Linux

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

ПИТЕР®

Краткое содержание

Оглавление

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 внутристрочной.
- **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 поведениям, 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 изобретательность, 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 вверху, 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.

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 удается 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 ISOC++ 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 внутристрочные 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 *ISOC++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 A.

Linux не может похвастаться большими достижениями в области обеспечения прямой совместимости[1], 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 9[1]. 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 удается 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 локальности[1].

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 каналы[1] 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 `peg`, то приложению потребуется считать из файла-устройства клавиатуры сначала `p`, потом `e` и, наконец, `g` — именно в таком порядке. Когда больше не остается символов, которые необходимо прочитать, 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 дискете 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 монтирована 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 каталогов[1]. 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 неперемещаемые 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
ECHILD	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 возвратить 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!

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 сокеты. 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`. Задает 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 дает 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 монтирования[1]. 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 забывайте.

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 «e». Кен Томпсон (Ken Thompson), автор UNIX, както 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 Linux[1] `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). Обрабатывать 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 прочитать 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 EIO) 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 последует 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 промежуток 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 вызвать 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:

- EBADF — 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 диске[1].

[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 получат 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.

Закрытие 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 инициирует 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 нуль:

```
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_SETSEEK_CUR или SEEK_END либо результирующая файловая позиция получится отрицательной. Факт, что EINVAL может соответствовать обеим 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 буквы `p`, за исключением того, что они игнорируют текущую позицию файла. Вместо использования ее 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 содержимым:

```
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 блокированный 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 блокированной 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 оказывается 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 дожидаются событий определенного типа. Файловые дескрипторы, перечисленные в `readfds`, отслеживают, не 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 дожидаются 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 считывания[1].

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;

        /* Дожидаемся 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 компенсируют 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 `nfds 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;
- `POLLNVAL` — 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,
           nfds_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 заполнение[1]. 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 приемов[1] 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 удается. Например, Linux может без проблем взаимодействовать с файловыми системами FAT и NTFS.

Преимуществ 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 *урезается*, 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.

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 однобайтовых 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 удается 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 ANSIC (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 *потоком 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 закроется 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 */
```

Закрытие 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.

Закрытие 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 малораспространенные. 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 возвратить символ обратно, если окажется, что он вам не подходит: