THE UNIX PROGRAMMING ENVIRONMENT

Brian W. Kernighan, Rob Pike

PRENTICE HALL

HIGH TECH

UNIX

ПРОГРАММНОЕ ОКРУЖЕНИЕ

Брайан Керниган, Роб Пайк

Санкт-Петербург — Москва
2003

Брайан Керниган, Роб Пайк

UNIX. Программное окружение

Перевод П. Шера

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Козлихин*
Редактор	*В. Овчинников*
Художник	*В. Гренда*
Корректура	*С. Беляева*
Верстка	*Н. Гриценко*

Керниган Б., Пайк Р.
UNIX. Программное окружение. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 416 с., ил.
ISBN 5-93286-029-4

Книга представляет собой введение в программное окружение UNIX и адресована тем, кто хочет научиться программировать с помощью всех тех инструментов, которые поставляются с операционной системой. Рассматривается вход в систему, работа с файлами (cat, mv, cp, rm) и каталогами (cd, mkdir, ...), основы окружения (переменные, маски), фильтры (grep, sed, awk), программирование оболочки (циклы, сигналы, аргументы, стандартный ввод-вывод), введение в системные вызовы (read, write, open, creat, ...), введение в программирование с использованием lex, yacc и make, работа с документацией с помощью troff, tbl и eqn. Приводимые примеры не придуманы специально для этой книги, – некоторые из них впоследствии стали частью комплекта программ, используемых каждый день. Программы написаны на Си. Предполагается, что читатель знает или хотя бы изучает этот язык.

Прочтение этой книги как новичками, так и опытными пользователями поможет понять, как сделать работу с системой эффективной и приносящей удовольствие.

ISBN 5-93286-029-4
ISBN 0-13-937681-X (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 20.03.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 26 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

«Количество инсталляций UNIX возросло до 10, а ожидается еще больший рост».

(Справочное руководство по системе UNIX, 2-е издание, июнь 1972 года.)

Операционная система UNIX[1] стартовала на неиспользовавшейся DEC PDP-7 в Bell Laboratories в 1969 году. Кен Томпсон (Ken Thompson) при помощи и поддержке Руда Канадея (Rudd Canaday), Дага Мак-Илроя (Doug McIlroy), Джо Оссанны (Joe Ossanna) и Денниса Ритчи (Dennis Ritchie) написал небольшую универсальную систему с разделением времени, достаточно удобную для того, чтобы привлечь пользователей, и в конечном счете создавшую достаточный кредит доверия для покупки более мощной машины – PDP-11/20. Одним из первых пользователей стал Ритчи, который помогал перенести систему на PDP-11 в 1970 году. Кроме того, Ритчи спроектировал и написал компилятор для языка программирования Си. В 1973 году Ритчи и Томпсон переписали ядро UNIX на Си, прервав таким образом традицию написания системного программного обеспечения на языке ассемблера. И после этой переделки система, по существу, стала тем, чем она и является сегодня.

Примерно в 1974 году система была разрешена для использования в университетах «в учебных целях», а через несколько лет стали доступны ее коммерческие версии. В это время системы UNIX процветали в Bell Laboratories – они проникли в лаборатории, проекты по разработке программного обеспечения, центры обработки текстов и системы поддержки операций в телефонных компаниях. С того времени UNIX распространилась по всему миру – десятки тысяч систем установлено на различное оборудование, от микрокомпьютеров до самых крупных мэйнфреймов.

Почему UNIX имела такой успех? Можно привести несколько причин. Во-первых, благодаря тому что она написана на языке Си, она перено-

---

[1] UNIX – это торговая марка Bell Laboratories. «UNIX» – это *не* акроним, это намек на MULTICS, операционную систему, над которой Томпсон и Ритчи работали до UNIX.

сима – UNIX-системы работают на всевозможных компьютерах, от микропроцессоров до мэйнфреймов; это важное коммерческое преимущество. Во-вторых, исходный код доступен и написан на языке высокого уровня, что делает простым адаптирование системы для специфических требований пользователей. Наконец, и это самое важное, UNIX – *хорошая* операционная система, особенно для программистов. Среда программирования UNIX поразительно богата и продуктивна.

Хотя UNIX и представляет ряд новаторских программ и технологий, действенность системы не определяется какой-то одной программой или концепцией. Эффективность UNIX определяется применением особого подхода к программированию, философией использования компьютера. Для того чтобы описать эту философию, одного предложения, конечно же, недостаточно, но вот ее основная идея – мощность системы в огромной степени определяется взаимодействием программ, а не их наличием. Многие UNIX-программы по отдельности выполняют довольно тривиальные задачи, но будучи объединенными с другими, образуют полный и полезный инструментарий.

Цель этой книги в том, чтобы представить философию программирования в UNIX. Основа этой философии – взаимодействие программ, поэтому, хотя основное место отводится обсуждению отдельных инструментов, на протяжении всей книги обсуждается и тема комбинирования программ, а также их использования для построения других программ. Чтобы правильно работать с системой UNIX и ее компонентами, надо не только понимать, как применять программы, но и знать, как они взаимодействуют с окружением.

По мере распространения UNIX становилось все меньше тех, кто квалифицированно использовал бы ее приложения. Нередко случалось так, что опытным пользователям, в том числе и авторам этой книги, удавалось найти только «топорное» решение задачи, или же они писали программы для выполнения заданий, которые без проблем обрабатывались уже существующими средствами. Конечно же, чтобы найти красивое решение, необходимы опыт и понимание системы. Хотелось бы надеяться, что прочтение этой книги поможет как новичкам, так и «бывалым» пользователям понять, как сделать работу с системой наиболее эффективной и приносящей удовольствие. Используйте систему UNIX правильно!

Мы адресуем книгу программистам-одиночкам в надежде, что, сделав их труд более продуктивным, мы тем самым сделаем более эффективной и групповую работу. Хотя книга предназначена в основном для специалистов, содержание первых четырех или даже пяти глав можно понять, не имея опыта программирования, так что они могут иметь ценность и для других пользователей.

Везде, где возможно, приводятся не специально придуманные, а реальные примеры. Случалось даже, что программы, написанные как примеры для этой книги, впоследствии становились частью комплек-

та программ, используемых каждый день. Все примеры были протестированы непосредственно из текста,[1] который представлен в машинно-считываемой форме.

Книга организована следующим образом. Глава 1 представляет собой введение в самые основы работы с системой. Она описывает процесс регистрации, почту, файловую систему, наиболее употребительные команды и содержит начальную информацию о командном процессоре. Опытные пользователи могут пропустить эту главу.

Глава 2 посвящена обсуждению файловой системы UNIX. Файловая система является центральным звеном в функционировании ОС и в ее использовании, поэтому вы должны хорошо понимать ее для правильной работы в UNIX. Обсуждаются файлы и каталоги, привилегии и права доступа к файлам, а также индексные дескрипторы. В конце главы дан обзор иерархии файловой системы и рассказано о файлах устройств.

Командный процессор, или *оболочка,* – это основной инструмент как для исполнения программ, так и для их написания. Глава 3 показывает, как использовать оболочку для создания новых команд, работы с аргументами команд, переменными оболочки, элементарной управляющей логикой и перенаправлением ввода-вывода.

Глава 4 рассказывает о фильтрах – программах, которые выполняют некоторые простые преобразования данных по мере «пропускания» последних через себя. Первый раздел знакомит читателей с командой поиска по шаблону `grep` и ее сородичами, в следующем разделе обсуждаются некоторые наиболее распространенные фильтры, такие как `sort`, а оставшаяся часть главы посвящена двум универсальным программам преобразования данных: `sed` и `awk`. Программа `sed` – это потоковый редактор, осуществляющий редактирование потока данных по мере его поступления. А `awk` – это язык программирования, обеспечивающий простой информационный поиск и генерирование отчетов. В ряде случаев применение этих программ (возможно, во взаимодействии с оболочкой) позволяет полностью избежать программирования в его традиционном виде.

В главе 5 рассматривается применение оболочки для создания программ, которые будут использоваться другими людьми. Обсуждается более сложная управляющая логика и переменные, обработка системных сигналов. Примеры в этой главе составлены с использованием как `sed` и `awk`, так и оболочки.

Со временем вы достигнете предела того, что может быть реализовано при помощи оболочки и других, уже существующих программ. Глава 6

[1] В процессе редактирования русского издания книги все предлагаемые авторами примеры также проверялись на работоспособность и возможность их компиляции, если она необходима, в современном окружении систем Solaris, Linux и FreeBSD. – *Примеч. науч. ред.*

посвящена написанию новых программ с использованием стандартной библиотеки ввода-вывода. Программы написаны на Си. Предполагается, что читатель знает или хотя бы изучает этот язык одновременно с прочтением книги. Приведены разумные методики разработки и организации новых программ, описано поэтапное проектирование, показано, как использовать уже существующий инструментарий.

Глава 7 знакомит с системными вызовами – фундаментом, на котором построены все остальные слои программного обеспечения. Обсуждаемые темы: ввод-вывод, создание файла, обработка ошибок, каталоги, индексы, процессы и сигналы.

