, отображаемая таким образом, высвобождается с помощью `munmap()`.

Этот подход связан с дополнительными издержками в виде лишнего системного вызова, применяемого для открытия и закрытия файла устройства. Следовательно, работа с анонимной памятью — более быстрое решение.

Расширенное выделение памяти

Многие операции выделения памяти, рассматриваемые в этой главе, ограничены и управляются `glibc` или параметрами ядра, которые программист может менять. Для этого применяется вызов `mallopt()`:

```
#include <malloc.h>
```

```
int mallopt (int param, int value);
```

Вызов `mallopt()` устанавливает параметр, связанный с управлением памятью (называемый `param`) в значение `value`. В случае успеха вызов возвращает ненулевое значение, при ошибке — 0. Обратите внимание: `mallopt()` не устанавливает `errno`. Кроме того, этот вызов практически всегда оканчивается успешно, поэтому не рассчитывайте, что сможете что-то прояснить на основе возвращенного значения.

В настоящее время Linux поддерживает семь значений `param`, все они определяются в `<malloc.h>`.

- `M_CHECK_ACTION` — значение переменной окружения `MALLOC_CHECK_` (будет рассмотрена в следующем разделе).
- `M_MMAP_MAX` — максимальное количество отображений, которые система может создать для удовлетворения запросов динамической памяти. Когда лимит достигнут, сегмент данных будет использоваться для всех операций выделения памяти, пока не будет высвобождено какое-либо из ранее созданных отображений. Значение 0 полностью отменяет использование любых анонимных отображений для выделения динамической памяти.
- `M_MMAP_THRESHOLD` — порог (измеряемый в байтах), вплоть до которого запросы на выделение памяти будут удовлетворяться с помощью анонимного отображения, без участия сегмента данных. Обратите внимание: выделения меньше этого порога также могут удовлетворяться посредством анонимных отображений по усмотрению системы. Значение 0 активирует использование анонимных отображений во всех случаях, фактически отменяя применение сегмента данных при выделениях динамической памяти.
- `M_MXFAST` — максимальный размер быстрой корзины (в байтах). *Быстрые корзины* (fast bins) — это особые фрагменты памяти в куче, которые никогда не объединяются со смежными областями и не возвращаются в систему. Таким образом, работая с быстрыми корзинами, мы добиваемся очень оперативного выделения памяти за счет увеличения фрагментации. Значение 0 деактивирует любое использование быстрых корзин.
- `M_PERTURB` — активирует *искажение памяти*, помогающее выявлять ошибки в управлении памятью. Получая ненулевое значение `value`, `glibc` устанавливает

все выделенные байты (кроме тех, которые были запрошены через `calloc()`) в значение, логически дополняющее наименьший значимый байт в `value`. Это помогает обнаруживать ошибки, связанные с использованием до инициализации. Более того, `glibc` устанавливает все высвобожденные байты в наименьший значимый байт из `value`. Так удастся идентифицировать ошибки, связанные с использованием после высвобождения.

- `M_TOP_PAD` — объем заполнения (в байтах), используемого при корректировке размера сегмента данных. Всякий раз, когда `glibc` задействует `brk()` для увеличения сегмента данных, вы можете запросить больше памяти, чем требуется, рассчитывая обойтись без дополнительного вызова `brk()`, который понадобился бы для получения новой порции памяти в ближайшем будущем. Аналогично всякий раз, когда `glibc` уменьшает размер сегмента данных, она может удерживать дополнительную память, отдав немного меньше, чем требовалось. Эти дополнительные байты и называются *заполнением*. Значение 0 отключает любое использование заполнения.
- `M_TRIM_THRESHOLD` — минимальное количество свободной памяти (в байтах), которое находится поверх сегмента данных перед тем, как `glibc` выполнит вызов `sbrk()` для возврата памяти в ядро.

Стандарт XPG, в свободной форме определяющий `mallopt()`, указывает три других параметра: `M_GRAIN`, `M_KEEP` и `M_NLBLKS`. Linux определяет эти параметры, но, задавая для них значения, мы ничего не добьемся. В табл. 9.1 приведен полный список всех допустимых параметров, значений, задаваемых для них по умолчанию, а также диапазон их допустимых значений.

Таблица 9.1. Параметры `mallopt()`

Параметр	Происхождение	По умолчанию	Допустимые значения	Специальные значения
<code>M_CHECK_ACTION</code>	Linux-специфичный	0	0–2	
<code>M_GRAIN</code>	Стандарт XPG	Не поддерживается в Linux		
<code>M_KEEP</code>	Стандарт XPG	Не поддерживается в Linux		
<code>M_MMAP_MAX</code>	Уникальный для Linux	$64 * 1024$	≥ 0	0 отключает использование <code>mmap()</code>
<code>M_MMAP_THRESHOLD</code>	Уникальный для Linux	$128 * 1024$	≥ 0	0 отключает использование кучи
<code>M_MXFAST</code>	Стандарт XPG	64	0 - 80	0 отключает использование быстрых корзин
<code>M_NLBLKS</code>	Стандарт XPG	Не поддерживается в Linux		
<code>M_PERTURB</code>	Уникальный для Linux	0	0 или 1	0 отключает пертурбацию
<code>M_TOP_PAD</code>	Уникальный для Linux	0	≥ 0	0 отключает заполнение сверху
<code>M_TRIM_THRESHOLD</code>	Уникальный для Linux	$128 * 1024$	≥ -1	-1 отключает отсечение

Любые вызовы `malloc()` в программе должны происходить до первого вызова `malloc()` или другого интерфейса выделения памяти. Использовать их просто:

```
int ret;

/* используем mmap() для всех операций выделения больше 64 Кбайт */
ret = malloc(M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
    fprintf(stderr, "malloc failed!\n");
```

Тонкая настройка с помощью `malloc_usable_size()` и `malloc_trim()`. Linux предоставляет пару функций, обеспечивающих низкоуровневое управление системой выделения памяти в `glibc`. Первая такая функция позволяет запрашивать, сколько пригодных для использования байтов содержится в указанном выделенном фрагменте памяти:

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

Успешный вызов `malloc_usable_size()` возвращает точный размер выделяемого фрагмента памяти, на который направлен указатель `ptr`. Поскольку `glibc` иногда округляет выделяемые области, чтобы они уместались в имеющийся фрагмент анонимного отображения, рабочий объем такой области может быть больше, чем запрошено. Разумеется, выделенная область не может быть меньше, чем запрошено. Вот пример использования этой функции:

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf) {
    perror ("malloc");
    return -1;
}

size = malloc_usable_size (buf);

/* на самом деле в нашем распоряжении есть 'size' байтов из буфера 'buf' ... */
```

Вторая функция позволяет программе принудить `glibc` к тому, чтобы вся память, которую компилятор может высвободить в данный момент, сразу же вернулась к ядру:

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

При успешном вызове `malloc_trim()` сегмент данных максимально укорачивается, без учета байтов `padding`, которые резервируются. В таком случае функция возвращает 1. При ошибке она возвращает 0. Как правило, `glibc` выполняет такое

уменьшение автоматически, сразу после того, как объем доступной для высвобождения памяти достигнет `M_TRIM_THRESHOLD` байт. При этом используется заполнение в `M_TOP_PAD`.

Эти функции практически никогда не пригодятся вам, кроме как при отладке или в учебных целях. Они не переносятся и предоставляют вашей программе низкоуровневые детали механизма `glibc` для выделения памяти.

Отладка при операциях выделения памяти

Программы могут устанавливать переменную окружения `MALLOC_CHECK_`, обеспечивающую перевод подсистемы памяти в режим отладки. Дополнительные отладочные проверки осуществляются за счет снижения эффективности выделения памяти, но на этапе отладки в ходе разработки приложения эти издержки оправданны.

Отладкой управляет переменная окружения, поэтому нет необходимости перекомпилировать программу. Например, можно просто выдать следующую команду:

```
$ MALLOC_CHECK_=1 ./rudder
```

Если `MALLOC_CHECK_` установлена в 0, то подсистема памяти бесшумно игнорирует все ошибки. Если здесь мы имеем значение 1, то в `stderr` выводится информативное сообщение. Если переменная получит значение 2, программа немедленно завершается с помощью `abort()`. Поскольку `MALLOC_CHECK_` изменяет поведение работающей программы, программы `setuid` игнорируют эту переменную.

Получение статистической информации. В Linux предоставляется функция `mallinfo()`, выдающая статистическую информацию о системе выделения памяти:

```
#include <malloc.h>
```

```
struct mallinfo mallinfo (void);
```

Вызов `mallinfo()` возвращает статистическую информацию в структуре `mallinfo`. Эта структура возвращается по значению, а не по указателю. Ее содержимое также определяется в `<malloc.h>`:

```
/* все размеры указаны в байтах */
```

```
struct mallinfo {
    int arena;      /* размер сегмента данных, используемого malloc */
    int ordblks;    /* количество свободных фрагментов */
    int smblks;     /* количество быстрых корзин */
    int hblks;      /* количество анонимных отображений */
    int hblkhd;     /* размер анонимных отображений */
    int usmblks;    /* максимальный общий выделяемый размер памяти */
    int fsmblks;    /* размер доступных быстрых корзин */
    int uordblks;   /* общий размер выделенного пространства */
    int fordblks;   /* размер доступных фрагментов */
    int keepcost;   /* размер пространства, которое можно отсечь */
};
```

Работать с ней просто:

```
struct mallinfo m;  
  
m = mallinfo ();  
  
printf ("free chunks: %d\n", m.ordblks);
```

В Linux также предоставляется функция `malloc_stats()`, выводящая в `stderr` статистику, относящуюся к памяти:

```
#include <malloc.h>  
  
void malloc_stats (void);
```

При вызове `malloc_stats()` в программе, интенсивно потребляющей память, получаются довольно большие числа:

```
Arena 0:  
system bytes      = 865939456  
in use bytes      = 851988200  
Total (incl. mmap):  
system bytes      = 3216519168  
in use bytes      = 3202567912  
max mmap regions  = 65536  
max mmap bytes    = 2350579712
```

Выделение памяти на основе стека

До сих пор мы обсуждали механизмы выделения динамической памяти, которые использовали для получения памяти кучу или отображения. Это неудивительно, поскольку куча и отображения в памяти являются динамическими по своей природе. В адресном пространстве процесса обычно находится еще одна программная конструкция — стек, в котором располагаются *автоматические переменные* программы.

Правда, программист вполне может использовать стек для выделения динамической памяти. Пока такое выделение не вызывает переполнения стека, подобный подход должен оставаться простым и осуществляться без проблем. Чтобы выполнить выделение динамической памяти из стека, пользуйтесь системным вызовом `alloca()`:

```
#include <alloca.h>  
  
void * alloca (size_t size);
```

В случае успеха вызов `alloca()` возвращает указатель на `size` байт в памяти. Эта память находится в стеке и автоматически высвобождается после возврата функции, которая инициировала вызов. Некоторые реализации при ошибке возвращают `NULL`, но большинство реализаций `alloca()` по природе своей не реагируют на ошибки и, соответственно, не могут о них сообщать. Сбой проявляется только на этапе переполнения стека.

Эта функция используется идентично `malloc()`, но вам не требуется высвобождать выделенную память (на самом деле этого и нельзя делать). Ниже приведен пример функции, которая открывает указанный файл в конфигурационном каталоге системы — вероятно, это каталог `/etc`, — который машинезависимо определяется во время компиляции. Функция должна выделить новый буфер, скопировать в него конфигурационный каталог системы, а потом сцепить этот буфер с предоставленным именем файла:

```
int open_sysconf (const char *file, int flags, int mode)
{
    const char *etc = SYSCONF_DIR; /* "/etc/" */
    char *name;

    name = alloca (strlen (etc) + strlen (file) + 1);
    strcpy (name, etc);
    strcat (name, file);

    return open (name, flags, mode);
}
```

После возврата память, выделенная с помощью функции `alloca()`, автоматически высвобождается, поскольку стек возвращается к вызывающей функции. Таким образом, после возврата функции, вызвавшей `alloca()`, вы уже не можете использовать эту память! Тем не менее, поскольку вам не приходится выполнять какую-либо очистку (вызывать `free()`), результирующий код получается немного аккуратнее. Вот та же функция, реализованная с помощью `malloc()`:

```
int open_sysconf (const char *file, int flags, int mode)
{
    const char *etc = SYSCONF_DIR; /* "/etc/" */
    char *name;
    int fd;

    name = malloc (strlen (etc) + strlen (file) + 1);
    if (!name) {
        perror ("malloc");
        return -1;
    }

    strcpy (name, etc);
    strcat (name, file);
    fd = open (name, flags, mode);
    free (name);

    return fd;
}
```

Обратите внимание: не следует использовать память, выделенную с помощью `alloca()`, в параметрах, передаваемых вызову функции, поскольку в таком случае выделенная память окажется посреди стекового пространства, зарезервированного для параметров функций. Например, следующий код совершенно неверен:

```
/* НЕ ДЕЛАЙТЕ ТАК! */  
ret = foo (x, alloca (10));
```

У интерфейса `alloca()` довольно темная история. Во многих системах он работал неправильно или провоцировал непредсказуемое поведение. В системах с небольшим стеком и стеком, имеющим фиксированный размер, использование `alloca()` приводило к переполнению стека и аварийному завершению программы. В некоторых других системах `alloca()` просто не существовал. Со временем полные ошибок и непоследовательные реализации окончательно подмочили репутацию `alloca()`.

Итак, если вы стремитесь обеспечить переносимость вашей программы, то старайтесь не использовать `alloca()`. Тем не менее в рамках Linux `alloca()` является превосходным и недооцененным инструментом. Он работает исключительно хорошо — во многих архитектурах при выделении памяти с помощью `alloca()` приходится всего лишь повысить указатель стека. Этот вызов оказывается удобнее и производительнее, чем `malloc()`. При выделении небольших объемов памяти в уникальном для Linux коде `alloca()` позволяет радикально улучшить рабочие характеристики системы.

Дублирование строк в стеке

Один из самых распространенных случаев использования `alloca()` связан с временным дублированием строки. Например:

```
/* мы хотим дублировать 'song' */  
char *dup;  
  
dup = alloca (strlen (song) + 1);  
strcpy (dup, song);  
  
/* bcgjkmptv 'dup'... */  
  
return; /* 'dup' автоматически высвобождается */
```

Такая операция требуется довольно часто, а также, учитывая существенное ускорение работы при применении `alloca()`, в системах Linux предоставляются варианты `strdup()`, дублирующие указанную строку в стеке:

```
#define _GNU_SOURCE  
#include <string.h>  
  
char * strdupa (const char *s);  
char * strndupa (const char *s, size_t n);
```

Вызов `strdupa()` возвращает дубликат `s`. При вызове `strndupa()` дублируется до `n` символов из `s`. Если `s` длиннее `n`, то дублирование прекращается на `n` и функция прикрепляет нулевой байт. Эти функции обладают всеми достоинствами `alloca()`. Дублированная строка автоматически высвобождается, как только возвращается функция, сделавшая вызов.

Стандарт POSIX не определяет функций `alloca()`, `strdupa()` или `strndupa()`, в других операционных системах они встречаются эпизодически. Если важно обеспечить переносимость программы, то использовать эти функции настоятельно не рекомендуется. Однако в Linux `alloca()` и ее аналоги работают вполне неплохо и помогают значительно повысить производительность, избавляя вас от хитрых манипуляций с выделением динамической памяти, — достаточно просто откорректировать указатель на кадр стека.

Массивы переменной длины

В C99 появились *массивы переменной длины* (VLA). Это массивы, геометрия которых задается во время выполнения, а не компиляции. В GNU C массивы переменной длины уже поддерживались некоторое время, но после их стандартизации в C99 появилось больше стимулов для их использования. Массивы переменной длины позволяют избежать издержек, связанных с выделением динамической памяти, почти по такому же принципу, что и `alloca()`.

Они работают именно так, как вы и предполагали:

```
for (i = 0; i < n; ++i) {
    char foo[i + 1];

    /* используем 'foo'... */
}
```

Здесь `foo` — это массив символов `char`, имеющий переменный размер `i + 1`. При каждой итерации цикла `foo` создается динамически и автоматически очищается, как только он оказывается за пределами области определения. Если вместо VLA мы воспользуемся `alloca()`, то память не будет освобождена вплоть до возврата функции. Массив переменной длины гарантирует, что высвобождение памяти происходит при каждой итерации цикла. Соответственно, используя массив переменной длины, мы потребуем не более `n` байт, тогда как `alloca()` потребует `n * (n + 1) / 2` байт.

Применив массив переменной длины, мы можем переписать нашу функцию `open_sysconf()` вот так:

```
int open_sysconf (const char *file, int flags, int mode)
{
    const char *etc; = SYSCONF_DIR; /* "/etc/" */
    char name[strlen (etc) + strlen (file) + 1];

    strcpy (name, etc);
    strcat (name, file);

    return open (name, flags, mode);
}
```

Основное различие между `alloca()` и массивами переменной длины заключается в следующем: память, полученная с помощью первой функции, существует на протяжении жизни функции, в то время как во втором случае память имеется

в распоряжении до выхода содержащей ее переменной из области определения; а такой выход может наступить до возврата функции. Это может иметь как положительные, так и отрицательные последствия. В цикле `for`, который мы только что рассмотрели, высвобождение памяти при каждой итерации цикла снижает общий объем потребляемой памяти без каких-либо побочных эффектов (нам не требуется держать на подхвате дополнительную память). Однако если мы по какой-то причине хотим, чтобы память имелась в наличии дольше, чем на протяжении одного цикла, то целесообразно использовать `alloca()`.

ПРИМЕЧАНИЕ

При смешивании `alloca()` и массивов переменной длины в рамках одной функции можно спровоцировать неожиданное поведение. Избегайте риска и используйте в конкретной функции или одно, или другое.

Выбор механизма выделения памяти

В этой главе мы рассмотрели множество операций, связанных с выделением памяти. У программиста вполне может возникнуть вопрос: какое решение оптимально для конкретной задачи? В большинстве случаев вы не прогадаете, если воспользуетесь `malloc()`. Тем не менее в некоторых ситуациях целесообразен иной подход. В табл. 9.2 обобщены рекомендации по выбору механизма выделения памяти.

Таблица 9.2. Способы выделения памяти в Linux

Способ выделения	За	Против
<code>malloc()</code>	Легкий, распространенный	Выделяемая память не обязательно заполнена нулями
<code>calloc()</code>	Обеспечивает простое выделение массивов, возвращаемая память заполняется нулями	Интерфейс сложен, если используется не для выделения массивов, а в других целях
<code>realloc()</code>	Изменяет размер имеющегося выделения	Полезен только при изменении размеров имеющихся выделений
<code>brk()</code> и <code>sbrk()</code>	Обеспечивают максимально полный контроль над кучей	Слишком низкоуровневые для большинства пользователей
Анонимные выделения памяти	Простые в использовании, разделяемые, позволяют разработчику настраивать уровень защиты и предлагать рекомендации-извещения. Оптимальны при больших отображениях	Недостаточно хороши при небольших отображениях; в случаях, когда это целесообразно, <code>malloc()</code> автоматически прибегает к анонимным отображениям в памяти
<code>posix_memalign()</code>	Выделяет память, выровненную по той или иной разумной границе	Относительно новая функция, поэтому ее переносимость осложнена; излишняя мера, если перед вами не стоит очень строгих требований к выравниванию
<code>memalign()</code> и <code>valloc()</code>	В других UNIX-системах более распространены, чем <code>posix_memalign()</code>	Не относятся к стандарту POSIX, в меньшей мере контролируют выравнивание, чем <code>posix_memalign()</code>

Продолжение ⇨

Таблица 9.2 (продолжение)

Способ выделения	За	Против
<code>alloca()</code>	Очень быстрое выделение, нет необходимости явно высвобождать память; отлично подходит для небольших выделений	Не может вернуть ошибку, неудобен при выделении больших областей, не работает в некоторых UNIX-системах
Массивы переменной длины	Аналогичны <code>alloca()</code> , но высвобождают память, как только массив выходит из области видимости, а не после возврата функции	Полезны только при работе с массивами; в некоторых ситуациях высвобождающее поведение <code>alloca()</code> может быть предпочтительнее; менее распространена в UNIX-системах, чем <code>alloca()</code>

Наконец, не следует забывать об альтернативных возможностях выделения памяти: автоматическом и статическом выделении. Выделение автоматических переменных в стеке или глобальных переменных в куче зачастую оказывается проще. В таких случаях программисту не приходится манипулировать указателями и заботиться о высвобождении памяти.

Управление памятью

В языке C предоставляется семейство функций для управления необработанными байтами памяти. По принципу работы эти функции во многом напоминают интерфейсы для манипуляций со строками, такие как `strcmp()` и `strcpy()`. Однако они взаимодействуют с буфером, размер которого задается пользователем, а не опираются на предположение, что строки завершаются нулем. Обратите внимание: ни одна из этих функций не может возвращать ошибок. Предотвращение сбоев — задача, которую приходится решать программисту. Достаточно передать неверную область памяти — и не останется выхода, кроме как идти на вынужденное нарушение сегментирования!

Установка байтов

Из функций для управления памятью чаще всего применяется `memset()`:

```
#include <string.h>
```

```
void * memset (void *s, int c, size_t n);
```

Вызов `memset()` устанавливает `n` байт, начиная с `s`, в байт `c` и возвращает `s`. Часто эта функция используется для заполнения блока памяти нулями:

```
/* обнуляем [s,s+256) */
memset (s, '\0', 256);
```

Функция `bzero()` — это ранний, устаревающий интерфейс, появившийся в системе BSD для решения аналогичной задачи. В новом коде следует использовать

`memset()`, но Linux предоставляет `bzero()` для обеспечения обратной совместимости и переносимости на другие системы:

```
#include <strings.h>

void bzero (void *s, size_t n);
```

Следующий вызов аналогичен приведенному выше примеру с `memset()`:

```
bzero (s, 256);
```

Обратите внимание: `bzero()` (наряду с другими b-интерфейсами) требует заголовка `<strings.h>`, а не `<string.h>`.

ВНИМАНИЕ

Избегайте выделения памяти с помощью `malloc()`, если после этого собираетесь сразу заполнить полученную память нулями посредством `memset()`. Да, вы получите желаемый результат, но нерационально вызывать две функции для достижения такого же эффекта, что от одной `calloc()`, которая также возвращает заполненную нулями память. Так вы не просто сэкономите один вызов функции; ведь `calloc()` может получить уже заполненную нулями память из ядра. В таком случае вам не придется вручную устанавливать каждый байт в 0, что положительно сказывается на производительности.

Сравнение байтов

Аналогично `strcmp()` `memcmp()` проверяет два фрагмента памяти на эквивалентность:

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

Вызов сравнивает первые `n` байт `s1` с первыми `n` байт в `s2`, после чего возвращает 0, если блоки памяти эквивалентны. Если `s1` меньше `s2`, то возвращается значение меньше 0, а если `s1` больше `s2`, то возвращается значение больше 0.

BSD, в свою очередь, предлагает интерфейс, который в настоящее время уже выходит из употребления, но предназначен для выполнения примерно таких же задач:

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

При вызове `bcmp()` сравниваются первые `n` байт в `s1` и `s2`. Если блоки памяти эквивалентны, то возвращается нулевое значение, если неэквивалентны — то ненулевое.

В силу применения структур данных (см. подразд. «Выравнивание» разд. «Выделение динамической памяти» в этой главе), сравнение двух структур на эквивалентность с помощью `memcmp()` или `bcmp()` — неточная операция. При заполнении может применяться неинициализированный мусор, состав которого у двух экземпляров

структуры отличается, тогда как в остальном эти структуры идентичны. Следовательно, приведенный далее код является небезопасным:

```
/* идентичны ли две шлюпки? (НЕБЕЗОПАСНО) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
    return memcmp (a, b, sizeof (struct dinghy));
}
```

Если программист собирается сравнивать структуры, то нужно сравнивать друг с другом каждый элемент этих структур по очереди. Такой поход допускает некоторую оптимизацию, но определенно является более трудоемким, чем небезопасная работа с применением `memcmp()`. Вот эквивалентный код:

```
/* Идентичны ли две шлюпки? */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
    int ret;
    if (a->nr_oars < b->nr_oars)
        return -1;
    if (a->nr_oars > b->nr_oars)
        return 1;

    ret = strcmp (a->boat_name, b->boat_name);
    if (ret)
        return ret;

    /* и т. д. для каждого элемента... */
}
```

Перемещение байтов

Функция `memmove()` копирует первые `n` байт из `src` в `dst`, возвращая `dst`:

```
#include <string.h>
```

```
void * memmove (void *dst, const void *src, size_t n);
```

Опять же BSD предоставляет выходящий из употребления интерфейс для решения аналогичной задачи:

```
#include <strings.h>
```

```
void bcopy (const void *src, void *dst, size_t n);
```

Обратите внимание: обе функции принимают одинаковые параметры, однако порядок первых двух в `bcopy()` является обратным.

И `bcopy()`, и `memmove()` позволяют безопасно обрабатывать пересекающиеся области памяти (например, если часть `dst` находится внутри `src`). Таким образом, мы можем, например, перемещать байты в определенной области памяти вверх-вниз. Данная ситуация встречается редко, а программисту необходимо знать о ее насту-

плении, поэтому стандарт C определяет вариант `memmove()`, не поддерживающий такого взаимного пересечения областей памяти. Потенциально этот вариант работает быстрее:

```
#include <string.h>
```

```
void * memcopy (void *dst, const void *src, size_t n);
```

Функция работает аналогично `memmove()`, с оговоркой, что пересечение `dst` и `src` не разрешается. Если это происходит, то результат не определен.

Еще одна безопасная функция для копирования называется `memccpy()`:

```
#include <string.h>
```

```
void * memccpy (void *dst, const void *src, int c, size_t n);
```

Функция `memccpy()` работает так же, как и `memcpy()`, но копирование останавливается, если функция находит байт `c` среди первых `n` байт в `src`. Вызов возвращает указатель на байт, который следует в `dst` после `c`, либо `NULL`, если `c` не был найден.

Наконец, можно использовать `mempcpy()` для пошагового прохода через память:

```
#define _GNU_SOURCE  
#include <string.h>
```

```
void * mempcpy (void *dst, const void *src, size_t n);
```

Функция `mempcpy()` работает так же, как и `memcpy()`, но она возвращает указатель на следующий байт после последнего скопированного байта. Она полезна, если множество данных требуется скопировать в последовательно расположенные области памяти. Однако улучшение незначительное, так как возвращаемое значение — это просто сумма `dst + n`. Эта функция является уникальной для GNU.

Поиск байтов

Функции `memchr()` и `memrchr()` находят заданный байт в блоке памяти:

```
#include <string.h>
```

```
void * memchr (const void *s, int c, size_t n);
```

Функция `memchr()` просматривает `n` байт в области памяти, на которую направлен указатель `s`, и ищет символ `c`, интерпретируемый как `unsigned char`:

```
#define _GNU_SOURCE  
#include <string.h>
```

```
void * memrchr (const void *s, int c, size_t n);
```

Вызов возвращает указатель на первый байт, совпадающий с `c`, либо `NULL`, если `c` не найден.

Функция `memrchr()` равнозначна `memchr()`, но она выполняет поиск назад, начиная с байта `n`, на который указывает `s`, а не вперед — от первого байта. В отличие от `memchr()`, `memrchr()` — это расширение GNU, а не часть языка C.

Более сложные поисковые задачи решаются с помощью функции `memmem()`. Она ищет в блоке памяти произвольный массив байтов:

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack,
               size_t haystacklen,
               const void *needle,
               size_t needlelen);
```

Функция `memmem()` возвращает указатель на первый экземпляр подблока `needle`, имеющего длину `needlelen` байт, который, в свою очередь, находится в блоке памяти `haystack` длиной `haystacklen` байт. Если функции не удастся найти `needle` в `haystack`, она возвращает `NULL`. Эта функция также является расширением GNU.

