Николай Иванов

Самоучитель Программирование в Linux

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
И20

Иванов Н. Н.

И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил.

ISBN 978-5-9775-0744-8

Рассмотрены фундаментальные основы программирования в Linux: инструментарий, низкоуровневый ввод-вывод, многозадачность, файловая система, межпроцессное взаимодействие и обработка ошибок. Книга главным образом ориентирована на практическое применение изложенных концепций. В ней есть все, что нужно начинающим, а углубленное изучение каждой темы делает ее ценной и для опытных программистов. Каждая тема проиллюстрирована большим числом примеров на языках C и C++ и Python, которые читатель сможет использовать в качестве образцов для собственных программ. На FTP-сервере издательства находятся исходные тексты программ.

Во втором издании материал актуализирован с учетом современных тенденций, добавлены 3 новые главы по программированию в Linux на языке Python, устранены замеченные ошибки.

Для начинающих и опытных Linux-программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Подписано в печать 05.10.11.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 32,25.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0744-8

Оглавление

Введение

Предисловие

В современном мире операционная система Linux играет особую роль. Некогда считалось, что "внутренности" операционных систем могут быть доступны и понятны только особому кругу избранных. Благодаря Linux, созданной тысячами увлеченных программистов-добровольцев, мир увидел обратную сторону медали. Теперь каждый желающий может свободно изучать или изменять исходный код операционной системы. Если вы тоже желаете взглянуть на Linux изнутри, то надеюсь, что эта книга станет вашим верным помощником.

Эта книга дает базовые универсальные знания и умения, не связанные с конкретными областями программирования, но необходимые любому Linux-программисту в равной степени. Цель книги — донести до читателя понимание концепций файлов и процессов в Linux, научить самостоятельно работать с данной системой и развивать ее.

Условные обозначения

Новые термины, а также слова и фразы, на которые следует обратить особое внимание, выделяются в книге *курсивом*. Адреса сети Интернет выделяются жирным шрифтом, например: **http://www.kernel.org**. Элементы листингов (имена функций и переменных, типы данных, именованные константы и т. д.) помечаются в основном тексте `пропорциональным шрифтом`.

В примерах интерактивной работы с оболочкой символ `$` (доллар) обозначает приглашение командной строки:

```
$ uname
Linux
```

Листинги программ представлены в виде обособленных блоков.

Листинг 0.1. Пример helloworld.c

```
#include <stdio.h>

int main (void)
{
       printf ("Hello World!\n");
       return 0;
}
```

Примеры программ

Исходные тексты рассмотренных в книге программ на языках C, C++ и Python можно скачать по ссылке **ftp://85.249.45.166/ 9785977507448.zip**, а также на странице книги на сайте **www.bhv.ru**.

Для отыскания файла, соответствующего конкретному листингу, используйте вспомогательную табл. П4.1 в *приложении 4*.

Благодарности

Хочу выразить глубокую признательность всем, кто помогал мне в работе над этой книгой. Это, в первую очередь, мои родные и близкие, которые с пониманием относились к непрерывному стуку клавиш и всячески поддерживали меня: жена Аня, мама Юлия Петровна, теща Татьяна Германовна. Еще хочу поблагодарить моих друзей Диму К., Надю К., Иру С., Сашу М., Иру Х. и Максима Ш., которые время от времени напоминали мне о том, что жизнь состоит не только из работы.

Отдельное спасибо руководителю проекта издательства "БХВ-Петербург" Евгению Рыбакову, предложившему мне написать эту книгу, терпеливо отвечавшему на мои "глупые" вопросы и оказавшему помощь в решении некоторых проблем технического характера.

Благодарю парней из форума LinuxForum.com, которые помогли мне в отладке примера на C++. Спасибо Ричарду Столлману, основателю проекта GNU, Линусу Торвальдсу, написавшему ядро Linux, и создателям текстового процессора OpenOffice.org Writer, в котором была написана эта книга.

Если вы не знакомы с языком C++, но хотите на нем программировать, то лучше всего начать с книги "Самоучитель C++" Герберта Шилдта, признанного автора книг по языку C++.

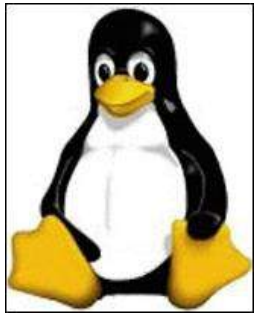
Обратная связь

Вы держите в руках второе издание книги, в котором были исправлены некоторые ошибки, а материал был переработан в соответствии с современным положением дел в области программирования для Linux. Кроме того, книга была дополнена но-

вой частью с главами, посвященными программированию в Linux на языке Python. В первых 28 главах книги мы будем писать программы на "родном" для Linux и проверенном годами языке C, а в *главах 29—31* опробуем свои силы в языке Python.

Много лет назад я узнал об одном из вечных законов программирования, согласно которому даже в самой простой и тщательно отлаженной программе имеется хотя бы одна ошибка. Мой опыт и опыт моих коллег доказывает, что это действительно так. Поэтому я прошу вас, уважаемый читатель, не верить в мою безупречность и сообщать мне об ошибках и опечатках, допущенных в книге. Это поможет сделать последующие переиздания лучше и качественнее.

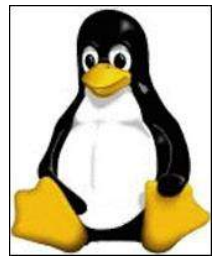
Я не люблю критику, но с удовольствием выслушаю ваши идеи, пожелания и дополнения, относящиеся к этой книге. Пишите мне по адресу **nnivanov@mail.ru**. Я постараюсь ответить каждому.



ЧАСТЬ I

Основы программирования в Linux

ГЛАВА 1



Создание программы

В начале книги рассматриваются технические вопросы создания программ в Linux. Процесс программирования обычно разделяют на несколько этапов, содержание которых определяется поставленной задачей. Прохождение каждого такого этапа требует наличия определенных инструментов, совокупный набор которых называется *инструментарием*. Эта глава описывает базовый инструментарий Linux-программиста, пишущего на языке C: *текстовый редактор*, *компилятор* и *компоновщик* — наиболее часто используемые инструменты создания программ. В последующих главах книги инструментарий будет постепенно дополняться новыми элементами.

Обычно программисты придерживаются некоторых общих приемов написания программ — *идиом программирования*. Описанная в *разд. 1.4* концепция создания многофайловых проектов — одна из таких идиом.

1.1. Исходный код

Создание любой программы начинается с постановки задачи, проектирования и написания исходного кода (source code). Обычно исходный код программы записывается в один или несколько файлов, которые называют *исходными файлами* или *исходниками*.

> ***Примечание***
>
> Иногда под словосочетанием "исходный код" понимают совокупность всех исходных файлов конкретного проекта (например, "исходный код ядра Linux").

Исходные файлы обычно создаются и набираются в текстовом редакторе. В принципе, для написания исходных кодов подойдет любой текстовый редактор. Но желательно, чтобы это был редактор с "подсветкой" синтаксиса, т. е. выделяющий визуально ключевые слова используемого языка программирования. В результате исходный код становится более наглядным, а программист делает меньше опечаток и ошибок.

В современных дистрибутивах Linux представлен большой выбор текстовых редакторов. Наибольшей популярностью среди программистов пользуются редакторы двух семейств:

- *vi* (Visual Interface) — полноэкранный редактор, созданный Биллом Джоем (Bill Joy) в 1976 г. С тех пор было написано немало клонов vi. Практически все Unix-подобные системы комплектуются той или иной версией этого текстового редактора. Наиболее популярные клоны vi в Linux — vim (Vi IMproved), Elvis и nvi;
- *Emacs* (Editor MACroS) — текстовый редактор, разработанный Ричардом Столлманом (Richard Stallman). Из всех существующих версий Emacs наиболее популярными являются GNU Emacs и XEmacs.

Среди других распространенных в Linux редакторов следует отметить pico (PIne COmposer), jed и mcedit (Midnight Commander EDITor). Они не обладают мощью vi или Emacs, но достаточно просты и удобны в использовании. В Linux также имеется множество текстовых редакторов с графическим интерфейсом: kate, gedit, nedit, bluefish, jedit (этот список можно продолжать очень долго). Редакторы vim и GNU Emacs тоже имеют собственные графические расширения.

Обычно программирование начинается с примера, выводящего на экран приветствие "Hello World!". Отступим от этой давней традиции и напишем сразу что-нибудь полезное, например программу часов.

Для начала создайте в своем текстовом редакторе файл myclock.c (листинг 1.1).

Листинг 1.1. Файл myclock.c

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

int main (void)
{
        time_t nt = time (NULL);
        printf ("%s", ctime (&nt));
        return 0;
}
```

Это исходный код нашей первой программы. Рассмотрим его по порядку:

1. Заголовочный файл stdio.h делает доступными механизмы ввода-вывода стандартной библиотеки языка C. Нам он нужен для вызова функции `printf()`.
2. Заголовочный файл time.h включается в программу, чтобы сделать доступными функции `time()` и `ctime()`, работающие с датой/временем.
3. Собственно программа начинается с функции `main()`, в теле которой создается переменная `nt`, имеющая тип `time_t`. Переменные этого типа предназначены для хранения числа секунд, прошедших с начала *эпохи отсчета компьютерного времени* (полночь 1 января 1970 г.).
4. Функция `time()` заносит в переменную `nt` текущее время.

5. Функция `ctime()` преобразовывает время, исчисляемое в секундах от начала *эпохи* (Epoch), в строку, содержащую привычную для нас запись даты и времени.
6. Полученная строка выводится на экран функцией `printf()`.
7. Инструкция `return 0;` осуществляет выход из программы.

1.2. Компиляция

Чтобы запустить программу, ее необходимо сначала перевести с понятного человеку исходного кода в понятный компьютеру исполняемый код. Такой перевод называется *компиляцией* (compilation).

Чтобы откомпилировать программу, написанную на языке C, нужно "пропустить" ее исходный код через *компилятор*. В результате получается *исполняемый* (бинарный) код. Файл, содержащий исполняемый код, обычно называют *исполняемым файлом* или *бинарником* (binary).

> ***Замечание***
>
> Компилятор не всегда генерирует код, готовый к непосредственному выполнению. Эти случаи будут рассмотрены в *разд. 1.3*.

Компилятором языка C в Linux обычно служит программа gcc (GNU C Compiler) из пакета компиляторов GCC (GNU Compiler Collection). Чтобы откомпилировать нашу программу (листинг 1.1), следует вызвать gcc, указав в качестве аргумента имя исходного файла:

```
$ gcc myclock.c
```

Если компилятор не нашел ошибок в исходном коде, то в текущем каталоге появится файл a.out. Теперь, чтобы выполнить программу, требуется указать командной оболочке путь к исполняемому файлу. Поскольку текущий каталог обычно обозначается точкой, то запуск программы можно осуществить следующим образом:

```
$ ./a.out
Wed Nov  8 03:09:01 2006
```

Исполняемые файлы программ обычно располагаются в каталогах, имена которых перечислены через двоеточие в особой переменной `PATH`. Чтобы просмотреть содержимое этой переменной, введите следующую команду:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/qt4/bin
```

Если бинарник находится в одном из этих каталогов, то для запуска программы достаточно ввести ее имя (например, ls). В противном случае потребуется указание пути к исполняемому файлу.

Имя a.out не всегда подходит для программы. Один из способов исправить положение — просто переименовать полученный файл:

```
$ mv a.out myclock
```

Но есть способ лучше. Можно запустить компилятор с опцией `-o`, которая позволяет явно указать имя файла на выходе:

```
$ gcc -o myclock myclock.c
```

Наша программа не содержит синтаксических ошибок, поэтому компилятор молча "проглатывает" исходный код. Проведем эксперимент, нарочно испортив программу. Для этого уберем в исходном файле первую инструкцию функции `main()`, которая объявляет переменную `nt`. Теперь снова попробуем откомпилировать полученный исходный код:

```
$ gcc -o myclock myclock.c
myclock.c: In function 'main':
myclock.c:6: error: 'nt' undeclared (first use in this function)
myclock.c:6: error: (Each undeclared identifier is reported only once
myclock.c:6: error: for each function it appears in.)
```

> ***Замечание***
>
> Комментарии используются программистами не только для пояснений и заметок. Перед компиляцией комментарии исключаются из исходного кода, как если бы их вообще не было. Поэтому программу myclock можно "испортить", просто закомментировав первую инструкцию функции `main()`.

Как и ожидалось, компиляция не удалась. Обратите внимание, что находящийся в текущем каталоге исполняемый файл myclock — это "детище" предыдущей компиляции. Новый файл не был создан.

> ***Замечание***
>
> Иногда говорят, что "компилятор ругается". Это программистский жаргон, означающий, что компиляция не удалась из-за ошибок в исходном коде.

Очень важно научиться понимать сообщения об ошибках, выводимых компилятором. В нашем случае сообщается, что произошло "нечто" в файле myclock.c внутри функции `main()`. Далее говорится, что строка номер 6 содержит ошибку (error): переменная `nt` не была объявлена к моменту ее первого использования в данной функции. В последних двух строках приводится пояснение: для каждой функции, где встречается необъявленный идентификатор (имя), сообщение об ошибке выводится только один раз.

Иногда вместо ошибки (error) выдается предупреждение (warning). В этом случае компиляция не останавливается, но до сведения программиста доводится информация о потенциально опасной конструкции исходного кода.

Теперь верните недостающую строку обратно или раскомментируйте, поскольку файл myclock.c нам еще понадобится.

1.3. Компоновка

В предыдущем разделе говорилось о том, что компилятор переводит исходный код программы в исполняемый. Но это не всегда так.

В достаточно объемных программах исходный код обычно разделяется для удобства на несколько частей, которые компилируются отдельно, а затем соединяются воедино. Каждый такой "кусок" содержит *объектный код* и называется *объектным модулем*.

Объектные модули записываются в *объектные файлы*, имеющие расширение `.o`. В результате объединения объектных файлов могут получаться исполняемые файлы (обычные запускаемые бинарники), а также библиотеки, о которых пойдет речь в *главе 4*.

Для объединения объектных файлов служит *компоновщик (линковщик)*, а сам процесс называют *компоновкой* или *линковкой*. В Linux имеется компоновщик GNU ld, входящий в состав пакета GNU binutils.

> ***Примечание***
>
> Иногда компоновщик называют также *загрузчиком*.

Ручная компоновка объектных файлов — довольно неприятный процесс, требующий передачи программе ld большого числа параметров, зависящих от многих факторов. К счастью, компиляторы из коллекции GCC сами вызывают линковщик с нужными параметрами, когда это необходимо.

> ***Примечание***
>
> Программист может самостоятельно передавать компоновщику дополнительные параметры через компилятор. Эта возможность будет рассмотрена в *главе 4*.

Теперь вернемся к нашему примеру (листинг 1.1). В предыдущем разделе компилятор "молча" вызвал компоновщик, в результате чего получился исполняемый файл. Чтобы отказаться от автоматической компоновки, нужно передать компилятору опцию `-c`:

```
$ gcc -c myclock.c
```

Если компилятор не нашел ошибок, то в текущем каталоге должен появиться объектный файл myclock.o. Других объектных файлов у нас нет, поэтому будем компоновать только его. Это делается очень просто:

```
$ gcc -o myclock myclock.o
```

> ***Замечание***
>
> В UNIX существуют различные форматы объектных файлов. Наиболее популярные среди них — a.out (Assembler OUTput) и COFF (Common Object File Format). В Linux чаще всего встречается открытый формат объектных и исполняемых файлов ELF (Executable and Linkable Format).

1.4. Многофайловые проекты

Современные программные проекты редко ограничиваются одним исходным файлом. Распределение исходного кода программы на несколько файлов имеет ряд существенных преимуществ перед однофайловыми проектами.

- Использование нескольких исходных файлов накладывает на *репозиторий* (рабочий каталог проекта) определенную логическую структуру. Такой код легче читать и модернизировать.
- В однофайловых проектах любая модернизация исходного кода влечет повторную компиляцию всего проекта. В многофайловых проектах, напротив, достаточно откомпилировать только измененный файл, чтобы обновить проект. Это экономит массу времени.
- Многофайловые проекты позволяют реализовывать одну программу на разных языках программирования.
- Многофайловые проекты позволяют применять к различным частям программы разные лицензионные соглашения.

Обычно процесс сборки многофайлового проекта осуществляется по следующему алгоритму:

1. Создаются и подготавливаются исходные файлы. Здесь есть одно важное замечание: каждый файл должен быть целостным, т. е. не должен содержать незавершенных конструкций. Функции и структуры не должны разрываться. Если в рамках проекта предполагается создание исполняемой программы, то в одном из исходных файлов должна присутствовать функция `main()`.
2. Создаются и подготавливаются заголовочные файлы. У заголовочных файлов особая роль: они устанавливают соглашения по использованию общих идентификаторов (имен) в различных частях программы. Если, например, функция `func()` реализована в файле a.c, а вызывается в файле b.c, то в оба файла требуется включить директивой `#include` заголовочный файл, содержащий объявление (прототип) нашей функции. Технически можно обойтись и без заголовочных файлов, но в этом случае функцию можно будет вызвать с произвольными аргументами, и компилятор, за отсутствием соглашений, не выведет ни одной ошибки. Подобный "слепой" подход потенциально опасен и в большинстве случаев свидетельствует о плохом стиле программирования.
3. Каждый исходный файл отдельно компилируется с опцией `-c`. В результате появляется набор объектных файлов.
4. Полученные объектные файлы соединяются компоновщиком в одну исполняемую программу.

Если необходимо скомпоновать несколько объектных файлов (`OBJ1.o`, `OBJ2.o` и т. д.), то применяют следующий простой шаблон:

```
$ gcc -o OUTPUT_FILE OBJ1.o OBJ2.o ...
```

Рассмотрим программу, которая принимает в качестве аргумента строку и переводит в ней все символы в верхний регистр, т. е. заменяет все строчные буквы на заглавные. Чтобы не усложнять пример, будем преобразовывать только латинские (англоязычные) символы.

Для начала создадим файл print_up.h (листинг 1.2), в котором будет находиться объявление (прототип) функции `print_up()`. Эта функция переводит символы строки в верхний регистр и выводит полученный результат на экран.

Листинг 1.2. Файл print_up.h

```
void print_up (const char * str);
```

Итак, объявление функции `print_up()` устанавливает соглашение, по которому любой исходный файл, включающий print_up.h директивой `#include`, обязан вызвать функцию `print_up()` с одним и только одним аргументом типа `const char*`. Теперь создадим файл print_up.c (листинг 1.3), в котором будет находиться тело функции `print_up()`.

Листинг 1.3. Файл print_up.c

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "print_up.h"

void print_up (const char * str)
{
        int i;
        for (i = 0; i < strlen (str); i++)
                printf ("%c", toupper (str[i]));

        printf ("\n");
}
```

Функция `print_up()` просматривает в цикле всю строку, посимвольно преобразовывая ее в верхний регистр. Вывод также производится посимвольно. В заключение выводится символ новой строки. Функция `toupper()`, объявленная в файле ctype.h и являющаяся частью стандартной библиотеки языка C, возвращает переданный ей символ в верхнем регистре, если это возможно. Без дополнительных манипуляций эта функция не работает с кириллическими (русскоязычными) символами, но сейчас это не важно. Теперь создадим третий файл main.c (листинг 1.4), который будет содержать функцию `main()`, необходимую для компоновки и запуска программы.

Листинг 1.4. Файл main.c

```
#include <string.h>
#include <stdio.h>
#include "print_up.h"

int main (int argc, char ** argv)
{
        if (argc < 2) {
                fprintf (stderr, "Wrong arguments\n");
                return 1;
        }
```

```
        print_up (argv[1]);
        return 0;
}
```

Сначала вспомним, что аргументы командной строки передаются в программу через функцию `main()`:

- `argc` — целое число, содержащее количество аргументов командной строки;
- `argv` — массив строк (двумерный массив символов), в котором находятся аргументы.

Следует помнить, что в первый аргумент командной строки обычно заносится имя программы. Таким образом, `argv[0]` — это имя программы, `argv[1]` — первый переданный при запуске аргумент, `argv[2]` — второй аргумент и т. д. до элемента `argv[argc-1]`.

Вообще говоря, аргументы в программу можно передавать не только из командной оболочки. Поэтому понятие "аргументы командной строки" не всегда отражает действительное положение вещей. В связи с этим, чтобы избежать неоднозначности, будем в дальнейшем пользоваться более точной формулировкой "аргументы программы".

> ***Примечание***
>
> Об аргументах программы и способах их обработки будет подробно рассказано в *главе 5*.

Теперь нужно собрать проект воедино. Сначала откомпилируем каждый файл с расширением `.c`:

```
$ gcc -c print_up.c
$ gcc -c main.c
```

В результате компиляции в репозитории программы должны появиться объектные файлы print_up.o и main.o, которые следует скомпоновать в один бинарный файл:

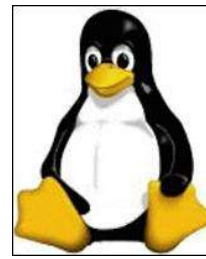
```
$ gcc -o printup print_up.o main.o
```

> ***Примечание***
>
> Если компилятор gcc вызывается с опцией `-c`, но без опции `-o`, то имя выходного файла получается заменой расширения `.c` на `.o`. Например, из файла foo.c получается файл foo.o.

Осталось только запустить и протестировать программу:

```
$ ./printup
Wrong arguments
$ ./printup Hello
HELLO
```

Обратите внимание на то, что заголовочный файл print_up.h не компилируется. Заголовочные файлы вообще никогда отдельно не компилируются. Дело в том, что на стадии *препроцессирования* (условно первая стадия компиляции) все директивы `#include` заменяются на содержимое указанных в них файлов.



ГЛАВА 2

Автосборка

Сборкой называется процесс подготовки программы к непосредственному использованию. Простейший пример сборки — компиляция и компоновка. Более сложные проекты могут также включать в себя дополнительные промежуточные этапы (операции над файлами, конфигурирование и т. п.). Существуют также языки программирования, позволяющие запускать программы сразу после подготовки исходного кода, минуя стадию сборки. Примером такого языка является Python, о котором речь пойдет в *главах 29—31*.

В этой главе описывается инструментарий для автоматической сборки программных проектов в Linux, написанных на языках семейства C/C++. Отдельно рассмотрены некоторые идиомы, связанные с процессом автосборки.

2.1. Обзор средств автосборки в Linux

В предыдущей главе рассматривался простейший многофайловый проект. Для его сборки нам приходилось сначала компилировать каждый исходник, а затем компоновать полученные объектные файлы в единый бинарник.

Собирать программы вручную неудобно, поэтому программисты, как правило, прибегают к различным приемам, позволяющим автоматизировать этот процесс. Самый простой способ — написать сценарий оболочки (shell-скрипт), который будет автоматически выполнять все то, что вы обычно вводите вручную. Тогда многофайловый проект из предыдущей главы можно собрать, например, при помощи скрипта, приведенного в листинге 2.1.

Листинг 2.1. Скрипт make_printup

```
#!/bin/sh
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Примечание

Любая командная оболочка является также интерпретатором собственного языка программирования. В результате ей можно передавать набор команд в виде одного файла. Подобные файлы называются *скриптами (или сценариями) оболочки*. Каждый такой сценарий начинается с последовательности символов `#!` (решетка и восклицательный знак), после которой следует (без пробела) полный путь к исполняемому файлу оболочки, под которой будет выполняться скрипт. Такую строку называют shebang или hashbang. В Linux ссылка /bin/sh обычно указывает на оболочку bash.

Теперь файлу make_printup необходимо дать права на выполнение:

```
$ chmod +x make_printup
```

Примечание

О правах доступа будет подробно рассказано в *главах 7 и 13*.

Осталось только вызвать скрипт, и проект будет создан:

```
$ ./make_printup
```

На первый взгляд, все прекрасно. Но настоящий программист должен предвидеть все возможные проблемы, и при детальном рассмотрении перспективы использования скрипта оболочки для сборки проекта уже не кажутся такими радужными. Перечислим некоторые проблемы.

- Скрипты оболочки статичны. Их работа не зависит от состояния текущей задачи. Даже если нужно заново откомпилировать только один файл, скрипт будет собирать проект "с нуля".
- В скриптах плохо просматриваются связи между различными элементами проекта.
- Скрипты не обладают возможностью самодиагностики.

К счастью, Linux имеет в наличии достаточно большой арсенал специализированных средств для автоматической сборки программных проектов. Такие средства называют *автосборщиками* или *утилитами автоматической сборки*. Благодаря специализированным автосборщикам программист может сосредоточиться, собственно, на программировании, а не на процессе сборки.

Все утилиты автосборки в Linux можно разделить на несколько условных категорий.

- Семейство make — это различные реализации стандартного для Unix-подобных систем автосборщика make. Из представителей данного семейства наибольшей популярностью в Linux пользуются утилиты GNU make, imake, pmake, cmake, fastmake и tmake.
- Надстройки над make — утилиты семейства make работают с особыми файлами, в которых содержится вся информация о сборке проекта. Такие файлы называют make-файлами (makefiles). При работе с классическими make-утилитами эти файлы создаются и редактируются вручную. Надстройки над make автоматизируют процесс создания make-файлов. Наиболее популярные представители этого семейства в Linux — утилиты из пакета GNU Autotools (automake, autoconf, libtool и т. д.).

- Специализированные автосборщики — обычно такие утилиты создаются для удобства при работе с определенными проектами. Типичный представитель этого семейства — утилита qmake (Qt make) — автосборщик для проектов, использующих библиотеку Qt. Библиотека Qt была создана норвежской компанией Trolltech, но в настоящее время принадлежит корпорации Nokia.

Изучить каждый из перечисленных автосборщиков в рамках данной книги не представляется возможным. Поэтому мы будем пользоваться самой популярной в Linux утилитой автосборки GNU make. Далее, говоря о make, будем подразумевать GNU make.

2.2. Утилита make

Утилита make (GNU make) — наиболее популярное и проверенное временем средство автоматической сборки программ в Linux. Даже "гигант" автосборки, пакет GNU Autotools, является лишь надстройкой над make. Автоматическая сборка программы на языке C обычно осуществляется по следующему алгоритму.

1. Подготавливаются исходные и заголовочные файлы.
2. Подготавливаются make-файлы, содержащие сведения о проекте. Порой даже крупные проекты обходятся одним make-файлом. Вообще говоря, make-файл может называться как угодно, однако обычно выбирают одно из трех стандартных имен (Makefile, makefile или GNUmakefile), которые распознаются автосборщиком автоматически.
3. Вызывается утилита make, которая собирает проект на основании данных, полученных из make-файла. Если в проекте используется нестандартное имя make-файла, то его нужно указать после опции `-f` при вызове автосборщика.

На протяжении всей книги, чтобы не путаться, для make-файлов мы будем указывать имя Makefile.

> ***Замечание***
>
> Разработчики GNU make рекомендуют использовать имя Makefile. В этом случае у вас больше шансов, что make-файл будет стоять обособленно в отсортированном списке содержимого репозитория.

2.3. Базовый синтаксис Makefile

Итак, чтобы работать с make, необходимо создать файл с именем Makefile. В make-файлах могут присутствовать следующие конструкции:

- *Комментарии.* В make-файлах допустимы однострочные комментарии, которые начинаются символом # (решетка) и действуют до конца строки.
- *Объявления констант.* Константы в make-файлах служат для подстановки. Они во многом схожи с константами препроцессора языка C.

- *Целевые связки.* Эти элементы несут основную нагрузку в make-файле. При помощи целевых связок задаются зависимости между различными частями программы, а также определяются действия, которые будут выполняться при сборке программы. В любом make-файле должна быть хотя бы одна целевая связка.

Для правильного составления make-файла необходимо определить основную цель сборки проекта. Затем следует выявить промежуточные цели, если таковые существуют. Вернемся к примеру из предыдущей главы (см. листинги 1.2—1.4). В нем основной целью является формирование бинарного файла printup. Чтобы его получить, требуются файлы print_up.o и main.o. Это *промежуточные цели*. В make-файлах за каждую цель отвечает своя целевая связка.

После определения целей нужно выявить зависимости. В нашем примере основная цель (файл printup) может быть достигнута только при наличии файлов print_up.o и main.o. А файл print_up.o может быть получен только при наличии исходного файла print_up.c (см. листинг 1.3) и заголовочного файла print_up.h (см. листинг 1.2). Аналогичным образом файл main.o может быть получен только при наличии файлов main.c (см. листинг 1.4) и print_up.h (см. листинг 1.2).

Итак, мы знаем, что в Makefile обязательны только целевые связки. Каждая целевая связка состоит из следующих компонентов:

- *Имя цели*. Если целью является файл, то указывается его имя. После имени цели следует двоеточие.
- *Список зависимостей*. Здесь просто перечисляются через пробел имена файлов или имена промежуточных целей. Если цель ни от чего не зависит, то этот список будет пустым.
- *Инструкции*. Это команды, которые должны выполняться для достижения цели. Например, в целевой связке print_up.o инструкцией будет являться команда компиляции файла print_up.c. Каждая инструкция пишется на новой строке и начинается с символа табуляции. Обратите внимание, что некоторые текстовые редакторы (например, mcedit) по умолчанию заменяют табуляцию группой пробелов. В этом случае для редактирования Makefile следует воспользоваться другим редактором или настроить существующий. Иногда целевая связка не подразумевает выполнение каких-либо команд, а призвана только установить зависимости. В таком случае список инструкций оставляют пустым.

Теперь проверим систему в действии, организовав автосборку проекта printup из предыдущей главы. Сначала создаем Makefile (листинг 2.2).

Листинг 2.2. Make-файл программы printup1

```
# Makefile for printup

printup: print_up.o main.o
	gcc -o printup print_up.o main.o

print_up.o: print_up.c print_up.h
	gcc -c print_up.c
```

```
main.o: main.c
	gcc -c main.c

clean:
	rm -f *.o
	rm -f printup
```

Далее вызываем утилиту make с указанием цели, которую нужно достичь. В нашем случае это будет выглядеть так:

```
$ make printup
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Итак, проект собран. Осталось только во всем разобраться. Первая строка в Makefile — это комментарий. Затем следуют целевые связки. Первая связка отвечает за создание исполняемого файла printup и формируется следующим образом:

1. Сначала записывается имя цели (printup).
2. После двоеточия перечисляются зависимости (print_up.o и main.o).
3. На следующей строке после знака табуляции пишется правило для получения бинарника printup.

Аналогичным образом оформляются остальные целевые связки. Последняя (clean) требует особого рассмотрения:

1. Сначала указывается имя цели (clean).
2. После двоеточия следует пустой список зависимостей. Это значит, что данная связка не требует наличия каких-либо файлов и не предполагает предварительного выполнения промежуточных целей.
3. На следующих двух строках прописаны инструкции, удаляющие объектные файлы и бинарник.

Эта цель очищает проект от всех файлов, автоматически созданных при сборке. Итак, чтобы очистить проект, достаточно набрать следующую команду:

```
$ make clean
rm -f *.o
rm -f printup
```

Очистка проекта обычно выполняется в следующих случаях:

- при подготовке исходного кода к отправке конечному пользователю или другому программисту, когда нужно избавить проект от лишних файлов;
- при изменении или добавлении в проект заголовочных файлов;
- при изменении make-файла.

Вообще говоря, при запуске make имя цели можно не указывать. Тогда основной целью будет считаться первая цель в Makefile. Следовательно, в нашем случае, чтобы собрать проект, достаточно вызвать make без аргументов:

```
$ make
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Иногда требуется вписать в make-файл нечто длинное, например инструкцию, не умещающуюся в одной строке. В таком случае строки условно соединяются символом \ (обратная косая черта):

```
gcc -Wall -pedantic -g -o my_very_long_output_file one.o two.o \
three.o four.o five.o
```

Автосборщик при обработке make-файла будет интерпретировать такую конструкцию как единую строку.

2.4. Константы make

В make-файлах для параметризации процесса сборки можно использовать константы. Для объявления и инициализации констант предусмотрен следующий шаблон:

```
NAME=VALUE
```

Здесь `NAME` — это имя константы, `VALUE` — ее значение. Имя константы не должно начинаться с цифры. Значение может содержать любые символы, включая пробелы. Признак окончания значения константы — конец строки. Иначе говоря, любые символы, стоящие между знаком "равно" и символом переноса строки, будут являться значением константы.

Значение константы можно подставить в любую часть make-файла (кроме комментария). Если имя константы состоит из одного символа, то для подстановки достаточно добавить перед именем литеру `$` (доллар). Когда имя состоит из нескольких символов, для подстановки применяется следующий шаблон:

```
$(NAME)
```

Теперь модернизируем make-файл проекта printup, добавив в него константы (листинг 2.3).

Листинг 2.3. Make-файл программы printup2

```
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
	$(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
	$(CC) -c print_up.c
```

```
main.o: main.c
	$(CC) -c main.c

clean:
	$(CLEAN) *.o
	$(CLEAN) $(PROGRAM_NAME)
```

Итак, мы заменили имя компилятора, команду удаления и имя конечной программы символическими именами. Теперь можно, например, сменить имя программы, просто изменив значение константы PROGRAM_NAME.

Примечание

Файл print_up.h был исключен из списков зависимостей. При отсутствии заголовочного файла компилятор всегда сообщает об этом. Таким образом, включение print_up.h в список зависимостей хотя и не ошибочно, но явно избыточно.

Обратите внимание, что константы допустимы при объявлении и инициализации других констант. В результате наш Makefile можно значительно модернизировать (листинг 2.4).

Листинг 2.4. Make-файл программы printup3

```
CC=gcc
CLEAN=rm
CLEAN_FLAGS=-f
CLEAN_COMMAND=$(CLEAN) $(CLEAN_FLAGS)
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
	$(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
	$(CC) -c print_up.c

main.o: main.c
	$(CC) -c main.c

clean:
	$(CLEAN_COMMAND) *.o
	$(CLEAN_COMMAND) $(PROGRAM_NAME)
```

На самом деле, объявляемые пользователем константы по существу не являются константами, поскольку их можно переопределять, т. е. повторно присваивать им значения. В этом легко убедиться, если слегка изменить предыдущий Makefile (листинг 2.5).

Листинг 2.5. Make-файл программы printup4

```
CC=gcc
CLEAN=some_value
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
	$(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
	$(CC) -c print_up.c

main.o: main.c
	$(CC) -c main.c

CLEAN=rm -f

clean:
	$(CLEAN) *.o
	$(CLEAN) $(PROGRAM_NAME)
```

Из листинга 2.5 видно, что константа `CLEAN` изменяется. Тем не менее такое переопределение редко встречается на практике, поэтому понятие "константа" вполне пригодно.

Утилита make поддерживает также целый ряд специализированных констант. Две из них используются в целевых связках и представляют особый интерес:

- `$@` — содержит имя текущей цели;
- `$^` — содержит список зависимостей в текущей связке.

Если теперь переписать Makefile, добавив в него эти две константы, то получится довольно симпатичный результат (листинг 2.6).

Листинг 2.6. Make-файл программы printup5

```
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
	$(CC) -o $@ $^

print_up.o: print_up.c
	$(CC) -c $^

main.o: main.c
	$(CC) -c $^
```

```
clean:
	$(CLEAN) *.o
	$(CLEAN) $(PROGRAM_NAME)
```

Makefile не только уменьшился в размере, но и стал более гибким. Теперь при изменении целей или списков зависимостей не требуется параллельно изменять инструкции. Итак, имея определенный багаж знаний, можно окончательно модернизировать Makefile для проекта printup (листинг 2.7).

Листинг 2.7. Make-файл программы printup6

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f
PROGRAM_NAME=printup
OBJECT_FILES=*.o
SOURCE_FILES=print_up.c main.c

$(PROGRAM_NAME): $(OBJECT_FILES)
	$(CC) $(CCFLAGS) -o $@ $^

$(OBJECT_FILES): $(SOURCE_FILES)
	$(CC) $(CCFLAGS) -c $^

clean:
	$(CLEAN) *.o $(PROGRAM_NAME)
```

Мы усовершенствовали Makefile до такой степени, что для добавления в проект нового исходного файла достаточно будет дописать его имя в константу `SOURCE_FILES`. В этой версии make-файла появляется новая константа `CCFLAGS` с "магическим" значением `-Wall`. Посредством этой константы компилятору могут передаваться какие-то общие опции. В нашем случае опция `-Wall` включает все виды предупреждений (warnings). Таким образом, если компилятор "заподозрит" что-то неладное, то немедленно сообщит об этом.

2.5. Рекурсивный вызов make

Иногда программные проекты разделяют на несколько независимых подпроектов. В этом случае каждый подпроект имеет свой make-файл. Но рано или поздно понадобится все соединить в один большой проект. Для этого используется концепция *рекурсивного вызова* make, предполагающая наличие главного make-файла, который инициирует автосборку каждого подпроекта, а затем объединяет полученные файлы.

Посредством опции `-C` можно передать автосборщику make имя каталога, в котором следует искать Makefile. Это позволяет вызвать make для каждого под-

проекта из главного make-файла. Чтобы понять, как это осуществляется на практике, рассмотрим пример многокомпонентного программного проекта.

Итак, задача состоит в написании программы, которая выводит текущую версию Linux, а затем повторяет вывод заглавными буквами. Проект разбивается на два подпроекта. В одном реализуется функция чтения версии Linux, в другом — перевод всех символов полученного результата в верхний регистр. Конечным продуктом каждого подпроекта будет объектный файл. В рамках главного проекта эти файлы будут компоноваться в один бинарник.

Сначала нужно создать два каталога с именами readver и toup. В первом каталоге будет размещаться проект чтения версии Linux, во втором — проект перевода полученного результата в верхний регистр. Оба подпроекта будут содержать по одному исходному и по одному заголовочному файлу.

Теперь создайте в каталоге readver файлы readver.h (листинг 2.8) и readver.c (листинг 2.9).

Листинг 2.8. Файл readver/readver.h

```
#define STR_SIZE 1024
int readver (char * str);
```

Листинг 2.9. Файл readver/readver.c

```
#include <stdio.h>
#include <string.h>
#include "readver.h"
int readver (char * str)
{
        int i;
        FILE * fp = fopen ("/proc/version", "r");
        if (!fp) {
                fprintf (stderr, "Cannot open /proc/version\n");
                return 1;
        }

        for (i = 0; (i < STR_SIZE) &&
                        ((str[i] = fgetc(fp)) != EOF); i++);
        str[i] = '\0';
        fclose (fp);
        return 0;
}
```

С заголовочным файлом все понятно: здесь определяется соглашение по использованию функции `readver()`, а также объявляется макроконстанта `STR_SIZE`, которая показывает максимальный размер буфера для считывания файла. Функция `readver()`, реализованная в readver.c, читает файл /proc/version, в котором находится

информация о текущей версии Linux. Эта функция записывает результат (содержимое файла /proc/version) в строку, переданную в качестве аргумента. Вся ответственность за выделение памяти лежит на вызывающей стороне.

Сначала /proc/version открывается в режиме "только для чтения" (read-only). После стандартной проверки начинается посимвольное считывание файла и занесение результата в строку `str`, которая завершается нуль-терминатором (`'\0'`). При успешном завершении функция возвращает ноль, в противном случае — ненулевое значение.

Теперь в каталоге readver следует создать make-файл, который будет собирать подпроект чтения версии Linux (листинг 2.10).

Листинг 2.10. Файл readver/Makefile

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f

readver.o: readver.c
	$(CC) $(CCFLAGS) -c $^

clean:
	$(CLEAN) *.o
```

Следующий шаг — создание подпроекта, реализующего механизм перевода полученных из /proc/version данных в верхний регистр. Для этого необходимо перейти в каталог toup и создать в нем заголовочный файл toup.h (листинг 2.11), исходный файл toup.c (листинг 2.12) и собственно make-файл (листинг 2.13).

Листинг 2.11. Заголовочный файл toup/toup.h

```
void toup (char * str);
```

Листинг 2.12. Исходный файл toup/toup.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "toup.h"

void toup (char * str)
{
	int i;
	for (i = 0; i < strlen (str); i++)
		str[i] = toupper (str[i]);
}
```

Листинг 2.13. Файл toup/Makefile

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f

toup.o: toup.c
	$(CC) $(CCFLAGS) -c $^

clean:
	$(CLEAN) *.o
```

Функция `toup()` не выводит строку на экран, а просто конвертирует каждый ее символ в верхний регистр, оставляя вызывающей стороне задачу выделения памяти.

Теперь, когда оба подпроекта готовы, поднимемся на уровень выше и создадим исходный файл upver.c (листинг 2.14), в котором будут вызываться функции `readver()` и `toup()`.

Листинг 2.14. Исходный файл upver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <readver.h>
#include <toup.h>

int main (void)
{
	char * str = (char*) malloc (STR_SIZE * sizeof(char));
	if (str == NULL) {
		fprintf (stderr, "Cannot allocate memory\n");
		return 1;
	}

	if (readver (str) != 0) {
		fprintf (stderr, "Failed\n");
		return 1;
	}

	printf ("%s\n", str);
	toup (str);
	printf ("%s\n", str);
	free (str);
	return 0;
}
```

Обратите внимание, что заголовочные файлы readver.h и toup.h записываются не в кавычках, как мы привыкли, а в угловых скобках. Это значит, что компилятор

(точнее — препроцессор) будет искать данные файлы в специально отведенных каталогах (наподобие /usr/include или /usr/local/include). Перед нами встает задача добавления в этот список каталогов readver и toup. Для этого используется опция компилятора `-I`, которая указывает каталог, в котором находятся заголовочные файлы.

Теперь создадим главный make-файл (листинг 2.15).

Листинг 2.15. Главный make-файл программы upver

```
CC=gcc
CCFLAGS=-Wall
MAKE=make
CLEAN=rm -f
PROGRAM_NAME=upver
OBJECTS=readver/readver.o toup/toup.o

$(PROGRAM_NAME): make-readver make-toup upver.o
	$(CC) $(CCFLAGS) -o $(PROGRAM_NAME) $(OBJECTS) upver.o

upver.o: upver.c
	$(CC) $(CCFLAGS) -c -Ireadver -Itoup $^

make-readver:
	$(MAKE) -C readver readver.o

make-toup:
	$(MAKE) -C toup toup.o

clean:
	$(CLEAN) *.o $(PROGRAM_NAME)
	$(MAKE) -C readver clean
	$(MAKE) -C toup clean
```

Цели `make-readver` и `make-toup` не являются файлами. Это так называемые *псевдоцели*. Соответствующие им целевые модули вызывают утилиту make с опцией `-C`. Проще говоря, целевой модуль `make-readver` собирает подпроект чтения версии Linux, а модуль `make-toup` — второй подпроект, переводящий символы результата в верхний регистр. Аналогичным образом происходит очистка (цель `clean`): сначала очищаются файлы главного проекта, а затем выполняется очистка подпроектов их собственными средствами.

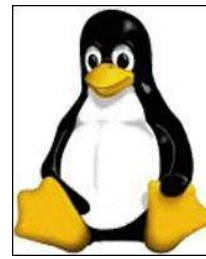
Как видно из рассмотренного примера, любой программный проект можно разделить на подпроекты, используя концепцию рекурсивного вызова make. Подобным образом, например, разрабатывается ядро Linux: в настоящее время в его исходниках насчитывается более 1400 make-файлов!

2.6. Получение дополнительной информации

В этой главе принципы автоматической сборки программ в Linux описаны лишь частично. Дополнительную информацию по этой теме можно получить из следующих источников:

- man 1 make — man-страница утилиты GNU make;
- info make — info-страница GNU make;
- **http://www.gnu.org/software/autoconf/manual/html_node/** — Autoconf;
- **http://www.gnu.org/software/automake/manual/html_node/** — Automake;
- **http://sources.redhat.com/autobook/autobook/autobook_toc.html** — Autotools;
- **http://tmake.sourceforge.net/** — tmake;
- **http://www.cmake.org/HTML/Documentation.html** — cmake;
- **http://www.snake.net/software/imake-stuff/** — imake;
- **http://doc.qt.nokia.com/** — qmake;
- **http://www.fastmake.org/doc.html** — Fastmake.

ГЛАВА 3



Окружение

Окружение — это набор переменных и их значений, с помощью которых можно передавать в программу какие-нибудь данные общего назначения. В этой главе описываются механизмы, позволяющие вашим программам читать и модифицировать окружение.

3.1. Понятие окружения

Чтобы понять, что такое окружение, нужно немного забежать вперед и разобраться в базовых терминах многозадачности Linux. В основе многозадачности лежит понятие "процесс". *Процесс* — это нечто, совершающее в системе какие-либо действия. Обычно это запущенная работающая программа. Один процесс может инициировать (породить) другой. В таком случае породивший процесс называют *родителем* или *родительским процессом*, а порожденный — *потомком* или *дочерним процессом*. Все процессы в системе — элементы одной иерархии, называемой *деревом процессов*. На вершине этой иерархии находится процесс init. Это единственный процесс, не имеющий родителя. Каждый раз, когда вы запускаете в командной строке какую-нибудь программу, например ls, в системе рождается новый процесс, для которого командная оболочка является родителем.

> ***Примечание***
>
> Бытует мнение, что родителем процесса init является ядро Linux. Не имеет смысла в очередной раз поднимать философский вопрос о курице и яйце. Для нас важно лишь то, что init всегда стоит на вершине иерархии процессов.

Окружение (environment) — это набор специфичных для конкретного пользователя пар `ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ`. Каждый процесс располагает и свободно распоряжается своей копией окружения, которую он получает от родителя. Таким образом, если процесс сначала модифицирует свое окружение, а потом породит новый процесс, то потомок получит копию уже измененного окружения.

Чтобы посмотреть, что представляет собой окружение в действительности, введите следующую команду:

```
$ env
```

На экран будет выведено окружение программы env. Можете не сомневаться, это точная копия окружения вашей командной оболочки. В современных Linux-дистрибутивах окружение представлено десятками переменных, но широко используются только некоторые из них:

- `USER` — имя пользователя;
- `HOME` — домашний каталог;
- `PATH` — список каталогов, в которых осуществляется поиск исполняемых файлов программ;
- `SHELL` — используемая командная оболочка;
- `PWD` — текущий каталог.

Следующий шаблон позволяет посмотреть значение конкретной переменной:

```
$ echo $VARIABLE
```

Здесь `VARIABLE` — это имя переменной. Вообще говоря, оболочка вместо записи `$VARIABLE` всегда подставляет значение соответствующей переменной окружения. Рассмотрим пример:

```
$ touch $USER
```

Вместо `$USER` здесь подставляется значение переменной. Такая команда создаст пустой файл, присвоив ему имя текущего пользователя.

Пользователь может модифицировать окружение командной оболочки. В разных оболочках это делается по-разному. Рассмотрим такую возможность на примере bash. Эта оболочка, помимо свойственного любому процессу окружения, поддерживает также свой собственный набор переменных, которые называют *локальными переменными оболочки*. Рассмотрим пример:

```
$ MYVAR=Hello
$ echo $MYVAR
$ env | grep MYVAR
$
```

Первой командой создается и инициализируется локальная переменная оболочки `MYVAR`. Следующая команда выводит значение этой переменной. Третья команда ищет переменную `MYVAR` среди окружения программы env (точная копия окружения оболочки). Но поскольку `MYVAR` не является переменной окружения, то последняя команда ничего не выводит.

Как видно из этого примера, локальные переменные "работают" только на уровне оболочки. Не являясь частью окружения, они не могут наследоваться дочерними процессами. Поэтому программа env не нашла переменную `MYVAR`. Но иногда требуется включить (экспортировать) локальную переменную в окружение. Для этого в оболочке bash есть внутренняя (не являющаяся отдельной программой) команда `export`:

```
$ export MYVAR
$ env | grep MYVAR
MYVAR=Hello
```

Команду `export` можно также использовать для инициализации и экспорта переменной в один прием:

```
$ export MYVAR_NEW=Goodbye
$ env | grep MYVAR_NEW
MYVAR_NEW=Goodbye
```

Важно понимать, что полученные переменные `MYVAR` и `MYVAR_NEW` существуют только в текущем окружении процесса оболочки и ее потомков. Очевидно, что после перезагрузки эти переменные "исчезнут", если их, например, не объявить в файле инициализации bash. Даже такая жизненно важная переменная, как `PATH`, не появляется в окружении оболочки сама собой: строка для ее инициализации обычно находится в файле /etc/profile.

Исключить переменную из окружения оболочки позволяет команда `export -n`. При этом остается локальная переменная. Когда же надо удалить переменную отовсюду, вызывают команду `unset`:

```
$ env | grep -i myvar
MYVAR_NEW=Goodbye
MYVAR=Hello
$ export -n MYVAR
$ env | grep MYVAR
MYVAR_NEW=Goodbye
$ echo $MYVAR
Hello
$ unset MYVAR_NEW
$ env | grep MYVAR
$ echo -n $MYVAR_NEW
$
```

Важно понимать, что `export`, `export -n` и `unset` — это не отдельные программы, а внутренние команды bash. Оболочки ksh (KornShell) и zsh (Z shell), например, не поддерживают опцию `-n` команды `export`. Оболочки семейства C-Shell (csh, tcsh) работают с окружением совсем другими командами `setenv` и `unsetenv`.

3.2. Чтение окружения: *environ*, *getenv()*

Программа может читать окружение двумя способами:

1. Посредством внешней переменной `environ`, представляющей собой массив строк `ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ`. Эта переменная объявлена в заголовочном файле unistd.h.
2. При помощи функции `getenv()`, которая возвращает значение указанной переменной окружения. Эта функция объявлена в заголовочном файле stdlib.h.

Функция `getenv()` имеет следующий прототип:

```
char * getenv (char * VAR_NAME);
```

Она возвращает значение указанной переменной `VAR_NAME`. Если переменная отсутствует в окружении, то возвращается `NULL`.

Рассмотрим сначала пример, использующий `environ`. Следующая программа (листинг 3.1) просто выводит на экран свое окружение.

Листинг 3.1. Программа myenv1.c

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

extern char ** environ;

int main (void)
{
        int i;
        for (i = 0; environ[i] != NULL; i++)
                        printf ("%s\n", environ[i]);

        return 0;
}
```

Массив `environ` обычно используется для просмотра окружения (как в нашем случае), а также для передачи окружения в другой процесс в ходе реализации многозадачности, но об этом речь пойдет в *главе 10*. Для получения значения конкретной переменной предназначена функция `getenv()`. Приведенная в листинге 3.2 программа демонстрирует ее работу.

Листинг 3.2. Программа myenv2.c

```
#include <stdio.h>

int main (int argc, char ** argv)
{
        char * value;
        if (argc < 2) {
                fprintf (stderr, "Usage: myenv2 <variable>\n");
                return 1;
        }

        value = (char*) getenv (argv[1]);
        if (value == NULL)
                printf ("%s not found\n", argv[1]);
        else
                printf ("%s=%s\n", argv[1], value);

        return 0;
}
```

Программа выводит значение переменной окружения, имя которой читается из первого аргумента командной строки (`argv[1]`). Функция `getenv()` возвращает значение `NULL`, если переменная окружения не найдена. В этом случае наша программа выводит соответствующее сообщение:

```
$ gcc -o myenv1 myenv1.c
$ ./myenv1 ABRAKADABRA
ABRAKADABRA not found
$ ./myenv1 USER
USER=nnivanov
$ ./myenv1 user
user not found
```

Обратите внимание, что имена переменных окружения чувствительны к регистру символов. Исторически сложилась практика именования переменных окружения заглавными буквами, однако символы нижнего регистра также допускаются. Действительно, запись "user not found" выглядит несколько двусмысленно: то ли переменная `user` не найдена, то ли пользователь не найден. А все из-за того, что имя переменной сливается с остальным текстом.

3.3. Модификация окружения: *setenv()*, *putenv()*, *unsetenv()*

Чаще всего модификация окружения требуется при реализации многозадачности. Ранее уже говорилось о том, что дочерний процесс получает копию окружения своего родителя. Приведем пример. Некоторые программы читают языковую информацию (локаль) из переменной окружения `LANG`. Этим можно воспользоваться. Если текущая локаль настроена на русский язык, то программа date, например, будет выводить примерно следующее:

```
$ date
Птн Апр 29 06:48:00 MSD 2011
```

Однако если изменить переменную окружения `LANG`, то программа date унаследует ее от оболочки. Можно, например, вывести дату на английском, французском или даже на финском языке:

```
$ export LANG=en_US
$ date
Fri Apr 29 06:48:47 MSD 2011
$ export LANG=fr_FR
$ date
ven. avril 29 06:52:16 MSD 2011
$ export LANG=fi_FI
$ date
pe 29.4.2011 06.52.38 +0400
```

Такие "трюки" возможны не везде, а лишь в версиях Linux с языковой поддержкой. Но смысл в том, что дочернему процессу можно "подсунуть" любое окружение.

Для этого даже не обязательно менять окружение оболочки. Передать программе модифицированное окружение можно при помощи уже знакомой нам программы env:

```
$ env LANG=fi_FI date
pe 29.4.2011 06.53.30 +0400
```

На практике модификация окружения осуществляется функциями `setenv()`, `putenv()` и `unsetenv()`, которые объявлены в заголовочном файле stdlib.h.

```
int setenv (const char * NAME, const char * VALUE, int OV);
```

Функция `setenv()` добавляет новую или изменяет существующую переменную окружения с именем `NAME`, присваивая ей значение `VALUE`. `OV` — это флаг перезаписи, показывающий, нужно ли перезаписывать переменную, если таковая существует. Если параметр `OV` не равен нулю, то переменная перезаписывается, в противном случае переменная остается нетронутой. `setenv()` возвращает 0 при успешном завершении и –1, если произошла ошибка.

```
int putenv (char * INITSTR);
```

Функция `putenv()` добавляет новую или заменяет существующую переменную окружения, используя строку инициализации `INITSTR` в формате `ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ`. `putenv()` возвращает 0 при успешном завершении и –1, если произошла ошибка. Следует отметить, что функция `putenv()` реализована не во всех Unix-подобных системах, что может сказаться на переносимости.

```
int unsetenv (const char * NAME);
```

Функция `unsetenv()` удаляет переменную с именем `NAME` из окружения. Ранее функция `unsetenv()` ничего не возвращала, и лишь в glibc-2.2.2 все изменилось. В настоящее время `unsetenv()` возвращает 0 при успешном завершении или –1 — в случае ошибки.

Рассмотрим сначала пример, демонстрирующий работу функции `setenv()` (листинг 3.3).

Листинг 3.3. Пример setenvdemo.c

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

int main (int argc, char ** argv)
{
        int ov_flag = 1;
        char * var;
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        setenv (argv[1], argv[2], ov_flag);
        var = getenv (argv[1]);
        if (var == NULL)
                printf ("Variable %s doesn't exist\n", argv[1]);
        else
                printf ("%s=%s\n", argv[1], var);

        return 0;
}
```

Приведенная программа читает два аргумента командной строки: имя переменной и ее значение. Особый интерес представляет переменная `ov_flag`. Если присвоить ей нулевое значение, то поведение программы изменится: функция `setenv()` не будет перезаписывать уже существующие переменные окружения, унаследованные от командной оболочки.

Рассмотрим теперь пример использования `putenv()` (листинг 3.4).

Листинг 3.4. Пример putenvdemo.c

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

int main (int argc, char ** argv)
{
        char * var;
        char initvar[1024];
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        sprintf (initvar, "%s=%s", argv[1], argv[2]);
        putenv (initvar);
        var = getenv (argv[1]);
        if (var == NULL)
                printf ("Variable %s doesn't exist\n", argv[1]);
        else
                printf ("%s=%s\n", argv[1], var);

        return 0;
}
```

Эта программа работает так же, как и setenvdemo (листинг 3.3) с установленным флагом перезаписи. Очевидно, что в ситуациях, когда имя переменной окружения и ее значение хранятся в разных строках, всегда целесообразнее использовать `setenv()`.