Глава 8 рассказывает об инструментах разработки программ, таких как: `yacc` – генератор грамматического анализатора; `make`, контролирующий процесс компиляции больших программ, и `lex`, генерирующий лексические анализаторы. Описание основано на создании большой программы – Си-подобного программируемого калькулятора.

В главе 9 обсуждаются средства подготовки документов, проиллюстрированные описанием (на уровне пользователя) и страницей руководства для калькулятора из главы 8. Данная глава может быть прочитана независимо от остальных.

Приложение A – это краткое изложение материалов о стандартном редакторе `ed`. Вероятно, некоторые читатели предпочитают какой-либо другой редактор для каждодневной работы, но, тем не менее, `ed` – это общедоступный, действенный и производительный редактор. Его регулярные выражения являются основой других программ (например, `grep` и `sed`), и уже по этой причине он заслуживает изучения.

Приложение B представляет собой справочное руководство по языку калькулятора из главы 8.

В приложении C содержится листинг окончательной версии программы калькулятора, для удобства весь код собран в одном месте.

Приведем несколько фактов. Во-первых, система UNIX приобрела большую популярность и в настоящий момент широко используется несколько ее версий. Например, седьмая версия происходит от первоисточника создания системы UNIX – Computing Science Research Center, Bell Laboratories (Центра исследования вычислительных систем лабораторий Белла). System III и System V – это версии, официально поддерживаемые Bell Laboratories. Университет Беркли, Калифорния, распространяет системы, являющиеся производной седьмой версии, известные как UCB 4.xBSD. Кроме того, существует множество вариантов, также полученных на основе седьмой версии, в частности для микрокомпьютеров.

Чтобы справиться с этим разнообразием, авторы рассматривают только те аспекты системы, которые (насколько этого можно ожидать) одинаковы во всех вариантах. Значительная часть представленного материала не зависит от версии системы, а в тех случаях, когда детали

реализации отличаются, предпочтение отдается седьмой версии, так как именно она распространена наиболее широко. Примеры также выполнялись на System V в Bell Laboratories и на 4.1BSD Университета Беркли; лишь в небольшом количестве случаев потребовались незначительные изменения. Ваш компьютер будет работать с ними вне зависимости от версии операционной системы, а обнаруженные различия будут минимальными.

Во-вторых, несмотря на то что в книге представлено много материала, это не справочное руководство. При создании этой книги более важным представлялось описание подхода и способа применения, а не конкретные детали. Стандартным источником информации является справочное руководство по UNIX. В нем можно найти ответы на вопросы, которых вы не получили в данной книге, там же можно прочитать об отличиях конкретной системы от той, которая описана здесь.

В-третьих, по мнению авторов, лучший способ научиться чему бы то ни было состоит в том, чтобы сделать это. Эту книгу надо читать за компьютером, чтобы иметь возможность экспериментировать, проверять или опровергать написанное, исследовать пределы возможностей. Прочитайте немного, попробуйте сделать то, что написано, потом вернитесь и читайте дальше.

UNIX – это, конечно же, не совершенная, но удивительная и непостижимая вычислительная среда. Надеемся, что читатели придут к этому же выводу.

Хотелось бы выразить многим людям благодарность за конструктивные замечания и критику, а также за помощь в усовершенствовании кода. В особенности Джону Бентли (Jon Bentley), Джону Линдерману (John Linderman), Дагу Мак-Илрою (Doug McIlroy) и Питеру Вейнбергеру (Peter Weinberger), которые с огромным вниманием читали многочисленные черновики. Мы признательны Алу Ахо (Al Aho), Эду Бредфорду (Ed Bradford), Бобу Фландрене (Bob Flandrena), Дейву Хансону (Dave Hanson), Рону Хардину (Ron Hardin), Марион Харрис (Marion Harris), Джерарду Хользманну (Gerard Holzmann), Стиву Джонсону (Steve Johnson), Нико Ломуто (Nico Lomuto), Бобу Мартину (Bob Martin), Ларри Рослеру (Larry Rosler), Крису Ван Вику (Chris Van Wyk) и Джиму Вейтману (Jim Weytman) за их замечания по первому варианту этой книги. Мы также благодарим Мика Бьянчи (Mic Bianchi), Элизабет Биммлер (Elizabeth Bimmler), Джо Карфагно (Joe Carfagno), Дона Картера (Don Carter), Тома Де Марко (Tom De Marco), Тома Дафа (Tom Duff), Дэвида Гея (David Gay), Стива Махани (Steve Mahaney), Рона Пинтера (Ron Pinter), Денниса Ритчи (Dennis Ritchie), Эда Ситара (Ed Sitar), Кена Томпсона (Ken Thompson), Майка Тилсона (Mike Tilson), Поля Туки (Paul Tukey) и Ларри Вера (Larry Wehr) за их ценные предложения.

Брайан Керниган (Brian Kernighan)
Роб Пайк (Rob Pike)

7

Системные вызовы UNIX

Данная глава посвящена самому нижнему уровню взаимодействия с операционной системой UNIX – системным вызовам, которые служат точками входа в ядро. Именно эти возможности предоставляет собственно операционная система; все остальное строится на этой основе.

Рассмотрим несколько основных тем. Прежде всего, систему ввода-вывода, лежащую в основе таких библиотечных функций, как `fopen` и `putc`. Также мы продолжим обсуждение файловой системы, в частности каталогов и индексных дескрипторов (inode). Затем обсудим процессы – способы запуска программ из других программ. После этого поговорим о сигналах и прерываниях: что происходит при нажатии клавиши *DELETE* и как правильно обработать это в программе.

Как и в главе 6, многие примеры представляют собой полезные программы, не вошедшие в седьмую верию. Даже если вы не будете использовать именно эти программы, знакомство с ними может подсказать вам идеи для своих собственных проектов.

Системные вызовы детально рассмотрены в man2, а в этой главе будут рассмотрены наиболее важные моменты (без претензии на полноту изложения).

7.1. Низкоуровневый ввод-вывод

Самый нижний уровень ввода-вывода напрямую взаимодействует с операционной системой. Программа читает и пишет файлы фрагментами произвольного размера. Ядро буферизует данные в блоках, размер которых определяется периферийным устройством, и устанавливает очередность работы устройств, оптимизируя производительность для всех пользователей.

Дескрипторы файлов

Весь ввод и вывод выполняется как чтение и запись файлов, поскольку все периферийные устройства, включая и терминал, представлены в файловой системе файлами. Это значит, что единый интерфейс управляет всем взаимодействием между программой и периферийными устройствами.

В самом общем случае перед тем как читать или писать файл, необходимо сообщить системе о своем намерении; этот процесс называется *открытием* файла. Если вы собираетесь записывать в файл, возможно, его надо предварительно *создать*. Система проверяет права пользователя на выполнение операции (существует ли файл? есть ли разрешение на доступ к нему?) и, если все в порядке, возвращает неотрицательное целое число, которое и называется *дескриптором файла*. Всякий раз, когда выполняется файловый ввод-вывод, дескриптор используется вместо имени для идентификации файла. Вся информация об открытом файле поддерживается системой, а программа обращается к нему только по дескриптору. Указатель типа `FILE`, как рассказывалось в главе 6, указывает на структуру, которая кроме прочего содержит дескриптор файла, а макрос `fileno(fp)`, определенный в `<stdio.h>`, возвращает этот дескриптор.

Существуют специальные соглашения, позволяющие сделать терминальный ввод-вывод более удобным. Программа, будучи запущена оболочкой, получает от нее три открытых файла с дескрипторами `0`, `1` и `2`, которые называются стандартным вводом, стандартным выводом и стандартным выводом ошибок. Все они по умолчанию связаны с терминалом, так что если программа ограничивается чтением файла с дескриптором `0` и записью в файлы с дескрипторами `1` и `2`, ей не нужно открывать файлы для ввода-вывода. Если же программа открывает иные файлы, то они получат дескрипторы `3`, `4` и т. д.

Если используется перенаправление ввода-вывода, то оболочка изменяет установленное по умолчанию соответствие для дескрипторов `0` и `1` с терминала на указанные файлы. Обычно файловый дескриптор `2` остается связанным с терминалом, чтобы на него могли выводиться сообщения об ошибках. Такие команды, как `2>`*имя-файла* и `2>&1`, вызовут переопределение умолчаний, причем переназначение файлов выполняется оболочкой, а не программой. (При желании программа может сама совершить переназначение, но это встречается редко.)

Файловый ввод-вывод – read и write

Весь ввод и вывод осуществляется двумя системными вызовами – `read` и `write`, которые доступны из языка Си через функции с теми же именами. В обеих функциях первый аргумент – это дескриптор файла. Второй аргумент – массив байт, служащий источником или приемником данных. Третий аргумент задает количество передаваемых байт.

```
int fd, n, nread, nwritten;
char buf[SIZE];

nread = read(fd, buf, n);
nwritten = write(fd, buf, n);
```

Оба вызова возвращают количество фактически переданных байт. При чтении возвращаемое число может быть меньше ожидаемого, если до конца файла осталось менее `n` байт. (Если файл является терминалом, вызов `read` обычно читает до символа новой строки, а это, как правило, меньше указанного значения.) При достижении конца файла возвращается значение `0`, а в случае ошибки – значение `-1`. При записи возвращается число фактически записанных байт; отличие этого значения от ожидаемого говорит о наличии ошибки.

Хотя количество байт для чтения и записи не ограничивается, на практике чаще всего задается значение `1`, то есть посимвольная передача данных (небуферизованный ввод-вывод), и значение, равное размеру дискового блока, чаще всего 512 или 1024. (Это значение содержится в константе `BUFSIZ` в файле `stdio.h`.)

Проиллюстрируем вышесказанное при помощи программы, копирующей свой ввод в вывод. Так как ввод и вывод могут быть перенаправлены в любой файл или устройство, эта программа копирует что угодно куда угодно: это схематичная реализация команды `cat`.

```
/* cat: минимальная версия */
#define SIZE 512 /* произвольное значение */

main()
{
    char buf[SIZE];
    int n;

    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    exit(0);
}
```

Если размер файла не кратен `SIZE`, то очередной вызов `read` вернет меньшее значение для записи, а следующий после этого вызов `read` вернет значение `0`.

Наиболее эффективно чтение и запись данных выполняются порциями, равными размеру дискового блока, хотя и посимвольный ввод-вывод годится для небольших объемов данных, так как ядро само выполняет буферизацию и основные затраты приходятся на системные вызовы. Редактор `ed`, к примеру, использует побайтное чтение из стандартного ввода. Мы измерили производительность этой версии `cat` на файле длиной 54 000 байт для шести значений `SIZE`:

	Время (пользовательское+системное, сек)	
SIZE	PDP-11/70	VAX-11/750
1	271,0	188,8
10	29,9	19,3
100	3,8	2,6
512	1,3	1,0
1024	1,2	0,6
5120	1,0	0,6

Размер блока составляет 512 байт в системе PDP-11 и 1024 в системе VAX.

Совершенно нормальна ситуация, когда несколько процессов одновременно обращаются к одному файлу; например, один процесс записывает данные, а другой в это время их читает. Такое поведение может сбивать с толку, но иногда бывает и полезным. Даже если вызов `read` вернул `0`, следующий вызов может обнаружить новые данные, если они в это время были записаны. Этот эффект положен в основу программы `readslow`, которая продолжает чтение ввода независимо от того, достигнут ли конец файла. Программа `readslow` удобна для наблюдения за процессом выполнения программ:

```
$ slowprog >temp &
5213                    Идентификатор процесса
$ readslow <temp | grep something
```

Другими словами, медленная программа выполняет вывод в файл, а программа `readslow`, возможно, взаимодействуя с какой-либо другой программой, следит за накоплением данных.

Программа `readslow` идентична программе `cat` за исключением того, что она продолжает выполнять цикл после достижения конца файла. Здесь обращение к низкоуровневому вводу-выводу неизбежно, так как функции стандартной библиотеки продолжают возвращать `EOF` после первого достижения конца файла.

```
/* readslow: непрерывное чтение в ожидании данных */
#define SIZE     512     /* произвольное значение */

main()
{
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

Функция sleep приостанавливает выполнение программы на заданное число секунд; это описано в sleep(3). Не нужно, чтобы readslow тормошила файл, постоянно обращаясь за новыми данными; это потребует слишком много процессорного времени. Таким образом, эта версия readslow копирует свой ввод, достигает конца файла и прерывается ненадолго, затем пытается повторить ввод. Если за время бездействия поступили новые данные, они будут считаны при следующем проходе.

Упражнение 7.1. Добавьте в программу readslow параметр *-n* так, чтобы функция sleep останавливала выполнение на *n* секунд. В некоторых системах команда tail имеет параметр -f (forever, то есть бесконечно), который включает в команде tail режим, подобный readslow. Прокомментируйте это решение. □

Упражнение 7.2. Что произойдет с программой readslow, если размер читаемого файла уменьшится? Как это исправить? Подсказка: прочитайте об fstat в разделе 7.3. □

Создание файла – open, creat, close, unlink

Помимо использования стандартных файлов ввода, вывода и вывода ошибок, вам потребуется явно открывать файлы для чтения и записи. Для этой цели существуют два системных вызова: open и creat.[1]

Функция open аналогична fopen из предыдущей главы за исключением того, что она возвращает не указатель файла, а файловый дескриптор, имеющий тип int.

```
char *name;
int fd, rwmode;

fd = open(name, rwmode);
```

Как и в fopen, параметр name – это строка, содержащая имя файла. А вот параметр типа доступа отличается: rwmode имеет значение 0 для чтения, 1 для записи и 2 для чтения и записи. Функция open возвращает -1 в случае ошибки, а при успешном завершении – дескриптор файла.

Попытка открытия существующего файла приводит к ошибке. Для создания новых файлов и для перезаписи уже существующих предназначен системный вызов creat.

```
int perms;

fd = creat(name, perms);
```

Функция creat возвращает дескриптор файла, если ей удалось его создать, и -1 в противном случае. Если файл не существует, то creat созда-

[1] Кена Томпсона (Ken Thompson) однажды спросили, что бы он изменил в UNIX, если бы довелось перепроектировать систему заново. Он ответил: «Я бы написал creat с буквой e».

ет его с *правами доступа*, указанными в параметре perms. Если же файл уже существует, то creat обрежет его до нулевой длины; вызов creat для уже существующего файла не является ошибкой. (Права доступа не изменятся.) Независимо от значения perms созданный файл открыт для записи.

Как рассказывалось в главе 2, информация о правах доступа к файлу хранится в девяти битах, определяющих разрешение на чтение, запись и исполнение, поэтому их принято представлять трехзначным восьмеричным числом. Например, 0755 определяет разрешение на чтение, запись и выполнение для владельца и на чтение и выполнение для группы и для остальных. Не забывайте начинать восьмеричные числа с 0, как это принято в языке Си.

Для иллюстрации приведем упрощенную версию команды cp. Главное упрощение заключается в том, что наша версия копирует только один файл и не позволяет в качестве второго аргумента указывать каталог. Второй изъян состоит в том, что данная версия не сохраняет права доступа исходного файла (дальше будет показано, как с этим справиться).

```
/* cp: минимальная версия */
#include <stdio.h>
#define PERMS 0644 /* RW для владельца, R для группы и остальных */
char *progname;

main(argc, argv)  /* cp: копировать f1 в f2 */
int argc;
    char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZ];

    progname = argv[0];
    if (argc != 3)
        error("Usage: %s from to", progname);
    if ((f1 = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("write error", (char *) 0);
    exit(0);
}
```

Функция error будет описана в следующем разделе.

Существует ограничение (значение NOFILE в sys/param.h, обычно около 20) количества файлов, одновременно открываемых программой. Следовательно, если программа собирается работать с многими файлами, ей следует позаботиться о повторном использовании файловых де-

скрипторов. Системный вызов close разрывает связь между именем файла и его дескриптором, освобождая таким образом дескриптор для использования с другим файлом. Завершение программы посредством вызова exit или выхода из main вызывает закрытие всех файлов.

Системный вызов unlink удаляет имя; файл удаляется, если это последняя ссылка.

Обработка ошибок – errno

Системные вызовы, описанные в этом разделе, а фактически все системные вызовы, могут завершаться с ошибками. Обычно они сообщают об этом, возвращая значение -1. Иногда бывает полезно знать, какая именно ошибка произошла; с этой целью все системные вызовы, когда это уместно, оставляют номер ошибки во внешней переменной errno. (Значения различных номеров ошибок приведены во введении раздела 2 руководства по UNIX для программиста.) Пользуясь errno, программа может, к примеру, определить, вызвана ошибка открытия файла его отсутствием или же отсутствием прав доступа к нему у пользователя. Для преобразования номеров в текстовые сообщения существует массив строк sys_errlist, индексом в котором служат значения errno.[1]

Наша версия функции error использует эти структуры данных:

```
error(s1, s2) /* вывести сообщение об ошибке и закончить работу */
char *s1, *s2;
{
    extern int errno, sys_nerr;
    extern char *sys_errlist[], *progname;

    if (progname)
        fprintf(stderr, "%s: ", progname);
    fprintf(stderr, s1, s2);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, " (%s)", sys_errlist[errno]);
    fprintf(stderr, "\n");
    exit(1);
}
```

Значение errno инициализируется нулем и не должно превышать sys_nerr. При отсутствии ошибок errno не обнуляется, следовательно, пользователь должен делать это самостоятельно, если программа продолжает работу после возникновения ошибки.

---

[1] В современных реализациях переменная sys_errlist объявлена в stdio.h как константный указатель, поэтому следует исключить предложенное ниже объявление sys_errlist, так как его присутствие в тексте программы будет вызывать ошибку компиляции из-за несоответствия типов в объявлениях sys_errlist, полученных из stdio.h и из текста программы. – *Примеч. науч. ред.*

Вот как выглядят сообщения об ошибках в данной версии cp:

```
$ cp foo bar
cp: can't open foo (No such file or directory)
$ date >foo; chmod 0 foo          Создан файл с запретом на чтение
$ cp foo bar
cp: can't open foo (Permission denied)
$
```

Произвольный доступ – lseek

Обычно ввод-вывод выполняется последовательно: каждая операция чтения или записи начинается там, где закончилась предыдущая. Но при необходимости файл может быть прочитан и записан в произвольном порядке. Системный вызов lseek позволяет перемещаться по файлу, не выполняя при этом чтение или запись:

```
int fd, origin;
long offset, pos, lseek();

pos = lseek(fd, offset, origin);
```

В этом фрагменте текущая позиция в файле с дескриптором fd перемещается на offset байт относительно позиции, определенной параметром origin. Следующая операция чтения или записи начнется с этой позиции. Параметр origin может принимать значения 0, 1 и 2, при этом смещение отсчитывается от начала файла, от текущей позиции и от конца файла соответственно. Функция возвращает новое значение абсолютной позиции или -1 в случае ошибки. Например, чтобы дописать данные в файл, переместитесь в его конец:

```
lseek(fd, 0L, 2);
```

Чтобы вернуться в начало («перемотать»),

```
lseek(fd, 0L, 0);
```

Чтобы определить текущую позицию,

```
pos = lseek(fd, 0L, 1);
```

Обратите внимание на аргумент 0L: смещение – это длинное целое. (Буква «l» в имени lseek означает long, т. е. «длинное», что отличает эту функцию от seek из шестой версии, использующей короткие целые.)

Функция lseek позволяет с некоторой степенью приближения трактовать файлы как большие массивы, правда ценой более медленного доступа. Вот, например, функция, читающая произвольное количество байт из произвольного места файла:

```
get(fd, pos, buf, n) /* прочитать n байт из позиции pos */
    int fd, n;
```

```
    long pos;
    char *buf;
{
    if (lseek(fd, pos, 0) == -1) /* перейти на позицию pos */
        return -1;
    else
        return read(fd, buf, n);
```

Упражнение 7.3. Измените функцию `readslow` так, чтобы она обрабатывала имя файла, если оно передано параметром. Добавьте параметр `-e`:

```
$ readslow -e
```

заставляющий `readslow` перемещаться в конец входного файла перед началом чтения. Как поведет себя `lseek` при работе с программным каналом (pipe)? □

Упражнение 7.4. Перепишите функцию `efopen` из главы 6 с использованием вызова `error`. □

7.2. Файловая система: каталоги

Теперь посмотрим, как перемещаться по иерархии каталогов. Для этого не потребуются новые системные вызовы, просто используем старые в новом контексте. Поясним на примере функции `spname`, которая попытается справиться с неправильным именем файла. Функция

```
n = spname (имя, новое-имя);
```

ищет файл с именем, «достаточно похожим» на заданное параметром *имя*. Если таковое найдено, оно копируется в параметр *новое-имя*. Возвращаемое значение равно `-1`, если ничего похожего не найдено, `0` – если найдено точное соответствие, и `1` – если потребовалось исправление.

Функция `spname` представляет собой удобное дополнение к программе `p`. Если пользователь хочет напечатать файл, но ошибся в написании его имени, программа `p` может попросить его уточнить, что имелось в виду:

```
$ p /urs/srx/ccmd/p/spnam.c      Абсолютно неправильное имя
"/usr/src/cmd/p/spname.c"? y     Предложенное исправление принято
/* spname: возвратить правильно написанное имя файла */
...
```

Напишем функцию так, чтобы она пыталась исправить в каждом элементе имени файла следующие ошибки: одна пропущенная или лишняя буква, одна буква неправильная, пара букв перепутана местами – все эти варианты есть в примере. Такая программа будет просто находкой для невнимательной машинистки.

Перед тем как начать программировать, совершим краткий экскурс в структуру файловой системы. Каталог – это файл, содержащий пере-

чень имен файлов с указанием их расположения. Под «расположением» здесь понимается индекс файла в другой таблице, называемой *таблицей индексных дескрипторов*. Запись в этой таблице, называемая индексным дескриптором (inode) файла, – это то место, где хранится вся информация о нем, за исключением имени. Таким образом, запись в каталоге состоит из двух элементов – номера индексного дескриптора и имени файла. Точная спецификация находится в файле `sys/dir.h`:

```
$ cat /usr/include/sys/dir.h
#define DIRSIZ 14 /* максимальная длина имени файла */

struct direct /* структура записи в каталоге */
{
    ino_t d_ino; /* номер индексного дескриптора */
    char d_name[DIRSIZ]; /* имя файла */
};
$
```

«Тип» данных `ino_t`, определенный с помощью `typedef`, описывает запись в таблице индексных дескрипторов. В системах PDP-11 и VAX это тип `unsigned short`, но в других системах он, скоре всего, будет другим, не стоит полагаться на это в программах, поэтому и использован `typedef`. Полный набор «системных» типов находится в файле `sys/types.h`, который должен быть включен до файла `sys/dir.h`.

Алгоритм работы `spname` достаточно прост. Предположим, надо обработать имя файла `/d1/d2/f`. Основная идея заключается в том, чтобы отбросить первый элемент (/) и искать в корневом каталоге имя, максимально совпадающее со следующим элементом (`d1`), затем в найденном каталоге искать что-либо подобное `d2` и так далее, пока не будут найдены соответствия для всех элементов. Если в очередном каталоге не удалось найти подходящее имя, поиск прекращается.

Вся работа разделена между тремя функциями: собственно `spname` отделяет элементы друг от друга и составляет из них «наиболее вероятное» имя файла. Функция `mindist` вызывается из `spname` и выполняет в заданном каталоге поиск файла с именем, максимально похожим на заданное, обращаясь при этом к функции `spdist`, вычисляющей «расстояние» между двумя именами.

```
/* spname: возвращает правильно написанное имя файла */
/*
 * spname(oldname, newname)  char *oldname, *newname;
 *  returns -1 if no reasonable match to oldname,
 *           0 if exact match,
 *           1 if corrected.
 *  stores corrected name in newname.
 */

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

```
spname(oldname, newname)
    char *oldname, *newname;
{
    char *p, guess[DIRSIZ+1], best[DIRSIZ+1];
    char *new = newname, *old = oldname;

    for (;;) {
        while (*old == '/') /* пропустить слэши */
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0')   /* правильно или исправлено */
            return strcmp(oldname,newname) != 0;
        p = guess;  /* скопировать следующий компонент в guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZ)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
            return -1;  /* неудачно */
        for (p = best; *new = *p++; ) /* добавить в конец */
            new++;                    /* нового имени */
    }
}

mindist(dir, guess, best)   /* просмотр dir в поиске guess */
    char *dir, *guess, *best;
{
    /* устанавливает best, возвращает расстояние 0..3 */
    int d, nd, fd;
    struct {
        ino_t ino;
        char  name[DIRSIZ+1];   /* на 1 больше, чем в dir.h */
    } nbuf;

    nbuf.name[DIRSIZ] = '\0';   /* +1 для заключительного '\0' */
    if (dir[0] == '\0')     /* текущий каталог */
        dir = ".";
    d = 3;  /* минимальное расстояние */
    if ((fd=open(dir, 0)) == -1)
        return d;
    while (read(fd,(char *) &nbuf,sizeof(struct direct)) > 0)
        if (nbuf.ino) {
            nd = spdist(nbuf.name, guess);
            if (nd <= d && nd != 3) {
                strcpy(best, nbuf.name);
                d = nd;
                if (d == 0)     /* точное совпадение */
                    break;
            }
        }
    close(fd);
    return d;
}
```

Если имя каталога, переданное в функцию `mindist`, пусто, то поиск выполняется в текущем каталоге (`.`), за один проход обрабатывается один каталог. Обратите внимание, что в качестве буфера для чтения выступает структура, а не символьный массив. Размер определяется посредством `sizeof`, а затем адрес преобразуется в указатель на тип `char`.

Когда запись в каталоге не используется (если файл был удален), то соответствующая запись, содержащая нулевое значение индексного дескриптора, пропускается. Расстояние проверятся выражением

```
if (nd <= d ...)
```

а не

```
if (nd < d ...)
```

так что любой символ считается более подходящим, чем точка «.», которая всегда является первым элементом каталога.

```
/* spdist: возвращает расстояние между двумя именами */
/*
 *  очень грубый показатель правильности написания:
 *  0, если две строки идентичны
 *  1, если два символа переставлены местами
 *  2, если один символ добавлен, удален или не совпадает
 *  3 - иначе
 */

#define EQ(s,t) (strcmp(s,t) == 0)

spdist(s, t)
    char *s, *t;
{
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;       /* точное совпадение */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1]
              && *t == s[1] && EQ(s+2, t+2))
                return 1;   /* перестановка */
            if (EQ(s+1, t+1))
                return 2;   /* 1 несовпадающий символ */
        }
        if (EQ(s+1, t))
            return 2;       /* лишний символ */
    }
    if (*t && EQ(s, t+1))
        return 2;           /* недостающий символ */
    return 3;
}
```

Теперь, когда у нас есть функция spname, очень просто добавить проверку написания к команде p:

```
/* p: печатает ввод порциями (версия 4) */

#include <stdio.h>
#define PAGESIZE    22
char  *progname; /* имя программы для сообщения об ошибке */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp, *efopen();
    int i, pagesize = PAGESIZE;
    char *p, *getenv(), buf[BUFSIZ];

    progname = argv[0];
    if ((p=getenv("PAGESIZE")) != NULL)
        pagesize = atoi(p);
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc == 1)
        print(stdin, pagesize);
    else
        for (i = 1; i < argc; i++)
            switch (spname(argv[i], buf)) {
            case -1:    /* совпадение невозможно */
                fp = efopen(argv[i], "r");
                break;
            case 1:     /* исправлено */
                fprintf(stderr, "\"%s\"? ", buf);
                if (ttyin() == 'n')
                    break;
                argv[i] = buf;
                /* неуадчно… */
            case 0: /* точное совпадение */
                fp = efopen(argv[i], "r");
                print(fp, pagesize);
                fclose(fp);
            }
    exit(0);
}
```

Автоматическое исправление имен файлов надо использовать с определенной осторожностью. Оно хорошо работает в интерактивных программах, таких как p, но не подходит для программ, которые могут работать без общения с пользователем.