Перещелкивание байтов

Библиотека Linux C предоставляет интерфейс для тривиального сворачивания байтов данных:

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

Вызов `memfrob()` затемняет первые `n` байт в памяти, начиная с `s`. При этом применяется исключающее «ИЛИ» (XOR) к каждому байту с числом 42. Вызов возвращает `s`.

Эффект от вызова `memfrob()` можно обратить, вновь вызвав `memfrob()` применительно к той же области памяти, поэтому следующий код является холостой командой, применяемой с `secret`:

```
memfrob (memfrob (secret, len), len);
```

Эта функция ни в коем случае не является полноценной (и даже приемлемой) заменой шифрования; ее использование ограничено несложным затемнением строк. Функция является уникальной для GNU.

Блокировка памяти

В Linux реализуется *подкачка страниц по требованию*. В соответствии с этой технологией страницы вызываются с диска по мере необходимости и сбрасываются на диск, как только становятся не нужны. Благодаря этому виртуальные адресные пространства процессов, работающих в системе, могут никак не соотно-

ситься с общим объемом физической памяти, поскольку вторичные носители способны создавать видимость наличия практически неограниченного объема физической памяти.

Такая подкачка происходит прозрачно, и приложения обычно вообще не должны знать, как подкачка организуется в ядре Linux. Однако возможны две ситуации, в которых приложению бывает целесообразно оказывать влияние на процедуры подкачки, происходящие в системе.

- *Детерминизм.* Приложения, работающие с ограничениями по времени, требуют детерминистского поведения. Если некоторые обращения к памяти приводят к возникновению страничных ошибок, связанных с затратными операциями дискового ввода-вывода, то приложение может испытывать необходимость в дополнительном времени на работу. Если приложение может гарантировать, что необходимые ему страницы всегда остаются в физической памяти и не сбрасываются на диск, то оно может обеспечить и отсутствие страничных ошибок при обращениях к памяти. Так достигается согласованность и детерминизм, а также улучшается производительность.
- *Безопасность.* Если в памяти хранятся конфиденциальные сведения, их можно оттуда выгрузить и хранить на диске в незашифрованном виде. Например, если закрытый ключ пользователя обычно хранится на диске в зашифрованном виде, то незашифрованная копия ключа, находившаяся в памяти, может оказаться в файле подкачки. В среде, где действуют повышенные требования к безопасности, такое поведение может оказаться неприемлемым. Если в приложении возникает подобная проблема, оно может потребовать, чтобы область, содержащая ключ, всегда оставалась в пределах физической памяти.

Разумеется, если изменить принципы подкачки, действующие в ядре, это может негативно сказаться на общей производительности. Детерминизм или безопасность в одном приложении могут улучшиться, но пока его страницы будут заблокированы в памяти, вместо них будут подкачиваться страницы другого приложения. Если мы доверяем алгоритмам ядра, то можем быть уверены, что оно оптимальным образом будет подбирать страницы для выгрузки из памяти — то есть выгружать страницы, которые с наименьшей вероятностью понадобятся нам в ближайшем будущем, поэтому, если мы изменим данное поведение, ядро может выгрузить какие-то сравнительно востребованные страницы.

Блокировка части адресного пространства

Стандарт POSIX 1003.1b-1993 определяет два интерфейса для «блокировки» одной или нескольких страниц в физической памяти. Это гарантирует, что данные страницы ни при каких условиях не будут выгружены на диск. Первая функция блокирует заданный интервал адресов:

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

```
int mlock (const void *addr, size_t len);
```

Вызов `mlock()` блокирует область виртуальной памяти, начинающуюся с `addr` и продолжающуюся в физическую память на `len` байт. При успехе этот вызов возвращает 0; при ошибке он возвращает -1 и устанавливает `errno` соответствующее значение.

Успешный вызов блокирует в памяти все те страницы, которые содержат `[addr, addr+len)`. Например, если при вызове указан всего один байт, то в памяти блокируется вся страница, на которой он находится. Стандарт POSIX диктует, чтобы `addr` был выровнен по границе страниц. Linux не требует строгого выполнения этого правила, при необходимости бесшумно округляя `addr` вниз до ближайшей целой страницы. Однако если в программе необходимо обеспечить переносимость в другие системы, то требуется гарантировать, что `addr` будет выровнен по границе страниц.

Допустимы следующие коды `errno`:

- `EINVAL` — параметр `len` является отрицательным;
- `ENOMEM` — вызывающая сторона попыталась заблокировать больше страниц, чем допускает лимит ресурса `RLIMIT_MEMLOCK` (подробнее см. в подразд. «Лимиты блокировки» ниже);
- `EPERM` — лимит ресурса `RLIMIT_MEMLOCK` был равен 0, но процесс не обладал возможностью `CAP_IPC_LOCK` (также см. подразд. «Лимиты блокировки» ниже).

ПРИМЕЧАНИЕ

Дочерний процесс не наследует заблокированный статус памяти через `fork()`. Однако, поскольку в адресных пространствах Linux действует поведение копирования при записи, страницы дочернего процесса фактически блокируются в памяти, пока этот потомок не запишет в них что-нибудь.

В качестве примера предположим, что программа держит в памяти расшифрованную строку. Процесс может заблокировать страницу, содержащую эту строку, с помощью кода вроде следующего:

```
int ret;

/* блокируем 'secret' в памяти */
ret = mlock (secret, strlen (secret));
if (ret)
    perror ("mlock");
```

Блокировка всего адресного пространства

Если процессу требуется удерживать все свое адресное пространство в физической памяти, то интерфейс `mlock()` для этого довольно неудобен. В таком случае, который вполне обычен в приложениях реального времени, POSIX определяет специальный системный вызов, блокирующий все адресное пространство:

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

int mlockall (int flags);
```


Вызов `mlockall()` блокирует все страницы в адресном пространстве актуального процесса, удерживая их в физической памяти. Параметр `flags`, управляющий таким поведением и представляющий собой побитовое «ИЛИ», может принимать одно из следующих двух значений:

- `MCL_CURRENT` — задав это значение, мы приказываем `mlockall()` заблокировать в адресном пространстве процесса все страницы, отображенные в данный момент: стек, сегмент данных, отображенные файлы и т. д.;
- `MCL_FUTURE` — указав данное значение, мы заставляем `mlockall()` гарантировать, чтобы все страницы, отображаемые в данное адресное пространство в будущем, также блокировались в памяти.

В большинстве приложений указывается побитовое «ИЛИ» из двух этих значений.

В случае успеха этот вызов возвращает 0; при ошибке он возвращает -1 и присваивает `errno` один из следующих кодов ошибки:

- `EINVAL` — параметр `flags` является отрицательным;
- `ENOMEM` — вызывающая сторона попыталась заблокировать больше страниц, чем допускает лимит ресурса `RLIMIT_MEMLOCK` (подробнее см. в подразд. «Лимиты блокировки» ниже);
- `EPERM` — лимит ресурса `RLIMIT_MEMLOCK` был равен 0, но процесс не обладал возможностью `CAP_IPC_LOCK` (также см. подразд. «Лимиты блокировки» ниже).

Разблокировка памяти

Чтобы разблокировать страницы, находящиеся в физической памяти, и вновь позволить ядру при необходимости выгрузить их на диск, POSIX стандартизирует еще два интерфейса:

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

```
int munlock (const void *addr, size_t len);  
int munlockall (void);
```

Системный вызов `munlock()` разблокирует страницы, начиная с адреса `addr` и вплоть до `len` байт. Он отменяет действие `mlock()`. Системный вызов `munlockall()` отменяет действие `mlockall()`. Оба вызова в случае успеха возвращают 0, а при ошибке возвращают -1, присваивая `errno` одно из следующих значений:

- `EINVAL` — параметр `len` является недопустимым (только для `munlock()`);
- `ENOMEM` — некоторые из указанных страниц являются недопустимыми;
- `EPERM` — лимит ресурса `RLIMIT_MEMLOCK` был равен 0, но процесс не обладал возможностью `CAP_IPC_LOCK` (см. подразд. «Лимиты блокировки» ниже).

Блокировки памяти не вкладываются друг в друга, поэтому отдельный вызов `mlock()` или `munlock()` высвобождает заблокированную страницу независимо от того, сколько раз она блокировалась с помощью `mlock()` или `mlockall()`.

Лимиты блокировки

Блокировка памяти может влиять на общую производительность системы. Действительно, если будет заблокировано слишком много страниц, то выделения памяти могут пробуксовывать, поскольку Linux лимитирует количество страниц, которые может заблокировать процесс.

Процессы, обладающие возможностью `CAP_IPC_LOCK`, могут блокировать в памяти любое количество страниц. Процессы, не имеющие такой характеристики, могут блокировать всего `RLIMIT_MEMLOCK` байт. По умолчанию данный лимит равен 32 Кбайт — вполне достаточно, чтобы хранить в памяти один-два секрета, но такой объем еще не оказывает негативного эффекта на общую производительность. Лимиты ресурсов рассматривались в гл. 6, там же было рассказано, как получать и изменять это значение.

Находится ли страница в физической памяти

Для отладки и диагностики Linux использует функцию `mincore()`, позволяющую определять, находится указанный диапазон в физической памяти или выгружен на диск:

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start,
             size_t length,
             unsigned char *vec);
```

Вызов `mincore()` предоставляет массив, указывающий, какие отображаемые страницы находятся в физической памяти в момент системного вызова. Вызов возвращает массив посредством `vec` и описывает страницы, начиная со `start` (требуется выравнивание по границам страниц) и до `length` байт (выравнивание по границам страниц не нужно). Каждый байт в `vec` соответствует одной странице в предоставленном диапазоне. Первый байт описывает первую страницу в этом диапазоне, второй — вторую и т. д. линейно. Следовательно, `vec` должен быть достаточно велик, чтобы содержать $(length - 1 + \text{размер_страницы}) / \text{размер_страницы}$ байт. Самый младший бит в каждом байте равен 1, если страница находится в физической памяти, и 0, если не находится. Остальные биты в настоящее время остаются неопределенными и зарезервированы для использования в будущем.

В случае успеха вызов возвращает 0. При ошибке он возвращает -1 и присваивает `errno` одно из следующих значений:

- `EAGAIN` — недостаточно ресурсов ядра для выполнения данного запроса;
- `EFAULT` — параметр `vec` указывает на недействительный адрес;
- `EINVAL` — значение параметра `start` не выровнено по границам страниц;
- `ENOMEM` — в диапазоне $[address, address+1)$ содержится память, не являющаяся частью файлового отображения.

В настоящее время этот системный вызов правильно работает только с файловыми отображениями, созданными с помощью `MAP_SHARED`. Это значительно ограничивает практическую применимость вызова.

Уступающее выделение памяти

Linux использует стратегию *уступающего выделения памяти*. Когда процесс требует из ядра дополнительную память — например, при увеличении своего сегмента данных или создании нового отображения в памяти, — ядро фиксирует *обязательство* выделить память, не предоставляя при этом никакого физического хранилища. Лишь когда процесс записывает информацию в новый выделенный объем памяти, ядро *удовлетворяет* это обязательство, предоставляя для обещанного объема памяти физическое пространство. Ядро делает это последовательно для каждой страницы, по мере необходимости выполняя подкачку страниц по требованию и копирование при записи.

Такое поведение имеет следующие сильные стороны. Во-первых, подобное ленивое выделение памяти позволяет ядру откладывать большую часть работы до самого последнего момента, в который ее необходимо выполнить, — притом что, возможно, некоторые выделения и не придется удовлетворять. Во-вторых, поскольку запросы удовлетворяются последовательно для каждой страницы, и при этом по требованию, физическое пространство требуется только для памяти, которая действительно используется в настоящий момент. В конечном счете объем обещанной памяти может значительно превышать количество доступной даже с учетом разделения подкачки. Последняя возможность называется *избыточным выделением*.

Избыточное выделение и полный расход памяти (ООМ). Избыточное выделение памяти позволяет системе одновременно запускать гораздо больше приложений (при этом довольно объемных), чем это было бы реализуемо в ситуации, когда каждая запрошенная страница памяти должна быть обеспечена физическим хранилищем уже в момент выделения, а не в момент фактического использования. При избыточном выделении для отображения в память файла, имеющего размер 2 Гбайт, место в памяти требуется только для страницы, на которую процесс в настоящий момент записывает данные. Аналогично без избыточного выделения каждый вызов `fork()` потребовал бы достаточно свободного пространства, чтобы продублировать все адресное пространство, несмотря на то что большинство страниц так и не подверглись бы копированию при записи.

Однако что произойдет, если процессы попытаются удовлетворить больше обязательств, чем позволяет доступный объем физической памяти и область подкачки в конкретной системе? В таком случае один запрос или более удовлетворены не будут. Ядро уже «пообещало» выделить память — системный вызов, запрашивавший память, был успешен, — поэтому ядру остается принудительно завершить (убить) процесс, чтобы высвободить нужную память.

Когда из-за избыточного выделения возникает недостаток памяти и не удастся удовлетворить запрос, несмотря на данное ранее обязательство, возникает условие

нехватки памяти (OOM). При наступлении такой ситуации ядро задействует механизм, называемый OOM killer, выбирающий процесс для принудительного завершения. В данном случае ядро пытается найти наименее важный процесс, потребляющий при этом максимальное количество памяти.

Ситуация OOM возникает редко. Поэтому если избыточное выделение будет разрешено, то в первую очередь это принесет вам огромную пользу. Тем не менее неожиданный дефицит памяти очень нежелателен, а труднопредсказуемое завершение процесса в подобных условиях часто является недопустимым.

В системах, в которых такой ситуации действительно лучше не допускать, ядро позволяет деактивировать избыточное выделение памяти в файле `/proc/sys/vm/overcommit_memory`, а также аналогичный параметр `sysctl vm.overcommit_memory`.

По умолчанию этот параметр имеет значение 0. При таком значении ядро должно применять эвристическую стратегию избыточного выделения: разумно использовать избыточное выделение, но отклонять запросы, которые кажутся вопиющими. Значение 1 обеспечивает успешное выполнение всех обязательств, при этом предупреждение пропадает впустую. Некоторые приложения, интенсивно потребляющие память (в частности, исследовательские инструменты), зачастую запрашивают гораздо больше памяти, чем им на самом деле требуется, поэтому такой параметр целесообразен.

Значение 2 полностью отключает избыточное выделение и активирует *строгий учет*. В этом случае обязательства по выделению памяти ограничены объемом раздела подкачки плюс настраиваемый процент физической памяти. Этот процент задается в файле `/proc/sys/vm/overcommit_ratio` или с помощью аналогичного параметра `sysctl`, равного `vm.overcommit_ratio`. По умолчанию действует значение 50, ограничивающее обязательства по выделению памяти объемом раздела подкачки плюс 50 % физической памяти. В физической памяти находится ядро, страницы подкачки, страницы, зарезервированные в системе, заблокированные страницы и т. д., поэтому только часть этой памяти действительно доступна для подкачки и гарантированно может использоваться для удовлетворения обязательств.

Будьте осторожны со строгим учетом! Многие разработчики систем, шарахающиеся от OOM killer, считают строгий учет панацеей. Тем не менее приложения часто выполняют множество ненужных операций выделения, сильно захватывающих «избыточную» область. Обеспечение такого поведения — одна из основных причин, по которым машины оснащаются виртуальной памятью.

10 Сигналы

Сигналы — это программные прерывания, обеспечивающие асинхронную обработку событий. Эти события могут приходить из-за пределов системы; например, пользователь может сгенерировать символ прерывания, нажав `Ctrl+C`. Другие источники прерываний — действия программы или ядра; например, сигнал возникнет, если процесс выполнит код, в котором происходит деление на нуль. В качестве примитивной формы межпроцессной коммуникации (IPC) один процесс также может послать сигнал другому процессу.

Основная черта сигналов заключается в том, что не только события происходят асинхронно — например, пользователь может нажать `Ctrl+C` в любой момент работы программы, — но и обработка сигналов в программе выполняется асинхронно. Функции обработки сигналов регистрируются в ядре, которое асинхронно вызывает функции из остальной части программы, когда программа получает тот или иной сигнал.

Еще на заре существования UNIX в этой операционной системе применялись сигналы. Со временем они развивались, причем наиболее значительный прогресс наблюдается в повышении их надежности (ранее сигналы могли теряться), а также функциональности, поскольку современные сигналы могут нести полезную нагрузку, определяемую пользователем. Раньше различные системы UNIX вносили в работу сигналов такие изменения, из-за которых сигналы из разных систем могли становиться несовместимыми. К счастью, эта проблема была решена в рамках POSIX, который стандартизировал обработку сигналов. Именно этот стандарт и используется в Linux, о нем мы здесь и поговорим.

В начале этой главы будет сделан обзор разных сигналов. Мы поговорим о том, как следует и как не следует их использовать. Затем мы обсудим разные интерфейсы Linux, обеспечивающие управление сигналами и манипулирование ими.

Самые нетривиальные приложения взаимодействуют с использованием сигналов. Даже если вы специально проектируете ваше приложение таким образом, что вся его коммуникация может выполняться без опоры на сигналы — и это зачастую хорошо! — в некоторых случаях без сигналов просто не обойтись. Так, они необходимы при завершении программы.

Концепции, связанные с сигналами

Сигналы обладают очень строгим жизненным циклом. Сначала сигнал *порождается* (иногда также говорят, что он *отсылается* или *генерируется*). Затем ядро *сохраняет* сигнал до тех пор, пока не сможет его доставить. Наконец, как только появляется такая возможность, ядро *обрабатывает* сигнал требуемым образом. В зависимости от требований процесса ядро может выполнить одно из трех действий.

- *Игнорировать сигнал.* Никаких действий не предпринимается. Есть два сигнала, которые не могут быть проигнорированы: SIGKILL и SIGSTOP. Дело в том, что системный администратор должен иметь возможность останавливать или завершать (убивать) процессы. Если бы процесс был способен проигнорировать SIGKILL (стать «бессмертным») или SIGSTOP (стать «неудержимым»), то системный администратор лишился бы такой возможности.
- *Перехватить сигнал и обработать его.* Ядро приостанавливает исполнение текущего кода в процессе и переключается на функцию, которая была зарегистрирована ранее. Затем процесс выполняет эту функцию. Когда он вернется после ее выполнения, процесс вновь перейдет к выполнению той работы, которую прервал в момент получения сигнала. Особенно часто приходится отлавливать сигналы SIGINT и SIGTERM. Процессы отлавливают SIGINT, реагируя на действие пользователя, сгенерировавшего символ прерывания. Например, терминал может перехватить этот сигнал и вернуться в основное окно с приглашением. Процессы отлавливают SIGTERM для выполнения необходимой очистки, например для отсоединения от сети или для удаления временных файлов — до завершения. Сигналы SIGKILL и SIGSTOP нельзя перехватить.
- *Выполнение действия, задаваемого по умолчанию.* Это действие зависит от того, какой именно сигнал был отправлен. Зачастую стандартное действие — это завершение процесса. Например, именно так обрабатывается сигнал SIGKILL. Тем не менее многие сигналы предоставляются для строго определенной цели и интересуют программиста лишь в конкретной ситуации. Поэтому по умолчанию подобные сигналы игнорируются, так как большинство программ в них «не заинтересованы». Ниже мы рассмотрим различные сигналы и стандартные действия, выполняемые по умолчанию при их получении.

В прежние времена, когда сигнал был доставлен, обработавшая его функция не имела никакой информации о том, что же произошло, — она знала только о самом факте возникновения определенного сигнала. В настоящее время ядро может предоставлять заинтересованному программисту широкий контекст происходящего. Как мы вскоре увидим, сигналы могут передавать даже данные, определяемые пользователем.

Идентификаторы сигналов

У каждого сигнала есть символьное имя, начинающееся с префикса SIG. Например, SIGINT отсылается, если пользователь нажал Ctrl+C. Сигнал SIGABRT генерируется,

когда процесс вызывает функцию `abort()`. Наконец, сигнал `SIGKILL` посылается при принудительном завершении процесса.

Все эти сигналы описываются в заголовочном файле, включаемом из `<signal.h>`. Сигналы — это просто препроцессорные определения, представляющие положительные целые числа. Таким образом, каждый сигнал ассоциирован с целочисленным идентификатором. У сигналов отображение имени на целое число зависит от реализации и отличается в разных системах UNIX. Правда, около десятка наиболее распространенных сигналов везде отображаются одинаково (например, `SIGKILL` — это печально известный *сигнал 9*). В портируемой программе каждый сигнал всегда должен иметь имя, удобное для чтения, и никогда — целочисленное значение.

Номера сигналов начинаются с 1 (обычно это `SIGHUP`) и далее возрастают. Всего существует около 30 сигналов, но большинство программ регулярно используют лишь несколько из них. Сигнала со значением 0 не существует; это специальное значение, известное как *нулевой сигнал*. С нулевым сигналом не связано решительно ничего интересного — он не заслуживает собственного имени. Но некоторые системные вызовы (например, `kill()`) используют значение 0 в качестве специального случая.

ПРИМЕЧАНИЕ Чтобы вывести список сигналов, поддерживаемых в вашей системе, выполните команду `kill -l`.

Сигналы, поддерживаемые в Linux

В табл. 10.1 перечислены сигналы, поддерживаемые в Linux.

Таблица 10.1. Сигналы

Сигнал	Описание	Действие по умолчанию
SIGABRT	Отправляется функцией <code>abort()</code>	Завершение с дампом ядра
SIGALRM	Отправляется функцией <code>alarm()</code>	Завершение
SIGBUS	Аппаратная ошибка или ошибка выравнивания	Завершение с дампом ядра
SIGCHLD	Дочерний процесс был завершен	Игнорируется
SIGCONT	Процесс продолжил работу и был остановлен	Игнорируется
SIGFPE	Арифметическое исключение	Завершение с дампом ядра
SIGHUP	Терминал, управляющий процессом, был закрыт (как правило, при выходе пользователя из системы)	Завершение
SIGILL	Процесс попытался выполнить недопустимую инструкцию	Завершение с дампом ядра
SIGINT	Пользователь сгенерировал символ прерывания (Ctrl+C)	Завершение
SIGIO	Событие асинхронного ввода/вывода	Завершение ¹
SIGKILL	Неотлавливаемое завершение процесса	Завершение

Продолжение ↗

¹ В других UNIX-системах, например в BSD, этот сигнал игнорируется.

Таблица 10.1 (продолжение)

Сигнал	Описание	Действие по умолчанию
SIGPIPE	Процесс записал информацию в конвейер, но считывателей не было	Завершение
SIGPROF	Истекло время, заданное на профилирующем таймере	Завершение
SIGPWR	Сбой в энергоснабжении	Завершение
SIGQUIT	От пользователя получен символ выхода (Ctrl+\)	Завершение с дампом ядра
SIGSEGV	Нарушение доступа к памяти	Завершение с дампом ядра
SIGSTKFLT	Ошибка в стеке сопроцессора	Завершение ¹
SIGSTOP	Приостанавливает выполнение процесса	Остановка
SIGSYS	Процесс попытался выполнить недопустимый системный вызов	Завершение с дампом ядра
SIGTERM	Отлавливаемое завершение процесса	Завершение
SIGTRAP	Достигнута точка останова	Завершение с дампом ядра
SIGTSTP	Пользователь сгенерировал символ приостановки (Ctrl+Z)	Остановка
SIGTTIN	Фоновый процесс выполнил считывание с управляющего терминала	Остановка
SIGTTOU	Фоновый процесс выполнил запись в управляющий терминал	Остановка
SIGURG	Операция ввода/вывода срочно требует обработки	Игнорируется
SIGUSR1	Сигнал, определяемый процессом	Завершение
SIGUSR2	Сигнал, определяемый процессом	Завершение
SIGVTALRM	Генерируется функцией <code>setitimer()</code> при вызове с флагом <code>ITIMER_VIRTUAL</code>	Завершение
SIGWINCH	Изменился размер окна управляющего терминала	Игнорируется
SIGXCPU	Превышен лимит ресурсов, выделенных процессу	Завершение с дампом ядра
SIGXFSZ	Превышен лимит ресурсов, выделенных файлу	Завершение с дампом ядра

Существует еще несколько значений сигналов, но в Linux они считаются эквивалентными другим значениям. `SIGINFO` определяется как `SIGPWR`², `SIGIOT` — как `SIGABRT`, а `SIGPOLL` и `SIGLOST` — как `SIGIO`.

Итак, для справки у нас есть вышеприведенная таблица, так что подробно рассмотрим все перечисленные в ней сигналы.

- `SIGABRT` — функция `abort()` отправляет этот сигнал процессу, который ее вызвал. После этого процесс завершается и генерирует файл ядра. В Linux `abort()` возникает, когда контрольные утверждения, например `assert()`, не выполняются.
- `SIGALRM` — функции `alarm()` и `setitimer()` (вторая — при флаге `ITIMER_REAL`) отправляют этот сигнал вызвавшему их процессу, когда завершается тревожный период. Эти функции и другие, похожие на них, рассмотрены в гл. 11.

¹ Ядро Linux больше не генерирует этот сигнал; он сохраняется только для обеспечения обратной совместимости.

² Этот сигнал определяется только в архитектуре Alpha. Во всех других машинных архитектурах этот сигнал не существует.

- SIGBUS — ядро порождает этот сигнал, когда процесс вызывает любую ошибку оборудования, не связанную с защитой памяти (в последнем случае генерируется SIGSEGV). В традиционных системах UNIX этот сигнал представляет различные неисправимые ошибки, например невыровненное обращение к памяти. Но в ядре Linux большинство подобных ошибок исправляются автоматически, без генерирования сигнала. Ядро порождает такой сигнал лишь в том случае, когда процесс ненадлежащим образом обращается к области памяти, созданной с помощью mmap() (мы подробно обсудили отображения в памяти в гл. 9). Если этот сигнал не перехвачен, то ядро завершит процесс и выполнит дамп ядра.
- SIGCHLD — каждый раз при завершении или остановке процесса ядро посылает этот сигнал его процессу-родителю. Поскольку по умолчанию SIGCHLD игнорируется, процессы должны явно отлавливать и обрабатывать его, если их интересует «жизнь собственных потомков». Обработчик этого сигнала обычно вызывает функцию wait() (см. гл. 5), чтобы узнать идентификатор pid процесса и код выхода.
- SIGCONT — ядро посылает этот сигнал процессу, когда процесс должен возобновить работу после остановки. По умолчанию этот сигнал игнорируется, но процессы могут отлавливать его, если «планируют» совершить какое-то действие, возвращаясь к работе. Данный сигнал обычно используется на терминалах или в редакторах, которые при возвращении к работе обновляют экран.
- SIGFPE — несмотря на название этот сигнал представляет любое арифметическое исключение. Его применение не ограничено операциями с плавающей точкой. К числу таких исключений относятся переполнения, выходы за нижнюю границу (отрицательные переполнения), а также деление на нуль. По умолчанию в данном случае происходит завершение процесса и генерируется файл ядра, но процессы могут отлавливать этот сигнал и обрабатывать его. Обратите внимание: если процесс решает продолжить работу, то поведение процесса и результат проблемной операции остаются неопределенными.
- SIGHUP — ядро посылает этот сигнал ведущему процессу сеанса при отключении терминала. Ядро также отправляет этот сигнал каждому из тех процессов, которые работают в приоритетном режиме, когда завершается ведущий процесс сеанса. В данном случае по умолчанию применяется завершение, и это логично — ведь сигнал означает, что пользователь вышел из системы. Служебные процессы (демоны) перегружают этот сигнал, обогащая его механизмом, требующим перезагрузить конфигурационные файлы. Например, посылая SIGHUP серверу Apache, мы приказываем ему заново прочитать httpd.conf. Использование SIGHUP для этой цели — распространенное соглашение, но такой метод не обязателен. Эта практика безопасна, поскольку у демонов нет управляющих терминалов, поэтому они, как правило, не должны получать такой сигнал.
- SIGILL — ядро генерирует этот сигнал, когда процесс пытается выполнить недопустимую машинную инструкцию. Стандартное действие — завершить процесс и выполнить дамп ядра. Процессы могут по своему усмотрению отлавливать

и обрабатывать SIGILL, но после его возникновения их поведение остается неопределенным.

- SIGINT — отправляется всем процессам из группы, работающей в приоритетном режиме, когда пользователь вводит символ прерывания (обычно Ctrl+C). Стандартное действие — завершение процесса, но по своему усмотрению процессы могут отлавливать и обрабатывать этот сигнал. Обычно они так и делают, чтобы выполнить очистку перед завершением.
- SIGIO — отправляется, когда генерируется событие асинхронного ввода/вывода в стиле BSD. Подобный ввод/вывод в Linux используется редко. Распространенные в Linux приемы расширенного ввода/вывода подробно рассмотрены в гл. 4.
- SIGKILL — порождается системным вызовом `kill()`. Он предоставляет системным администраторам надежный способ безусловного завершения процесса. Этот сигнал не может быть перехвачен или проигнорирован, при его получении всегда происходит завершение процесса.
- SIGPIPE — если процесс записывает информацию в конвейер, но другой процесс, который должен был ее прочитать, уже завершился, то ядро порождает этот сигнал. По умолчанию в данном случае процесс завершается, но этот сигнал можно перехватить и обработать.
- SIGPROF — его генерирует функция `setitimer()`, если она используется с флагом `ITIMER_PROF`. Сигнал отправляется, когда истекает время, заданное на профилирующем таймере. По умолчанию процесс в данном случае завершается.
- SIGPWR — работа сигнала зависит от конкретной системы. В Linux он возникает, когда наступает низкий уровень заряда батареи (например, при работе с блоком бесперебойного питания UPS). Демон, выполняющий мониторинг UPS, посылает этот сигнал процессу `init`, который в ответ осуществляет очистку и завершает работу системы — остается надеяться, еще до того, как питание окончательно иссякнет!
- SIGQUIT — ядро генерирует этот сигнал для всех процессов из группы, работающей в приоритетном режиме, когда пользователь вводит символ завершения сеанса на терминале (обычно Ctrl+\\). По умолчанию все процессы завершаются, и генерируется дамп ядра.
- SIGSEGV — этот сигнал, название которого является сокращением от выражения *segmentation violation* («*нарушение сегментирования*»), передается процессу, когда тот пытается совершить недопустимое обращение к памяти. В частности, к таким ситуациям относится попытка обращения к неотображенной памяти, попытка считывания из памяти, которая закрыта для чтения, выполнение кода в области памяти, не допускающей выполнения, либо запись в область памяти, не предназначенную для записи. Процессы могут перехватывать и обрабатывать этот сигнал, но по умолчанию процесс завершается и генерируется дамп ядра.
- SIGSTOP — может быть отправлен только системным вызовом `kill()`. Он безусловно останавливает процесс и не может быть перехвачен или обработан.

- SIGSYS — ядро отправляет этот сигнал процессу, который пытается сделать недопустимый системный вызов. Это может произойти, если двоичный файл был создан в более новой версии операционной системы, а используется в более старой. Правильно построенные двоичные файлы, выполняющие свои системные вызовы через `glibc`, не получают такого сигнала. Вместо этого недопустимые системные вызовы должны возвращать `-1` и присваивать переменной `errno` значение `ENOSYS`.
- SIGTERM — сигнал генерируется только системным вызовом `kill()` и позволяет пользователю аккуратно завершить процесс (это действие выполняется по умолчанию). На усмотрение процесса такой сигнал может перехватываться, после чего происходит очистка и лишь потом — завершение. Но практика перехвата этого сигнала, после которого не происходит своевременного завершения, считается грубой.
- SIGTRAP — ядро отправляет этот сигнал процессу, когда он пересекает точку останова. Как правило, этот сигнал перехватывают отладчики, остальные процессы его игнорируют.
- SIGTSTP — ядро отправляет этот сигнал всем процессам из группы, работающей в приоритетном режиме, когда пользователь вводит символ приостановки (обычно `Ctrl+Z`).
- SIGTTIN — направляется к процессу, работающему в фоновом режиме, когда он пытается выполнить считывание из своего управляющего терминала. По умолчанию процесс завершается.
- SIGTTOU — направляется к процессу, работающему в фоновом режиме, когда он пытается записать информацию в свой управляющий терминал. По умолчанию процесс завершается.
- SIGURG — ядро отправляет этот сигнал процессу, когда на сокет прибывают внеполосные данные (в этой книге они не рассматриваются).
- SIGUSR1 и SIGUSR2 — доступны для выполнения задач, определяемых пользователем. Ядро никогда их не посылает. Процессы могут использовать SIGUSR1 и SIGUSR2 для любых целей по своему усмотрению. Как правило, с их помощью даются инструкции процессу-демону, чтобы он действовал не так, как обычно. По умолчанию при получении любого из этих сигналов процесс завершается.
- SIGVTALRM — функция `setitimer()` отсылает этот сигнал, когда истекает время на таймере, созданном с применением флага `ITIMER_VIRTUAL`. Таймеры будут рассмотрены в гл. 11.
- SIGWINCH — ядро отправляет этот сигнал всем процессам из группы, работающей в приоритетном режиме, когда на их терминале изменяется размер окна. По умолчанию процессы игнорируют этот сигнал, но по усмотрению процесса сигнал также может быть перехвачен и обработан, если при работе программа учитывает размер окна терминала. Хороший пример программы, перехватывающей такой сигнал, — `top`. Попробуйте изменить размер ее окна, пока она работает, и посмотрите, как она отреагирует.

- SIGXCPU — ядро порождает этот сигнал, когда процесс превышает отведенный ему лимит ресурсов конфигурируемого процессора. Ядро продолжит выдавать этот сигнал раз в секунду до тех пор, пока процесс не закончит работу или не исчерпает свой лимит ресурсов аппаратного процессора. Как только будет исчерпан лимит ресурсов аппаратного процессора, ядро пошлет процессу сигнал SIGKILL.
- SIGXFSZ — ядро посылает этот сигнал, как только процесс превышает отведенный ему лимит размера файла. По умолчанию в данном случае процесс завершается, но если этот сигнал перехвачен или проигнорирован, то системный вызов, спровоцировавший превышение максимального размера файла, возвращает -1 и присваивает `errno` значение `EFBIG`.

Основы управления сигналами

Итак, мы разобрались со значениями всех сигналов. Теперь рассмотрим, как управлять ими внутри вашей программы. Простейший и самый старый интерфейс для работы с сигналами — это функция `signal()`. Она определяется в стандарте ISO C89, который регламентирует лишь наименьший «общий знаменатель» поддержки сигналов. Поэтому данный системный вызов очень прост. Linux предоставляет и другие интерфейсы, обеспечивающие гораздо более полный контроль при работе с сигналами. Эти функции будут рассмотрены ниже в текущей главе. Поскольку функция `signal()` — самая простая из себе подобных, а также описана в ISO C и потому довольно распространена, мы для начала рассмотрим именно ее:

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal (int signo, sighandler_t handler);
```

Успешный вызов `signal()` отменяет актуальное действие, предпринятое в ответ на получение сигнала `signo`. Вместо этого сигнал обрабатывается с помощью механизма, указанного в `handler`. `signo` — это название одного из сигналов, рассмотренных в предыдущем разделе, например `SIGINT` или `SIGUSR1`. Как вы помните, процесс не сможет перехватить сигналы `SIGKILL` и `SIGSTOP`, поэтому устанавливать один из них в качестве значения для обработчика не имеет смысла.

Функция `handler` должна возвращать `void`. И это правильно, поскольку для этой функции (в отличие от обычных) в программе не предусмотрено стандартное место для возврата. Эта функция принимает один аргумент — целое число, являющееся идентификатором обрабатываемого сигнала (например, `SIGUSR2`). Он позволяет одной функции обрабатывать сразу много сигналов. Прототип имеет следующую форму:

```
void my_handler (int signo);
```

Для определения этого прототипа Linux использует объявление `sighandler_t`. В других системах UNIX напрямую задействуются указатели функций. В отдельных

системах вы найдете собственные типы, которые могут называться не `sighandler_t`, а как-то иначе. Если необходимо обеспечить переносимость программы, то в ней не следует прямую ссылаться на типы.

Когда эта функция направляет сигнал к процессу, зарегистрировавшему обработчик сигналов, ядро приостанавливает выполнение основного потока инструкций программы и вызывает обработчик сигнала. Обработчик получает значение сигнала, которое представляет собой `signo`, изначально переданное функции `signal()`.

Вы также можете применять функцию `signal()`, чтобы приказать ядру проигнорировать указанный сигнал для актуального процесса либо чтобы сбросить сигнал и применить поведение, заданное по умолчанию. Это делается с помощью специальных значений, присваиваемых параметру `handler`:

- `SIG_DFL` — устанавливает поведение сигнала, указанного как `signo`, в качестве действующего по умолчанию. Например, в случае `SIGPIPE` процесс будет завершен;
- `SIG_IGN` — игнорирует сигнал, указанный как `signo`.

Функция `signal()` возвращает предыдущее поведение сигнала. Это может быть применение указателя на обработчик сигнала, значение `SIG_DFL` или `SIG_IGN`. При ошибке функция возвращает `SIG_ERR`. Она не устанавливает значения `errno`.

Ожидание любого сигнала

Стандарт POSIX определяет системный вызов `pause()`, полезный при отладке и при написании демонстрационных фрагментов кода. Этот вызов переводит процесс в спящий режим до тех пор, пока процесс не получит сигнал, который либо будет обработан, либо завершит сам процесс:

```
#include <unistd.h>
```

```
int pause (void);
```

Функция `pause()` возвращается лишь в том случае, если сигнал получен (и, соответственно, этот сигнал обрабатывается). В такой ситуации `pause()` возвращает `-1` и присваивает `errno` значение `EINTR`. Если ядро порождает игнорируемый данным процессом сигнал, то процесс не просыпается.

В ядре Linux `pause()` является одним из простейших системных вызовов. Он выполняет всего два действия. Во-первых, переводит процесс в непрерывный спящий режим. Во-вторых, вызывает `schedule()`, чтобы активизировать планировщик процессов Linux и найти другой процесс для запуска. Поскольку этот процесс фактически ничего не ожидает, ядро не станет его «будить», пока он не получит сигнал. Все эти перипетии выражаются всего в двух строках кода на C¹.

¹ Следовательно, `pause()` не является простейшим из существующих системных вызовов. Вызовы `getpid()` и `gettid()` еще проще — каждый из них занимает всего одну строку.

Примеры

Рассмотрим пару простых примеров. В первом из них мы регистрируем обработчик сигнала SIGINT, который просто выводит сообщение, а затем завершает программу (как обычно при использовании SIGINT):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* обработчик для SIGINT */
static void sigint_handler (int signo)
{
    /*
     * Технически не следует использовать printf()
     * в обработчике сигнала, но если вы так сделаете, это не смертельно.
     * Подробнее об этом мы поговорим в разделе
     * «Реентерабельность».
     */
    printf ("Захвачен сигнал SIGINT!\n");
    exit (EXIT_SUCCESS);
}

int main (void)
{
    /*
     * Регистрируем sigint_handler как наш обработчик
     * для сигнала SIGINT.
     */
    if (signal (SIGINT, sigint_handler) == SIG_ERR) {
        fprintf (stderr, "Невозможно обработать SIGINT!\n");
        exit (EXIT_FAILURE);
    }

    for (;;)
        pause ();

    return 0;
}
```

В следующем примере мы зарегистрируем точно такой же обработчик для SIGTERM и SIGINT. Кроме того, мы сбросим поведение сигнала SIGPROF, чтобы он действовал, как установлено по умолчанию (процесс будет завершаться), и проигнорируем SIGHUP (который в противном случае завершил бы процесс):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* обработчик для SIGINT и SIGTERM */
```

```
static void signal_handler (int signo)
{
    if (signo == SIGINT)
        printf ("Захвачен сигнал SIGINT!\n");
    else if (signo == SIGTERM)
        printf ("Захвачен сигнал SIGTERM!\n");
    else {
        /* этого ни в коем случае не должно произойти */
        fprintf (stderr, "Неожиданный сигнал!\n");
        exit (EXIT_FAILURE);
    }
    exit (EXIT_SUCCESS);
}

int main (void)
{
    /*
     * Регистрируем signal_handler как наш обработчик сигнала
     * для SIGINT.
     */
    if (signal (SIGINT, signal_handler) == SIG_ERR) {
        fprintf (stderr, "Невозможно обработать SIGINT!\n");
        exit (EXIT_FAILURE);
    }

    /*
     * Регистрируем signal_handler как наш обработчик сигнала
     * для SIGTERM.
     */
    if (signal (SIGTERM, signal_handler) == SIG_ERR) {
        fprintf (stderr, "Невозможно обработать SIGTERM!\n");
        exit (EXIT_FAILURE);
    }

    /* Сбрасываем поведение SIGPROF, чтобы он действовал как по умолчанию. */
    if (signal (SIGPROF, SIG_DFL) == SIG_ERR) {
        fprintf (stderr, "Невозможно сбросить SIGPROF!\n");
        exit (EXIT_FAILURE);
    }

    /* Игнорируем SIGHUP. */
    if (signal (SIGHUP, SIG_IGN) == SIG_ERR) {
        fprintf (stderr, "Невозможно игнорировать SIGHUP!\n");
        exit (EXIT_FAILURE);
    }

    for (;;)
        pause ();

    return 0;
}
```

Выполнение и наследование

При ветвлении дочерний процесс наследует все сигнальные действия от своего родительского процесса. Это означает, что процесс-потомок копирует у родителя зарегистрированные действия (игнорировать, применить действие по умолчанию, обработать) для каждого из сигналов. Сигналы, ожидающие обработки, *не наследуются*, и это правильно: такой сигнал будет отправлен процессу с конкретным pid, а определенно не его потомку.

Когда процесс создается с помощью одного из системных вызовов, относящихся к семейству `ехес`, все сигналы действуют так, как это задано по умолчанию, если только родительский процесс их не игнорирует. В последнем случае новый отображенный процесс также будет их игнорировать. Иными словами: любой сигнал, перехваченный процессом-предком до `ехес`, после `ехес` выполняет действие, определенное по умолчанию, а все остальные сигналы остаются без изменений. Это действительно целесообразно, поскольку свежезапущенный процесс не использует совместно с предком его адресное пространство. Следовательно, могут отсутствовать какие-либо зарегистрированные обработчики сигналов. Сигналы, ожидающие обработки, наследуются. Информация о наследовании обобщена в табл. 10.2.

Таблица 10.2. Наследуемое поведение сигналов

Поведение сигнала	Между ветвлениями	Между вызовами <code>ехес</code>
Игнорирование	Наследуется	Наследуется
По умолчанию	Наследуется	Наследуется
Обработка	Наследуется	Не наследуется
Ожидающие сигналы	Не наследуется	Наследуется

Эти варианты поведения при выполнении процесса на практике имеют интересное практическое преимущество: когда оболочка выполняет процесс в фоновом режиме (или когда фоновый процесс выполняет другой процесс), такой новый процесс должен игнорировать символы прерывания и выхода. Следовательно, прежде чем оболочка запустит фоновый процесс, она должна задать для сигналов `SIGINT` и `SIGQUIT` флаг `SIG_IGN`. Поэтому зачастую в программах, обрабатывающих эти сигналы, сначала требуется удостовериться, что они не игнорируются. Например:

```
/* обрабатываем SIGINT, но только если он не игнорируется */
if (signal (SIGINT, SIG_IGN) != SIG_IGN) {
    if (signal (SIGINT, sigint_handler) == SIG_ERR)
        fprintf (stderr, "Невозможно обработать SIGINT!\n");
}

/* обрабатываем SIGQUIT, но только если он не игнорируется */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN) {
    if (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
        fprintf (stderr, "Невозможно обработать SIGQUIT!\n");
}
```


Необходимость устанавливать поведение сигнала таким образом, чтобы это поведение можно было проверить, красноречиво свидетельствует о недостатке интерфейса `signal()`. Ниже мы изучим функцию, не имеющую такого недостатка.

Сопоставление номеров сигналов и строк

Во всех примерах, рассмотренных до сих пор, мы жестко задавали имена сигналов. Но иногда бывает более удобно (или даже необходимо) иметь возможность преобразовать номер сигнала в строковое представление его имени. Это можно сделать несколькими способами. Например, мы можем получить строку из статически определенного списка:

```
extern const char * const sys_siglist[];
```

`sys_siglist` — это массив строк, содержащий имена сигналов, поддерживаемых в системе. Они индексируются в массиве номерами сигналов.

Существует альтернативный интерфейс, который определен в BSD и называется `psignal()`. Эта функция достаточно распространена, она поддерживается и в Linux.

```
#include <signal.h>
```

```
void psignal (int signo, const char *msg);
```

Вызов `psignal()` выводит в `stderr` ту строку, которую вы предоставили в качестве аргумента `msg`. Далее следует двоеточие, пробел и сигнал, указанный в `signo`. Если `msg` пропустить, то в `stderr` будет выводиться только имя сигнала. Если значение `signo` недопустимо, то это будет указано в сообщении, которое вы увидите в `stderr`.

Более удобный интерфейс — функция `strsignal()`. Она не стандартизирована, но поддерживается и в Linux, и во многих других системах:

```
#define _GNU_SOURCE  
#include <string.h>
```

```
char * strsignal (int signo);
```

Вызов `strsignal()` возвращает указатель на описание сигнала, заданного как `signo`. Если значение `signo` является недопустимым, то это обычно четко описывается в возвращаемом сообщении (некоторые другие системы UNIX, поддерживающие эту функцию, вместо подобного сообщения возвращают значение `NULL`). Возвращаемая строка остается валидной только до следующего вызова `strsignal()`, поэтому такая функция небезопасна для потоков.

Как правило, удобнее всего работать с `sys_siglist`. Воспользовавшись таким подходом, мы можем следующим образом переписать рассмотренный выше обработчик сигнала:

```
static void signal_handler (int signo)  
{  
    printf ("Захвачен сигнал %s\n", sys_siglist[signo]);  
}
```

Отправка сигнала

Системный вызов `kill()`, лежащий в основе распространенной утилиты `kill`, отправляет сигнал от одного процесса к другому:

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill (pid_t pid, int signo);
```

В обычной ситуации (когда `pid` больше нуля) `kill()` посылает сигнал `signo` процессу с идентификатором `pid`.

Если параметр `pid` равен нулю, `signo` отправляется всем процессам, относящимся к той же группе, что и вызывающий процесс.

Если `pid` равен `-1`, то сигнал `signo` отсылается всем тем процессам, которым вызывающий процесс может отправить сигнал (имеет на это право доступа), за исключением `init` и себя самого. В следующем подразделе мы обсудим права доступа, управляющие доставкой сигналов.

Если `pid` меньше `-1`, то сигнал отсылается группе процессов `-pid`.

В случае успеха `kill()` возвращает `0`. Вызов считается успешным, как только удастся послать один сигнал. При ошибке (ни один сигнал послать не удалось) вызов возвращает `-1` и присваивает `errno` одно из следующих значений:

- `EINVAL` — сигнал, обозначенный `signo`, является недопустимым;
- `EPERM` — вызывающий процесс не обладает достаточными правами доступа, чтобы послать сигнал какому-либо запрошенному процессу;
- `ESRCH` — процесс или группа процессов, определенные `pid`, не существуют либо (в случае процесса) это процесс-зомби.

Права доступа

Для того чтобы отправить сигнал другому процессу, отсылающий процесс должен располагать соответствующими правами доступа. Процесс, обладающий возможностью `CAP_KILL` (обычно таким процессом владеет администратор), может послать сигнал любому процессу. Без такой возможности фактический или реальный пользовательский ID процесса-отправителя должен быть равен реальному или сохраненному пользовательскому ID процесса-получателя. Проще говоря, обычный пользователь может послать сигнал лишь такому процессу, каким владеет сам.

ПРИМЕЧАНИЕ

В UNIX-системах и в Linux в частности определяется исключение для `SIGCONT`: процесс может послать этот сигнал любому процессу из того же сеанса. Совпадения пользовательского ID не требуется.

Если значение `signo` равно нулю (уже упоминавшийся выше нулевой сигнал), то вызов не отправляет никакого сигнала, но выполняет проверку ошибок. Это удобный инструмент, позволяющий проверить, имеет ли тот или иной процесс нужные права доступа, чтобы послать сигнал указанному процессу или процессам.

Примеры

Далее показан пример отправки сигнала `SIGHUP` процессу с ID 1722:

```
int ret;

ret = kill (1722, SIGHUP);
if (ret)
    perror ("kill");
```

Фактически этот фрагмент кода аналогичен следующему, вызывающему утилиту `kill`:

```
$ kill -HUP 1722
```

Чтобы проверить, есть ли у нас право отправить сигнал процессу 1722, не посылая при этом сам сигнал, можно поступить так:

```
int ret;

ret = kill (1722, 0);
if (ret)
    ; /* право доступа отсутствует */
else
    ; /* право доступа имеется */
```

Отправка сигнала самому себе

С помощью функции `raise()` процесс легко может отправить сигнал сам себе:

```
#include <signal.h>

int raise (int signo);
```

Этот вызов:

```
raise (signo);
```

эквивалентен следующему:

```
kill (getpid (), signo);
```

В случае успеха этот вызов возвращает 0, а при ошибке — ненулевое значение. Он не устанавливает переменную `errno`.

Отправка сигнала целой группе процессов

Еще одна вспомогательная функция позволяет с удобством отправлять сигнал всем процессам, относящимся к заданной группе, в тех случаях, когда отрицание ID группы процессов и использование `kill()` представляется слишком затратной операцией:

```
#include <signal.h>
```

```
int killpg (int pgrp, int signo);
```

Этот вызов

```
killpg (pgrp, signo);
```

эквивалентен следующему:

```
kill (-pgrp, signo);
```

Вызов остается верным, даже если `pgrp` равно нулю. В таком случае `killpg()` посылает сигнал `signo` каждому процессу из той группы, к которой относится вызывающий процесс.

При успехе `killpg()` возвращает 0. При ошибке он возвращает -1 и присваивает `errno` одно из следующих значений:

- `EINVAL` — сигнал, указанный как `signo`, является недопустимым;
- `EPERM` — вызывающий процесс не обладает достаточными правами доступа, чтобы отправить сигнал всем требуемым процессам;
- `ESRCH` — группа процессов, обозначенная `pgrp`, не существует.

Реентерабельность¹

Когда ядро порождает сигнал, тот процесс, которому он адресован, может быть занят выполнением какого-либо кода. Например, он может быть как раз занят важной операцией, которая, будучи прерванной, оставит процесс в несогласованном состоянии. Допустим, структура данных будет обновлена лишь наполовину или вычисление выполнено лишь частично. Возможно, процесс даже занят обработкой другого сигнала.

Обработчики сигнала не могут определить, какой код исполняется в процессе в тот момент, когда приходит сигнал. Поэтому очень важно обеспечить, чтобы любой обработчик сигналов, устанавливаемый вашим процессом, очень аккуратно выполнял любые действия, требуемые ему для работы, и бережно обращался с затрагиваемыми данными. Обработчики сигналов не должны гадать о том, чем занимался процесс перед тем, как его прервали. В частности, особая осторожность необходима при изменении глобальных (то есть разделяемых) данных. Действи-

¹ Подробнее об этой возможности см. <http://ru.wikipedia.org/wiki/Реентерабельность>.

тельно, было бы неплохо, если бы обработчик сигналов вообще не касался глобальных данных. Но в следующем разделе мы рассмотрим способ временного блокирования доставки сигналов — так, чтобы можно было безопасно манипулировать данными, которые находятся в совместном использовании между обработчиком сигналов и остальным процессом.

А что же насчет системных вызовов и других библиотечных функций? Что, если ваш процесс как раз записывает информацию в файл или выделяет память, а обработчик сигнала пытается записать данные в тот же файл или пробует вызвать `malloc()`? Что делать, если процесс сейчас вызывает функцию, использующую статический буфер, например `strsignal()`, а тут приходит сигнал?

Некоторые функции, разумеется, не допускают реентерабельности. Если программа как раз выполняет нереентерабельную функцию и возникает сигнал, после которого обработчик сигнала вызывает ту же самую функцию, может возникнуть хаос. *Реентерабельной* называется такая функция, которую можно безопасно вызывать из нее же (либо параллельно, из другого потока в том же процессе). Чтобы функция считалась реентерабельной, она не должна манипулировать статическими данными, а должна иметь дело только с данными, выделенными в стеке и предоставляемыми вызывающей стороной. Кроме того, она не должна вызывать никакую нереентерабельную функцию.

Гарантированно реентерабельные функции. При написании обработчика сигналов вы должны учитывать, что прерываемый процесс может быть остановлен в ходе выполнения нереентерабельной функции (или любой другой, если уж на то пошло). Таким образом, обработчики сигналов должны оперировать только реентерабельными функциями.

В разных стандартах подготовлены списки функций, которые являются *сигналобезопасными*: реентерабельными и, соответственно, допустимыми для применения в обработчике сигналов. Наиболее примечательно, что в POSIX.1-2003 и Single UNIX Specification дается список функций, которые гарантированно являются реентерабельными и сигналобезопасными на всех платформах, соответствующих данному стандарту или спецификации. Эти функции перечислены в табл. 10.3.

Таблица 10.3. Функции, которые гарантированно реентерабельны и безопасны для работы с сигналами

<code>abort()</code>	<code>accept()</code>	<code>access()</code>
<code>aio_error()</code>	<code>aio_return()</code>	<code>aio_suspend()</code>
<code>alarm()</code>	<code>bind()</code>	<code>cfgetispeed()</code>
<code>cfgetospeed()</code>	<code>cfsetispeed()</code>	<code>cfsetospeed()</code>
<code>chdir()</code>	<code>chmod()</code>	<code>chown()</code>
<code>clock_gettime()</code>	<code>close()</code>	<code>connect()</code>
<code>creat()</code>	<code>dup()</code>	<code>dup2()</code>
<code>execle()</code>	<code>execve()</code>	<code>_Exit()</code>
<code>_exit()</code>	<code>fchmod()</code>	<code>fchown()</code>
<code>fcntl()</code>	<code>fdatasync()</code>	<code>fork()</code>

Продолжение ➞

Таблица 10.3 (продолжение)

fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()
getpgrp()	getpid()	getppid()
getsockname()	getsockopt()	getuid()
kill()	link()	listen()
lseek()	lstat()	mkdir()
mkfifo()	open()	pathconf()
pause()	pipe()	poll()
posix_trace_event()	pselect()	raise()
read()	readlink()	recv()
recvfrom()	recvmsg()	rename()
rmdir()	select()	sem_post()
send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()
setsockopt()	setuid()	shutdown()
sigaction()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()
sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()
socketpair()	stat()	symlink()
sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	timer_getoverrun()	timer_gettime()
timer_settime()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

Безопасных функций гораздо больше, но Linux и другие системы, соответствующие POSIX, гарантируют реентерабельность лишь перечисленных функций.

Наборы сигналов

Отдельным функциям, которые мы рассмотрим ниже в этой главе, требуется манипулировать наборами сигналов. Например, процесс может блокировать набор сигналов либо набор сигналов может ожидать обработки. Для этого применяются *операции с наборами сигналов*:

```
#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);

int sigismember (const sigset_t *set, int signo);
```

Функция `sigemptyset()` инициализирует набор сигналов, указанный в `set`, помечая его как пустое множество (из этого набора исключаются все сигналы). Функция `sigfillset()` инициализирует набор сигналов, указанный в `set`, помечая его как полный (все сигналы включаются в набор). Обе функции возвращают 0. Эти две функции нужно применить к набору сигналов, прежде чем приступить к дальнейшему его использованию.

Функция `sigaddset()` добавляет сигнал `signo` в набор, указанный в `set`, а `sigdelset()` удаляет `signo` из того набора сигналов, который указан в `set`. Обе функции возвращают 0 в случае успеха или -1 при ошибке. Во втором случае переменной `errno` присваивается код ошибки `EINVAL`, и это указывает, что `signo` является недопустимым идентификатором сигнала.

Функция `sigismember()` возвращает 1, если `signo` — это набор сигналов, указанный в `set`, 0 — если это другой набор, и -1 — при ошибке. В последнем случае переменной `errno` присваивается код `EINVAL`, указывая, что значение `signo` является недопустимым.

Дополнительные функции для работы с сигналами. Все вышеописанные функции для работы с сигналами соответствуют стандарту POSIX и имеются в любой современной системе UNIX. Linux также предоставляет несколько нестандартных функций:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);

int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

Функция `sigisemptyset()` возвращает 1, если набору сигналов `set` соответствует пустое множество, и 0 — в противном случае.

Функция `sigorset()` помещает объединение (двоичное «ИЛИ») наборов сигналов `left` и `right` в `dest`.

Функция `sigandset()` помещает пересечение (двоичное «И») наборов сигналов `left` и `right` в `dest`. Обе функции возвращают 0 в случае успеха и -1 при ошибке, присваивая `errno` значение `EINVAL`.

Эти функции полезны, но ими не следует пользоваться в программах, которые должны соответствовать POSIX.

Блокировка сигналов

Выше мы обсудили реентерабельность и проблемы, возникающие при обращении с такими обработчиками сигналов, которые могут работать асинхронно, в любое время. Мы обсудили такие функции, которые нельзя вызывать внутри обработчика сигнала, так как они являются нереентерабельными.

Но что делать, если вашей программе требуется разделять данные, чтобы они совместно использовались обработчиком сигнала и какой-то другой сущностью в коде? Что, если на определенных этапах выполнения программы вы не хотите допускать никаких прерываний, в том числе провоцируемых обработчиками сигналов? Такие области в программе называются *критическими регионами*. Мы защищаем их, временно приостанавливая доставку сигналов, пока программа идет через критический регион. Принято говорить, что в это время сигналы *блокируются*. Все сигналы, которые были порождены в период блокировки, не обрабатываются до тех пор, пока не наступит разблокировка. Процесс может заблокировать любое количество сигналов. Набор сигналов, заблокированный процессом, называется его *сигнальной маской*.

POSIX определяет, а Linux реализует специальную функцию для управления сигнальной маской процесса:

```
#include <signal.h>

int sigprocmask (int how,
                 const sigset_t *set,
                 sigset_t *oldset);
```

Поведение `sigprocmask()` зависит от значения `how`, в качестве которого может служить один из следующих флагов:

- `SIG_SETMASK` — сигнальная маска вызывающего процесса изменяется на `set`;
- `SIG_BLOCK` — сигналы `set` добавляются к сигнальной маске вызывающего процесса. Иными словами, сигнальная маска заменяется объединением (двоичным «ИЛИ») из актуальной маски и `set`;
- `SIG_UNBLOCK` — сигналы `set` удаляются из сигнальной маски вызывающего процесса. Иными словами, сигнал заменяется пересечением (двоичным «НЕ») `set`. Не допускается разблокировка сигнала, который ранее не был заблокирован.

Если `oldset` не равно `NULL`, то функция помещает предыдущий набор сигналов в `oldset`.

Если `set` равно `NULL`, то функция игнорирует `how` и не меняет сигнальную маску, а помещает ее в `oldset`. Иными словами, передавая в качестве `set` нулевое значение, мы можем получить актуальную сигнальную маску.

При успехе вызов возвращает 0. При ошибке он возвращает -1 и присваивает `errno` либо `EINVAL` (это говорит о том, что значение `how` является недопустимым), либо `EFAULT` (если `set` или `oldset` является недопустимым указателем).

Блокирование `SIGKILL` или `SIGSTOP` не разрешается. Функция `sigprocmask()` бесшумно игнорирует любые попытки добавить сигнал к сигнальной маске.

Получение сигналов, ожидающих обработки

Когда ядро генерирует заблокированный сигнал, он не доставляется. Такие сигналы называются *ожидающими*. Когда ожидающий сигнал разблокируется, ядро передаст его процессу для обработки.

POSIX определяет функцию для получения набора ожидающих сигналов:

```
#include <signal.h>
```

```
int sigpending (sigset_t *set);
```

Успешный вызов функции `sigpending()` помещает набор ожидающих сигналов в `set` и возвращает 0. При ошибке вызов возвращает -1 и присваивает `errno` значение `EFAULT`. Это говорит о том, что `set` является недействительным указателем.

Ожидание набора сигналов

Третья функция, определяемая в POSIX, позволяет процессу временно изменить его сигнальную маску, а затем дожидаться, пока будет сгенерирован новый сигнал, такой, который будет обработан процессом либо завершит этот процесс.

```
#include <signal.h>
```

```
int sigsuspend (const sigset_t *set);
```

Если сигнал завершает процесс, то функция `sigsuspend()` не возвращается. Если сигнал генерируется, а потом обрабатывается, то `sigsuspend()` выдает -1 после возвращения обработчика сигнала, присваивая `errno` значение `EINTR`. Если `set` является недопустимым указателем, то `errno` присваивается значение `EFAULT`.

Обычный сценарий применения функции `sigsuspend()` связан с получением сигналов, которые могли прибыть и оказаться заблокированными, пока программа проходила критический регион. Процесс сначала использует `sigprocmask()`, чтобы заблокировать набор сигналов, и сохраняет старую маску в `oldset`. Когда критический регион останется позади, процесс вызовет `sigsuspend()` и предоставит для `set` значение `oldset`.

Расширенное управление сигналами

Функция `signal()`, с которой мы познакомились в начале этой главы, очень проста. Поскольку она входит в состав базовой библиотеки C и, следовательно, должна отражать минимальный набор предположений о возможностях той операционной системы, в которой она используется, она служит лишь наименьшим общим знаменателем при управлении сигналами. В качестве альтернативы POSIX также стандартизирует системный вызов `sigaction()`, предоставляющий более широкие возможности при управлении сигналами. В частности, эту функцию можно изменять для того, чтобы блокировать прием указанных сигналов в период действия

вашего обработчика, а также для получения разнообразной информации о системе и о состоянии процесса в момент порождения сигнала:

```
#include <signal.h>

int sigaction (int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```

Вызов функции `sigaction()` изменяет поведение сигнала, обозначенного аргументом `signo`. Этот аргумент может принимать любые значения, кроме тех, что ассоциируются с `SIGKILL` и `SIGSTOP`. Если `act` не равно `NULL`, то этот системный вызов изменяет актуальное поведение сигнала так, как это указано в `act`. Если `oldact` не равно `NULL`, то вызов сохраняет там предшествующее (а в случае `NULL` — актуальное) поведение указанного сигнала.

Структура `sigaction` обеспечивает детализированный контроль над сигналами. Заголовок `<sys/ signal.h>`, включаемый из `<signal.h>`, определяет эту структуру следующим образом:

```
struct sigaction {
    void (*sa_handler)(int); /* обработчик сигнала или действие */
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; /* сигналы, которые следует блокировать */
    int sa_flags; /* флаги */
    void (*sa_restorer)(void); /* устаревшее поле, не соответствует POSIX */
};
```

В поле `sa_handler` указывается действие, которое должно быть осуществлено после получения сигнала. Как и в случае с функцией `signal()`, это поле может быть снабжено флагом `SIG_DFL`, обозначающим действие, заданное по умолчанию, флагом `SIG_IGN`, при котором ядро игнорирует сигнал для процесса, либо указателем на функцию, обрабатывающую сигнал. Эта функция имеет такой же прототип, как и обработчик сигналов, устанавливаемый функцией `signal()`:

```
void my_handler (int signo);
```

Если флаг `SA_SIGINFO` установлен в поле `sa_flags`, то именно поле `sa_sigaction`, а не `sa_handler` определяет ту функцию, которая будет обрабатывать сигнал. Прототип этой функции немного отличается от предыдущего:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

Эта функция получает номер сигнала в качестве своего первого параметра, структуру `siginfo_t` — в качестве второго и структуру `ucontext_t` (приведенную к указателю `void`) — в качестве третьего параметра. Она не имеет возвращаемого значения. Структура `siginfo_t` предоставляет обработчику сигналов исчерпывающую информацию, с которой мы познакомимся чуть ниже.

Обратите внимание: в некоторых машинных архитектурах (и, возможно, в других системах UNIX) поля `sa_handler` и `sa_sigaction` находятся в объединении, и вы не должны присваивать значения обоим полям.

В поле `sa_mask` предоставляется набор сигналов, которые система должна блокировать на протяжении выполнения обработчика сигнала. Таким образом, программист может самостоятельно налагать нужную защиту от повторного входа сразу для многих обработчиков сигналов. Сигнал, обрабатываемый в текущий момент, также блокируется, если в поле `sa_flags` не установлен флаг `SA_NODEFER`. Вы не можете блокировать `SIGKILL` или `SIGSTOP`. Вызов будет бесшумно игнорировать любой из них в `sa_mask`.

Поле `sa_flags` — это битовая маска, включающая нуль, один или несколько флагов, изменяющих обработку сигнала, указанного как `signo`. Мы уже познакомились с флагами `SA_SIGINFO` и `SA_NODEFER`. В поле `sa_flags` также могут находиться следующие значения.

- `SA_NOCLDSTOP` — если `signo` соответствует сигнал `SIGCHLD`, этот флаг приказывает системе не выдавать уведомления, когда дочерний процесс останавливает или возобновляет работу.
- `SA_NOCLDWAIT` — если `signo` соответствует сигнал `SIGCHLD`, этот флаг обеспечивает *автоматическое снятие дочерних процессов*: при завершении дочерние процессы не превращаются в зомби, а родителю не приходится вызывать для них `wait()` (более того, он и не может этого сделать). В гл. 5 подробно рассказано о дочерних процессах, процессах-зомби и вызове `wait()`.
- `SA_NOMASK` — этот флаг является устаревшим эквивалентом `SA_NODEFER`, не поддерживаемым в POSIX (флаг `SA_NODEFER` рассмотрен выше в этой главе). Вместо `SA_NOMASK` используйте `SA_NODEFER`, но не удивляйтесь, если встретите такое значение в старом коде.
- `SA_ONESHOT` — этот флаг является устаревшим эквивалентом `SA_RESETHAND`, не поддерживаемым в POSIX (флаг `SA_RESETHAND` был рассмотрен выше в этой главе). Вместо `SA_ONESHOT` используйте `SA_RESETHAND`, но не удивляйтесь, если встретите такое значение в старом коде.
- `SA_ONSTACK` — этот флаг приказывает системе вызвать указанный обработчик сигналов для *альтернативного сигнального стека*, предоставляемого функцией `sigaltstack()`. Если вы не предоставите альтернативный стек, то будет использоваться стек, заданный по умолчанию. Соответственно, система будет действовать так, словно этот флаг вообще отсутствует. Альтернативные сигнальные стеки встречаются редко, хотя они и полезны в некоторых Pthreads-приложениях с небольшими стеками потоков, которые могут переполняться в результате определенных действий обработчика сигналов. Мы не будем более подробно обсуждать `sigaltstack()` в этой книге.
- `SA_RESTART` — этот флаг обеспечивает перезапуск в стиле BSD для тех системных вызовов, выполнение которых было прервано сигналами.
- `SA_RESETHAND` — сигнал активизирует «одноразовый» режим. Как только возвращается обработчик сигнала, поведение сигнала, снабженного этим флагом, сбрасывается на заданное по умолчанию.

Поле `sa_restorer` является устаревшим и больше не используется в Linux. В любом случае оно даже не описывается в POSIX. Считайте, что его не существует, и просто не трогайте его.

В случае успеха вызов `sigaction()` возвращает 0. При ошибке вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- EFAULT — `act` или `oldact` содержит недопустимый указатель;
- EINVAL — `signo` содержит недопустимый сигнал, SIGKILL или SIGSTOP.

Структура `siginfo_t`

Структура `siginfo_t` также определяется в `<sys/signal.h>` следующим образом:

```
typedef struct siginfo_t {
    int si_signo;      /* номер сигнала */
    int si_errno;      /* значение errno */
    int si_code;       /* код сигнала */
    pid_t si_pid;      /* идентификатор pid отсылающего процесса */
    uid_t si_uid;      /* реальный идентификатор uid отсылающего процесса */
    int si_status;      /* значение выхода или сигнал */
    clock_t si_utime;   /* затраченное пользовательское время */
    clock_t si_stime;   /* затраченное системное время */
    sigval_t si_value;  /* значение полезной нагрузки сигнала */
    int si_int;         /* сигнал POSIX.1b */
    void *si_ptr;       /* сигнал POSIX.1b */
    void *si_addr;      /* местоположение в памяти, спровоцировавшее ошибку */
    int si_band;        /* событие полосы */
    int si_fd;          /* дескриптор файла */
};
```

Эта структура изобилует информацией, передаваемой обработчику сигналов (если вы используете `sa_sigaction` вместо `sa_sighandler`). В современных вычислительных системах сигнальная модель UNIX воспринимается многими как очень неудобный способ выполнения межпроцессной коммуникации. Вероятно, такие скептики просто привыкли применять `signal()` в тех случаях, когда следовало бы прибегнуть к `sigaction()` с флагом SA_SIGINFO. Структура `siginfo_t` позволяет выжать из сигналов гораздо больше функциональных возможностей.

В этой структуре немало интересных данных. В частности, здесь мы находим информацию о процессе, который отправил сигнал, и о причине этого сигнала. Ниже приводится подробное описание каждого из полей этой структуры.

- `si_signo` — номер интересующего нас сигнала. Первый аргумент вашего обработчика сигналов также предоставляет эту информацию (что позволяет избежать разыменования указателя).
- `si_errno` — если это поле ненулевое, то оно содержит код ошибки, ассоциированный с этим сигналом. Поле действительно для всех сигналов.
- `si_code` — описывает, почему и откуда процесс получил этот сигнал (например, от `kill()`). В следующем разделе мы рассмотрим все возможные значения этого поля. Поле действительно для всех сигналов.
- `si_pid` — для SIGCHLD предоставляет pid-идентификатор того процесса, который был завершен.

- `si_status` — для `SIGCHLD` предоставляет статус выхода того процесса, который был завершен.
- `si_utime` — для `SIGCHLD` предоставляет затраченное пользовательское время того процесса, который был завершен.
- `si_stime` — для `SIGCHLD` предоставляет затраченное системное время того процесса, который был завершен.
- `si_value` — объединение `si_int` и `si_ptr`.
- `si_int` — для сигналов, посланных посредством функции `sigqueue()`, предоставленная полезная нагрузка типизируется как целое число.
- `si_ptr` — для сигналов, посланных посредством функции `sigqueue()`, предоставленная полезная нагрузка типизируется как указатель `void`.
- `si_addr` — для сигналов `SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` и `SIGTRAP` этот указатель `void` содержит номер ошибки, вызвавшей проблему. Например, в случае `SIGSEGV` в этом поле содержится тот адрес, в котором произошло нарушение доступа к памяти (следовательно, здесь нередко встречается `NULL!`).
- `si_band` — для сигнала `SIGPOLL` здесь содержится внеполосная и приоритетная информация для того файлового дескриптора, который указан в `si_fd`.
- `si_fd` — для сигнала `SIGPOLL` здесь содержится дескриптор того файла, чья операция завершилась.

Поля `si_value`, `si_int` и `si_ptr` особенно сложны в работе, так как процесс может использовать их для передачи произвольных данных другому процессу. Следовательно, вы можете задействовать их, чтобы передавать как простое целое число, так и указатель на структуру данных (кстати, в данном случае указатель вам практически не поможет, если процессы не разделяют общего адресного пространства). Эти поля подробно рассматриваются в разд. «Отправка сигнала с полезной нагрузкой» далее.

POSIX гарантирует, что лишь три поля будут действительны для всех сигналов. Доступ к остальным полям должен происходить лишь при обработке применимого с ними сигнала. Например, вы должны обращаться к полю `si_fd` только при обработке сигнала `SIGPOLL`.

Удивительный мир `si_code`

Поле `si_code` описывает причину сигнала. Для сигналов, отправляемых пользователем, это поле указывает, как был послан сигнал. Если отправителем сигнала является ядро, то поле объясняет, почему был послан сигнал.

Следующие значения поля `si_code` действительны для любого сигнала. Они указывают, как/почему был послан сигнал:

- `SI_ASYNCIO` — отправлен по причине завершения асинхронного ввода/вывода (см. гл. 5);
- `SI_KERNEL` — сгенерирован ядром;

- SI_MESGQ — отправлен в результате изменения состояния очереди сообщений POSIX (в этой книге не рассматривается);
- SI_QUEUE — отправлен функцией `sigqueue()` (см. в следующем разделе);
- SI_TIMER — отправлен, поскольку истекло время на таймере POSIX;
- SI_TKILL — послан `tkill()` или `tgkill()`. Эти системные вызовы используются библиотеками многопоточной обработки и не рассматриваются в этой книге;
- SI_SIGIO — отправлен, поскольку в очередь был поставлен SIGIO;
- SI_USER — отправлен функцией `kill()` или `raise()`.

Следующие значения `si_code` действительны только для SIGBUS. Они указывают тип произошедшей аппаратной ошибки:

- BUS_ADRALN — процесс вызвал ошибку выравнивания (выравнивание рассматривается в гл. 9);
- BUS_ADRERR — процесс обратился по недействительному физическому адресу;
- BUS_OBJERR — процесс вызвал какую-то другую аппаратную ошибку.

При работе с SIGCHLD следующие значения указывают, что сделал потомок для генерации сигнала, отосланного его предку:

- CLD_CONTINUED — работа потомка была остановлена, но позже возобновилась;
- CLD_DUMPED — произошло аварийное завершение процесса-потомка;
- CLD_EXITED — произошло обычное завершение потомка с помощью вызова `exit()`;
- CLD_KILLED — потомок был принудительно завершен (убит);
- CLD_STOPPED — потомок был остановлен;
- CLD_TRAPPED — потомок попал в ловушку.

Следующие значения допустимы только для SIGFPE. Они описывают тип произошедшей арифметической ошибки:

- FPE_FLTDIV — процесс выполнил действие над числом с плавающей точкой, в результате чего произошло деление на нуль;
- FPE_FLTOVF — процесс выполнил действие над числом с плавающей точкой, в результате чего произошло переполнение;
- FPE_FLTINV — процесс выполнил недопустимое действие над числом с плавающей точкой;
- FPE_FLTRES — процесс выполнил операцию над числом с плавающей точкой, в результате чего был получен неточный или недопустимый результат;
- FPE_FLTSUB — процесс выполнил операцию над числом с плавающей точкой, что привело к появлению нижнего индекса за пределами допустимого диапазона;
- FPE_FLTUND — процесс выполнил операцию с плавающей точкой, которая привела к выходу за нижнюю границу;
- FPE_INTDIV — процесс выполнил целочисленную операцию, которая привела к делению на нуль;

- FPE_INTOVF — процесс выполнил действие над целым числом, в результате чего произошло переполнение.

Следующие значения `si_code` допустимы только для SIGILL. Они описывают суть той недопустимой инструкции, которая была выполнена:

- ILL_ILLOPC — процесс попытался перейти в недопустимый режим адресации;
- ILL_ILLOPC — процесс попытался выполнить недопустимый код операции;
- ILL_ILLOPN — процесс попытался выполнить недопустимый операнд;
- ILL_PRVOPC — процесс попытался выполнить привилегированный код операции;
- ILL_PRVREG — процесс попытался сработать в привилегированном регистре;
- ILL_ILLTRP — процесс попытался выполнить недопустимое прерывание.

Во всех этих значениях `si_addr` указывает на адрес, по которому произошло нарушение.

При работе с SIGPOLL следующие значения указывают событие ввода/вывода, в результате которого был сгенерирован сигнал:

- POLL_ERR — возникла ошибка ввода/вывода;
- POLL_HUP — устройство зависло либо произошло разъединение с сокетом;
- POLL_IN — в файле есть данные, доступные для считывания;
- POLL_MSG — доступно сообщение;
- POLL_OUT — в данный файл может быть записана информация;
- POLL_PRI — в файле доступны для считывания высокоприоритетные данные.

Следующие значения действительны для SIGSEGV. Они описывают два типа недопустимых обращений к памяти:

- SEGV_ACCERR — процесс обратился к доступной области памяти недопустимым образом, то есть нарушил права доступа к памяти;
- SEGV_MAPERR — процесс обратился к недопустимой области памяти.

При любом из этих двух значений в `si_addr` содержится адрес, по которому произошло нарушение.

При работе с сигналом SIGTRAP применяются следующие два значения `si_code`, описывающие тип возникшей ловушки:

- TRAP_BRKPT — процесс дошел до точки останова;
- TRAP_TRACE — процесс столкнулся с прерыванием трассировки.

Обратите внимание: `si_code` — это поле со значением, а не битовое поле.

Отправка сигнала с полезной нагрузкой

Как было показано в предыдущем разделе, обработчикам сигнала, зарегистрированным с флагом SA_SIGINFO, передается параметр `siginfo_t`. В этой структуре

содержится поле `si_value`, которое может содержать опциональную полезную нагрузку, передаваемую от генератора сигнала к получателю сигнала.

Функция `sigqueue()`, стандартизированная в POSIX, позволяет процессу отправлять сигнал с такой полезной нагрузкой:

```
#include <signal.h>
```

```
int sigqueue (pid_t pid,
              int signo,
              const union sigval value);
```

Функция `sigqueue()` работает примерно так же, как и `kill()`. В случае успеха сигнал, идентифицированный как `signo`, ставится в очередь процесса или группы процессов, имеющих идентификатор `pid`, а функция возвращает 0. Полезная нагрузка сигнала задается в `value`, которое представляет собой объединение (`union`) целого числа и указателя `void`:

```
union sigval {
    int sival_int;
    void *sival_ptr;
};
```

При ошибке вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- `EAGAIN` — вызывающий процесс достиг предельного количества сигналов, которые могут быть поставлены в очередь;
- `EINVAL` — сигнал, указанный `signo`, недействителен;
- `EPERM` — вызывающий процесс не имеет достаточных прав, чтобы передать сигнал любому из запрошенных процессов. Права доступа для отправки сигнала такие же, как и при работе с `kill()` (см. разд. «Отправка сигнала» этой главы);
- `ESRCH` — процесс или группа процессов, обозначенные `pid`, не существуют, либо в случае с процессом мы имеем дело с зомби.

Как и при работе с `kill()`, вы можете передать `signo` нулевой сигнал, чтобы протестировать права доступа.

Пример сигнала с полезной нагрузкой. В этом примере мы посылаем процессу с `pid 1722` сигнал `SIGUSR2`. В качестве полезной нагрузки сигнал несет целочисленное значение 404:

```
sigval value;
int ret;

value.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, value);
if (ret)
    perror ("sigqueue");
```


Если процесс 1722 обслужит SIGUSR2 с помощью обработчика SA_SIGINFO, то обнаружит signo со значением SIGUSR2, si->si_int со значением 404 и si->si_code со значением SI_QUEUE.

Изъян в UNIX?

У сигналов дурная репутация среди многих UNIX-программистов. Это не просто старый, а устаревший механизм обмена информацией между пользователем и ядром, и в лучшем случае сигналы могут считаться примитивной формой межпроцессной коммуникации. В мире многопоточных программ и циклов событий сигналы кажутся анахронизмом. UNIX — операционная система, которая выдержала столь серьезную проверку временем и до сих пор развивает ту самую парадигму программирования, что действовала в ней с первых дней существования. В ней сигналы воспринимаются как досадное недоразумение. Я бы не замахнулся на то, чтобы «немного переделать сигналы», но начал бы с создания нового решения на базе файловых дескрипторов, которое было бы более выразительным, расширяемым, потокобезопасным.

Тем не менее хорошо это или нет, но мы по-прежнему довольствуемся сигналами. Сигналы — единственный механизм для получения многих уведомлений от ядра (например, уведомлений о выполнении недопустимого кода операции). Кроме того, именно с помощью сигналов в UNIX (а значит, и в Linux) завершаются процессы и организуются взаимоотношения «предок — потомок». Следовательно, программист должен понимать сигналы и уметь ими пользоваться.

Одна из основных причин, по которым качество работы с сигналами все сильнее падает, — сложность написания подходящего обработчика сигналов, с которым не возникали бы проблемы из области реентерабельности. Но если вы будете держать обработчики максимально простыми и будете пользоваться лишь теми функциями, что перечислены в табл. 10.3 (если вообще решите применять функции), то все должно быть безопасным.

Еще одна брешь в броне сигналов состоит в том, что многие программисты по-прежнему предпочитают использовать для управления ими вызовы `signal()` и `kill()`, а не `sigaction()` и `sigqueue()`. Как вы могли убедиться в двух последних разделах, сигналы получаются гораздо более мощными и выразительными, если применять обработчики в стиле SA_SIGINFO. Хотя я не являюсь поклонником сигналов, следует признать, что большинство их острых углов удается обходить, пользуясь продвинутыми интерфейсами Linux. Правда, капризность сигналов при этом никуда не девается.

11

Время

В современных операционных системах время выполняет разнообразные функции, и во многих программах требуется отслеживать время. Ядро измеряет ход времени тремя различными способами.

- **Фактическое время.** Это точное время и дата из нашего реального мира — то самое время, которое вы можете проверить по настенным часам. Процессы применяют фактическое время, взаимодействуя с пользователем или снабжая событие временной меткой.
- **Процессное время.** Это время, которое процесс тратит на работу в процессоре. Это может быть количество времени, потраченное на выполнение самого процесса (пользовательское время), или время, которое израсходовано ядром на работу по запросу процесса (системное время). Процессное время актуально для процессов при выполнении таких задач, как профилирование, учет, статистическая работа. Например, к подобным задачам относится измерение количества процессорного времени, потребовавшегося для завершения заданного алгоритма. В таких ситуациях фактическое время нам не подходит, поскольку, учитывая многозадачную природу Linux, за единицу фактического времени обычно протекает менее единицы процессного времени. Но бывает и наоборот: при наличии нескольких процессоров, а также какого-то числа потоков в процессе процессное время для отдельно взятой операции может превышать фактическое время!
- **Монотонное время.** Время такого рода течет по строго линейному принципу. В большинстве операционных систем, в том числе в Linux, в таком качестве используется время доступности системы (время, истекшее с момента загрузки системы). Фактическое время в компьютере может изменяться — например, если пользователь его самостоятельно установит или если система будет автоматически корректировать время, справляясь с рассинхронизацией часов. Правда, при этом могут возникать дополнительные погрешности — например, иногда теряются секунды. С другой стороны, период доступности системы есть детерминированное и неизменное представление времени. Важнейшим свойством монотонного времени является не его конкретное значение, а гарантия того, что оно течет строго линейно, и поэтому подходит для расчета временной разницы между двумя дискретными выборками.

Три эти разновидности времени могут быть представлены в одном из двух форматов.

- **Относительное время.** Это значение, которое отсчитывается относительно определенной контрольной точки, например от настоящего момента: «через 5 секунд» или «10 минут назад». Для подсчета относительного времени используется монотонное время.
- **Абсолютное время.** В данном случае мы говорим о времени, которое определяется без такой контрольной точки; например, полдень 25 марта 1968 года. Для отсчета таких временных значений идеально подходит фактическое время.

На практике используется как относительное, так и абсолютное время. Например, процессу может потребоваться отменить запрос через 500 миллисекунд, обновлять экран 60 раз в секунду или отметить, что с момента начала операции уже истекло 7 секунд. Во всех этих случаях применяется относительное время. Напротив, приложение-календарь может сохранить дату пользовательской вечеринки как 8 февраля, а файловая система полностью запишет время и дату создания файла (а не «5 секунд назад»). При этом на часах пользователя будет отображаться дата по григорианскому календарю, а не количество секунд, истекших с момента загрузки системы.

В системах UNIX абсолютное время представляется как количество секунд, истекших с момента *epoch*, который наступил в 00:00:00 по Гринвичу 1 января 1970 года. Как ни странно, но это означает, что в UNIX на самом низком уровне абсолютное время также является относительным. В UNIX существует специальный тип данных для хранения «секунд, истекших с момента epoch». Этот формат данных мы рассмотрим в следующем разделе.

Операционные системы отсчитывают ход времени с помощью *программных часов*. Это часы, ход которых программно поддерживается ядром. Ядро инстанцирует периодический таймер, который называется *системным* и тикает с определенной частотой. Когда интервал таймера заканчивается, ядро увеличивает количество истекшего времени на единицу, называемую *тактом* или *циклом*. Ранее такты были 32-битными значениями, но, начиная с версии ядра Linux 2.6, они стали 64-битными¹.

В Linux рабочая частота системного таймера называется *Hz*, поскольку именно такое самоочевидное имя определяет для нее препроцессор. Значение *Hz* зависит от конкретной архитектуры и не является частью Linux ABI. Следовательно, программа не может зависеть от того или иного значения частоты или ориентироваться на него. Исторически в архитектуре x86 использовалось значение 100, то есть система делала 100 тактов в секунду, а системный таймер имел частоту 100 Герц. Таким образом, каждый такт занимал 0,01 секунды. В версии 2.6 ядра Linux разработчики взвинтили значение *Hz* сразу до 1000, продолжительность такта там составляла всего 0,001 секунды. Но в версии 2.6.13 и выше значение *Hz* равно 250

¹ В настоящее время ядро Linux поддерживает и «бестактовый» режим работы.

(такт занимает 0,004 секунды¹). Значение HZ по определению включает в себе компромисс: чем больше значение, тем выше его разрешающая способность и тем выше издержки, связанные с таймером.

Хотя процессы и не должны опираться на какое-либо фиксированное значение HZ, POSIX описывает механизм, позволяющий во время выполнения определять частоту системного таймера:

```
long hz;
```

```
hz = sysconf ( _SC_CLK_TCK );  
if ( hz == -1 )  
    perror ( "sysconf" ); /* этого происходить не должно */
```

Этот интерфейс полезен, если программе требуется определить разрешение системного таймера, но он не нужен для преобразования системных значений времени в секунды. Дело в том, что большинство интерфейсов POSIX экспортируют временные величины, которые уже преобразованы в определенную частоту или масштабированы в нее. Данная частота не зависит от значения HZ. В отличие от HZ эта фиксированная частота входит в состав системного ABI; в архитектуре x86 ее значение равно 100. Функции POSIX, возвращающие время в пересчете на такты таймера, используют CLOCKS_PER_SEC для представления фиксированной частоты.

Иногда бывает и так, что компьютер неожиданно отключается. Иногда даже вилку вынимают из розетки, но после загрузки компьютерное время оказывается верным. Дело в том, что на большинстве компьютеров установлены работающие от батареи *аппаратные часы*, на которых отслеживается дата и время, пока компьютер остается выключен. Когда ядро загружается, оно инициализирует концепцию текущего времени, исходя из показаний аппаратных часов. Аналогично, когда пользователь выключает систему, ядро записывает текущее время на аппаратные часы. Системный администратор может синхронизировать время и в другие моменты с помощью команды `hwclock`.

Управление ходом времени в системе UNIX связано с решением нескольких задач, причем для любого отдельно взятого процесса актуальны лишь некоторые из них. В частности, к таким задачам относятся установка и определение текущего фактического времени, «засыпание» на определенный период времени, выполнение высокоточных измерений времени и управление таймерами. В этой главе мы обсудим все разнообразные рутинные процедуры, связанные с обслуживанием хода времени. Для начала рассмотрим структуры данных, применяемые в Linux для представления времени.

¹ В настоящее время HZ является одним из параметров компиляции ядра, причем в архитектуре x86 можно выбрать любое из следующих значений: 100, 250, 300 и 1000. Тем не менее программы из пользовательского пространства не могут зависеть от конкретного значения HZ.

Структуры данных, связанные с представлением времени

Системы UNIX постепенно развивались, и в них реализовывались собственные интерфейсы для управления временем. Поэтому понемногу появлялись все новые структуры данных, которые представляют все тот же обманчиво простой феномен времени. Эти структуры данных варьируются от обычных целых чисел до различных структур с множеством полей. Мы рассмотрим их здесь, а потом перейдем к подробному изучению отдельно взятых интерфейсов.

Оригинальное представление

Простейшая структура данных называется `time_t`, она определяется в заголовке `<time.h>`. Предполагалось, что `time_t` будет непрозрачным типом. Однако в большинстве систем UNIX, в том числе в Linux, он является простым определением типа `long` из языка C:

```
typedef long time_t;
```

`time_t` представляет количество секунд, истекших с момента `epoch`. «Это ведь так много, тут наверняка возникнет переполнение!» — подумают многие читатели. На самом деле до переполнения еще далеко, но оно действительно может возникнуть, так как UNIX-системы будут еще долго использоваться. При работе с 32-битным числовым типом `long` структура `time_t` может представлять до 2 147 483 647 секунд, истекших с момента `epoch`. Таким образом, новая «проблема 2000 года» ожидает нас в обозримом будущем — это будет 2038 год. Остается надеяться, что в 22:14:07 в понедельник 18 января 2038 года большинство систем и программ уже будут 64-битными.

А теперь — с микросекундной точностью!

Серьезная проблема, связанная с `time_t`, заключается в том, что за одну секунду может произойти довольно много событий. Структура `timeval` расширяет `time_t`, обеспечивая точность до микросекунд. В заголовке `<sys/time.h>` эта структура определяется следующим образом:

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

```
struct timeval {  
    time_t tv_sec;          /* секунды */  
    suseconds_t tv_usec;    /* микросекунды */  
};
```

Поле `tv_sec` измеряет секунды, а `tv_usec` — микросекунды. Странное поле `suseconds_t` обычно является определением целочисленного типа.

И еще лучше: наносекундная точность

Иногда бывает недостаточно и микросекундного разрешения, и структура `timespec` поднимает планку до наносекундной точности. В заголовке `<time.h>` эта структура определяется следующим образом:

```
#include <time.h>

struct timespec {
    time_t tv_sec;    /* секунды */
    long tv_nsec;    /* наносекунды */
};
```

При наличии выбора наносекундное разрешение обычно предпочтительнее микросекундного. Кроме того, структура `timespec` отказалась от глупого компонента `suseconds_t` в пользу простого и нетребовательного `long`. Таким образом, после появления структуры `timespec` на нее переключилось большинство интерфейсов, применяемых при работе с временем, и общая точность измерений выросла. Тем не менее, как мы вскоре увидим, одна важная функция по-прежнему использует `timeval`.

На практике ни одна из структур обычно не обеспечивает заявленной точности, так как наносекундное и даже микросекундное разрешение недостижимо для системного таймера. Тем не менее предпочтительно иметь в интерфейсе информацию о разрешении, чтобы он мог приспособиться к тому разрешению, которое действительно обеспечивается в системе.

Разбиение времени

Функции, которые мы рассмотрим, выполняют преобразования между временем UNIX и строками либо программно строят строковое представление указанной даты. Для облегчения этого процесса стандарт C описывает структуру `tm`. Она позволяет выражать «разбитое» время в формате, более удобочитаемом для человека. Эта структура также определяется в заголовке `<time.h>`:

```
#include <time.h>

struct tm {
    int tm_sec;    /* секунды */
    int tm_min;    /* минуты */
    int tm_hour;   /* часы */
    int tm_mday;   /* день месяца */
    int tm_mon;    /* месяц */
    int tm_year;   /* год */
    int tm_wday;   /* день недели */
    int tm_yday;   /* день года */
    int tm_isdst;  /* летнее время? */
#ifdef _BSD_SOURCE
    long tm_gmtoff; /* временной пояс, коррекция относительно Гринвича */
    const char *tm_zone; /* сокращенное обозначение временного пояса */
#endif
};
```

Структура `tm` помогает с большей определенностью судить, к какому именно дню относится значение структуры `time_t`, равное, например, 314159: к воскресенью или к субботе (правильный ответ — суббота). Учитывая, как много места она занимает, это, конечно, не лучший вариант представления даты и времени. Но она удобна для преобразования значений из пользовательского формата в машинный и наоборот.

Вот краткое описание полей этой структуры:

- `tm_sec` — количество секунд, истекших с момента окончания последней минуты. Обычно это значение находится в диапазоне от 0 до 59, но может достигать и 61, чтобы учитывать до двух корректировочных секунд;
- `tm_min` — количество минут, истекших с момента окончания последнего часа. Это значение находится в диапазоне от 0 до 59;
- `tm_hour` — количество часов, истекших после полуночи. Это значение находится в диапазоне от 0 до 23;
- `tm_mday` — день месяца. Это значение находится в диапазоне от 0 до 31. Стандарт POSIX не указывает значение 0, однако в Linux оно используется для последнего дня предыдущего месяца;
- `tm_mon` — количество месяцев, истекших с января. Это значение находится в диапазоне от 0 до 11;
- `tm_year` — количество лет, истекших с 1900-го;
- `tm_wday` — количество дней, истекших с воскресенья. Это значение находится в диапазоне от 0 до 6;
- `tm_yday` — количество дней, истекших с 1 января. Это значение находится в диапазоне от 0 до 365;
- `tm_isdst` — особое поле, указывающее, должен ли учитываться переход на летнее время при интерпретации значений, записанных в других полях структуры. Если это значение положительное, то применяется летнее время. Если оно равно нулю, то летнее время не применяется. Если значение отрицательное, то статус летнего времени остается неизвестным;
- `tm_gmtoff` — смещение в секундах для текущего временного пояса относительно гринвичского времени. Это поле присутствует в структуре лишь при условии, что значение `_BSD_SOURCE` будет определено до включения `<time.h>`;
- `tm_zone` — сокращенное обозначение актуального временного пояса — например, EST. Это поле присутствует в структуре лишь при условии, что значение `_BSD_SOURCE` будет определено до включения `<time.h>`.

Тип для процессного времени

Тип `clock_t` представляет такты процессорных часов. Это целочисленный тип, зачастую `long`. Такты, представляемые `clock_t`, в зависимости от применяемого интерфейса, могут соответствовать либо фактической частоте системного таймера (HZ), либо `CLOCKS_PER_SEC`.

Часы POSIX

Некоторые системные вызовы, рассматриваемые в этой главе, используют *часы POSIX* — стандарт, применяемый для реализации и представления источников времени. Тип `clockid_t` соответствует конкретной разновидности часов POSIX, в Linux поддерживается пять таких разновидностей.

- `CLOCK_REALTIME` — часы реального (фактического) времени, действующие в пределах всей системы. Установка этих часов требует специальных привилегий.
- `CLOCK_MONOTONIC` — часы с монотонным отсчетом времени, которые не может устанавливать какой-либо процесс. Представляют время, истекшее с какого-либо нежестко задаваемого момента в прошлом — например, с момента загрузки системы.
- `CLOCK_MONOTONIC_RAW` — значение похоже на `CLOCK_MONOTONIC` с той оговоркой, что эти часы не подходят для подвода (коррекции возникающих погрешностей). Соответственно, если аппаратные часы будут спешить или отставать относительно фактического времени, то при считывании данных часов вы не сможете исправить такую погрешность. Эти часы специфичны для Linux.
- `CLOCK_PROCESS_CPUTIME_ID` — это попроцессные часы высокого разрешения, предоставляемые процессором. Например, в архитектуре x86 они используют регистр счетчика цифровых меток (TSC).
- `CLOCK_THREAD_CPUTIME_ID` — напоминают попроцессные часы, но уникальны для каждого потока в процессе.

Стандарт POSIX требует наличия только одних часов — `CLOCK_REALTIME`. Поэтому, хотя Linux и обеспечивает нормальную работу всех пяти часов, в коде, предназначенном для портирования, следует использовать только `CLOCK_REALTIME`.

Разрешение источника времени. Стандарт POSIX определяет функцию `clock_getres()`, позволяющую установить разрешение конкретного источника времени:

```
#include <time.h>
```

```
int clock_getres (clockid_t clock_id,  
                 struct timespec *res);
```

Успешный вызов `clock_getres()` сохраняет в `res` разрешение часов, указанное в `clock_id`, и, если оно не равно `NULL`, вызов возвращает 0. При ошибке функция возвращает -1 и устанавливает `errno` в один из следующих двух кодов ошибок:

- `EFAULT` — `res` является недопустимым указателем;
- `EINVAL` — `clock_id` не является допустимым источником времени, применяемым в данной системе.

В следующем примере мы выводим на экран разрешение пяти источников времени, рассмотренных в предыдущем разделе:


```
clockid_t clocks[] = {
    CLOCK_REALTIME,
    CLOCK_MONOTONIC,
    CLOCK_PROCESS_CPUTIME_ID,
    CLOCK_THREAD_CPUTIME_ID,
    CLOCK_MONOTONIC_RAW,
    (clockid_t) -1 };
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++) {
    struct timespec res;
    int ret;

    ret = clock_getres (clocks[i], &res);
    if (ret)
        perror ("clock_getres");
    else
        printf ("время=%d секунд=%ld наносекунд=%ld\n",
            clocks[i], res.tv_sec, res.tv_nsec);
}
```

В современной системе с архитектурой x86 получим примерно такой вывод:

```
clock=0 sec=0 nsec=4000250
clock=1 sec=0 nsec=4000250
clock=2 sec=0 nsec=1
clock=3 sec=0 nsec=1
clock=4 sec=0 nsec=4000250
```

Обратите внимание: 4 000 250 наносекунд здесь равны 4 миллисекундам, или 0,004 секунды. В свою очередь, разрешение 0,004 секунды обусловлено работой системных часов в архитектуре x86, значение HZ для которых равно 250 (об этом мы говорили в начале главы). Следовательно, мы видим, что оба значения — CLOCK_REALTIME и CLOCK_MONOTONIC — привязаны к тактам процессора и к разрешению, обеспечиваемому системным таймером. Напротив, и CLOCK_PROCESS_CPUTIME_ID, и CLOCK_THREAD_CPUTIME_ID используют источник времени с более высоким разрешением. На данной машине с архитектурой x86 таким источником является TSC, который, как было показано выше, обеспечивает разрешение с наносекундной точностью.

В Linux, а также в большинстве других UNIX-систем все функции, использующие часы POSIX, требуют связывания с результирующим объектом посредством `librt`. Например, если скомпилировать предыдущий фрагмент кода в исполняемый вид, то у нас может получиться следующая команда:

```
$ gcc -Wall -W -O2 -lrt -g -o snippet snippet.c
```

Получение текущего времени суток

Существует несколько причин, по которым в приложениях может потребоваться определить текущие дату и время. Например, программа должна отображать эту

информацию пользователю, подсчитывать истекшее или относительное время, снабжать событие цифровой печатью и т. д. Простейшим и исторически наиболее распространенным инструментом для получения актуального времени является функция `time()`:

```
#include <time.h>

time_t time (time_t *t);
```

Вызов `time()` возвращает текущее время, выраженное в количестве секунд, истекших с момента `epoch`. Если параметр `t` не равен `NULL`, то функция также записывает актуальное время в предоставленный указатель.

При ошибке функция возвращает `-1` (с приведением типа к `time_t`) и присваивает `errno` соответствующее значение. Единственной возможной ошибкой в данном случае является `EFAULT`. Вы получите такое значение, если указатель `t` является недопустимым.

Например:

```
time_t t;

printf ("текущее значение: %ld\n", (long) time (&t));
printf ("то же значение: %ld\n", (long) t);
```

ПРИМЕЧАНИЕ

Представление времени в поле `time_t`, выражаемое как «количество секунд, истекших с момента `epoch`», не соответствует точному количеству секунд, прошедших с того судьбоносного момента времени. В действующей в UNIX системе подсчета времени високосными считаются все годы, делящиеся на четыре, а корректировочные секунды просто игнорируются. Поэтому значение времени в `time_t` не совсем точное. Но оно хотя бы является согласованным — и этого достаточно.

Более удобный интерфейс

Функция `gettimeofday()` расширяет `time()`, обеспечивая разрешение с точностью до миллисекунд:

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

int gettimeofday (struct timeval *tv,
                  struct timezone *tz);
```

При успешном вызове `gettimeofday()` текущее время записывается в структуру `timeval`, на которую направлен указатель `tv`, после чего функция возвращает `0`. Структура `timezone` и параметр `tz` уже выходят из употребления; в Linux их не следует применять. Для `tz` всегда передавайте значение `NULL`.

При ошибке этот вызов возвращает `-1` и присваивает `errno` значение `EFAULT`. Это единственная возможная ошибка; она говорит о том, что указатель `tv` или `tz` недопустим.

Например:

```
struct timeval tv;
int ret;

ret = gettimeofday (&tv, NULL);
if (ret)
    perror ("gettimeofday");
else
    printf ("секунды=%ld микросекунды=%ld\n",
           (long) tv.sec, (long) tv.tv_usec);
```

Структура `timezone` выходит из употребления, поскольку ядро при работе не учитывает временные пояса, а `glibc` отказывается работать с полем `tz_dsttime` структуры `timezone`. Работа с временными поясами будет рассмотрена в следующем разделе.

Продвинутый интерфейс

Стандарт POSIX предоставляет интерфейс `clock_gettime()` для получения значения времени из конкретного источника. Но эта функция гораздо интереснее по той причине, что теоретически работает с точностью до наносекунд:

```
#include <time.h>
```

```
int clock_gettime (clockid_t clock_id,
                  struct timespec *ts);
```

В случае успеха вызов возвращает 0 и сохраняет текущее значение источника времени, указанное с помощью `clock_id` в `ts`. При ошибке этот вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- `EFAULT` — `ts` является недопустимым указателем;
- `EINVAL` — в данной системе `clock_id` является недопустимым источником времени.

В следующем примере мы получаем актуальное время из всех четырех стандартных источников времени:

```
clockid_t clocks[] = {
    CLOCK_REALTIME,
    CLOCK_MONOTONIC,
    CLOCK_PROCESS_CPUTIME_ID,
    CLOCK_THREAD_CPUTIME_ID,
    CLOCK_MONOTONIC_RAW,
    (clockid_t) -1 };
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++) {
    struct timespec ts;
    int ret;

    ret = clock_gettime (clocks[i], &ts);
```

```

    if (ret)
        perror ("clock_gettime");
    else
        printf ("часы=%d секунды=%ld наносекунды=%ld\n",
                clocks[i], ts.tv_sec, ts.tv_nsec);
}

```

Получение процессного времени

Системный вызов `times()` получает процессное время работающего процесса и его потомков. Значение времени выражается в тактах процессора:

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

```

struct tms {
    clock_t tms_utime; /* потребленное пользовательское время */
    clock_t tms_stime; /* потребленное системное время */
    clock_t tms_cutime; /* пользовательское время, потребленное потомками */
    clock_t tms_cstime; /* системное время, потребленное потомками */
};

```

```
clock_t times (struct tms *buf);
```

В случае успеха вызов заполняет предоставленную структуру `tms`, на которую направлен указатель `buf`, записывая в нее значения процессного времени, потребленного вызывающим процессом и его потомками. Полученное количество времени подразделяется на пользовательское и системное время. *Пользовательское время* тратится на выполнение кода в пользовательском пространстве. *Системное время* тратится на выполнение кода в пространстве ядра — например, в ходе системного вызова или при отказе страницы. Сообщаемые значения времени для каждого потомка учитываются только после того, как потомок завершится, а предок вызовет в процессе `waitpid()` или другую подобную функцию. Вызов возвращает количество тактов процессора, которое монотонно повышается. Это время истекает с произвольного, но фиксированного момента в прошлом. Ранее в качестве такой точки отсчета принимался момент загрузки системы — следовательно, функция `times()` возвращала количество тактов, равное последнему непрерывному периоду работы системы, — но в настоящее время точка отсчета отнесена примерно на 429 миллионов секунд в прошлое от момента загрузки. Разработчики реализовали это изменение, чтобы перехватывать такой системный код, который не мог обрабатывать оборачивание системного времени и попадание в нуль. Таким образом, абсолютное значение, возвращаемое этой функцией, практически не имеет смысла; но относительные изменения между двумя вызовами по-прежнему остаются актуальными.