Теперь рассмотрим программу, работающую с функцией `unsetenv()` (листинг 3.5).

Листинг 3.5. Программа unsetenvdemo.c

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

extern char ** environ;

int main (int argc, char ** argv)
{
        int i;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (unsetenv (argv[1]) != 0) {
                fprintf (stderr, "Cannot unset %s\n", argv[1]);
                return 1;
        }

        for (i = 0; environ[i] != NULL; i++)
                printf ("%s\n", environ[i]);

        return 0;
}
```

Эта программа удаляет из окружения переменную, переданную в качестве аргумента, а затем выводит на экран все окружение, через механизм просмотра массива `environ`. Таким образом можно убедиться, что переменная действительно удалена из окружения:

```
$ ./unsetenvdemo USER | grep USER
$
```

Обратите внимание, что передача в функцию `unsetenv()` имени несуществующей переменной не считается ошибкой. Это говорит о том, что `unsetenv()` не обязуется, а только пытается удалить переменную из окружения. Другое дело, когда в имя переменной "вкрадывается", например, знак равенства:

```
$ ./unsetenvdemo USER=
```

В результате будет выдано сообщение

```
Cannot unset USER=
```

Поскольку знак равенства является служебным символом, функция `unsetenv()` выдала ошибку.

3.4. Очистка окружения

Есть два способа очистить окружение процесса:

1. Присвоить массиву `environ` значение `NULL`.
2. Воспользоваться функцией `clearenv()`.

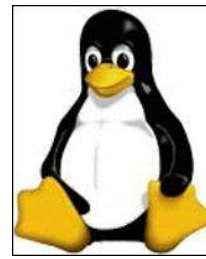
Функция `clearenv()` объявлена в заголовочном файле stdlib.h следующим образом:

```
int clearenv (void);
```

В обычных приложениях, не требующих повышенных мер безопасности, целесообразнее выбрать первый способ, поскольку функция `clearenv()` реализована не во всех Unix-подобных системах.

В тех случаях, когда необходимо предотвратить возможность доступа к окружению потенциальных злоумышленников (в серверах, браузерах, сетевых приложениях и т. д.), используют функцию `clearenv()`. Дело в том, что присвоение массиву `environ` значения `NULL` лишь ликвидирует указатель. При этом бывшее окружение остается в памяти процесса. Функция `clearenv()` освобождает память, делая бывшее окружение практически недоступным. Здесь есть свои тонкости, но они выходят за рамки данной книги.

ГЛАВА 4



Библиотеки

Библиотеки позволяют разным программам использовать один и тот же объектный код. Это избавляет программиста от необходимости создавать то, что уже создано. В этой главе рассматриваются методы создания и применения библиотек.

4.1. Библиотеки и заголовочные файлы

Библиотека (library) — это набор соединенных определенным образом объектных файлов. Библиотеки подключаются к программе на стадии компоновки. Функция `printf()`, например, реализована в стандартной библиотеке языка C, которая автоматически подключается к программе, когда компоновка осуществляется посредством gcc.

Чтобы подключить библиотеку к программе, нужно передать компоновщику опцию `-l`, указав после нее (можно без разделяющего пробела) имя библиотеки. Если, например, к программе необходимо подключить библиотеку mylibrary, то для осуществления компоновки следует задать такую команду:

```
$ gcc -o myprogram myprogram.o -lmylibrary
```

Чаще всего файлы библиотек располагаются в специальных каталогах (/lib, /usr/lib, /usr/local/lib и т. п.). Если же требуется подключить библиотеку из другого места, то при компоновке следует указать опцию `-L`, после которой (можно без разделяющего пробела) указывается нужный каталог. Таким образом, при необходимости подключения библиотеки mylibrary, находящейся в каталоге /usr/lib/particular, для выполнения компоновки указывают такую команду:

```
$ gcc -o myprogram myprogram.o -L/usr/lib/particular -lmylibrary
```

Следует отметить, что имена файлов библиотек обычно начинаются с префикса `lib`. Имя библиотеки получается из имени файла отбрасыванием префикса `lib` и расширения. Если, например, файл библиотеки имеет имя libmylibrary.so, то сама библиотека будет называться mylibrary.

Библиотеки подразделяются на две категории:

- *статические* (архивы);
- *совместно используемые* (динамические).

Статические библиотеки (static libraries) создаются программой ar. Файлы статических библиотек имеют расширение `.a`. Если, например, статическая библиотека foo (представленная файлом libfoo.a) создана из двух объектных файлов foo1.o и foo2.o, то следующие две команды будут эквивалентны:

```
$ gcc -o myprogram myprogram.o foo1.o foo2.o
$ gcc -o myprogram myprogram.o -lfoo
```

Совместно используемые библиотеки (shared libraries) создаются компоновщиком при вызове gcc с опцией `-shared` и имеют расширение `.so`. Совместно используемые библиотеки не помещают свой код непосредственно в программу, а лишь создают специальные ссылки. Поэтому любая программа, скомпонованная с динамической библиотекой, при запуске требует наличия данной библиотеки в системе.

Иногда заголовочные файлы тоже называют библиотеками. Это не так! Библиотеки — это объектный код, сгруппированный таким образом, чтобы им могли пользоваться разные программы. Заголовочные файлы — это часть исходного кода программы, в которых, как правило, определяются соглашения по использованию общих идентификаторов (имен). Когда в заголовочном файле определены механизмы, реализованные в библиотеке, то правильно будет называть такой файл (или группу файлов) *интерфейсом библиотеки*.

4.2. Подключение библиотек

Рассмотрим в качестве примера программу для возведения числа в степень (листинг 4.1).

Листинг 4.1. Программа power.c

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

int main (int argc, char ** argv)
{
       if (argc < 3) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       printf ("%f\n", pow (atof (argv[1]), atof (argv[2])));
       return 0;
}
```

Функции `printf()` и `atof()` реализованы в стандартной библиотеке языка C, которая автоматически подключается к проекту во время компоновки. Функция `pow()` принадлежит библиотеке математических функций, которую нужно подключать отдельно. Математическая библиотека, представленная файлами libm.so (динамический вариант) и libm.a (статический вариант), имеется практически в любой Linux-системе. Следовательно, для сборки приведенной программы необходимо указать следующую команду:

```
$ gcc -o power1 power.c -lm
```

Возникает вопрос: какая именно библиотека была подключена? Оба варианта математической библиотеки (статическая и динамическая) подходят под шаблон `-lm`. В таких ситуациях предпочтение отдается динамическим библиотекам. Но если при компоновке указать опцию `-static`, то приоритет изменится в сторону статической библиотеки:

```
$ gcc -static -o power2 power.c -lm
```

> ***Замечание***
>
> Чтобы использовать статический вариант математической библиотеки, в вашей Linux-системе должен быть установлен пакет glibc-static-devel.

Теперь сравните размеры исполняемых файлов. Бинарник, полученный в результате линковки с опцией `-static`, значительно больше:

```
-rwxr-xr-x 1 nnivanov nnivanov 5.9K 2011-04-29 07:14 power1*
-rwxr-xr-x 1 nnivanov nnivanov 615K 2011-04-29 07:20 power2*
```

Это обусловлено тем, что статическая библиотека полностью внедряется в исполняемый файл, а совместно используемая библиотека лишь оставляет информацию о себе.

4.3. Создание статических библиотек

Статическая библиотека — это архив, создаваемый специальным архиватором ar из пакета GNU binutils. Многие пользователи (особенно пользователи Windows) полагают, что архиватор — это средство сжатия файлов. На самом деле архиватор лишь объединяет несколько файлов в один с возможностью выполнения обратной процедуры.

> ***Примечание***
>
> Программы сжатия файлов называются *компрессорами*. Зачастую (особенно в ОС Windows) компрессор вместе с архиватором составляют один программный пакет, который по ошибке называют просто архиватором.

Утилита ar создает архив, который может подключаться к программе во время компоновки на правах библиотеки. Этот архиватор поддерживает много опций, однако нас интересуют только некоторые из них.

Опция `r` создает архив, а также добавляет или заменяет файлы в существующем архиве. Например, если нужно создать статическую библиотеку libfoo.a из файлов foo1.o и foo2.o, то для этого достаточно ввести следующую команду:

```
$ ar r libfoo.a foo1.o foo2.o
```

Опция `x` извлекает файлы из архива:

```
$ ar x libfoo.a
```

Опция `c` приостанавливает вывод сообщений о том, что создается библиотека:

```
$ ar cr libfoo.a foo1.o foo2.o
```

Опция `v` включает режим подробных сообщений (verbose mode):

```
$ ar crv libfoo.a foo1.o foo2.o
```

Рассмотрим теперь пример создания статической библиотеки, в которой реализуются две функции для работы с окружением. Для наглядности разобьем исходный код на два файла: mysetenv.c (листинг 4.2) и myprintenv.c (листинг 4.3).

Листинг 4.2. Файл mysetenv.c

```
#include <stdlib.h>
#include <stdio.h>
#include "myenv.h"

void mysetenv (const char * name, const char * value)
{
       printf ("Setting variable %s\n", name);
       setenv (name, value, 1);
}
```

Листинг 4.3. Файл myprintenv.c

```
#include <stdlib.h>
#include <stdio.h>
#include "myenv.h"

void myprintenv (const char * name)
{
       char * value = getenv (name);
       if (value == NULL) {
              printf ("Variable %s doesn't exist\n");
              return;
       }
       printf ("%s=%s\n", name, value);
}
```

Чтобы функции `mysetenv()` и `myprintenv()` вызывались по правилам, нужно создать заголовочный файл (листинг 4.4).

Листинг 4.4. Заголовочный файл myenv.h

```
#ifndef MYENV_H
#define MYENV_H

void mysetenv (const char * name, const char * value);
void myprintenv (const char * name);

#endif
```

Обратите внимание на следующие строки листинга 4.4:

```
#ifndef MYENV_H
#define MYENV_H
#endif
```

Этот распространенный "фокус" с препроцессором позволяет избежать множественных включений в исходный код одного и того же заголовочного файла.

Теперь организуем автоматическую сборку библиотеки. Для этого необходимо создать make-файл (листинг 4.5).

Листинг 4.5. Файл Makefile

```
libmyenv.a: mysetenv.o myprintenv.o
	ar rv $@ $^

mysetenv.o: mysetenv.c
	gcc -c $^

myprintenv.o: myprintenv.c
	gcc -c $^

clean:
	rm -f libmyenv.a *.o
```

Напишем программу, к которой будет подключаться полученная библиотека (листинг 4.6).

Листинг 4.6. Программа envmain.c

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

int main (int argc, char ** argv)
{
	if (argc < 3) {
		fprintf (stderr, "Too few arguments\n");
		return 1;
	}
```

```
        mysetenv (argv[1], argv[2]);
        myprintenv (argv[1]);
        return 0;
}
```

Осталось только собрать эту программу с библиотекой libmyenv.a:

```
$ gcc -o myenv envmain.c -L. -lmyenv
```

Проверим, что получилось:

```
$ ./myenv MYVAR Hello
Setting variable MYVAR
MYVAR=Hello
```

Кроме того, сборку программы и создание библиотеки можно объединить в один проект. Для этого создадим универсальный make-файл (листинг 4.7).

Листинг 4.7. Универсальный файл Makefile

```
myenv: envmain.o libmyenv.a
        gcc -o myenv envmain.o -L. -lmyenv

envmain.o: envmain.c
        gcc -c $^

libmyenv.a: mysetenv.o myprintenv.o
        ar rv $@ $^

mysetenv.o: mysetenv.c
        gcc -c $^

myprintenv.o: myprintenv.c
        gcc -c $^

clean:
        rm -f myenv libmyenv.a *.o
```

В этом варианте сборка программы envmain осуществляется в два этапа. В качестве главной цели теперь используется `myenv`, а цель `libmyenv.a` попала в список зависимостей. Но это не мешает, например, собирать в рамках данного проекта только библиотеку:

```
$ make libmyenv.a
gcc -c mysetenv.c
gcc -c myprintenv.c
ar rv libmyenv.a mysetenv.o myprintenv.o
ar: creating libmyenv.a
a — mysetenv.o
a — myprintenv.o
```

4.4. Создание совместно используемых библиотек

Процесс создания и подключения совместно используемых библиотек несколько сложнее, чем статических. Динамические библиотеки создаются при помощи gcc по следующему шаблону:

```
$ gcc -shared -o LIBRARY_NAME FILE1.o FILE2.o ...
```

В действительности gcc вызывает компоновщик ld с опциями для создания совместно используемой библиотеки. В принципе, этот процесс внешне ничем (кроме опции `-shared`) не отличается от компоновки обычного исполняемого файла. Но не все так просто. При создании динамических библиотек следует учитывать два нюанса:

- в процессе компоновки совместно используемой библиотеки должны участвовать объектные файлы, содержащие *позиционно-независимый код* (Position Independent Code). Этот код имеет возможность подгружаться к программе в момент ее запуска. Чтобы получить объектный файл с позиционно-независимым кодом, нужно откомпилировать исходный файл с опцией `-fPIC`;
- опция `-L`, указанная при компоновке, позволяет дополнить список каталогов, в которых будет выполняться поиск библиотек. По умолчанию в исполняемом файле сохраняется лишь имя библиотеки, а во время запуска программы происходит повторный поиск библиотек. Поэтому программист должен учитывать месторасположение динамической библиотеки не только на своем компьютере, но и там, где будет впоследствии запускаться программа.

В момент запуска программы для поиска библиотек просматриваются каталоги, перечисленные в файле /etc/ld.so.conf и в переменной окружения `LD_LIBRARY_PATH`.

> ***Примечание***
>
> Синтаксис файла /etc/ld.so.conf позволяет при помощи инструкции `include` подгружать содержимое других файлов, содержащих информацию о расположении библиотек.

Переменная окружения `LD_LIBRARY_PATH` имеет тот же формат, что и переменная `PATH`, т. е. содержит список каталогов, разделенных двоеточиями. Известно, что окружение нестабильно и может изменяться в ходе наследования от процесса к процессу. Поэтому использование `LD_LIBRARY_PATH` — не самый разумный ход.

В процессе компоновки программы можно отдельно указать каталог, где будет размещаться библиотека. Для этого линковщику ld необходимо передать опцию `-rpath` при помощи опции `-Wl` компилятора gcc. Например, чтобы занести в исполняемый файл prog месторасположение библиотеки libfoo.so, нужно сделать следующее:

```
$ gcc -o prog prog.o -L./lib/foo -lfoo -Wl,-rpath,/lib/foo
```

Итак, опция `-Wl` сообщает gcc о необходимости передать линковщику определенную опцию. Далее, после запятой, следует сама опция и ее аргументы, также разде-

ленные запятыми. Такой подход выглядит лучше, чем применение LD_LIBRARY_PATH, однако и здесь есть существенный недостаток. Нет никаких гарантий, что на компьютере у конечного пользователя библиотека libfoo.so будет также находиться в каталоге /lib/foo.

Есть еще один способ заставить программу искать совместно используемую библиотеку в нужном месте. Во время инсталляции программы можно добавить запись с каталогом месторасположения библиотеки в файл /etc/ld.so.conf либо в один из файлов, добавленных в ld.so.conf инструкцией include. Но это делают крайне редко, поскольку слишком длинный список каталогов в этом файле может отразиться на скорости загрузки системы. Обычно к такому подходу прибегают только такие "именитые" проекты, как Qt или X11.

Примечание

Многие Linux-системы при загрузке читают файл /etc/ld.so.conf и создают кэш динамических библиотек.

Наилучший выход из сложившегося положения — размещать библиотеки в специально предназначенных для этого каталогах (/usr/lib или /usr/local/lib). Естественно, программист в ходе работы над проектом может для удобства пользоваться переменной LD_LIBRARY_PATH или опциями -Wl и -rpath, но в конечной программе лучше избегать этих приемов и просто располагать библиотеки в обозначенных ранее каталогах.

Теперь, уяснив все тонкости, переходим к делу. За основу возьмем пример из предыдущего раздела. Рассмотрим сначала концепцию использования переменной окружения LD_LIBRARY_PATH. Чтобы переделать предыдущий пример для работы с динамической библиотекой, требуется лишь изменить make-файл (листинг 4.8).

Листинг 4.8. Модифицированный файл Makefile

```
myenv: envmain.o libmyenv.so
       gcc -o myenv envmain.o -L. -lmyenv

envmain.o: envmain.c
       gcc -c $^

libmyenv.so: mysetenv.o myprintenv.o
       gcc -shared -o libmyenv.so $^

mysetenv.o: mysetenv.c
       gcc -fPIC -c $^

myprintenv.o: myprintenv.c
       gcc -fPIC -c $^

clean:
       rm -f myenv libmyenv.so *.o
```

Обратите внимание, что файлы mysetenv.o и myprintenv.o, участвующие в создании библиотеки, компилируются с опцией `-fPIC` для генерирования позиционно-независимого кода. Файл envmain.o не добавляется в библиотеку, поэтому он компилируется без опции `-fPIC`. Если теперь попытаться запустить исполняемый файл myenv, то будет выдано сообщение об ошибке:

```
$ ./myenv MYVAR Hello
./myenv: error while loading shared libraries: libmyenv.so:
cannot open shared object file: No such file or directory
```

Проблема в том, что программа не нашла библиотеку в стандартном списке каталогов. После установки переменной `LD_LIBRARY_PATH` проблема исчезнет:

```
$ export LD_LIBRARY_PATH=.
$ ./myenv MYVAR Hello
Setting variable MYVAR
MYVAR=Hello
```

Если подняться в родительский каталог и попытаться оттуда запустить программу myenv, то опять будет обнаружена ошибка:

```
$ cd ..
$ myenv/myenv MYVAR Hello
myenv/myenv: error while loading shared libraries: libmyenv.so:
cannot open shared object file: No such file or directory
```

Очевидно, что ошибка произошла из-за того, что в текущем каталоге не нашлась требуемая библиотека. Этот поучительный пример говорит о том, что в `LD_LIBRARY_PATH` лучше заносить абсолютные имена каталогов, а не относительные. Попробуем еще раз:

```
$ cd myenv
$ export LD_LIBRARY_PATH=$PWD
$ cd ..
$ myenv/myenv MYVAR Hello
Setting variable MYVAR
MYVAR=Hello
```

Переменная окружения `PWD` содержит абсолютный путь к текущему каталогу, а запись `$PWD` подставляет это значение в команду.

Попробуем теперь указать линковщику опцию `-rpath`. Для этого изменим в make-файле первую целевую связку:

```
myenv: envmain.o libmyenv.so
       gcc -o myenv envmain.o -L. -lmyenv -Wl,-rpath,.
```

Помимо этого, для чистоты эксперимента удалим из окружения переменную `LD_LIBRARY_PATH`:

```
$ unset LD_LIBRARY_PATH
```

Теперь программа запускается из любого каталога без манипуляций с окружением.

4.5. Взаимодействие библиотек

Бывают случаи, когда динамическая библиотека сама компонуется с другой библиотекой. В этом нет ничего необычного. Такие зависимости между библиотеками можно увидеть при помощи программы ldd. Даже библиотека libmyenv.so из предыдущего раздела автоматически скомпонована со стандартной библиотекой языка C:

```
$ ldd libmyenv.so
linux-gate.so.1 =>  (0xffffe000)
libc.so.6 => /lib/i686/libc.so.6 (0xb76c0000)
/lib/ld-linux.so.2 (0xb7833000
```

Статические библиотеки не могут иметь таких зависимостей, но это не мешает им участвовать в создании динамических библиотек.

Рассмотрим практический пример создания динамической библиотеки с использованием других библиотек. Для этого нужно создать четыре исходника (по одному на каждую библиотеку и еще один — для исполняемой программы), что иллюстрируют листинги 4.9—4.12.

Листинг 4.9. Исходный файл first.c

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

void do_first (void)
{
       printf ("First library\n");
}
```

Листинг 4.10. Исходный файл second.c

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

void do_second (void)
{
       printf ("Second library\n");
}
```

Листинг 4.11. Исходный файл third.c

```
#include "common.h"

void do_third (void)
{
       do_first ();
       do_second ();
}
```

Листинг 4.12. Программа program.c

```
#include "common.h"

int main (void)
{
       do_third ();
       return 0;
}
```

Общий для всех файл common.h содержит объявления библиотечных функций (листинг 4.13).

Листинг 4.13. Общий файл common.h

```
#ifndef COMMON_H
#define COMMON_H

void do_first (void);
void do_second (void);
void do_third (void);

#endif
```

Теперь создадим make-файл, который откомпилирует все исходные файлы и создаст три библиотеки, одна из которых будет статической (листинг 4.14).

Листинг 4.14. Файл Makefile

```
program: program.c libthird.so
       gcc -o program program.c -L. -lthird \
       -Wl,-rpath,.

libthird.so: third.o libfirst.so libsecond.a
       gcc -shared -o libthird.so third.o -L. \
       -lfirst -lsecond -Wl,-rpath,.

third.o: third.c
       gcc -c -fPIC third.c

libfirst.so: first.c
       gcc -shared -fPIC -o libfirst.so first.c

libsecond.a: second.o
       ar rv libsecond.a second.o

second.o: second.c
       gcc -c second.c
```

```
clean:
	rm -f program libfirst.so libsecond.a \
	libthird.so *.o
```

Итак, make-файл содержит семь целевых связок. Рассмотрим каждую из них по порядку.

1. Бинарник program — создается в один прием из исходного файла program.c с подключением динамической библиотеки libthird.so. Символ \ (бэкслэш) применяется в make-файлах для разбиения длинных строк.
2. Динамическая библиотека libthird.so — получается в результате компоновки файла third.o с подключением динамической библиотеки libfirst.so и архива libsecond.a.
3. Объектный файл third.o, содержащий позиционно-независимый код, — создается компиляцией исходного файла third.c с опцией `-fPIC`.
4. Совместно используемая библиотека libfirst.so — создается из исходного файла first.c. Компиляция и компоновка полученного позиционно-независимого объектного кода осуществляются в один прием.
5. Статическая библиотека libsecond.a — создается путем архивирования единственного файла second.o.
6. Файл second.o — получается в результате компиляции исходника second.c. Как уже отмечалось, для создания статических библиотек используются файлы, содержащие обычный объектный код с фиксированными позициями.
7. Целевая связка `clean` — очищает проект, удаляя исполняемую программу, библиотеки и объектные файлы.

Теперь при помощи программы ldd проверим зависимости, образовавшиеся между библиотеками:

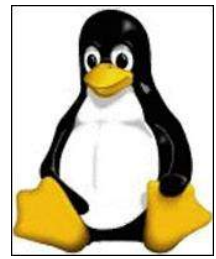
```
$ ldd libthird.so
linux-gate.so.1 =>  (0xffffe000)
libfirst.so => ./libfirst.so (0xb787c000)
libc.so.6 => /lib/i686/libc.so.6 (0xb770c000)
/lib/ld-linux.so.2 (0xb7881000)
```

Как и ожидалось, библиотека libthird.so зависит от libfirst.so. Если вызвать программу ldd для исполняемого файла program, то можно увидеть образовавшуюся цепочку зависимостей полностью:

```
$ ldd program
linux-gate.so.1 =>  (0xffffe000)
libthird.so => ./libthird.so (0xb77ef000)
libc.so.6 => /lib/i686/libc.so.6 (0xb767f000)
libfirst.so => ./libfirst.so (0xb767d000)
/lib/ld-linux.so.2 (0xb77f2000)
```

Очевидно, что архив libsecond.a никак не фигурирует в выводе команды ldd. Статические библиотеки никогда не образуют зависимости, поскольку для обеспечения автономной работы их код полностью включается в результирующий файл.

ГЛАВА 5



Аргументы и опции программы

Каждая программа обрабатывает переданные ей *аргументы* по своему усмотрению. Чтобы такой "индивидуализм" не превращался в головную боль для пользователя, программисты используют механизм *опций*, который значительно облегчает понимание смысла передаваемых в программу аргументов.

В этой главе описываются механизмы работы с опциями. В *разд. 5.4* приводятся ссылки на источники дополнительной информации по данной теме.

5.1. Аргументы программы

Программа, созданная на языке C, может получать данные извне посредством чтения аргументов функции `main()`. Такие данные называют *аргументами программы*. Не следует отождествлять понятия "аргументы программы" и "аргументы командной строки". Дело в том, что программа получает свои аргументы от родительского процесса, в качестве которого не всегда выступает командная оболочка. Таким процессом может оказаться, например, браузер, в котором никакой "командной строки" нет.

Прототип (объявление) функции `main()` можно представить следующим образом:

```
int main (int argc, char ** argv, char ** envp)
```

Третий аргумент, идентичный массиву `environ` и содержащий окружение программы, исторически не прижился. Поэтому функцию `main()` обычно представляют в укороченном виде:

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

Если в программе не предусмотрено чтение аргументов, то функцию `main()` можно объявить еще проще:

```
int main (void)
```

или даже так:

```
int main ()
```

Возможность подобных вариаций с прототипами обусловлена отсутствием единого объявления для функции `main()`. Работая с ее аргументами, компилятор рассчитывает лишь на "здравый рассудок" программиста.

Значение `argc` показывает общее число аргументов программы. Всегда следует помнить, что первый аргумент (`argv[0]`) формально содержит имя программы. Реально в `argv[0]` может находиться что угодно. Об этом будет подробно рассказано в *главе 10*.

Вместо "исторических" имен `argc` и `argv` можно выбрать любые другие. В этом можно легко убедиться, написав небольшую программу (листинг 5.1). Но имена `argc` и `argv` настолько укоренились в культуре программирования, что любой опытный программист наверняка скажет, что приведенный в листинге 5.1 пример написан в дурном стиле.

Листинг 5.1. Программа mainargs.c

```
#include <stdio.h>

int main (int a, char ** b)
{
        int i;
        for (i = 0; i < a; i++)
                printf ("%d: %s\n", i, b[i]);
        return 0;
}
```

Аргументы программы могут интерпретироваться как угодно. Но это, как ни странно, может создавать большие неудобства. Если каждая программа ведет себя по-своему, то работа пользователя может превратиться в кошмар. Чтобы избежать подобных ситуаций, подавляющее большинство программ в Linux используют *опции*.

Опции (options) — это унифицированный интерфейс для передачи данных в программу через аргументы. Механизм опций позволяет разделить все аргументы программы на следующие категории:

- *Опции*. Обычно перед опцией стоит дефис. Различают *короткие* (односимвольные) и *длинные* (многосимвольные) опции. Длинные опции часто начинаются с последовательности из двух дефисов.
- *Зависимые аргументы* (аргументы опций). Некоторые опции предполагают наличие аргумента. Например, имя файла, которое указывается вслед за опцией `-o` компилятора gcc, является зависимым аргументом. Опции, не требующие наличия зависимых аргументов, называют *флагами* (flags). Как правило, одна опция может принимать только один такой аргумент.
- *Свободные аргументы*. Эти аргументы не связаны напрямую с опциями. Например, имя каталога, передаваемое программе rmdir, является свободным аргументом.

Примечание

В *главе 4* мы рассматривали программу ar в контексте создания статических библиотек. Это один из редких случаев, когда стандартная программа в Linux принимает опции без дефиса.

Рассмотрим пример вызова программы с использованием опций:

```
$ gcc -pedantic -c -o program.o program.c
```

Итак, команда состоит из шести частей.

1. `gcc` — это имя программы, которое командная оболочка обычно заносит в `argv[0]`.
2. `-pedantic` — длинная опция, не требующая наличия зависимого аргумента (флаг).
3. `-c` — короткая опция, не требующая наличия свободного аргумента (флаг).
4. `-o` — короткая опция, требующая наличия зависимого аргумента.
5. `program.o` — это зависимый аргумент опции `-o`.
6. `program.c` — свободный аргумент.

5.2. Использование опций

Чтобы использовать опции, не обязательно выполнять самостоятельный синтаксический разбор аргументов программы. В этом разделе будут описаны средства обработки односимвольных (коротких) опций при помощи функции `getopt()`.

Функция `getopt()` объявлена в заголовочном файле getopt.h следующим образом:

```
int getopt (int ARGC, char ** ARGV, const char * OPTS);
```

`ARGC` и `ARGV` — это знакомые нам аргументы функции `main()`. В строке `OPTS` перечислены опции (без разделителей и дефисов), используемые программой. Если после имени следует двоеточие, это означает, что опция требует наличия зависимого аргумента. Например, если `OPTS` указывает на строку `a:bc`, то программа принимает три опции: `-a`, `-b` и `-c`, причем `-a` подразумевает наличие зависимого аргумента, а опции `-b` и `-c` являются флагами.

Обработка опций программы подразумевает неоднократный вызов функции `getopt()`, которая возвращает код очередной фактически переданной опции. Если пользователь ввел опцию, не указанную в строке `OPTS`, то `getopt()` возвращает код символа "?" (знак вопроса). Когда все имеющиеся опции обработаны, `getopt()` возвращает –1.

Для чтения аргументов опций и независимых аргументов предназначены следующие внешние переменные, также объявленные в файле getopt.h:

```
extern char * optarg;
extern int optind;
```

Строка `optarg` содержит зависимый аргумент обрабатываемой опции, а в переменной `optind` хранится индекс первого свободного аргумента в массиве `argv`. При

этом следует учитывать, что `getopt()` перемещает все свободные аргументы в конец `argv`. Таким образом, они оказываются в диапазоне между `optind` и `argc-1`. Если `optind` больше или равно `argc`, то это означает, что свободные аргументы отсутствуют.

Рассмотрим пример программы, которая выводит каждый переданный ей свободный аргумент в отдельной строке (листинг 5.2). Опция `-o` будет перенаправлять вывод в файл, имя которого указывается в зависимом аргументе. А флаг `-h` будет выводить краткую справку по работе с программой.

Листинг 5.2. Программа getoptdemo.c

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

int main (int argc, char ** argv)
{
       FILE * outfile = stdout;
       int i, opt;
       char * filename = NULL;
       char help_str[] =
              "Usage: getoptdemo [OPTIONS] ARGUMENTS ...\n"
              "-h              — Print help and exit\n"
              "-o <outfile>    — Write output to file\n";
       while ((opt = getopt (argc, argv, "ho:")) != -1) {
              switch (opt) {
                     case 'h':
                     printf ("%s", help_str);
                     return 0;

                     case 'o':
                     filename = optarg;
                     break;

                     case '?':
                     fprintf (stderr, "%s", help_str);
                     return 1;

                     default:
                     fprintf (stderr, "Unknown error\n");
                     return 1;
              }
       }

       if (optind >= argc) {
              fprintf (stderr,
              "No argument(s) found\n%s", help_str);
```

```
                return 1;
        }

        if (filename != NULL) {
                outfile = fopen (filename, "w");
                if (outfile == NULL) {
                        fprintf (stderr, "Cannot open "
                        "output file (%s)\n", filename);
                        return 1;
                }
        }

        for (i = optind; i < argc; i++)
                fprintf (outfile, "%s\n", argv[i]);

        if (outfile != stdout) fclose (outfile);
        return 0;
}
```

Рассмотрим все по порядку. Сначала объявляется файловый указатель `outfile`, который инициализируется значением `stdout`. Эта запись говорит о том, что по умолчанию (если не указана опция `-o`) вывод будет осуществляться в стандартный вывод. Указатель `filename` несет двойную нагрузку: если указана опция `-o`, в него будет занесено имя файла; если же в указателе `filename` останется значение `NULL`, это будет означать, что опция `-o` не передавалась. Строка `help_str` содержит краткую справку по работе с программой. Если указать опцию `-h`, то эта справка будет выводиться в стандартный вывод (`stdout`). В случае неправильного набора опций или аргументов, справочная информация будет выводиться в стандартный поток ошибок (`stderr`).

Затем следует цикл обработки опций. Функция `getopt()` вызывается до тех пор, пока не возвратит значение –1, сигнализирующее о том, что все опции обработаны. Опция `-h` подразумевает вызов справки и завершение программы, поэтому вместо инструкции `break` стоит `return`.

Далее обрабатывается опция `-o`, зависимый аргумент которой (`optarg`) заносится в `filename`. Следующий блок выводит в поток ошибок (`stderr`) справочную информацию и завершает программу, если пользователь ввел неверную опцию. Наконец, если `getopt()` возвратила что-то еще, то выводится сообщение о неизвестной ошибке, и программа завершается.

После выхода из цикла проверяется наличие свободных аргументов. Если таковые отсутствуют, то программа завершается, выводя сообщение об ошибке и справочную информацию в `stderr`.

Затем проверяется, пожелал ли пользователь перенаправить вывод в файл. Если опция `-o` была указана, то в `outfile` заносится указатель на открытый файл. В противном случае вывод будет производиться в `stdout` (стандартный вывод).

5.3. Использование длинных опций

Функция `getopt_long()`, объявленная в заголовочном файле getopt.h, позволяет обрабатывать не только короткие, но и длинные (многосимвольные) опции. Следует заметить, что `getopt_long()` устанавливает правило, по которому длинным опциям предшествует последовательность из двух дефисов (`--version`, `--line-number` и т. д.). Итак, функция `getopt_long()` имеет следующий прототип:

```
int getopt_long (int ARGC, char ** ARGV,
                 const char * SHORT_OPTS,
                 const struct option * LONG_OPTS,
                 int * OPT_INDEX);
```

Аргументы `ARGC`, `ARGV` и `SHORT_OPTS` аналогичны соответствующим аргументам функции `getopt()` из предыдущего раздела. `LONG_OPTS` — это массив структур `option`, описывающий длинные опции и соответствующие им короткие опции. По адресу, находящемуся в `OPT_INDEX`, располагается индекс текущей опции в массиве `LONG_OPTS`. Структура `option` состоит из четырех элементов:

```
struct option {
       char * LONG_OPT;
       int WITH_ARG;
       int * OPT_FLAG;
       int SHORT_OPT;
};
```

Строка `LONG_OPT` содержит имя длинной опции (без дефисов). Элемент `WITH_ARG` может принимать три значения:

- 0 или `no_argument` — если опция является флагом, т. е. не требует наличия зависимого аргумента;
- 1 или `required_argument` — если опция требует наличия зависимого аргумента;
- 2 или `optional_argument` — если зависимый аргумент предполагается, но не является обязательным.

Указатель `OPT_FLAG` необходим для работы с опциями, не требующими аргументов. В рамках данной книги мы не будем использовать этот элемент. Если `OPT_FLAG` установлен в `NULL`, то в `SHORT_OPT` заносится код символа соответствующей короткой опции.

Последний элемент массива `LONG_OPTS` должен содержать только нулевые значения:

```
LONG_OPT = NULL;
WITH_ARG = 0;
OPT_FLAG = NULL;
SHORT_OPT = 0;
```

Если в каждом элементе массива `LONG_OPTS` устанавливать `OPT_FLAG` в `NULL`, то функция `getopt_long()` будет работать так же, как и `getopt()`. Указатель `OPT_FLAG` позво-

ляет задействовать расширенные возможности `getopt_long()`, описание которых выходит за рамки данной книги.

Рассмотрим в качестве примера программу из предыдущего раздела, добавив в нее возможность обработки длинных опций (листинг 5.3). Для этого определим следующие соответствия между длинными и короткими опциями:

- длинная опция `--help` будет аналогична короткой `-h`;
- длинная опция `--output` будет аналогична короткой `-o`.

Листинг 5.3. Программа longoptdemo.c

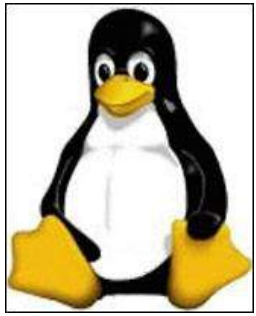
```
#include <stdio.h>
#include <getopt.h>

int main (int argc, char ** argv)
{
       FILE * outfile = stdout;
       int i, opt;
       char * filename = NULL;
       char help_str[] =
       "Usage: longoptdemo [OPTIONS] ARGUMENTS ...\n"
       "-h, --help              — Print help and exit\n"
       "-o, --output <outfile>  — Write output to file\n";
       const struct option long_opts[] = {
              { "help", no_argument, NULL, 'h' },
              { "output", required_argument, NULL, 'o' },
              { NULL, 0, NULL, 0 }
       };

       while ((opt = getopt_long (argc, argv, "ho:",
                           long_opts, NULL)) != -1) {
              switch (opt) {
                     case 'h':
                     printf ("%s", help_str);
                     return 0;

                     case 'o':
                     filename = optarg;
                     break;

                     case '?':
                     fprintf (stderr, "%s", help_str);
                     return 1;

                     default:
                     fprintf (stderr, "Unknown error\n");
```

```
                return 1;
            }
        }

        if (optind >= argc) {
            fprintf (stderr,
            "No argument(s) found\n%s", help_str);
            return 1;
        }

        if (filename != NULL) {
            outfile = fopen (filename, "w");
            if (outfile == NULL) {
                fprintf (stderr, "Cannot open "
                "output file (%s)\n", filename);
                return 1;
            }
        }

        for (i = optind; i < argc; i++)
            fprintf (outfile, "%s\n", argv[i]);

        if (outfile != stdout) fclose (outfile);
        return 0;
}
```

5.4. Получение дополнительной информации

В этой главе рассмотрены далеко не все возможности обработки аргументов программы и применения опций. Дополнительную информацию по этой теме можно получить из следующих источников:

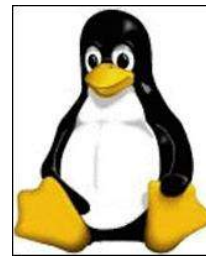
- man 3 getopt — man-страница функции `getopt()`;
- man 3 getopt_long — man-страница функции `getopt_long()`;
- man 3 popt — man-страница библиотеки popt (для доступа к ней может потребоваться установка дополнительных пакетов, таких как libpopt-devel);
- man 1 getopt — man-страница утилиты getopt;
- **http://www.gnu.org/software/gengetopt/gengetopt.html** — обработчик опций GNU Gengetopt;
- **http://www.gnu.org/software/libc/manual/html_node/Argp.html** — обработчик опций Argp.



ЧАСТЬ II

Низкоуровневый ввод-вывод в Linux

ГЛАВА 6



Концепция ввода-вывода в Linux

В этой главе описываются и сравниваются между собой следующие механизмы ввода-вывода в Linux:

- механизмы ввода-вывода стандартной библиотеки языка C;
- низкоуровневые механизмы ввода-вывода Linux;
- механизмы ввода-вывода стандартной библиотеки языка C++.

6.1. Библиотечные механизмы ввода-вывода языка C

Совокупность операций, при помощи которых программа читает и записывает файлы, называется *вводом-выводом* (input/output). Если программа читает и записывает файлы посредством прямого обращения к ядру операционной системы, то такой ввод-вывод называется *низкоуровневым*.

Чтобы понять работу механизмов низкоуровневого ввода-вывода в Linux, рассмотрим сначала принципы чтения и записи файлов в стандартной библиотеке языка C.

Любую модель файлового ввода-вывода можно условно поделить на следующие составляющие:

- *абстракция файла*;
- механизмы *открытия* и *закрытия* файлов;
- механизмы *чтения* и *записи* файлов;
- механизмы *произвольного доступа* к данным в файле.

Применяя эти понятия к вводу-выводу стандартной библиотеки языка C, мы имеем возможность изучать низкоуровневый ввод-вывод в сравнении с аналогичными механизмами высокого уровня.

Абстракция файла в стандартной библиотеке языка C реализуется через тип данных `FILE`. Для этого создается указатель, который в дальнейшем будет участвовать в файловых операциях:

```
FILE * myfile;
```

Особого рассмотрения требуют следующие глобальные экземпляры:

```
extern FILE * stdin;
extern FILE * stdout;
extern FILE * stderr;
```

Эта "троица" представляет собой *стандартный ввод* (`stdin`), *стандартный вывод* (`stdout`) и *стандартный поток ошибок* (`stderr`). Ввод-вывод в Linux построен на одной аксиоме, которая гласит, что любая операция чтения или записи данных сводится к файловому вводу-выводу. Таким образом, любое "нечто", способное принимать или отправлять данные (клавиатура, экран, аппаратные устройства, сетевые соединения и т. п.), можно представить в виде файла.