Упражнение 7.5. Насколько можно было бы усовершенствовать эвристический алгоритм наилучшего совпадения в `spname`? Например, нет смысла обрабатывать имя файла как каталог, а в данной версии это возможно. □

Упражнение 7.6. Имя `tx` всегда будет считаться подходящим, если каталог заканчивается на `t`*c*, независимо от значения *c*. Можно ли улучшить определение расстояния? Напишите программу и посмотрите, как она понравится пользователям. □

Упражнение 7.7. Функция `mindist` читает каталог поэлементно. Можно ли существенно ускорить программу `p`, выполняя чтение большими блоками? □

Упражнение 7.8. Измените функцию `spname` так, чтобы она возвращала префикс предполагаемого имени, если не удалось найти достаточно близкое соответствие. Что делать, если несколько имен соответствуют этому префиксу? □

Упражнение 7.9. В каких еще программах можно использовать `spname`? Напишите автономную программу, исправляющую полученные аргументы перед тем, как передать их в другую программу, например такую, как

```
$ fix prog имена-файлов...
```

Можно ли написать версию `cd` с использованием `spname`? Как ее проинсталлировать? □

7.3. Файловая система: индексные дескрипторы

В этом разделе будут рассмотрены системные вызовы, имеющие дело с файловой системой и, в частности, с информацией о файлах, такой как размер, даты, права доступа и т. д. Эти системные вызовы дают доступ ко всей той информации, которую мы обсуждали в главе 2.

Давайте теперь обратимся собственно к индексным дескрипторам. Часть их описывается структурой `stat`, определяемой в `sys/stat.h`:

```
struct stat /* структура, возвращаемая stat */
{
dev_t    st_dev;      /* устройство индексного дескриптора */
ino_t    st_ino;      /* номер индексного дескриптора */
short    st_mode      /* биты доступа */
short    st_nlink;    /* количество ссылок на файл */
short    st_uid;      /* идентификатор владельца */
short    st_gid;      /* идентификатор группы владельца */
dev_t    st_rdev;     /* для специальных файлов */
off_t    st_size;     /* размер файла в символах */
time_t   st_atime;    /* время последнего чтения из файла */
```

```
time_t   st_mtime;    /* время последней записи или создания файла */
time_t   st_ctime;    /* время последнего изменения файла или индексного
                         дескриптора */
};
```

Большинство полей описано в комментариях. Такие типы, как `dev_t` и `ino_t`, определены в `sys/types.h`, это обсуждалось в предыдущих разделах. Элемент `st_mode` содержит набор флагов, описывающих файл; для удобства флаги определены также в файле `sys/stat.h`:

```
#define S_IFMT  0170000    /* тип файла */
#define    S_IFDIR 0040000 /* каталог */
#define    S_IFCHR 0020000 /* специальный символьный */
#define    S_IFBLK 0060000 /* специальный блочный */
#define    S_IFREG 0100000 /* обычный */
#define S_ISUID  0004000   /* при запуске установить эффективный
                              идентификатор пользователя как у владельца */
#define S_ISGID  0002000   /* при запуске установить эффективный
                              идентификатор группы как у владельца */
#define S_ISVTX  0001000   /* сохранить перекачанный текст даже после
                              использования */
#define S_IREAD  0000400   /* права на чтение, владелец */
#define S_IWRITE 0000200   /* права на запись, владелец */
#define S_IEXEC  0000100   /* права на выполнение/поиск, владелец */
```

Доступ к индексному дескриптору файла выполняется при помощи двух системных вызовов: `stat` и `fstat`. Вызов `stat` принимает имя файла и возвращает информацию индексного дескриптора для этого файла (или `-1` в случае ошибки), `fstat` делает то же самое из файлового дескриптора для открытого файла (не из указателя `FILE`). Таким образом,

```
char *name;
int fd;
struct stat stbuf;

stat(name, &stbuf);
fstat(fd, &stbuf);
```

Структура `stbuf` заполняется информацией из индексного дескриптора для файла с именем `name` или дескриптором `fd`.

Теперь, когда мы располагаем всеми этими фактами, попробуем написать какой-нибудь полезный код. Начнем с того, что напишем на языке Си программу `checkmail`, которая будет наблюдать за почтовым ящиком пользователя. Если файл увеличивается в размере, программа пишет `You have mail` и подает звуковой сигнал. (Если файл становится меньше, это, видимо, потому, что вы только что прочитали и удалили какое-то письмо, сообщения не требуются). Для первого шага этого вполне достаточно; когда эта программа заработает, можно будет ее усовершенствовать.

```
/* checkmail: следит за почтовым ящиком пользователя */
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
char *progname;
char *maildir = "/usr/spool/mail";  /* зависит от системы */

main(argc, argv)
    int argc;
    char *argv[];
{
    struct stat buf;
    char *name, *getlogin();
    int lastsize = 0;

    progname = argv[0];
    if ((name = getlogin()) == NULL)
        error("can't get login name", (char *) 0);
    if (chdir(maildir) == -1)
        error("can't cd to %s", maildir);
    for (;;) {
        if (stat(name, &buf) == -1) /* нет почтового ящика */
            buf.st_size = 0;
        if (buf.st_size > lastsize)
            fprintf(stderr, "\nYou have mail\007\n");
        lastsize = buf.st_size;
        sleep(60);
    }
}
```

Функция `getlogin(3)` возвращает имя, с которым зарегистрирован пользователь, или `NULL`, если имя не получено. Системный вызов `chdir` переводит `checkmail` в почтовый каталог, так что последующим вызовам `stat` не приходится просматривать каждый каталог, начиная с корневого и заканчивая почтовым. В вашей системе, может быть, придется изменить значение `maildir`. Программа `checkmail` написана так, чтобы попытки предпринимались даже в том случае, когда почтовый ящик не существует, поскольку большинство почтовых программ удаляют почтовый ящик, если он пуст.

Эта программа была представлена в главе 5 для иллюстрации циклов оболочки. Та версия создавала несколько процессов при каждом просмотре почтового ящика, что могло вызвать большую, чем хотелось бы, нагрузку на систему. Версия на языке Си – это единственный процесс, запускающий `stat` для файла каждую минуту. Во что обойдется программа проверки почтового ящика, которая все время работает в фоновом режиме? По оценкам авторов, будет затрачено менее одной секунды в час, что довольно мало, и это чрезвычайно важно.

Иллюстрация к обработке ошибок: sv

Теперь напишем программу под названием `sv` (похожую на `cp`), которая копирует набор файлов в каталог, но при этом перезаписывает файл назначения, только если он не существует или является более

старым, чем исходный файл. Название sv происходит от английского слова save (сохранить) – идея в том, что sv не перезаписывает более новые версии файлов. Команда sv учитывает больше информации из индексного дескриптора, чем checkmail.

Будем запускать sv следующим образом:

```
$ sv file1 file2 ... dir
```

Эта команда должна копировать file1 в dir/file1, file2 в dir/file2 и т. д., за исключением тех случаев, когда файл назначения новее соответствующего исходного файла – в этом случае копия не делается и выдается предупреждение. Для того чтобы избежать многократного копирования ссылок, программа sv не разрешает использовать / в именах исходных файлов.

```
/* sv: сохраняет новые файлы */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    struct stat stbuf;
    char *dir = argv[argc-1];

    progname = argv[0];
    if (argc <= 2)
        error("Usage: %s files... dir", progname);
    if (stat(dir, &stbuf) == -1)
        error("can't access directory %s", dir);
    if ((stbuf.st_mode & S_IFMT) != S_IFDIR)
        error("%s is not a directory", dir);
    for (i = 1; i < argc-1; i++)
        sv(argv[i], dir);
    exit(0);
}
```

Значения времени в индексном дескрипторе задаются в секундах после 0:00 GMT, 1 января 1970, так что более старые файлы имеют меньшие значения в поле st_mtime.

```
sv(file, dir)   /* сохраняет file в dir */
char *file, *dir;
{
    struct stat sti, sto;
    int fin, fout, n;
```

```
    char target[BUFSIZ], buf[BUFSIZ], *index();

    sprintf(target, "%s/%s", dir, file);
    if (index(file, '/') != NULL)  /* strchr() в некоторых системах */
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1)   /* файл назначения не существует */
        sto.st_mtime = 0;   /* пусть он будет старее */
    if (sti.st_mtime < sto.st_mtime)    /* файл назначения более новый */
        fprintf(stderr, "%s: %s not copied\n",
            progname, file);
    else if ((fin = open(file, 0)) == -1)
        error("can't open file %s", file);
    else if ((fout = creat(target, sti.st_mode)) == -1)
        error("can't create %s", target);
    else
        while ((n = read(fin, buf, sizeof buf)) > 0)
            if (write(fout, buf, n) != n)
                error("error writing %s", target);
    close(fin);
    close(fout);
}
```

Вместо стандартных функций ввода-вывода использована `creat` для того, чтобы `sv` имела возможность сохранить код прав доступа исходного файла. (Обратите внимание на то, что `index` и `strchr` – это разные названия одной и той же процедуры; посмотрите в руководстве описание `string(3)`, чтобы узнать, какое имя использует ваша система.)