При ошибке этот вызов возвращает `-1` и присваивает `errno` соответствующее значение. В Linux возможен только один код ошибки — `EFAULT`; он означает, что указатель `buf` является недопустимым.

Установка текущего времени суток

В предыдущих разделах мы обсуждали, как получать значения времени. Но иногда в приложении приходится устанавливать для даты и времени определенное значение. Практически всегда для этого используется специальная процедура, созданная именно для такой цели, например `date`.

Аналог `time()` называется `stime()`:

```
#define _SVID_SOURCE
#include <time.h>

int stime (time_t *t);
```

При успешном вызове `stime()` системному времени присваивается значение, на которое указывает `t`, и функция возвращает 0. Вызов требует, чтобы у пользователя, который его выполняет, была возможность `CAP_SYS_TIME`. Как правило, такая возможность есть только у администратора.

При ошибке вызов возвращает -1 и присваивает `errno` значение `EFAULT` (если указатель `t` недопустим) или `EPERM`, если у вызывающего пользователя отсутствует возможность `CAP_SYS_TIME`.

Применять эту функцию очень просто:

```
time_t t = 1;
int ret;

/* устанавливаем время в значение «одна секунда с момента epoch» */
ret = stime (&t);
if (ret)
    perror ("stime");
```

В следующем разделе мы рассмотрим функции, которые упрощают преобразование человекочитаемых обозначений времени в формы, пригодные для использования в поле `time_t`.

Установка времени с заданной точностью

Парная функция для `gettimeofday()` называется `settimeofday()`:

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

int settimeofday (const struct timeval *tv,
                  const struct timezone *tz);
```

Успешный вызов `settimeofday()` присваивает времени значение, на которое указывает `tv`, после чего функция возвращает 0. Как и при работе с `gettimeofday()`, рекомендуется передавать `tz` значение `NULL`. При ошибке этот вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- EFAULT — `tv` или `tz` указывает на недопустимую область памяти;
- EINVAL — поле в одной из предоставленных структур содержит ошибку;
- EPERM — у вызывающего процесса отсутствует возможность `CAP_SYS_TIME`.

В следующем примере мы задаем для времени значение «суббота, середина декабря 1979 года»:

```
struct timeval tv = { .tv_sec = 31415926,  
                     .tv_usec = 27182818 };  
  
int ret;  
  
ret = settimeofday (&tv, NULL);  
if (ret)  
    perror ("settimeofday");
```

Продвинутый интерфейс для установки времени

Так же как функция `clock_gettime()` превосходит `gettimeofday()`, функция `clock_settime()` избавляет нас от необходимости пользоваться `settimeofday()`:

```
#include <time.h>  
  
int clock_settime (clockid_t clock_id,  
                  const struct timespec *ts);
```

В случае успеха этот вызов возвращает 0, а для источника, указанного как `clock_id`, устанавливается время, заданное в `ts`. При ошибке этот вызов возвращает -1 и присваивает `errno` одно из следующих значений:

- EFAULT — указатель `ts` недопустим;
- EINVAL — `clock_id` является в данной системе недопустимым источником времени;
- EPERM — процесс не обладает необходимыми правами доступа, чтобы установить указанный источник времени, либо указанный источник времени не может быть установлен.

В большинстве систем может быть установлен только один источник времени: `CLOCK_REALTIME`. Следовательно, единственное преимущество этой функции над `settimeofday()` заключается в том, что она обеспечивает разрешение с точностью до наносекунд (кроме того, нам не приходится возиться с бесполезной структурой `timezone`).

Эксперименты с временем

В системах UNIX и в языке C предоставляется семейство функций для преобразования разделенного времени (строковое представление времени в кодировке ASCII). Так, функция `time_tasctime()` преобразует структуру `tm` — разделенное время — в ASCII-строку:

```
#include <time.h>
```

```
char * asctime (const struct tm *tm);  
char * asctime_r (const struct tm *tm, char *buf);
```

Она возвращает указатель на статически выделенную строку. Последующий вызов любой функции, предназначенной для работы с временем, может перезаписать эту строку, поэтому `asctime()` небезопасна при работе с потоками.

Соответственно, для многопоточных программ (и для программистов, которые терпеть не могут плохо сработанные интерфейсы) рекомендуется использовать функцию `asctime_r()`. Она не возвращает указатель на статически выделенную строку, а использует строку, на которую направлен указатель `buf`. Эта строка должна содержать не менее 26 символов.

В случае ошибки обе функции возвращают `NULL`.

Вызов `mktime()` также преобразует структуру `tm`, но на этот раз в `time_t`:

```
#include <time.h>
```

```
time_t mktime (struct tm *tm);
```

При вызове `mktime()` мы также задаем временной пояс с помощью `tzset()` так, как указано в `tm`. При ошибке она возвращает `-1` (с приведением типа к `time_t`).

Вызов `ctime()` преобразует `time_t` в соответствующее ASCII-представление:

```
#include <time.h>
```

```
char * ctime (const time_t *timep);  
char * ctime_r (const time_t *timep, char *buf);
```

При ошибке функция возвращает `NULL`. Например:

```
time_t t = time (NULL);
```

```
printf ("время всего лишь одну строку назад: %s", ctime (&t));
```

Обратите внимание: здесь отсутствует переход на новую строку. Может быть, это и неудобно, но `ctime()` прикрепляет новую строку к своей возвращенной строке.

Как и `asctime()`, `ctime()` возвращает указатель на статическую строку. Поскольку эта функция небезопасна при использовании потоков, в многопоточных программах вместо нее требуется применять `ctime_r()`. Эта функция оперирует буфером, предоставленным в `buf`. Длина буфера должна составлять не менее 26 символов.

Функция `gmtime()` преобразует полученное значение `time_t` в структуру `tm`, выраженную в контексте временного пояса UTC:

```
#include <time.h>
```

```
struct tm * gmtime (const time_t *timep);  
struct tm * gmtime_r (const time_t *timep, struct tm *result);
```

При ошибке она возвращает `NULL`. Эта функция статически выделяет возвращенную структуру и, следовательно, небезопасна для потоков. В многопоточных программах необходимо использовать функцию `gmtime_r()`; она оперирует структурой, на которую указывает `result`.

Вызовы `localtime()` и `localtime_r()` функционально напоминают `gmtime()` и `gmtime_r()` соответственно, но они выражают полученный тип `time_t` в контексте часового пояса пользователя:

```
#include <time.h>
struct tm * localtime (const time_t *timep);
struct tm * localtime_r (const time_t *timep, struct tm *result);
```

Как и в случае с `mktime()`, вызов `localtime()` сопровождается вызовом `tzset()` и инициализацией часового пояса. Необходимость такого поведения для `localtime_r()` не указана.

Функция `difftime()` возвращает количество секунд, составляющих период между двумя значениями `time_t` (ее возвращаемое значение приводится к типу `double`):

```
#include <time.h>

double difftime (time_t time1, time_t time0);
```

Во всех системах POSIX `time_t` является арифметическим типом, а `difftime()` эквивалентна следующему коду (с оговоркой, что она не обнаруживает переполнения при вычитании):

```
(double) (time1 - time0)
```

Поскольку в Linux `time_t` является целочисленным типом, нет необходимости приводить его к `double`. Для обеспечения переносимости программы пользуйтесь функцией `difftime()`.

Настройка системных часов

Большие и внезапные скачки фактического времени могут ввергать в хаос такие приложения, чье нормальное функционирование завязано на абсолютных значениях времени. Рассмотрим, к примеру, команду `make`, которая выстраивает программные проекты в соответствии с подробным описанием, заданным в `Makefile`. При каждом вызове программы не происходит полной перестройки деревьев исходников. Если бы такая перестройка происходила, то при изменении единственного файла в больших программных проектах пересборка могла бы занимать до нескольких часов. Вместо этого `make` просматривает временные метки, находящиеся в файле исходников (допустим, `wolf.c`), соответствующие сделанным изменениям, и сравнивает их с объектным файлом (в данном случае — `wolf.o`). Если файл исходников или любой из сопровождающих его файлов, например `wolf.h`, окажется новее объектного файла, то `make` перестроит файл исходников в обновленный

объектный файл. Если файл исходников не окажется новее объектного файла, то не будут предприниматься никакие действия.

Теперь представим себе следующую ситуацию. Пользователь обнаруживает, что его часы отстали на пару часов, и запускает `date`, чтобы обновить системные часы. Если после этого пользователь обновит и пересохранит `wolf.c`, возникнут проблемы. Если пользователь отмотал текущее время назад, то файл `wolf.c` окажется системе старше, чем `wolf.o`, хотя это и не так. Пересборка не состоится.

Чтобы не допускать такого беспорядка, UNIX предоставляет функцию `adjtime()`, которая постепенно корректирует текущее время в направлении заданного перевода часов. Это делается для обеспечения правильности работы фоновых процессов, таких как демоны сетевого протокола времени (NTP), которые постоянно корректируют время с учетом рассинхронизации часов. Функция `adjtime()` призвана минимизировать негативное воздействие этих демонов на систему. Периодическая корректировка часов для преодоления рассинхронизации называется *подстройкой*:

```
#define _BSD_SOURCE
#include <sys/time.h>

int adjtime (const struct timeval *delta,
             struct timeval *olddelta);
```

При успешном вызове функции `adjtime()` мы приказываем ядру начать медленную корректировку времени в направлении, указанном в параметре `delta`, до полного выполнения этой корректировки. Если в `delta` указано отрицательное значение времени, то ядро замедляет ход системных часов до тех пор, пока не будет достигнута желаемая коррекция. Все изменения ядро применяет таким образом, что значение часов обязательно изменяется монотонно, резких скачков времени никогда не происходит. Даже при отрицательном показателе `delta` коррекция времени не связана с переводом «стрелок» назад. Просто системные часы будут «тикать» медленнее, пока компьютерное время не совпадет с фактическим.

Если значение `delta` не равно `NULL`, то ядро приостанавливает обработку всех ранее зарегистрированных корректировок. Однако если корректировки уже частично внесены, то эти изменения продолжают учитываться. Если значение `olddelta` не равно `NULL`, то все ранее зарегистрированные, но еще не примененные корректировки записываются в предоставленную структуру `timeval`. Передав значение `delta`, равное `NULL`, и допустимое значение `olddelta`, можно будет получить все текущие корректировки.

Все изменения, вносимые с помощью `adjtime()`, должны быть небольшими. Идеальный вариант использования функции — протокол NTP, который работает с незначительными изменениями (не превышающими пары секунд). В обоих направлениях Linux определяет максимальный и минимальный порог корректировки, оба этих порога достигают нескольких тысяч секунд.

При ошибке вызов `adjtime()` возвращает `-1` и присваивает `errno` одно из следующих значений:

- EFAULT — delta или olddelta является недопустимым указателем;
- EINVAL — изменение, указанное в delta, слишком мало или слишком велико;
- EPERM — вызывающий пользователь не обладает возможностью CAP_SYS_TIME.

Стандарт RFC 1305 определяет значительно более мощный и, соответственно, более сложный алгоритм корректировки часов, чем поступательный подход, применяемый adjtime(). Linux реализует этот алгоритм с помощью системного вызова adjtimex():

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

```
int adjtimex (struct timex *adj);
```

При вызове adjtimex() считываются параметры ядра, связанные с временем, после чего они записываются в структуру timex, на которую указывает adj. Возможен вариант (зависящий от значения поля modes в этой структуре), когда этот вызов может дополнительно задавать некоторые параметры.

Структура timex определяется в заголовке <sys/timex.h> следующим образом:

```
struct timex {
    int modes;           /* селектор режима */
    long offset;         /* смещение времени (микросекунд) */
    long freq;           /* смещение частоты */
                        /* (масштабированное значение ppm) */
    long maxerror;       /* максимальная ошибка (микросекунд) */
    long esterror;       /* расчетная ошибка (микросекунд) */
    int status;          /* состояние часов */
    long constant;       /* временная константа PLL */
    long precision;      /* точность часов (микросекунд) */
    long tolerance;      /* допустимая погрешность частоты часов (ppm) */
    struct timeval time; /* текущее время */
    long tick;           /* количество микросекунд между тактами часов */
};
```

Поле modes содержит побитовое «ИЛИ» из нуля или более следующих флагов:

- ADJ_OFFSET — задает смещение времени с помощью offset;
- ADJ_FREQUENCY — устанавливает используемую частоту с помощью freq;
- ADJ_MAXERROR — задает максимальную ошибку с помощью maxerror;
- ADJ_ESTERROR — определяет расчетную ошибку с помощью esterror;
- ADJ_STATUS — задает состояние часов с помощью status;
- ADJ_TIMECONST — устанавливает временную константу PLL (фазовая автоподстройка частоты) с помощью constant;
- ADJ_TICK — задает значение такта с помощью tick;
- ADJ_OFFSET_SINGLESHOT — с помощью offset однократно задает смещение времени при применении простого алгоритма, например adjtime().

Если значение поля `modes` равно 0, то никакие значения не устанавливаются. Ненулевое значение поля `modes` может предоставить лишь такой пользователь, который обладает характеристикой `CAP_SYS_TIME`. Любой другой пользователь может задать для поля значение 0, получив, таким образом, все параметры, но не имея возможности установить ни один из них.

При успехе вызов `adjtimex()` возвращает текущее состояние часов, которое может соответствовать одному из следующих значений:

- `TIME_OK` — часы синхронизированы;
- `TIME_INS` — будет вставлена корректировочная секунда;
- `TIME_DEL` — будет удалена корректировочная секунда;
- `TIME_OOP` — идет корректировочная секунда;
- `TIME_WAIT` — корректировочная секунда только что сработала;
- `TIME_BAD` — часы не синхронизированы.

При ошибке вызов `adjtimex()` возвращает -1 и присваивает `errno` одно из следующих значений ошибки:

- `EFAULT` — аргумент `adj` содержит недопустимый указатель;
- `EINVAL` — одно из полей `modes`, `offset` либо `tick` содержит недопустимое значение;
- `EPERM` — значение `modes` является ненулевым, но вызывающий пользователь не обладает характеристикой `CAP_SYS_TIME`.

Системный вызов `adjtimex()` является специфичным для Linux. Если приложение должно быть переносимым, то в данном случае следует использовать функцию `adjtime()`.

Стандарт RFC 1305 определяет довольно сложный алгоритм, поэтому подробное обсуждение функции `adjtimex()` выходит за рамки этой книги. Более развернуто о ней можно почитать в документации по RFC.

Засыпание и ожидание

Различные функции позволяют процессу засыпать (приостанавливать работу) на определенный период времени. Первая из таких функций, `sleep()`, переводит указанный процесс в спящий режим на такое количество секунд, которое дано в `seconds`:

```
#include <unistd.h>
```

```
unsigned int sleep (unsigned int seconds);
```

Вызов возвращает количество секунд, которые были проведены в *неспящем* режиме. Следовательно, в случае успеха вызов возвращает 0, но эта функция может возвращать и иные значения в диапазоне от 0 до `seconds` включительно (например, если сигнал прерывает состояние ожидания). Эта функция не устанавливает значения `errno`. Для большинства потребителей `sleep()` неважно, как

долго процесс находился в спящем режиме. Поэтому возвращаемое значение также не проверяется.

```
sleep (7); /* засыпание на семь секунд */
```

Если необходимо, чтобы процесс действительно «проспал» весь указанный период, то вы можете продолжать вызывать `sleep()` с ее возвращаемым значением, пока эта функция не вернет 0:

```
unsigned int s = 5;
```

```
/* спим пять секунд; именно пять и точка! */  
while ((s = sleep (s)))  
    ;
```

Засыпание с микросекундной точностью

Если система засыпает на период, состоящий из целого количества секунд, нас это не устроит. В современных системах секунда — это практически вечность, поэтому нередко в программах требуется большая временная детализация спящего режима. Познакомьтесь с функцией `usleep()`:

```
/* Версия из BSD */  
#include <unistd.h>
```

```
void usleep (unsigned long usec);
```

```
/* Версия из SUSv2 */  
#define _XOPEN_SOURCE 500  
#include <unistd.h>
```

```
int usleep (useconds_t usec);
```

При успешном вызове `usleep()` мы приказываем процессу заснуть на `usec` микросекунд. К сожалению, стандарты BSD и Single UNIX Specification (SUS) по-разному определяют прототип этой функции. Функция из BSD получает `unsigned long` и не имеет возвращаемого значения. В варианте для SUS функция `usleep()` принимает тип `useconds_t` и возвращает `int`. Linux следует спецификации SUS, если `_XOPEN_SOURCE` имеет значение 500 или выше. Если значение `_XOPEN_SOURCE` не определено или менее 500, то Linux следует BSD.

Версия из SUS возвращает 0 при успехе и -1 при ошибке. Допустимыми значениями `errno` являются `EINTR`, если период сна был прерван сигналом, или `EINVAL`, если значение `usecs` было слишком велико (в Linux допустимым считается весь диапазон значений этого типа, поэтому такая ошибка никогда не возникает).

Согласно спецификации тип `useconds_t` является беззнаковым целым числом и может содержать значения вплоть до 1 000 000.

Из-за различий между конфликтующими вариантами прототипов, а также в силу того, что некоторые системы UNIX могут поддерживать либо один, либо другой вариант, разумно вообще явно не включать тип `useconds_t` в ваш код. Для обеспечения

максимальной переносимости следует действовать так, как будто этот параметр является `unsigned int`, и не полагаться на возвращаемое значение функции `usleep()`:

```
void usleep (unsigned int usec);
```

Тогда она будет использоваться так:

```
unsigned int usecs = 200;
```

```
usleep (usecs);
```

Так мы можем работать с обоими вариантами функции, а также проверять наличие ошибок:

```
errno = 0;
usleep (1000);
if (errno)
    perror ("usleep");
```

Но в большинстве программ такая проверка не выполняется, а ошибки `usleep()` вообще игнорируются.

Засыпание с наносекундной точностью

В Linux на смену функции `usleep()` приходит `nanosleep()`, обеспечивающая разрешение с точностью до наносекунд и более интеллектуальный интерфейс:

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep (const struct timespec *req,
               struct timespec *rem);
```

При успешном вызове функция `nanosleep()` переводит вызывающий процесс в спящий режим на период, указанный в `req`, а потом возвращает 0. При ошибке функция возвращает -1 и присваивает `errno` соответствующее значение. Если сон процесса прерывается из-за сигнала, то вызов может вернуться прежде, чем истечет заданное время. В таком случае функция `nanosleep()` возвращает -1 и присваивает `errno` значение `EINTR`. Если `rem` не равно `NULL`, то функция записывает остаток длительности спящего режима (тот фрагмент `req`, который функция еще должна «проспать») в `rem`. Затем программа может повторить вызов, передав `rem` в качестве `req` (как показано ниже в этом разделе).

Вот другие возможные в данном случае значения `errno`:

- `EFAULT` — `req` или `rem` является недопустимым указателем;
- `EINVAL` — одно из полей в `req` является недопустимым.

Как правило, работать с этой функцией очень просто:

```
struct timespec req = { .tv_sec = 0,
                       .tv_nsec = 200 };

/* засыпаем на 200 нс */
```

```
ret = nanosleep (&req, NULL);
if (ret)
    perror ("nanosleep");
```

А вот пример использования второго параметра для продолжения работы, если сон был прерван:

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 1369 };
struct timespec rem;
int ret;

/* засыпаем на 1369 нс */
retry:
ret = nanosleep (&req, &rem);
if (ret) {
    if (errno == EINTR) {
        /* повторяем попытку, учитывая оставшийся период времени */
        req.tv_sec = rem.tv_sec;
        req.tv_nsec = rem.tv_nsec;
        goto retry;
    }
    perror ("nanosleep");
}
```

Наконец, далее приведен альтернативный подход достижения той же цели (пожалуй, более эффективный, но менее удобочитаемый):

```
struct timespec req = { .tv_sec = 1,
                        .tv_nsec = 0 };
struct timespec rem, *a = &req, *b = &rem;

/* засыпаем на 1 с */
while (nanosleep (a, b) && errno == EINTR) {
    struct timespec *tmp = a;
    a = b;
    b = tmp;
}
```

Функция `nanosleep()` обладает несколькими преимуществами по сравнению с `sleep()` и `usleep()`:

- разрешение с точностью до наносекунд, а не с точностью до микросекунд или секунд;
- стандартизирована в POSIX.1b;
- реализация не связана с использованием сигналов (недостатки чего рассмотрены ниже).

Несмотря на устаревание `usleep()`, во многих программах эта функция предпочитается `nanosleep()`. Поскольку функция `nanosleep()` стандартизирована в POSIX и при работе не использует сигналы, новые программы должны применять ее (либо интерфейс, рассмотренный в следующем разделе) вместо `sleep()` и `usleep()`.

Продвинутая работа со спящим режимом

Как и в случаях со всеми другими классами функций времени, рассмотренными выше, в семействе часов POSIX имеется и очень продвинутый интерфейс для работы со спящим режимом:

```
#include <time.h>

int clock_nanosleep (clockid_t clock_id,
                    int flags,
                    const struct timespec *req,
                    struct timespec *rem);
```

Функция `clock_nanosleep()` работает аналогично `nanosleep()`. На самом деле следующий вызов:

```
ret = nanosleep (&req, &rem);
```

делает то же самое, что и этот:

```
ret = clock_nanosleep (CLOCK_REALTIME, 0, &req, &rem);
```

Разница заключается в параметрах `clock_id` и `flags`. Первый указывает эталонный источник времени. Большинство источников времени подойдут в таком качестве, хотя вы и не сможете задать процессорные часы вызывающего процесса (например, `CLOCK_PROCESS_CPUTIME_ID`). Это было бы попросту бессмысленно, так как вызов приостанавливает выполнение процесса и значение процессорного времени перестает расти.

Выбор эталонного источника времени зависит от того, с какой целью ваша программа переходит в спящий режим. Если она должна «проспать» до какого-то абсолютного момента во времени, то разумнее всего выбрать `CLOCK_REALTIME`. Если спящий режим должен продолжаться относительно количество времени, то идеальным вариантом станет `CLOCK_MONOTONIC`.

Параметр `flags` может иметь значение `TIMER_ABSTIME` или 0. При флаге `TIMER_ABSTIME` то значение, которое указано в `req`, считается абсолютным, а не относительным. Так исключается потенциальное возникновение условий гонки. Чтобы пояснить значение этого параметра, предположим, что во время $T+0$ процесс собирается «проспать» до времени $T+1$. В момент $T+0$ процесс вызывает функцию `clock_gettime()`, чтобы узнать текущее время ($T+0$). Затем он вычитает $T+0$ из $T+1$, получая значение Y . Это значение он передает функции `clock_nanosleep()`. Но какое-то время пройдет между моментом, когда было определено время, и моментом, когда процесс заснет. Хуже, если процесс был деактивизирован процессором, обратился к несуществующей странице или произошло нечто подобное. Таким образом, всегда существует возможность возникновения условий гонки между получением текущего времени, расчетом временной разницы и самим моментом засыпания.

Флаг `TIMER_ABSTIME` сводит к нулю вероятность условий гонки, позволяя процессу прямо указать $T+1$. Ядро приостанавливает процесс до тех пор, пока в указанном

источнике времени не наступит момент $T+1$. Если время в указанном источнике уже превышает значение $T+1$, то вызов сразу же возвращается.

Рассмотрим относительное и абсолютное засыпание. В следующем примере спящий режим включается на 1,5 секунды:

```
struct timespec ts = { .tv_sec = 1, .tv_nsec = 500000000 };
int ret;
```

```
ret = clock_nanosleep (CLOCK_MONOTONIC, 0, &ts, NULL);
if (ret)
    perror ("clock_nanosleep");
```

Напротив, в следующем примере период сна продолжается до абсолютного момента во времени. Этот момент наступает ровно через одну секунду после момента времени, возвращенного функцией `clock_gettime()` из источника `CLOCK_MONOTONIC`:

```
struct timespec ts;
int ret;

/* процесс собирается проспать одну секунду с НАСТОЯЩЕГО МОМЕНТА */
ret = clock_gettime (CLOCK_MONOTONIC, &ts);
if (ret) {
    perror ("clock_gettime");
    return;
}

ts.tv_sec += 1;
printf ("Мы хотим заснуть до секунды=%ld наносекунды=%ld\n",
        ts.tv_sec, ts.tv_nsec);
ret = clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME,
&ts, NULL);

if (ret)
    perror ("clock_nanosleep");
```

Большинству программ требуется засыпать лишь на относительный период времени, так как их требования к засыпанию не очень строгие. Но некоторые процессы реального времени предъявляют очень точные требования к хронометражу и должны спать в течение абсолютного периода времени, чтобы исключить возникновения условий гонки, чреватых катастрофическими последствиями.

Переносимый способ засыпания

Вспомним функцию `select()`, знакомую нам из гл. 2:

```
#include <sys/select.h>
int select (int n,
            fd_set *readfds,
            fd_set *writefds,
```



```
fd_set *exceptfds,  
struct timeval *timeout);
```

Как было указано в той главе, функция `select()` обеспечивает переносимый способ перехода в спящий режим с детализацией менее одной секунды. В течение долгого времени для управления спящим режимом в переносимых программах UNIX можно было пользоваться только функцией `sleep()`. Функция `usleep()` поддерживалась далеко не везде, а `nanosleep()` еще даже не была написана. Разработчики открыли эффективный и удобный при переносимости способ перевода процессов в спящий режим: достаточно передать функции `select()` значения 0 для `n`, NULL для всех трех указателей `fd_set` и желаемую длительность сна для `timeout`.