Несмотря на то что `sv` – это специализированная программа, она иллюстрирует несколько важных идей. Есть множество программ, хотя и не являющихся «системными», но тем не менее способных использовать информацию, хранимую операционной системой, к которой можно получить доступ посредством системных вызовов. Для таких программ ключевым моментом является то, что системные типы данных определены только в стандартных заголовочных файлах, таких как `stat.h` и `dir.h`, и программа включает в себя эти файлы, а не использует собственные определения типов. У такого кода гораздо больше шансов оказаться переносимым из одной системы в другую.

Не удивляйтесь, что как минимум две трети кода программы `sv` – это проверка ошибок. На начальных этапах написания программы есть соблазн сэкономить на обработке ошибок, так как это не входит в постановку основной задачи. А когда программа «заработает», уже не хватает энтузиазма для того, чтобы вернуться назад и вставить проверки, которые превратят частную программу в устойчиво работающий продукт.

Программа `sv` не защищена от всех возможных несчастий, например она не обрабатывает прерывания в неудобное время, но она более осто-

рожна, чем большинство программ. Остановимся буквально на одном моменте – рассмотрим последний оператор `write`. Редко случается, что `write` не выполняется успешно, поэтому многие программы игнорируют такую возможность. Но на дисках заканчивается свободное место; пользователи превышают квоты; происходят обрывы линий связи. Все эти обстоятельства могут послужить причиной ошибок записи, и будет гораздо лучше, если пользователь узнает об этом, вместо того чтобы программа молча делала вид, что все хорошо.

Мораль в том, что проверка ошибок скучна и утомительна, но чрезвычайно важна. Из-за ограниченного пространства и желания сконцентрировать внимание на наиболее интересных вопросах авторы опускали эту часть во многих программах, рассмотренных в данной книге. Но в настоящих программных продуктах вы не можете позволить себе игнорировать ошибки.

Упражнение 7.10. Измените `checkmail` таким образом, чтобы идентифицировать отправителя письма в сообщении `You have mail` (Вы получили письмо). Подсказка: `sscanf`, `lseek`. □

Упражнение 7.11. Измените `checkmail` так, чтобы каталог не изменялся на почтовый до входа в цикл. Будет ли это иметь значение для производительности? (Сложнее.) Можете ли вы написать версию `checkmail`, которой требовался бы только один процесс для оповещения всех пользователей? □

Упражнение 7.12. Напишите программу `watchfile`, которая проверяет файл и печатает его с начала каждый раз, когда он изменяется. Где можно использовать такую программу? □

Упражнение 7.13. Программа `sv` не обладает гибкостью в обработке ошибок. Измените ее так, чтобы она продолжалась даже в том случае, когда она не может обработать некоторые файлы. □

Упражнение 7.14. Сделайте `sv` рекурсивной: если один из исходных файлов – это каталог, пусть каталог и файлы из него обрабатываются одинаково. Сделайте `cp` рекурсивной. Подумайте, должны ли `cp` и `sv` быть одной и той же программой, так чтобы `cp -v` не создавала копию в том случае, когда файл назначения более новый. □

Упражнение 7.15. Напишите программу `random`:

```
$ random имя-файла
```

которая выводит одну строку, случайным образом выбранную из файла. Если `random` обрабатывает файл с именами, то она может быть использована в программе `scapegoat`, которая помогает найти виноватого:

```
$ cat scapegoat
echo "It's all `random people`'s fault!"
$ scapegoat
It's all Ken's fault!
$
```

Убедитесь, что random работает правильно вне зависимости от распределения длин строк. □

Упражнение 7.16. В индексном дескрипторе содержится и другая информация, как, например, адреса на диске, в которых расположены блоки файлов. Исследуйте файл sys/ino.h, а потом напишите программу icat, которая будет читать файлы, определяемые по индексному дескриптору и дисковому устройству. (Естественно, она будет работать только при условии, что рассматриваемый диск читаем.) При каких условиях полезна icat? □

7.4. Процессы

Этот раздел описывает выполнение одной программы из другой. Самый простой способ сделать это – обратиться к стандартной библиотечной функции system, описанной, но осужденной в главе 6. Команда system получает один аргумент – командную строку точно в том виде, как она набрана на терминале (за исключением символа новой строки в конце), и выполняет ее в подоболочке. Если командная строка должна быть составлена из нескольких частей, то могут пригодиться возможности форматирования в памяти, которыми обладает sprintf. В конце данного раздела будет представлена более надежная версия system для использования в интерактивных программах, но сначала надо исследовать части, из которых она состоит.

Низкоуровневое создание процессов – execlp и execvp

Базовая операция – это выполнение другой программы *без ожидания завершения* системным вызовом execlp. Например, чтобы напечатать дату в качестве последнего действия выполняющейся программы, используйте

```
execlp("date", "date", (char *) 0);
```

Первый аргумент execlp – это имя файла команды; execlp получает путь поиска (т. е. $PATH) из окружения и осуществляет такой же поиск, как оболочка. Второй и последующие аргументы – это имя команды и ее параметры; они становятся массивом argv для новой программы. Конец списка помечен нулевым значением (чтобы понять конструкцию execlp, обратитесь к exec(2)).

Вызов execlp перекрывает существующую программу новой, запускает ее и выходит. Первичная программа получает управление обратно только в случае ошибки, например, если файл не найден или не является выполняемым:

```
execlp("date", "date", (char *) 0);
```

```
fprintf(stderr, "Couldn't execute 'date'\n");
exit(1);
```

Разновидность execlp, именуемая execvp, применяется в тех случаях, когда количество аргументов заранее не известно. Вызов выглядит так:

```
execvp(filename, argp);
```

где argp — это массив указателей на аргументы (как argv); последний указатель в массиве должен быть NULL, чтобы execvp имел возможность определить, где кончается список. Как и для execlp, filename — это файл, в котором находится программа, а argp — это массив argv для новой программы; argp[0] — это имя программы.

Ни одна из этих программ не допускает наличия метасимволов <, >, *, кавычек и т. д. в списке аргументов. Если это необходимо, вызывайте посредством execlp оболочку /bin/sh, которая выполнит всю работу. Сформируйте командную строку, которая будет содержать всю команду, как если бы она была напечатана на терминале, затем скажите:

```
execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

Аргумент -c определяет, что следующий аргумент должен рассматриваться как целая командная строка, а не как отдельный аргумент.

Рассмотрим в качестве иллюстрации программу waitfile. Команда

```
$ waitfile имя-файла [ команда ]
```

периодически проверяет указанный файл. Если он не изменился с момента последней проверки, то *команда* выполняется. Если *команда* не определена, то файл копируется на стандартное устройство вывода. Для мониторинга работы troff мы используем waitfile:

```
$ waitfile troff.out echo troff done &
```

Реализация waitfile извлекает время изменения файла при помощи fstat.

```
/* waitfile: ждет, пока файл не перестанет изменяться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int fd;
    struct stat stbuf;
    time_t old_time = 0;
```

```
    progname = argv[0];
    if (argc < 2)
        error("Usage: %s filename [cmd]", progname);
    if ((fd = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    fstat(fd, &stbuf);
    while (stbuf.st_mtime != old_time) {
        old_time = stbuf.st_mtime;
        sleep(60);
        fstat(fd, &stbuf);
    }
    if (argc == 2) {    /* копировать файл */
        execlp("cat", "cat", argv[1], (char *) 0);
        error("can't execute cat %s", argv[1]);
    } else {            /* запустить процесс */
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    exit(0);
}
```

Тут проиллюстрированы оба вызова: `execlp` и `execvp`.

Выбран именно такой подход, потому что он полезен для понимания, но приемлемы и другие варианты. Например, `waitfile` может просто завершаться, после того как файл перестал изменяться.

Упражнение 7.17. Измените программу `watchfile` (упражнение 7.12) таким образом, чтобы она имела те же свойства, что и `waitfile`: если нет параметра *команда*, она копирует файл; в ином случае выполняет команду. Могут ли `watchfile` и `waitfile` совместно использовать исходный код? Подсказка: `argv[0]`. □

Управление процессами – fork и wait

Следующий шаг – это восстановление управления после того, как программа выполнена с помощью `execlp` или `execvp`. Поскольку эти программы просто накладывают новую программу поверх старой, то чтобы сохранить первичную программу, необходимо сначала разделить ее на две копии; одна из них может быть перезаписана, в то время как вторая ожидает окончания новой, наложенной программы. Разделение осуществляется системным вызовом `fork`:

```
proc_id = fork();
```

программа разделяется на две копии, каждая из которых продолжает выполняться. Единственное различие между ними заключается в значении, возвращаемом `fork`, – это *идентификатор процесса*. Для одного из этих процессов (*дочернего*) `proc_id` равен нулю. Для другого (*родительского*) `proc_id` нулю не равен. Таким образом, элементарный способ вызвать другую программу и вернуться из нее выглядит так:

```
if (fork() == 0)
    execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

И этого вполне достаточно, если не считать обработки ошибок. `fork` создает две копии программы. Для дочернего процесса значение, возвращаемое `fork`, равно нулю, поэтому вызывается `execlp`, который выполняет командную строку `commandline` и умирает. Для родительского процесса `fork` возвращает ненулевую величину, поэтому `execlp` пропускается. (В случае наличия ошибок `fork` возвращает `-1`.)