```
struct timeval tv = { .tv_sec = 0,  
                     .tv_usec = 757 };
```

```
/* засыпание на 757 мс */  
select (0, NULL, NULL, NULL, &tv);
```

Если программа должна переноситься на сравнительно старые UNIX-системы, то лучше ограничиться использованием функции `select()`.

Превышение пределов

Все интерфейсы, рассмотренные в этой главе, гарантируют, что спящий режим продлится *как минимум столько, сколько было запрошено* (либо будет возвращена ошибка, свидетельствующая об обратном). Такая функция никогда не возвращается успешно, прежде чем истечет заданная задержка. Но вполне возможно, что задержка продлится *дольше*, чем было запрошено.

Такой феномен объясняется лишь принципом работы планировщика — возможно, запрошенное на спящий режим время уже истекло, и ядро в нужный момент «разбудило» процесс, но планировщик решил запустить какую-то другую задачу.

Но существуют и другие случаи, называемые *превышением пределов таймера*. Такая ситуация может возникать, если таймер обеспечивает недостаточную степень дискретизации и интервал между его «тикаем» превышает запрошенный интервал времени. Допустим, системный таймер «тикает» с интервалом 10 миллисекунд, а процесс запрашивает переход в спящий режим на 1 миллисекунду. Система способна измерять время и реагировать на события, связанные с временем (например, будить процесс), но с интервалом не менее 10 миллисекунд. Если процесс выдает вышеупомянутый запрос на переход в спящий режим, а таймеру остается 1 миллисекунда до следующего такта, то все будет хорошо — 1 миллисекунда истечет и ядро разбудит процесс. Если же запрос на переход в спящий режим поступает от процесса именно в тот момент, когда таймер отбивает очередной такт, то следующий такт наступит только через 10 миллисекунд. Соответственно, процессу придется проспать лишние 9 миллисекунд! Произойдет девять актов превышения предела, каждый продлится по 1 микросекунде. В среднем таймер с периодом X теряет на таких превышениях $X / 2$ времени.

Чтобы минимизировать превышение таймера, следует использовать высокоточные источники времени — например, те, что предоставляются часами POSIX. Кроме того, целесообразно устанавливать высокие значения HZ.

Альтернативы засыпанию

По возможности следует избегать перехода в спящий режим. Иногда этого сделать не удастся, но в засыпании нет ничего страшного — особенно если процесс будет спать не дольше секунды. Но код, наштабированный периодами сна ради «активного ожидания», как правило, является некачественным. Более подходящим будет код, блокируемый в файловом дескрипторе и позволяющий ядру обработать спящий режим, а потом разбудить процесс. Чтобы процесс не крутился в цикле, дожидаясь прихода очередного события, можно просто заблокировать его выполнение и разбудить процесс, лишь когда он действительно будет нужен.

Таймеры

Таймеры предоставляют механизм для уведомления процесса о том, когда истечет заданный период времени. Период времени, проходящий до *истечения* таймера, также называется *задержкой*. Конкретный способ, которым ядро уведомляет процесс об истечении таймера, зависит от таймера. В ядре Linux предлагается несколько типов таймеров, и мы рассмотрим их все.

Таймеры полезны во многих ситуациях. Например, с их помощью легко обновлять экран по 60 раз в секунду или отменять ожидающую транзакцию, если она пробудет в очереди на протяжении более 500 миллисекунд.

Простые варианты сигнализации

Функция `alarm()` — простейший интерфейс для работы с таймером:

```
#include <unistd.h>
```

```
unsigned int alarm (unsigned int seconds);
```

Когда вы вызываете эту функцию, доставка сигнала SIGALRM вызывающему процессу планируется на тот момент, когда истечет `seconds` секунд реального времени. Если ранее назначенный сигнал находился в состоянии ожидания, то вызов отменяет сигнал тревоги, заменяет его новым таким запрошенным сигналом и возвращает количество секунд, не истраченных предыдущим «будильником». Если значение `seconds` равно 0, то любой предыдущий сигнал тревоги отменяется (при его наличии), но новый тревожный сигнал не назначается.

Таким образом, для успешного использования этой функции также требуется зарегистрировать обработчик для сигнала SIGALRM. О сигналах и обработчиках сигналов мы подробно говорили в предыдущей главе. Ниже приведен фрагмент

кода с функцией `alarm_handler()`. Здесь регистрируется обработчик сигнала `SIGALRM`, длительность тревожного сигнала устанавливается равной 5 секундам:

```
void alarm_handler (int signum)
{
    printf ("Пять секунд прошло!\n");
}

void func (void)
{
    signal (SIGALRM, alarm_handler);
    alarm (5);

    pause ();
}
```

Интервальные таймеры

Системные вызовы *интервальных таймеров*, которые впервые появились в 4.2BSD, с тех пор были стандартизированы в POSIX. Они обеспечивают более полный контроль над временем, чем `alarm()`:

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

int getitimer (int which,
               struct itimerval *value);

int setitimer (int which,
               const struct itimerval *value,
               struct itimerval *ovalue);
```

Интервальные таймеры работают подобно `alarm()`, но опционально их можно настроить так, чтобы они автоматически сбрасывались и могли работать в одном из следующих режимов.

- `ITIMER_REAL` — измеряет фактическое время. Когда истечет указанный период фактического времени, ядро отправляет процессу сигнал `SIGALRM`.
- `ITIMER_VIRTUAL` — значение таймера уменьшается только в то время, когда процесс выполняет код из пользовательского пространства. Когда указанное количество процессорного времени истечет, ядро отправляет процессу сигнал `SIGVTALRM`.
- `ITIMER_PROF` — значение уменьшается и в ходе выполнения процесса, и в ходе работ ядра по поручению процесса (например, при завершении системного вызова). Когда истечет указанное количество времени, ядро отправит процессу сигнал `SIGPROF`. Обычно этот режим используется в сочетании с `ITIMER_VIRTUAL`, чтобы программа могла измерять пользовательское и процессное время, израсходованное процессом.
- `ITIMER_REAL` — измеряет то же время, что и `alarm()`; два других режима полезны при профилировании.

В структуре `itimerval` пользователь может задавать количество времени, которое должно пройти до истечения таймера. Кроме того, здесь при необходимости указывается новый срок окончания, с которым в следующий раз должен запускаться таймер:

```
struct itimerval {
    struct timeval it_interval; /* следующее значение */
    struct timeval it_value;    /* актуальное значение */
};
```

Выше мы говорили о структуре `timeval`, которая обеспечивает разрешение с точностью до микросекунд:

```
struct timeval {
    long tv_sec;    /* секунды */
    long tv_usec;   /* микросекунды */
};
```

Функция `setitimer()` настраивает таймер типа `which`, срок истечения которого указывается в поле `it_value`. Как только истечет время, определенное в `it_value`, ядро заново запустит таймер со значением, заданным в `it_interval`. Следовательно, `it_value` — это время, остающееся до истечения актуального таймера. Как только значение `it_value` достигает нуля, оно становится равным `it_interval`. Если срок таймера истечет, а `it_interval` будет равно 0, то таймер не перезапустится. Аналогично если значение `it_value` активного таймера равно 0, то таймер останавливается и не перезапускается.

Если `ovalue` не равно `NULL`, то возвращаются предыдущие значения интервального таймера типа `which`.

Функция `getitimer()` возвращает актуальные значения интервального таймера типа `which`.

Обе функции возвращают 0 при успехе и -1 при ошибке, причем в последнем случае `errno` присваивается одно из следующих значений:

- `EFAULT` — `value` или `ovalue` является недопустимым указателем;
- `EINVAL` — `which` не соответствует допустимому типу интервального таймера.

В следующем фрагменте кода создается обработчик сигнала `SIGALRM` (см. гл. 10), после чего интервальный таймер запускается с первоначальным периодом истечения, равным 5 секундам, и с последующим интервалом истечения, равным 1 секунде:

```
void alarm_handler (int signo)
{
    printf ("Таймер сработал!\n");
}

void foo (void)
{
    struct itimerval delay;
    int ret;
```

```
signal (SIGALRM, alarm_handler);

delay.it_value.tv_sec = 5;
delay.it_value.tv_usec = 0;
delay.it_interval.tv_sec = 1;
delay.it_interval.tv_usec = 0;
ret = setitimer (ITIMER_REAL, &delay, NULL);
if (ret) {
    perror ("setitimer");
    return;
}

pause ();
}
```

В некоторых системах UNIX с помощью сигнала SIGALRM реализуются вызовы `sleep()` и `usleep()`. Функции `alarm()` и `setitimer()` также используют SIGALRM. Поэтому программист должен внимательно следить за тем, чтобы вызовы этих функций не перекрывали друг друга, — результат такого сочетания не определен. При необходимости краткого ожидания программисту следует пользоваться функцией `nanosleep()`, которая, согласно стандарту POSIX, не применяет сигналов. При работе с таймерами следует использовать `setitimer()` или `alarm()`.

Функции для расширенной работы с таймерами

Самый мощный интерфейс для работы с таймерами относится к семейству часов POSIX.

При работе с таймерами, работа которых основана на использовании часов POSIX, акты инстанцирования, инициализации и, наконец, удаления таймера распределены на три самостоятельные функции: `timer_create()` создает таймер, `timer_settime()` инициализирует таймер, а `timer_delete()` уничтожает его.

ПРИМЕЧАНИЕ

Семейство часов POSIX для работы с интерфейсами таймеров — несомненно, самое продвинутое, но в то же время и самое новое (соответственно, наименее удобное при портировании) и самое сложное в использовании. Если ваши основные приоритеты — простота и переносимость программы, то лучше всего остановиться на функции `setitimer()`.

Создание таймера

Чтобы создать таймер, воспользуйтесь функцией `timer_create()`:

```
#include <signal.h>
#include <time.h>

int timer_create (clockid_t clockid,
                 struct sigevent *evp,
                 timer_t *timerid);
```

При успешном вызове `timer_create()` создается новый таймер, ассоциированный с часами POSIX `clockid`, в `timerid` сохраняется уникальный идентификатор времени, после чего функция возвращает 0. Этот вызов просто создает нужные условия для запуска таймера. До запуска таймера фактически ничего не происходит, в чем вы убедитесь, прочитав следующий раздел.

В приведенном далее фрагменте кода создается новый таймер, связанный с часами POSIX `CLOCK_PROCESS_CPUTIME_ID`. В `timer` сохраняется ID таймера:

```
timer_t timer;
int ret;

ret = timer_create (CLOCK_PROCESS_CPUTIME_ID,
                   NULL,
                   &timer);
if (ret)
    perror ("timer_create");
```

При ошибке вызов возвращает -1, значение `timerid` остается неопределенным, а `errno` присваивается одно из следующих значений:

- `EAGAIN` — система не обладает достаточным количеством ресурсов для завершения запроса;
- `EINVAL` — часы POSIX, указанные в `clockid`, являются недопустимыми;
- `ENOTSUP` — часы POSIX, указанные в `clockid`, являются допустимыми, но система не поддерживает использования этих часов с таймерами. Стандарт POSIX гарантирует, что в любых реализациях при работе с таймерами будут поддерживаться часы `CLOCK_REALTIME`. Поддержка других часов зависит от конкретной реализации.

Если значение параметра `evp` не равно `NULL`, то он определяет асинхронное уведомление, происходящее по истечении срока таймера. Структура задается в заголовке `<signal.h>`. Предполагается, что ее содержимое должно быть скрытым от программиста, но в ней определяются как минимум следующие поля:

```
#include <signal.h>

struct sigevent {
    union sigval sigev_value;
    int sigev_signo;
    int sigev_notify;
    void (*sigev_notify_function)(union sigval);
    pthread_attr_t *sigev_notify_attributes;
};

union sigval {
    int sival_int;
    void *sival_ptr;
};
```

Таймеры, работающие на базе часов POSIX, обеспечивают более полный контроль над тем, как именно ядро уведомляет процесс об истечении времени таймера.

Процесс может точно указывать сигнал, который будет испускаться ядром, даже само ядро может породить поток и выполнить функцию в ответ на истечение таймера. Процесс задает поведение, наступающее при истечении времени таймера, в поле `sigev_notify`, которое должно иметь одно из следующих значений:

- `SIGEV_NONE` — «нулевое» уведомление. При истечении таймера ничего не происходит;
- `SIGEV_SIGNAL` — по истечении таймера ядро направляет процессу сигнал, указанный в `sigev_signo`. В обработчике сигналов аргументу `si_value` присваивается значение `sigev_value`;
- `SIGEV_THREAD` — при истечении таймера ядро порождает новый поток (внутри данного процесса) и приказывает ему выполнить `sigev_notify_function`, передавая `sigev_value` в качестве единственного аргумента. Поток завершается, как только возвращается от этой функции. Если `sigev_notify_attributes` не равно `NULL`, то предоставленная структура `pthread_attr_t` определяет поведение нового потока.

Если `evp` равно `NULL`, как в одном из наших предыдущих примеров, то уведомление об истечении таймера устанавливается, как если бы `sigev_notify` было равно `SIGEV_SIGNAL`, `sigev_signo` — `SIGALRM`, а `sigev_value` — ID таймера. Таким образом, по умолчанию эти таймеры работают во многом так же, как и интервальные таймеры POSIX. Но стоит их немного настроить, и их возможности значительно расширятся!

В следующем примере создается таймер, работающий на основе часов `CLOCK_REALTIME`. Как только срок этого таймера истечет, ядро выдаст сигнал `SIGUSR1`, а в качестве значения `si_value` установит адрес, по которому хранится ID таймера:

```
struct sigevent evp;
timer_t timer;
int ret;

evp.sigev_value.sival_ptr = &timer;
evp.sigev_notify = SIGEV_SIGNAL;
evp.sigev_signo = SIGUSR1;
ret = timer_create (CLOCK_REALTIME,
                   &evp,
                   &timer);

if (ret)
    perror ("timer_create");
```

Установка таймера

Таймер, создаваемый с помощью `timer_create()`, еще не установлен ни на какое конкретное время. Чтобы ассоциировать его со сроком истечения и запустить часы, используйте `timer_settime()`:

```
#include <time.h>

int timer_settime (timer_t timerid,
                  int flags,
```

```
const struct itimerspec *value,
struct itimerspec *ovalue);
```

При успешном вызове `timer_settime()` начинает работать таймер, указанный с помощью `timerid`, со сроком истечения `value`, представляющий собой структуру `itimerspec`:

```
struct itimerspec {
    struct timespec it_interval; /* следующее значение */
    struct timespec it_value;    /* актуальное значение */
};
```

Как и в случае с `setitimer()`, `it_value` указывает актуальный срок истечения таймера. Когда таймер доходит до конца своего интервала, поле `it_value` обновляется значением из поля `it_interval`. Если `it_interval` равно 0, то таймер не является интервальным, поэтому прекращает работать, как только закончится срок `it_value`.

Выше мы обсуждали структуру `timespec`, обеспечивающую разрешение с точностью до наносекунд:

```
struct timespec {
    time_t tv_sec; /* секунды */
    long tv_nsec; /* наносекунды */
};
```

Если параметр `flags` равен `TIMER_ABSTIME`, то время, указанное в `value`, интерпретируется как абсолютное (напротив, по умолчанию это значение отсчитывается относительно текущего времени). Такое модифицированное поведение позволяет избежать условий гонки на этапах получения текущего времени, расчета относительной разницы между этим значением времени и целевым моментом в будущем. Подробнее этот вопрос обсуждается в подразд. «Продвинутая работа со спящим режимом» разд. «Засыпание и ожидание» этой главы.

Если значение `ovalue` не равно `NULL`, то предыдущий срок истечения таймера сохраняется в `itimerspec`. Если ранее таймер был отключен, то всем членам структуры присваивается 0.

С помощью значения `timer`, ранее инициализированного в `timer_create()`, в следующем примере создается периодический таймер, срок которого истекает раз в секунду:

```
struct itimerspec ts;
int ret;

ts.it_interval.tv_sec = 1;
ts.it_interval.tv_nsec = 0;
ts.it_value.tv_sec = 1;
ts.it_value.tv_nsec = 0;
ret = timer_settime (timer, 0, &ts, NULL);
if (ret)
    perror ("timer_settime");
```


Получение времени истечения таймера

Можно узнать срок истечения таймера, не сбрасывая его с помощью `timer_gettime()`:

```
#include <time.h>
```

```
int timer_gettime (timer_t timerid,  
                  struct itimerspec *value);
```

Успешный вызов `timer_gettime()` сохраняет время истечения того таймера, который указан с помощью аргумента `timerid`, в структуре, на которую указывает `value`, после чего функция возвращает 0. При ошибке вызов возвращает -1 и присваивает переменной `errno` одно из следующих значений:

- EFAULT — `value` является недопустимым указателем;
- EINVAL — `timerid` является недопустимым таймером.

Например:

```
struct itimerspec ts;  
int ret;  
  
ret = timer_gettime (timer, &ts);  
if (ret)  
    perror ("timer_gettime");  
else {  
    printf ("текущее время истечения: секунды=%ld наносекунды=%ld\n",  
           ts.it_value.tv_sec, ts.it_value.tv_nsec);  
    printf ("следующее время истечения: секунды=%ld наносекунды=%ld\n",  
           ts.it_interval.tv_sec, ts.it_interval.tv_nsec);  
}
```

Получение превышения пределов таймера

Стандарт POSIX содержит интерфейс для определения количества превышений пределов, возникших у конкретного таймера, — если такие превышения имели место:

```
#include <time.h>
```

```
int timer_getoverrun (timer_t timerid);
```

В случае успеха функция `timer_getoverrun()` возвращает количество дополнительных фактов истечения таймера, которые были зафиксированы между исходным истечением таймера и отправкой уведомления о таком истечении процессу — допустим, с помощью сигнала. Например, в одном из наших предыдущих примеров, где односекундный таймер успел проработать 10 миллисекунд, этот вызов вернул бы 9.

Согласно POSIX, если количество выходов за пределы равно или превышает `DELAYTIMER_MAX`, то вызов возвращает `DELAYTIMER_MAX`. К сожалению, Linux не реализует такое поведение. Вместо этого, как только количество выходов за пределы

у определенного таймера превышает `DELAYTIMER_MAX`, он возвращается к нулю и начинает работать заново.

При ошибке эта функция возвращает `-1` и присваивает `errno` значение `EINVAL`. Это единственно возможное условие ошибки, указывающее, что таймер, заданный в `timerid`, является недопустимым.

Например:

```
int ret;

ret = timer_getoverrun (timer);
if (ret == -1)
    perror ("timer_getoverrun");
else if (ret == 0)
    printf ("превышений не было\n");
else
    printf ("%d превышений\n", ret);
```

Удаление таймера

Удалить таймер просто:

```
#include <time.h>

int timer_delete (timer_t timerid);
```

Успешный вызов функции `timer_delete()` уничтожает таймер, ассоциированный с `timerid`, и возвращает `0`. При ошибке вызов возвращает `-1` и присваивает `errno` значение `EINVAL`. Это единственно возможное условие ошибки, указывающее, что таймер, заданный в `timerid`, является недопустимым.

Приложение А

Расширения GCC для языка C

В наборе компиляторов GNU (GNU Compiler Connection, GCC) содержится множество расширений для языка C, многие из которых зарекомендовали себя как очень ценные инструменты системных программистов. Большинство дополнений к языку C, которые мы рассмотрим в этом приложении, сообщают компилятору дополнительную информацию о поведении и предполагаемом назначении их кода. Компилятор, в свою очередь, использует эту информацию, чтобы генерировать более эффективный машинный код. Другие расширения заполняют пробелы, существующие в языке C, — особенно на низком уровне.

GCC предоставляет несколько расширений, в настоящее время входящих в новейший стандарт языка C, ISO C11. Одни из этих расширений работают аналогично соответствующим функциям из C11, но другие расширения реализованы в ISO C11 существенно иначе. Новый код должен представлять собой стандартизированные разновидности этих возможностей. Мы не будем рассматривать здесь такие расширения, а коснемся лишь тех, которые встречаются только в GCC.

GNU C

Вариант языка C, поддерживаемый GCC, часто именуется GNU C. В 1990-е годы GNU C заполнил множество пробелов, существовавших в языке C, и предоставил такие компоненты, как сложные переменные, массивы нулевой длины, встраиваемые функции и именованные инициализаторы. Но спустя примерно десятилетие язык C наконец был усовершенствован, а после появления стандартов ISO C99 и ISO C11 расширения GCC стали менее востребованы. Тем не менее GNU C по-прежнему содержит немало полезных возможностей, а многие программисты, пишущие на Linux, продолжают использовать поднабор GNU C — иногда всего по одному-два расширения — в своем коде, соответствующем стандартам C99 или C11.

Одна из наиболее известных баз кода, основанных на GCC, — это само ядро Linux, написанное только на GNU C. Правда, не так давно компания Intel приступила к разработкам, которые позволили бы компилятору Intel C Compiler (ICC) понимать расширения GNU C, используемые ядром. Следовательно, многие из этих расширений сегодня утрачивают свою GCC-специфичность.

Встраиваемые функции

Компилятор копирует весь код *встраиваемой* функции в то место, где эта функция вызывается. Вместо того чтобы хранить функцию внешне и переходить к ней всякий раз при ее вызове, компилятор выполняет содержимое такой функции безотрывно от остального кода. Благодаря такому поведению удастся избавиться от издержек, связанных с вызовом функции. Кроме того, открываются потенциальные возможности для оптимизаций в точке вызова, так как компилятор может одновременно корректировать работу и вызывающей и вызываемой стороны. Последнее особенно актуально, если параметры, передаваемые функции на стороне вызова, постоянны. Однако при копировании функции во все мельчайшие фрагменты кода, где она вызывается, код может увеличиться до совершенно недопустимых размеров. Следовательно, функции следует встраивать лишь в тех случаях, когда они маленькие и простые либо вызываются в ограниченном количестве мест.

На протяжении многих лет GCC поддерживал ключевое слово `inline`, приказывающее компилятору встроить данную функцию. В C99 это ключевое слово было формализовано:

```
static inline int foo (void) { /* ... */ }
```

Однако на практике оно представляет собой не более чем подсказку — напоминание компилятору о том, чтобы он рассмотрел возможность встраивания данной функции. GCC дополнительно предоставляет расширение, приказывающее компилятору *всегда* встраивать указанную функцию:

```
static inline __attribute__((always_inline)) int foo (void) { /* ... */ }
```

Наиболее очевидные кандидаты на роль встраиваемых функций — препроцессорные макрокоманды. Встраиваемая функция в GCC будет вести себя точно так же, как и макрокоманда, а также получать проверку типов. Например, вместо такой макрокоманды

```
#define max(a,b) ({ a > b ? a : b; })
```

можно использовать следующую встраиваемую функцию:

```
static inline max (int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

Зачастую программисты чрезмерно увлекаются встраиваемыми функциями. Нагрузка, связанная с вызовом функций в современных архитектурах — в частности, в x86, — очень и очень мала. Встраивайте функцию, лишь если она действительно стоит этого!

Подавление встраивания

Когда GCC работает в наиболее агрессивном режиме оптимизации, он автоматически выбирает подходящие функции и встраивает их. Как правило, такой подход себя оправдывает, но иногда программист точно знает, что функция будет работать неправильно, если ее встроить. В частности, это происходит при использовании `__builtin_return_address` (рассматривается далее в этом приложении). Для подавления встраивания применяйте ключевое слово `noinline`:

```
__attribute__((noinline)) int foo (void) { /* ... */ }
```

Чистые функции

Чистой называется функция, которая не оказывает побочных эффектов и чье возвращаемое значение отражает только параметры функции или неизменяемые глобальные переменные. Любой доступ к параметрам или глобальным переменным должен быть открыт только для чтения. С такими функциями может применяться оптимизация циклов и удаление подвыражений. Функции помечаются как чистые с помощью ключевого слова `pure`:

```
__attribute__((pure)) int foo (int val) { /* ... */ }
```

Классический пример такой функции — `strlen()`. При наличии идентичных образцов ввода возвращаемое значение этой функции инвариантно для множества вызовов. Поэтому ее можно вычленить из цикла и вызвать всего однажды. Рассмотрим следующий код:

```
/* символ за символом записываем каждую букву в 'p' в верхнем регистре */  
for (i = 0; i < strlen (p); i++)  
    printf ("%c", toupper (p[i]));
```

Если компилятору неизвестно, что функция `strlen()` является чистой, то ему придется вызывать ее в каждой итерации цикла.

Толковый программист — а также компилятор, знающий, что функция `strlen()` помечена как чистая, — напишет (сгенерирует) такой код:

```
size_t len;  
  
len = strlen (p);  
for (i = 0; i < len; i++)  
    printf ("%c", toupper (p[i]));
```

Между прочим, особо умные программисты (такие, как читатели этой книги) напишут так:

```
while (*p)  
    printf ("%c", toupper (*p++));
```

Это недопустимо. Действительно, для чистой функции нет никакого смысла возвращать `void`, так как возвращаемое значение является единственной сутью таких функций. Пример функции, которую нельзя назвать чистой, — `random()`.

Постоянные функции

Постоянная функция — ужесточенный вариант чистой. Такие функции не могут обращаться к глобальным переменным и принимать указатели в качестве параметров. Следовательно, возвращаемое значение постоянной функции не отражает ничего, кроме параметров, переданных по значению. С этими функциями могут выполняться дополнительные виды оптимизаций — кроме тех, что применимы с чистыми функциями. К постоянным относятся, например, такие чистые функции, как `abs()` (предполагается, что они не сохраняют состояния и не выкидывают других фокусов ради оптимизации). Программист помечает функцию как постоянную с помощью ключевого слова `const`:

```
__attribute__((const)) int foo (int val) { /* ... */ }
```

Как и в случае с чистыми функциями, нет смысла, чтобы постоянная функция возвращала значение типа `void`.

Невозвращаемые функции

Если функция не возвращает результат — например, потому, что она инвариантно вызывает `exit()`, — то программист может пометить ее ключевым словом `noreturn`, подчеркнув этот факт для компилятора:

```
__attribute__((noreturn)) void foo (int val) { /* ... */ }
```

В свою очередь, компилятор может выполнить дополнительные виды оптимизации, учитывая, что вызванная функция, сопровождаемая таким словом, ни при каких обстоятельствах не вернется. Для такой функции целесообразен только один вариант типа возвращаемого значения — `void`.

Функции, выделяющие память

Если функция возвращает указатель, который никогда не сможет ссылаться на уже существующую память¹, — что практически гарантируется тем фактом, что

¹ Конфликты при доступе к памяти возникают в тех случаях, когда один или несколько указателей направлены на один и тот же фрагмент памяти. Это может происходить в тривиальных случаях, когда один указатель присваивается значению другого, а также в более сложных и менее очевидных случаях. Если функция возвращает адрес только что выделенного фрагмента памяти, то не должно существовать никаких других указателей на этот фрагмент.

функция сама для себя выделяет свежий фрагмент памяти и возвращает указатель на него, — то программист может пометить такую функцию ключевым словом `malloc`, а компилятор, в свою очередь, выполнит соответствующую оптимизацию:

```
__attribute__((malloc)) void * get_page (void)
{
    int page_size;

    page_size = getpagesize ();
    if (page_size <= 0)
        return NULL;

    return malloc (page_size);
}
```

Принудительная проверка возвращаемого значения вызывающей стороной

Атрибут `warn_unused_result` используется не столько для оптимизации, сколько для помощи программисту. Он приказывает компилятору генерировать предупреждение каждый раз, когда возвращаемое значение функции не сохраняется или используется в условном операторе:

```
__attribute__((warn_unused_result)) int foo (void) { /* ... */ }
```

Таким образом программист может гарантировать, что все вызывающие компоненты проверяют и обрабатывают возвращаемое значение функции, если оно особенно важно. Функции с важными, но зачастую игнорируемыми возвращаемыми значениями — например, `read()` — отлично подходят для использования с этим атрибутом. Такие функции не могут возвращать `void`.

Как пометить функцию как устаревшую

Атрибут `deprecated` приказывает компилятору сгенерировать предупреждение в точке вызова в момент вызова такой функции:

```
__attribute__((deprecated)) void foo (void) { /* ... */ }
```

Это помогает программистам отвыкать от использования устаревающих и выходящих из употребления функций.

Как пометить функцию как используемую

Иногда случается, что та или иная функция вызывается кодом, который невидим для компилятора. Помечая ее атрибутом `used`, мы сообщаем компилятору, что программа использует данную функцию, хотя и складывается впечатление, что на эту программу нет никаких ссылок:

```
static __attribute__ ((used)) void foo (void) { /* ... */ }
```

Следовательно, компилятор выводит результирующий язык сборки и не выдает предупреждения о неиспользуемой функции. Этот атрибут полезен, если статическая функция вызывается только из написанного вручную кода ассемблера. Как правило, если компилятору неизвестно о вызовах, он генерирует предупреждение и пытается избавиться от такой функции в целях оптимизации.

Как пометить функции или параметры как неиспользуемые

Атрибут `unused` сообщает компилятору, что данная функция или параметр функции не используются, и приказывает компилятору не выдавать соответствующих предупреждений:

```
int foo (long __attribute__ ((unused)) value) { /* ... */ }
```

Это полезно, если компиляция происходит с параметрами `-W` или `-Wunused` и вы хотите перехватывать неиспользуемые параметры функций, но по тем или иным причинам вам приходится работать с функциями, которые должны иметь жестко заданную сигнатуру (такие случаи распространены при событийно-ориентированном программировании графических пользовательских интерфейсов либо в обработчиках сигналов).

Упаковка структуры

Атрибут `packed` сообщает компилятору, что тип или переменная должны быть упакованы в памяти максимально компактно. При этом зачастую нарушаются требования к выравниванию. Если атрибут указан для `struct` или `union`, то все переменные в этой структуре или объединении упаковываются. Если атрибут задан для конкретной переменной, то упаковывается только этот объект.

Следующий код упаковывает все переменные в структуре в минимальное пространство:

```
struct __attribute__ ((packed)) foo { ... };
```

Например, если структура содержит поле типа `char`, за которым следует поле типа `int`, то целое число, скорее всего, будет выровнено в памяти не по адресу, сразу же следующему за символьным значением, а, скажем, на три байта дальше. Компилятор выравнивает переменные, просто заполняя пустующие байты между ними. В упакованной структуре такое заполнение отсутствует, в результате чего эта структура потенциально потребляет меньше памяти, но не выполняет требований к выравниванию, действующих на уровне архитектуры.

Увеличение границы выравнивания переменной

GCC не только допускает упаковку переменных, но и позволяет программисту указывать альтернативное минимальное выравнивание для заданной переменной. На основании этой информации GCC выровняет переменную *как минимум* по указанному количеству байтов, а не по минимальному значению выравнивания, предписываемому архитектурой и интерфейсом ABI. Например, в следующем операторе мы объявляем для целого числа с именем `beard_length` минимальное значение выравнивания 32 байт (а не типичное выравнивание 4 байт, действующее на машинах с 32-битными целыми числами):

```
int beard_length __attribute__((aligned (32))) = 0;
```

Принудительное определение границы выравнивания для типа, как правило, имеет смысл только при работе с аппаратным обеспечением, которое может предъявлять более высокие требования к выравниванию, чем сама архитектура, либо когда вы вручную смешиваете код на C и ассемблерный код или используете инструкции, требующие специфического выравнивания. Один из примеров, где применяется такая функциональность выравнивания, — хранение часто используемых переменных в процессорных строках данных для оптимизации работы кэша. Такая техника применяется, например, в ядре Linux.

Если не указывать определенное минимальное выравнивание, то можно приказывать GCC выровнять заданный тип по максимальному значению из всех значений минимального выравнивания, используемых с какими-либо типами данных. Например, следующий код приказывает GCC выровнять таким образом `parrot_height` — вероятно, будет применяться выравнивание `double`:

```
short parrot_height __attribute__((aligned)) = 5;
```

Обычно такое решение подразумевает компромисс при выборе между пространством и временем: выровненные таким образом переменные занимают больше места, но копирование из них или в них, а также другие сложные манипуляции могут протекать быстрее, так как компилятор может выдавать машинные инструкции, работающие с максимальным заданным объемом памяти.

Различные аспекты архитектуры или системного инструментария могут ограничивать максимальное значение выравнивания переменной. Например, в некоторых архитектурах Linux линковщик не может распознавать значения выравнивания, превышающих достаточно небольшое значение, заданное по умолчанию. В таком случае значение выравнивания, задаваемое таким ключевым словом, округляется вниз до минимального допустимого выравнивания. Например, если вы запрашиваете выравнивание 32, но системный линковщик не допускает выравнивания более чем по 8 байт, то переменная будет выровнена по 8-байтной границе.

Помещение глобальных переменных в регистр

GCC позволяет программисту размещать глобальные переменные в специальном машинном регистре, где переменные затем будут находиться на протяжении всей длительности работы программы. В GCC такие переменные называются *глобальными регистровыми переменными*.

В соответствии с синтаксисом программист обязан указать машинный регистр. В следующем примере используется регистр `ebx`:

```
register int *foo asm ("ebx");
```

Программист должен выбрать переменную, которая не затирается функциями: то есть выбранная переменная должна быть пригодна для использования локальными функциями, сохраняться и восстанавливаться при вызове функции и не выделяться для каких бы то ни было специальных целей на уровне ABI-интерфейсов данной архитектуры или операционной системы. Если выбран неподходящий регистр, то компилятор сгенерирует предупреждение. Если регистр подходит, например `ebx`, использованный в данном примере, полностью удовлетворяет требованиям архитектуры x86, то компилятор, в свою очередь, прекратит применять этот регистр.

Такая оптимизация позволяет радикально повысить производительность, если переменная часто используется. Хороший пример — работа с виртуальными машинами. Если поместить в регистр переменную, которая, скажем, содержит указатель фрейма виртуального стека, то можно значительно выиграть в производительности. С другой стороны, если архитектура с самого начала испытывает дефицит (а в архитектуре x86 ситуация именно такова), то подобная оптимизация нецелесообразна.

Глобальные регистровые переменные нельзя использовать в обработчиках сигналов или сразу в нескольких потоках исполнения. Кроме того, они не могут содержать исходных значений, поскольку отсутствует механизм, с помощью которого исполняемые файлы могли бы передавать в регистры содержимое, задаваемое по умолчанию. Объявления глобальных регистровых переменных должны предшествовать любым определениям функций.

Аннотирование ветвей

GCC позволяет программисту аннотировать ожидаемое значение выражения — например, сообщать компилятору: должно быть условное выражение истинным или ложным. GCC, в свою очередь, может выполнять переупорядочение блоков и другие виды оптимизации, позволяющие повысить производительность условных ветвлений.

Синтаксис GCC для записи ветвления некрасив до безобразия. Чтобы повысить удобочитаемость аннотирования ветвей, мы используем препроцессорные макросы:

```
#define likely(x)    __builtin_expect (!! (x), 1)
#define unlikely(x) __builtin_expect (!! (x), 0)
```

Программист может аннотировать выражение как вероятно истинное или вероятно ложное, обернув его в функцию `likely()` или `unlikely()` соответственно.

В следующем примере ветвь аннотируется как вероятно ложная:

```
int ret;

ret = close (fd);
if (unlikely (ret))
    perror ("close");
```

А в этом примере ветвь, наоборот, обозначается как вероятно истинная:

```
const char *home;

home = getenv ("HOME");
if (likely (home))
    printf ("Ваш домашний каталог — %s\n", home);
else
    fprintf (stderr, "Переменная среды HOME не установлена!\n");
```

Как и в случае со встраиваемыми функциями, программисты зачастую злоупотребляют аннотированием ветвей. Как только вы начинаете это практиковать, возникает соблазн пометить *все* выражения. Но будьте осторожны — такие аннотации выражений допустимы лишь в случаях, когда вы *априори* уверены и практически не сомневаетесь, что данное выражение будет истинным или ложным практически всегда (например, в 99 % случаев). Скажем, функцию `unlikely()` удобно использовать с редко встречающимися ошибками. Не забывайте, что неверный прогноз хуже, чем отсутствие какого-либо прогноза.

Получение типа выражения

GCC предоставляет ключевое слово `typeof()` для получения типа заданного выражения. Семантически это ключевое слово действует точно так же, как и `sizeof()`. Например, следующее выражение возвращает любой тип, на который указывает `x`:

```
typeof (*x)
```

Мы можем воспользоваться выражением для объявления массива `y`, состоящего из этих типов:

```
typeof (*x) y[42];
```

Популярный вариант использования `typeof()` — написание «безопасных» макросов, способных оперировать любыми арифметическими выражениями и интерпретировать их параметры только один раз:

```
#define max(a,b) ({ \
    typeof (a) _a = (a); \
    typeof (b) _b = (b); \
    _a > _b ? _a : _b; \
})
```

Получение границы выравнивания типа

GCC предоставляет ключевое слово `alignof`, позволяющее определить выравнивание заданного объекта. Это значение зависит от конкретной архитектуры и ABI. Если в актуальной архитектуре не обеспечивается требуемое выравнивание, то ключевое слово возвращает выравнивание, рекомендуемое в данном ABI. В противном случае ключевое слово возвращает минимальное требуемое выравнивание.

Синтаксис идентичен синтаксису `sizeof()`:

```
__alignof__(int)
```

В зависимости от архитектуры данная функция, вероятно, вернет 4, так как 32-битные целые числа обычно выравниваются по 4-байтным границам.

ПРИМЕЧАНИЕ

В C11 и C++11 появилась функция `alignof()`, которая действует идентично `alignof()`, но при этом стандартизирована. При написании программ на C11 или C++11 следует отдавать предпочтение `alignof()`.

Это ключевое слово работает и с адресами переменных (`lvalue`). В данном случае возвращается минимальное выравнивание базового типа, а не актуальное выравнивание конкретного адреса переменной. Если минимальное выравнивание было изменено с помощью атрибута `aligned` (описано выше, в разд. «Увеличение границы выравнивания переменной» этого приложения), то это изменение отражается ключевым словом `__alignof__`.

Например, рассмотрим следующую структуру:

```
struct ship {
    int year_built;
    char cannons;
    int mast_height;
};
```

а также этот фрагмент кода:

```
struct ship my_ship;

printf ("%d\n", __alignof__(my_ship.cannons));
```

Ключевое слово `alignof` для этого фрагмента кода вернет 1, хотя из-за забивки структуры возможна ситуация, в которой на `cannons` потратится 4 байт.

Смещение члена внутри структуры

GCC предоставляет встроенное ключевое слово для определения смещения члена структуры внутри структуры. Макрос `offsetof()`, определенный в `<stddef.h>`, входит в состав стандарта ISO C. Большинство определений просто ужасны, они содержат непристойную арифметику над указателями и не рекомендуются несовершенным читателям. Расширение из GCC проще, а также потенциально быстрее:

```
#define offsetof(type, member) __builtin_offsetof (type, member)
```

Вызов возвращает смещение члена `member` внутри типа, то есть количество байтов от нуля (с начала этой структуры) до данного члена. Рассмотрим, например, следующую структуру:

```
struct rowboat {  
    char *boat_name;  
    unsigned int nr_oars;  
    short length;  
};
```

Точные значения смещений зависят от размера переменных, требований, предъявляемых к выравниванию в данной архитектуре, а также от принципа заполнения. На 32-битной машине мы вполне можем встретить вызов `offsetof()` в структуре `rowboat` с полями `boat_name`, `nr_oars` и `length`, чьи возвращаемые значения будут равны 0, 4 и 8 соответственно.

В системе Linux макрос `offsetof()` должен определяться с применением ключевого слова из GCC. Повторно определять его не требуется.

Получение возвращаемого адреса функции

GCC предоставляет ключевое слово для получения возвращаемого адреса актуальной функции либо одного из «вызывателей» этой функции:

```
void * __builtin_return_address (unsigned int level)
```

Параметр `level` указывает в цепочке вызовов ту функцию, адрес которой должен быть возвращен. Значение 0 требует возвращаемый адрес актуальной функции, значение 1 — адрес той функции, которая вызвала актуальную, значение 2 — функцию, вызвавшую ту, которой соответствует значение 1, и т. д.

Если актуальная функция является встроенной, то возвращается адрес вызывающей функции. Если это неприемлемо, то пользуйтесь ключевым словом `noinline`. Так вы принудите компилятор не встраивать функции.

Существует несколько способов применения ключевого слова `__builtin_return_address`. В частности, оно используется в отладочных и справочных целях.

Другой пример практического использования — раскручивание цепочки вызовов, требуемое для реализации интроспекции, утилиты для аварийного дампа, отладчика и т. д.

Обратите внимание: некоторые архитектуры позволяют возвращать только адрес вызывающей функции. В таких архитектурах ненулевое значение параметра может привести к выводу случайного возвращаемого значения. Следовательно, любые ненулевые значения параметров не допускают портирования и должны использоваться только в отладочных целях.

Диапазоны оператора case

GCC позволяет указывать в метках оператора case диапазон значений для отдельно взятого блока. Синтаксис таков:

case low ... high:

Например:

```
switch (val) {
    case 1 ... 10:
        /* ... */
        break;
    case 11 ... 20:
        /* ... */
        break;
    default:
        /* ... */
}
```

Эта функциональность очень полезна при работе с диапазоном значений ASCII:

case 'A' ... 'Z':

Обратите внимание, что до и после многоточия должны быть пробелы. В противном случае компилятор может «запутаться», особенно в целочисленных диапазонах. Всегда делайте так:

case 4 ... 8:

а так никогда не делайте:

case 4...8:

Арифметика указателей типа void и указателей на функции

В GCC операции сложения и вычитания допускаются над указателями типа void и над указателями на функции. Как правило, стандарт ISO C не позволяет выполнять

арифметические операции над такими указателями, поскольку размер значения типа `void` — это фактически нонсенс, он зависит от того, на какую именно сущность направлен указатель. Для обеспечения подобной арифметики GCC принимает размер такого адресуемого объекта за один байт. Следующий фрагмент кода увеличивает значение `a` на единицу:

```
a++; /* a — это void-указатель */
```

Параметр `-Wpointer-arith` заставляет GCC генерировать предупреждение, когда используются такие расширения.

Более переносимо и красиво

Согласитесь, синтаксис `attribute` очень некрасивый. Некоторые расширения, рассмотренные в этом приложении, требуют использования препроцессорных макросов — только так работа с ними становится терпимой. Но любые расширения лишь выиграют, если немного улучшить их внешний вид.

Немного препроцессорных уловок — и все получается как по волшебству. Более того, выполняя такую работу, мы также обеспечиваем портируемость GCC-расширений, определяя их таким образом, что они будут понятны любому компилятору (а не только GCC).

Для этого включите следующий фрагмент кода в заголовок и добавьте этот заголовок в ваши файлы исходников:

```
#if __GNUC__ >= 3
# undef inline
# define inline inline __attribute__((always_inline))
# define __noinline __attribute__((noinline))
# define __pure __attribute__((pure))
# define __const __attribute__((const))
# define __noreturn __attribute__((noreturn))
# define __malloc __attribute__((malloc))
# define __must_check __attribute__((warn_unused_result))
# define __deprecated __attribute__((deprecated))
# define __used __attribute__((used))
# define __unused __attribute__((unused))
# define __packed __attribute__((packed))
# define __align(x) __attribute__((aligned (x)))
# define __align_max __attribute__((aligned))
# define likely(x) __builtin_expect (!!(x), 1)
# define unlikely(x) __builtin_expect (!!(x), 0)
#else
# define __noinline /* не noinline */
# define __pure /* не pure */
# define __const /* не const */
# define __noreturn /* не noreturn */
# define __malloc /* не malloc */
# define __must_check /* не warn_unused_result */
#endif
```

```
# define __deprecated /* не deprecated */
# define __used /* не used */
# define __unused /* не unused */
# define __packed /* не packed */
# define __align(x) /* не aligned */
# define __align_max /* не align_max */
# define likely(x) (x)
# define unlikely(x) (x)
#endif
```

Например, в следующем фрагменте мы помечаем эту функцию как чистую, пользуясь одним из вышеперечисленных сокращений:

```
__pure int foo (void) { /* ... */ }
```

Если применяется GCC, то эта функция помечается атрибутом `pure`. Если мы работаем не с GCC, а с другим компилятором, то препроцессор однозначно требует метку `__pure`. Обратите внимание: в каждом определении вы можете указывать несколько атрибутов и соответственно одновременно применять несколько из вышеперечисленных комбинаций.

Проще, красивее, портabelнее!

Приложение Б

Библиография

В этом приложении перечислена рекомендуемая литература, имеющая отношение к системному программированию. Все приведенные источники разбиты на четыре подкатегории. Ни одна из этих работ не является обязательной к прочтению. Это просто моя подборка книг, которые кажутся мне наиболее ценными материалами по той или иной теме. Если вы желаете ознакомиться с дополнительной информацией, расширяющей материал моей книги, то именно эти книги я бы вам посоветовал.

Некоторые работы из данной подборки посвящены темам, в которых вы уже должны хорошо ориентироваться, — например, языку С. Другие тексты значительно дополняют книгу, подробно рассказывая, в частности, о gdb, Git и о проектировании операционных систем. Разумеется, данный список неполный — наверняка вы найдете и другие, не менее ценные ресурсы.

Книги по языку программирования С

В книгах из этого раздела рассматривается язык С — лингва франка¹ системного программирования. Если вы не умеете писать код на С столь же непринужденно, как говорить на родном языке, то одна или несколько следующих книг (а также большая практическая работа) помогут вам улучшить знания этого языка. Особо рекомендую изучить первую книгу, иногда сокращенно именуемую *K & R*. Ее краткость красноречиво показывает, насколько прост язык С.

- Керниган Б., Ритчи Д. Язык программирования Си (The C programming language). — 2-е изд. — М.: Вильямс, 2007.

¹ Лингва франка — смесь арабского, турецкого и различных романских языков, сложившаяся в Средние века на берегах Средиземного моря; использовалась в основном в купеческой среде для межнационального делового общения. — *Примеч. пер.*

Эта книга написана одним из авторов языка C и его соратником. Представляет собой наиболее фундаментальную работу по языку C.

- *Prinz Peter, Crawford Tony. C in a Nutshell.* — O'Reilly Media, 2005.

Отличная книга, рассказывающая как о языке C, так и о стандартной библиотеке C.

- *Prinz Peter, Kirch-Prinz Ulla. C Pocket Reference.* — O'Reilly Media, 2002.

Краткий справочник по языку C, обновленный с учетом ANSI C99.

- *Van der Linden Peter. Expert C Programming.* — Prentice Hall, 1994.

Прекрасная книга о сравнительно малоизвестных аспектах языка C, написанная в остроумном стиле и одобренная авторским юмором.

- *Summit Steve. C Programming FAQs: Frequently Asked Questions.* — 2nd edition. — Addison-Wesley, 1995.

Эта книга — настоящий шедевр. Она содержит более 400 часто задаваемых вопросов (с ответами) по языку программирования C. Ответы на многие из вопросов покажутся банальными знатокам языка C, но некоторые глубокие вопросы и ответы впечатлят даже самых эрудированных C-программистов. У этой книги есть и интернет-версия, которая обновляется гораздо оперативнее, чем бумажная.

Книги по программированию в Linux

Следующие тексты посвящены программированию в Linux. В них рассматриваются некоторые темы, не затронутые в этой книге, а также многие интересные инструменты для программирования.

- *Стивенс У., Феннер Б., Рудолфф Э. UNIX: Разработка сетевых приложений.* — 3-е изд. — СПб.: Питер, 2004.

Исчерпывающая работа, посвященная API сокетов. К сожалению, книга рассказывает не только о Linux, однако радуется, что она была обновлена с учетом стандарта IPv6.

- *Стивенс У. UNIX: взаимодействие процессов.* — СПб.: Питер, 2001.

Отличная книга, посвященная межпроцессной коммуникации (IPC).

- *Nichols Bradford. PThreads Programming.* — O'Reilly Media, 1996.

Обзор интерфейса прикладного программирования потоков в POSIX.

- *Mecklenburg Robert. Managing Projects with GNU Make.* — O'Reilly Media, 2004.

В этой книге отлично рассмотрен GNU Make — классический инструмент для выстраивания программных проектов в Linux.

- *Collins-Sussman Ben. Version Control with Subversion.* — O'Reilly Media, 2004.

Исчерпывающая работа о Subversion — системе контроля версий, пришедшей на смену CVS. В UNIX эта система применяется как для контроля версий,

так и для управления исходным кодом. Книга написана тремя авторами Subversion.

- *Loeliger Jon*. Version Control with Git. — O'Reilly Media, 2012.

Великолепное введение в работу с Git — иногда немного запутанной, но неизменно мощной распределенной системой контроля версий.

- *Robbins Arnold*. GDB Pocket Reference. — O'Reilly Media, 2005.

Удобный карманный справочник по gdb — отладчику для Linux.

- *Siever Ellen*. Linux in a Nutshell. — O'Reilly Media, 2009.

Потрясающий справочник по всем функциям Linux, в частности по многим из тех инструментов, из которых состоит среда разработки Linux.

Книги, посвященные ядру Linux

Две книги, упомянутые в этом разделе, рассказывают о ядре Linux. Существует три основные причины, по которым следует подробнее познакомиться с этой темой. Во-первых, ядро предоставляет интерфейс для взаимодействия между системными вызовами и программами из пользовательского пространства, поэтому оно имеет ключевое значение для системного программирования. Во-вторых, книга проливает свет на различные особенности поведения и странности, возникающие при взаимодействии ядра с приложением. Наконец, ядро Linux — образец отличного кода, и эти книги просто интересны.

- *Лав Р.* Разработка ядра Linux. — 2-е изд. — М.: Вильямс, 2006.

Это моя работа, относящаяся к данной категории. Она отлично подойдет системным программистам, желающим разобраться в конструкции и реализации ядра Linux. Эта книга не является справочником по API. В ней подробно обсуждаются алгоритмы, применяемые в ядре Linux, рассматриваются принципы принятия решений в ядре.

- *Corbet Jonathan*. Linux Device Drivers. — O'Reilly Media, 2005.

Отличное руководство по программированию драйверов устройств для ядра Linux с великолепным справочником по API. Хотя центральной темой этой книги являются драйверы устройств, материал будет интересен программистам различных специализаций, в частности системным программистам, желающим лучше понять принципы функционирования ядра Linux. Прекрасное дополнение к моей книге по ядру Linux.

Книги об организации операционных систем

Две упомянутые здесь книги касаются не только Linux — они описывают общие принципы организации операционных систем. Как я не раз указывал в этой книге,

глубокое понимание операционной системы, в которой вы программируете, значительно повышает качество создаваемого кода.

- *Silberschatz Abraham*. Operating System Concepts. — Prentice Hall, 2012.

Отличное введение в тему операционных систем, их историю и базовые алгоритмы. В книге содержится хороший набор ситуативных исследований.

- *Schimmel Curt*. UNIX Systems for Modern Architectures. — Addison-Wesley, 1994.

Эта книга рассказывает не столько об UNIX, сколько о современных процессорных и кэш-архитектурах, является отличным введением в тему взаимодействия операционных систем и аппаратного обеспечения. Хотя книга немного устарела, я настоятельно ее рекомендую.

Р. Лав

Linux. Системное программирование

2-е издание

Перевел с английского О. Сивченко

Заведующий редакцией	Д. Веницкий
Ведущий редактор	Н. Гринчик
Литературный редактор	В. Конаш
Художник	Л. Адуевская
Корректоры	Т. Курьянович, Е. Павлович
Верстка	А. Барцевич

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 27.03.14. Формат 70×100/16. Усл. п. л. 36,120. Тираж 1500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.



Нет времени ходить по магазинам?



наберите:



www.piter.com



Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD



**И наконец, вы нигде не купите
наши книги дешевле!**





КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.



ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным
направлениям: история и публицистика, экономика и финансы, менеджмент
и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com


УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com


Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых
партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

 Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

 Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