Чаще всего родительская программа ждет, пока закончится дочерняя, прежде чем продолжить свое собственное выполнение. Это реализуется при помощи системного вызова `wait`:

```
int status;

if (fork() == 0)
    execlp(...);          /* дочерняя */
wait(&status);            /* родительская */
```

Здесь еще не обрабатываются никакие аномальные обстоятельства, такие как сбой в работе `execlp` или `fork` или возможность наличия нескольких одновременно выполняющихся дочерних процессов (`wait` возвращает идентификатор процесса для закончившегося дочернего процесса, так что можно сравнить его со значением, возвращаеммым `fork`). Наконец, этот фрагмент не имеет никакого отношения к обработке сколь бы то ни было странного поведения со стороны дочернего процесса. Однако эти три строки – основа стандартной функции `system`.

Значение переменной `status`, возвращаемое `wait`, содержит в младших восьми битах представление системы о коде завершения для дочернего процесса; ноль означает нормальное завершение, а ненулевое значение указывает, что возникли какие-либо проблемы. Следующие по старшинству восемь бит берутся из аргумента функции `exit` или оператора `return` в `main`, вызвавшего завершение дочернего процесса.

Когда программа вызывается оболочкой, создаются три дескриптора: `0`, `1` и `2`, которые указывают на соответствующие файлы, а все остальные файловые дескрипторы доступны для использования. Когда эта программа вызывает другую, правила этикета предписывают убедиться, что выполняются те же условия. Ни `fork`, ни `exec` никоим образом не влияют на открытые файлы; оба процесса – и родительский, и дочерний, имеют один и тот же набор открытых файлов. Если родительская программа буферизует вывод, который должен появиться раньше, чем вывод дочерней, то родительская программа должна сбросить содержимое буфера на диск до вызова `execlp`. И наоборот, если родительская программа буферизует входящий поток, то дочерняя потеряет информацию, прочитанную родителем. Вывод может быть сброшен на диск, но невозможно вернуть назад ввод. Эти соображения возникают, если ввод или вывод осуществляется при помощи стандартной

библиотеки ввода-вывода, описанной в главе 6, так как она обычно буферизует и ввод, и вывод.

Наследование дескрипторов файлов в `execlp` может «подвесить» систему: если вызывающая программа не имеет своего стандартного ввода и вывода, связанного с терминалом, то и вызываемая программа не будет его иметь. Возможно, что это как раз то, что нужно; например, в скрипте для редактора `ed` ввод для команды, запущенной с восклицательным знаком `!`, вероятно, будет производиться из скрипта. Даже в этом случае `ed` должен читать ввод посимвольно, чтобы избежать проблем с буферизацией.

Однако для интерактивных программ, таких как `p`, система должна повторно подключить стандартный ввод-вывод к терминалу. Один из способов сделать это – подключить их к `/dev/tty`.

Системный вызов `dup(fd)` дублирует дескриптор файла `fd` в незанятый дескриптор файла с наименьшим номером, возвращая новый дескриптор, который ссылается на тот же самый открытый файл. Этот код связывает стандартный ввод программы с файлом:

```
int fd;

fd = open("file", 0);
close(0);
dup(fd);
close(fd);
```

Вызов `close(0)` освобождает дескриптор `0` (стандартный ввод), но, как всегда, не влияет на родительскую программу.

Приведем версию `system` для интерактивных программ, использующую для сообщений об ошибках `progname`. Можете пока не обращать внимания на части функции, относящиеся к обработке сигналов; вернемся к ним в следующем разделе.

```
/*
 * Более надежная версия system для интерактивных программ
 */
#include <signal.h>
#include <stdio.h>

system(s)   /* выполнить командную строку s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();
    extern char *progname;

    fflush(stdout);
    tty = open("/dev/tty", 2);
    if (tty == -1) {
        fprintf(stderr, "%s: can't open /dev/tty\n", progname);
        return -1;
```

```
    }
    if ((pid = fork()) == 0) {
        close(0); dup(tty);
        close(1); dup(tty);
        close(2); dup(tty);
        close(tty);
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    close(tty);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

Обратите внимание на то, что `/dev/tty` открывается в режиме `2` (чтения и записи), а затем дублируется для создания стандартного ввода и вывода. Именно так система назначает стандартные устройства ввода, вывода и ошибок при входе в нее. Поэтому можно выводить данные в стандартный ввод:

```
$ echo hello 1>&0
hello
$
```

Это означает, что можно было скопировать файловый дескриптор `2`, чтобы повторно подключить стандартный ввод-вывод, но корректнее и надежнее будет открыть `/dev/tty`. Даже у этого варианта `system` есть потенциально возможные проблемы: файлы, открытые в вызывающей программе, как, например, `tty` в программе `ttyin` в `p`, будут переданы дочернему процессу.

Урок не в том, что именно представленная в этом разделе версия `system` должна использоваться во всех ваших программах (она, например, не подошла бы для неинтерактивного `ed`), а в том, чтобы понять, как управлять процессами и как корректно использовать базовые возможности; значение слова «корректно» зависит от конкретного приложения и может не совпадать со стандартной реализацией `system`.

7.5. Сигналы и прерывания

В этом разделе поэтапно рассмотрим процесс обработки сигналов (таких как прерывания), поступающих из внешнего мира, а также ошибок программы. Ошибки программ возникают в основном из-за непра-

вильных обращений к памяти, при выполнении специфических инструкций или из-за операций с плавающей точкой. Наиболее распространенные сигналы, поступающие из внешнего мира: *прерывание* (*interrupt*) – этот сигнал посылается, когда вы нажимаете клавишу *DEL*; *выход* (*quit*) – порождается символом FS (*ctl-*); *отключение* (*hangup*) – вызван тем, что повешена телефонная трубка, и *завершение* (*terminate*) – порождается командой `kill`. Когда происходит одно из вышеуказанных событий, сигнал посылается всем процессам, запущенным с данного терминала, и если не существует соглашений, предписывающих иное, сигнал завершает процесс. Для большинства сигналов создается дамп памяти, который может потребоваться для отладки. (См. `adb(1)` и `sdb(1)`.)

Системный вызов `signal` изменяет действие, выполняемое по умолчанию. Он имеет два аргумента: первый – это номер, который определяет сигнал, второй – это или адрес функции, или же код, предписывающий игнорировать сигнал или восстанавливать действия по умолчанию. Файл `<signal.h>` содержит описания различных аргументов. Так,

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

приводит к игнорированию прерывания, в то время как

```
signal(SIGINT, SIG_DFL);
```

восстанавливает действие по умолчанию – завершение процесса. Во всех случаях `signal` возвращает предыдущее значение сигнала. Если второй аргумент – это имя функции (которая должна быть объявлена в этом же исходном файле), то она будет вызвана при возникновении сигнала. Чаще всего эта возможность используется для того, чтобы позволить программе подготовиться к выходу, например удалить временный файл:

```
#include <signal.h>
char *tempfile = "temp.XXXXXX";

main()
{
    extern onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    mktemp(tempfile);

    /* Обработка ... */

    exit(0);
}

onintr()  /* очистить в случае прерывания */
{
```

```
    unlink(tempfile);
    exit(1);
}
```

Зачем нужны проверка и повторный вызов `signal` в `main`? Вспомните, что сигналы посылаются во *все* процессы, запущенные на данном терминале. Соответственно, когда программа запущена не в интерактивном режиме (а с помощью `&`), командный процессор позволяет ей игнорировать прерывания, таким образом, программа не будет остановлена прерываниями, предназначенными для не фоновых процессов. Если же программа начинается с анонсирования того, что все прерывания должны быть посланы в `onintr`, невзирая ни на что, это сводит на нет попытки командного процессора защитить программу, работающую в фоновом режиме.

Решение, представленное выше, позволяет проверить состояние управления прерываниями и продолжать игнорировать прерывания, если они игнорировались ранее. Код учитывает тот факт, что `signal` возвращает предыдущее состояние конкретного сигнала. И если сигналы ранее игнорировались, процесс будет и далее их игнорировать; в противном случае они должны быть перехвачены.

В более сложной программе может потребоваться перехватить прерывание и интерпретировать его как запрос на отмену выполняемой операции и возврат в ее собственный цикл обработки команд. Возьмем, например, текстовый редактор: прерывание слишком долгой печати не должно приводить к выходу из программы и потере всей сделанной работы. В этой ситуации можно написать такой код:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    setjmp(sjbuf); /* сохранение текущей позиции в стеке*/
for (;;) {
    /* основной цикл обработки */
    }
    ...
}

onintr() /* переустановить в случае прерывания */
{
    signal(SIGINT, onintr); /* переустановить для следующего прерывания */
    printf("\nInterrupt\n");
    longjmp(sjbuf, 0);      /* возврат в сохраненное состояние */ }
```

Файл setjmp.h объявляет тип jmp_buf как объект, в котором может сохраняться положение стека; sjbuf объявляется как объект такого типа. Функция setjmp(3) сохраняет запись о месте выполнения программы. Значения переменных не сохраняются. Когда происходит прерывание, инициируется обращение к программе onintr, которая может напечатать сообщение, установить флаги или сделать что-либо другое. Функция longjmp получает объект, сохраненный в setjmp, и возвращает управление в точку программы, следующую за вызовом setjmp. Таким образом, управление (и положение стека) возвращаются к тому месту основной программы, где происходит вход в основной цикл.

Обратите внимание на то, что сигнал снова устанавливается в onintr, после того как произойдет прерывание. Это необходимо, так как сигналы при их получении автоматически восстанавливают действие по умолчанию.

Некоторые программы просто не могут быть остановлены в произвольном месте, например в процессе обработки сложной структуры данных, поэтому необходимо иметь возможность обнаруживать сигналы. Возможно следующее решение – надо сделать так, чтобы программа обработки прерываний установила флаг и возвратилась обратно вместо того, чтобы вызывать exit или longjmp. Выполнение будет продолжено с того самого места, в котором оно было прервано, а флаг прерывания может быть проверен позже.

С таким подходом связана одна трудность. Предположим, что программа читает с терминала в то время, когда послано прерывание. Надлежащим образом вызывается указанная подпрограмма, которая устанавливает флаги и возвращается обратно. Если бы дело действительно обстояло так, как было указано выше, то есть выполнение программы возобновлялось бы «с того самого места, где оно было прервано», то программа должна была бы продолжать читать с терминала до тех пор, пока пользователь не напечатал бы новую строку. Такое поведение может сбивать с толку, ведь пользователь может и не знать, что программа читает, и он, вероятно, предпочел бы, чтобы сигнал вступал в силу незамедлительно. Чтобы разрешить эту проблему, система завершает чтение, но со статусом ошибки, который указывает, что произошло; errno устанавливается в EINTR, определенный в errno.h, чтобы обозначить прерванный системный вызов.

Поэтому программы, которые перехватывают сигналы и возобновляют работу после них, должны быть готовы к «ошибкам», вызванным прерванными системными вызовами. (Системные вызовы, по отношению к которым надо проявлять осторожность, – это чтение с терминала, ожидание и пауза). Такая программа может использовать код, приведенный ниже, для чтения стандартного ввода:

```
#include <errno.h>
extern int errno;
```

```
...
if (read(0, &c, 1) <= 0)  /* EOF или прерывание */
    if (errno == EINTR) { /* EOF, вызванный прерыванием */
        errno = 0; /* переустановить для следующего раза */
        ...
    } else {       /* настоящий конец файла */
        ...
    }
```

И последняя тонкость, на которую надо обратить внимание, если перехват сигналов сочетается с выполнением других программ. Предположим, что программа обрабатывает прерывания и, к тому же, содержит метод (как ! в ed), посредством которого могут выполняться другие программы. Тогда код будет выглядеть примерно так:

```
if (fork() == 0)
    execlp(...);
signal(SIGINT, SIG_IGN); /* предок игнорирует прерывания */
wait(&status);           /* пока выполняется потомок */
signal(SIGINT, onintr);  /* восстановить прерывания */
```

Почему именно так? Сигналы посылаются всем вашим процессам. Предположим, что программа, которую вы вызвали, обрабатывает свои собственные прерывания, как это делает редактор. Если вы прерываете дочернюю программу, она получит сигнал и вернется в свой основной цикл, и, вероятно, прочитает ваш терминал. Но вызывающая программа также выйдет из состояния ожидания дочерней программы и прочитает ваш терминал. Наличие двух процессов чтения терминала все запутывает, так как на самом деле система «подкидывает монетку», чтобы решить, какая программа получит каждую из строк ввода. Чтобы избежать этого, родительская программа должна игнорировать прерывания до тех пор, пока не выполнится дочерняя. Это умозаключение отражено в обработке сигналов в system:

```
#include <signal.h>

system(s)   /* выполнить командную строку s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();

    ...
    if ((pid = fork()) == 0) {
        ...
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    ...
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
```

```
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

В отступление от описания, функция signal очевидно имеет несколько странный второй аргумент. На самом деле это указатель на функцию, которая возвращает целое число, и это также тип самой функции signal. Два значения, SIG_IGN и SIG_DFL, имеют правильный тип, но выбираются таким образом, чтобы они не совпадали ни с какими возможными реальными функциями. Для особо интересующихся приведем пример того, как они описываются для PDP-11 и VAX; описания должны быть достаточно отталкивающими, чтобы побудить к использованию signal.h.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

Сигналы alarm

Системный вызов alarm(*n*) вызывает отправку вашему процессу сигнала SIGALRM через *n* секунд. Сигнал alarm может применяться для того, чтобы убедиться, что нечто произошло в течение надлежащего промежутка времени если что-то произошло, SIGALRM может быть выключен, если же нет, то процесс может вернуть управление, получив сигнал alarm.

Чтобы пояснить ситуацию, рассмотрим программу, называемую timeout, она запускает некоторую другую команду; если эта команда не закончилась к определенному времени, она будет аварийно прервана, когда alarm выключится. Например, вспомните команду watchfor из главы 5. Вместо того чтобы запускать ее на неопределенное время, можно установить часовой лимит:

```
$ timeout -3600 watchfor dmg &
```

Код в timeout иллюстрирует практически все, о чем говорилось в двух предыдущих разделах. Потомок создан; предок устанавливает аварийный сигнал и ждет, пока потомок закончит свою работу. Если alarm приходит раньше, то потомок уничтожается. Предпринимается попытка вернуть статус выхода потомка.

```
/* timeout: устанавливает временное ограничение для процесса */
#include <stdio.h>
#include <signal.h>
int pid;        /* идентификатор дочернего процесса */
```

```
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int sec = 10, status, onalarm();

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        sec = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc < 2)
        error("Usage: %s [-10] command", progname);
    if ((pid=fork()) == 0) {
        execvp(argv[1], &argv[1]);
        error("couldn't start %s", argv[1]);
    }
    signal(SIGALRM, onalarm);
    alarm(sec);
    if (wait(&status) == -1 || (status & 0177) != 0)
        error("%s killed", argv[1]);
    exit((status >> 8) & 0377);
}

onalarm()   /* завершить дочерний процесс в случае получения alarm */
{
    kill(pid, SIGKILL);
}
```

Упражнение 7.18. Можете ли вы предположить, как реализован `sleep`? Подсказка: `pause(2)`. При каких условиях (если такие условия существуют) `sleep` и `alarm` могут создавать помехи друг для друга? □

История и библиография

В книге не представлено подробного описания реализации системы UNIX, в частности из-за того, что существуют имущественные права на код. Доклад Кена Томпсона (Ken Thompson) «UNIX implementation» (Реализация UNIX), изданный в «BSTJ» в июле 1978 года, описывает основные идеи. Эту же тему поднимают статьи «The UNIX system – a retrospective» (Система UNIX в ретроспективе) в том же номере «BSTJ» и «The evolution of the UNIX time-sharing system» (Эволюция UNIX – системы разделения времени), напечатанная в материалах Symposium on Language Design and Programming Methodology в журнале издательства Springer-Verlag «Lecture Notes in Computer Science» № 79 в 1979 году. Оба труда принадлежат перу Денниса Ритчи (Dennis Ritchie).

Программа `readslow` была придумана Питером Вейнбергером (Peter Weinberger) в качестве простого средства для демонстрации зрителям игры шахматной программы Belle Кена Томсона и Джо Кондона (Joe Condon) во время шахматного турнира. Belle записывала состояние игры в файл; наблюдатели опрашивали файл с помощью `readslow`, чтобы не занимать слишком много драгоценных циклов. (Новая версия оборудования для Belle осуществляет небольшие расчеты на своей главной машине, поэтому больше такой проблемы не существует.)

Том Дафф (Tom Duff) вдохновил нас на написание `spname`. Статья Айвора Дерхема (Ivor Durham), Дэвида Лэмба (David Lamb) и Джеймса Сакса (James Saxe) «Spelling correction in user interfaces» (Проверка орфографии в пользовательских интерфейсах), изданная CACM в октябре 1983 года, представляет несколько отличающийся от привычного проект реализации исправления орфографических ошибок в контексте почтовой программы.