Стандартный ввод, стандартный вывод и стандартный поток ошибок — это виртуальные "порталы", через которые система взаимодействует с пользователем. Как правило, стандартный ввод связан с драйвером клавиатуры, а стандартный вывод и стандартный поток ошибок ассоциируются с дисплеем компьютера. Но можно, например, связать стандартный вывод с драйвером принтера (который тоже представлен в виде файла). В этом случае вы будете получать информацию не на экране, а на бумаге. И это не просто концепции, а реальность, которую можно воплотить в жизнь одной командой, наподобие этой:

```
$ bash 1>/dev/printer
```

Механизмы открытия и закрытия файлов в стандартной библиотеке языка C представлены следующими функциями:

```
FILE * fopen (const char * FLOCATION, const char * OPEN_MODE);
FILE * freopen (const char * FLOCATION, const char * OPEN_MODE,
                      FILE * FP);
FILE * fclose (FILE * FP);
```

Механизмы чтения и записи файлов реализованы в стандартной библиотеке в виде следующих функций:

```
int fgetc (FILE * FP);
int fputc (int byte, FILE * FP);
char * fgets (char * STR, int SIZE, FILE * FP);
int fputs (const char * STR, FILE * FP);
int fscanf (FILE * FP, const char * FMT, ...);
int fprintf (FILE * FP, const char * FMT, ...);
int vfscanf (FILE * FP, const char * FMT, va_list ap);
int vfprintf (FILE * FP, const char * FMT, va_list ap);
```

Файловый ввод-вывод предполагает, что чтение и запись данных осуществляются последовательно. Для этого к абстракции файла привязывается понятие *текущей позиции ввода-вывода*. Текущая позиция устанавливается в начало файла в момент его открытия. Операции чтения/записи смещают эту позицию вперед на количество прочитанных или записанных байтов. Для принудительного изменения текущей позиции используются механизмы *произвольного доступа* (random access). В стан-

дартной библиотеке языка C произвольный доступ к данным осуществляется при помощи следующих механизмов:

- `#define SEEK_SET 0`
- `#define SEEK_CUR 1`
- `#define SEEK_END 2`
- `typedef struct {...} fpos_t;`
- `int fseek (FILE * FP, long int OFFSET, int WHENCE);`
- `int fgetpos (FILE * FP, fpos_t * POSITION);`
- `int fsetpos (FILE * FP, fpos_t * POSITION);`
- `long int ftell (FILE * FP);`
- `void rewind (FILE * FP);`

Рассмотрим пример программы, которая читает из файла "хвост" заданного размера и выводит его на экран (листинг 6.1). Первый аргумент программы — имя файла, второй аргумент — размер "хвоста". Все операции ввода-вывода осуществляются средствами стандартной библиотеки языка C.

Листинг 6.1. Программа mytail.c

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

int main (int argc, char ** argv)
{
        FILE * infile;
        int ch;
        long int nback;

        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        infile = fopen (argv[1], "r");
        if (infile == NULL) {
        fprintf (stderr, "Cannot open "
                        "input file (%s)\n", argv[1]);
                return 1;
        }

        nback = abs (atoi (argv[2]));
        fseek (infile, 0, SEEK_END);

        if (nback > ftell (infile))
                fseek (infile, 0, SEEK_SET);
```

```
        else
                fseek (infile, -nback, SEEK_END);

        while ((ch = fgetc (infile)) != EOF)
                fputc (ch, stdout);

        fclose (infile);
        return 0;
}
```

Рассмотрим вкратце, что делает эта программа. Сначала создается абстракция файла — указатель `infile`. Функция `fopen()` открывает файл в режиме "только для чтения" (read only). Обратите внимание, что введенный размер "хвоста" может превышать размер файла. Поэтому в программу включен алгоритм проверки, который устанавливает текущую позицию ввода-вывода в начало файла, если "хвост" слишком велик.

После установки текущей позиции ввода-вывода в начало "хвоста" осуществляется его посимвольное чтение с одновременной записью в выходной файл, в качестве которого выступает стандартный поток вывода (`stdout`). Цикл продолжается до тех пор, пока не будет достигнут конец входного файла, который обозначается в стандартной библиотеке языка C константой `EOF`.

Функция `fclose()` закрывает файл. В этом процессе есть свои тонкости. Дело в том, что в Linux применяются механизмы буферизации файлового ввода-вывода. Это означает, что функция `fputc()`, например, реально записывает данные не в устройство ввода-вывода, а в буфер, находящийся в оперативной памяти. Реальная запись данных в устройство производится по усмотрению системы. Такой подход позволяет более эффективно использовать ресурсы операционной системы. Функция `fclose()` отправляет системе запрос на *синхронизацию*. Но следует понимать, что такой запрос лишь принимается на рассмотрение, но не гарантирует реальную запись данных.

Иногда нужно направить системе запрос на синхронизацию, не закрывая файл. Для этого в стандартной библиотеке языка C предусмотрена следующая функция:

```
int fflush (FILE * FP);
```

Существует еще один нюанс: в стандартной библиотеке языка C используется собственный механизм буферизации. Таким образом, `fflush()` отвечает лишь за отправку на запись собственных буферов, а гарантированно заставить ядро синхронизировать данные средствами стандартной библиотеки языка C невозможно.

Даже если файл открыт в режиме "только для чтения", его все равно следует закрывать после всех операций ввода-вывода. Во-первых, это хороший стиль программирования. Во-вторых, открытые файлы расходуют системные ресурсы, которые нужно своевременно освобождать.

6.2. Концепция низкоуровневого ввода-вывода

Рассмотрим теперь особенности низкоуровневого ввода-вывода в Linux, сравнивая его с библиотечным вводом-выводом языка C.

В ядре Linux в качестве абстракций файлов используются *файловые дескрипторы*. В отличие от указателей типа `FILE`, дескрипторы представляют собой просто целые числа (`int`). Ядро создает для каждого процесса индивидуальную *таблицу файловых дескрипторов*, в которой хранится информация об открытых файлах. Итак, абстракция файла в рамках концепции низкоуровневого ввода-вывода представляет собой целое число, являющееся номером записи в таблице дескрипторов текущего процесса.

После открытия файла в таблице дескрипторов создается новая запись с уникальным номером. Чтобы избежать "узурпации" всех системных ресурсов одним процессом, ядро ограничивает размер таблицы файловых дескрипторов определенным значением. В оболочках bash, ksh и zsh присутствует внутренняя команда `ulimit`, с помощью которой можно узнать, сколько файлов может открыть один процесс в вашей системе:

```
$ ulimit -n
1024
```

В оболочках C Shell (csh, tcsh) для этих целей предназначена команда `limit`:

```
$ limit descriptors
descriptors 1024
```

Механизмы открытия и закрытия файлов представлены в ядре следующими системными вызовами:

```
int open (const char * FNAME, int OFLAGS, mode_t MODE);
int open (const char * FNAME, int OFLAGS);
int creat (const char * FNAME, mode_t MODE);
int close (int FD);
```

Чтение и запись файлов осуществляются следующими механизмами:

```
ssize_t read (int FD, void * BUFFER, size_t SIZE);
ssize_t write (int FD, const void * BUFFER, size_t SIZE);
ssize_t readv (int FD, const struct iovec * VECTOR, int SIZE);
ssize_t writev (int FD, const struct iovec * VECTOR, int SIZE);
```

Наконец, при помощи следующего системного вызова осуществляется произвольный доступ к данным в файле:

```
off_t lseek (int FD, off_t OFFSET, int WHENCE);
```

Представленные функции являются системными вызовами. Они отличаются от обычных функций тем, что реализованы в ядре Linux. Иными словами, ядро является для них подобием библиотеки.

6.3. Консольный ввод-вывод

Вернемся ненадолго к механизмам файлового ввода-вывода стандартной библиотеки языка C. Мы уже говорили, что файловые указатели с именами `stdin`, `stdout` и `stderr` связаны со стандартным вводом, стандартным выводом и со стандартным потоком ошибок соответственно. Чтение и запись этих файлов называется *консольным вводом-выводом*. Проведем небольшой эксперимент. Для этого напишем простую программу (листинг 6.2).

Листинг 6.2. Программа stdoutreopen.c

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

int main (void)
{
        fclose (stdout);
        stdout = fopen ("anyfile", "w");
        if (stdout == NULL)
                abort ();

        printf ("Hello World!\n");
        return 0;
}
```

Теперь откомпилируем и запустим эту программу:

```
$ gcc -o stdoutreopen stdoutreopen.c
$ ./stdoutreopen
$ cat anyfile
Hello World!
```

Программа на экран ничего не вывела, поскольку мы сначала закрыли стандартный вывод, а потом открыли его заново, но на этот раз связали его не с терминалом, а с файлом на диске.

Механизмы низкоуровневого ввода-вывода тоже имеют свою абстракцию консольного ввода-вывода. Это файлы с дескрипторами 0 (стандартный ввод), 1 (стандартный вывод) и 2 (стандартный поток ошибок). Реже используется их символическое представление в виде констант препроцессора, объявленных в файле unistd.h:

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

Иногда начинающие программисты не могут понять, чем технически отличается стандартный поток ошибок от стандартного вывода. Во-первых, это разные файлы. Следовательно, они могут быть перенаправлены в разные места. В оболочке bash, например, можно заставить любую программу писать свои ошибки в обычный файл:

```
$ gcc 2>errors
$ cat errors
gcc: no input files
```

Приведенный пример демонстрирует концепцию *перенаправления* (redirection) потоков ввода-вывода средствами оболочки bash. Если бы ошибки выводились функцией `printf()`, то их трудно было бы отделить от обычных сообщений программы.

Во-вторых, стандартный вывод (в отличие от `stderr`) задействует механизм буферизации. Это означает, что сообщения, которые программа пишет в стандартный вывод, отправляются не на терминал, а в буфер. Когда система сочтет нужным, содержимое буфера отправляется на экран терминала.

6.4. Ввод-вывод в C++

В стандартной библиотеке языка C++ ввод-вывод организован в виде мощных объектно-ориентированных конструкций. Абстракцией файла здесь являются объекты классов `ifstream` (для чтения) и `ofstream` (для записи).

В качестве механизмов открытия и закрытия файлов выступают инкапсулированные в классы `ifstream` и `ofstream` функции-члены. Для открытия файлов используется функция-член `open()`, для закрытия — функция-член `close()`. Можно также открывать файлы посредством передачи конструкторам классов `ifstream` и `ofstream` нужных параметров (имя файла и/или режим открытия файла).

Механизмы чтения и записи также инкапсулированы в классы `ifstream` и `ofstream`. Чтение осуществляется функцией `get()` или перегруженным оператором `>>`, запись — функцией `put()` или оператором `<<`.

Произвольный доступ осуществляется инкапсулированными в классы `ifstream` и `ofstream` функциями-членами. Для `ifstream()` это функция `seekg()`, для класса `ofstream` — `seekp()`. Для чтения текущей позиции служат функции-члены `tellg()` и `tellp()`. Начало файла, конец файла и текущая позиция обозначаются в стандартной библиотеке языка C++ следующими символическими именами:

```
ios::beg
ios::end
ios::cur
```

В качестве примера перепишем программу mytail (см. листинг 6.1) на язык C++ (листинг 6.3).

Листинг 6.3. Программа mytail.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
```

```
int main (int argc, char ** argv)
{
        long int nback;
        if (argc < 3) {
                cerr << "Too few arguments" << endl;
                return 1;
        }

        char ch;
        ifstream infile (argv[1], ios::in | ios::binary);
        if (!infile) {
                cerr << "Cannot open input file "
                        << argv[1] << endl;
                return 1;
        }

        infile.seekg (0, ios::end);
        nback = abs (atoi (argv[2]));

        if (nback > infile.tellg ())
                infile.seekg (0, ios::beg);
        else
                infile.seekg (-nback, ios::end);

        while (infile.get (ch)) cout << ch;

        infile.close ();
        return 0;
}
```

Для компиляции программ на C++ используется компилятор g++ (он же c++), входящий в состав пакета компиляторов GCC (GNU Compiler Collection):

```
$ g++ -o mytail mytail.cpp
```

Рассмотрим эту программу по порядку. Сначала пространство имен `std` включается в глобальное пространство имен следующей инструкцией:

```
using namespace std;
```

Это делается для того, чтобы перед каждым именем из стандартной библиотеки языка C++ не писать префикс `std::`.

В качестве абстракции входного файла создается экземпляр `infile` класса `ifstream`. Файл открывается при помощи конструктора. Хотя это можно было бы сделать функцией-членом `open()`:

```
ifstream infile;
infile.open (argv[1], ios::in | ios::binary);
```

Затем мы устанавливаем текущую позицию ввода-вывода в начало "хвоста". Это делается в несколько приемов, поскольку пользователь может ввести слишком длинный "хвост", превышающий размер файла. Функция-член `tellg()` позволяет узнать текущую позицию относительно начала файла. Следующим приемом мы фактически определяем размер файла:

```
infile.seekg(0, ios::end);
filesize = infile.tellg ();
```

Таким образом, если число байтов, на которое нужно отступить назад от конца файла (`nback`), превышает размер этого файла, то текущая позиция ввода-вывода устанавливается в `ios::beg`.

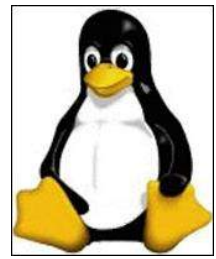
Ввод осуществляется посимвольно функцией `get()`. Вывод на экран выполняется оператором <<. Функция-член `close()` закрывает файл.

Особого внимания заслуживают механизмы осуществления консольного ввода-вывода в C++. В C++ есть объекты консольного ввода-вывода для обычных и *широких* (16-разрядных) символов, а также поддерживается еще один поток, предназначенный для *журналирования* (logging), который по умолчанию направлен на экран и использует механизм буферизации. Все указанные объекты ввода-вывода перечислены в табл. 6.1.

Таблица 6.1. *Объекты ввода-вывода в C++*

Объект	Класс	Что символизирует
`cin`	`istream`	Стандартный ввод
`cout`	`ostream`	Стандартный вывод
`cerr`	`ostream`	Стандартный поток ошибок
`wcin`	`wistream`	Стандартный ввод для широких символов
`wcout`	`wostream`	Стандартный вывод для широких символов
`wcerr`	`wostream`	Стандартный поток ошибок для широких символов
`clog`	`ostream`	Поток журнала для 8-разрядных символов
`wclog`	`wostream`	То же для 16-разрядных символов

ГЛАВА 7



Базовые операции ввода-вывода

Эта глава описывает основные операции низкоуровневого ввода-вывода в Linux:

- создание и открытие файлов (системные вызовы `creat()` и `open()`);
- закрытие файлов (системный вызов `close()`);
- чтение и запись файлов (системные вызовы `read()` и `write()`);
- перемещение текущей позиции ввода-вывода в файлах (системный вызов `lseek()`).

Отдельно будут рассмотрены основные положения, связанные с понятиями *режима файла* и *прав доступа*.

7.1. Создание файла: *creat()*

Прежде чем мы перейдем к детальному рассмотрению механизмов низкоуровневого ввода-вывода в Linux, определим понятие *режима файла*. К каждому файлу в Linux привязана цепочка из 16 битов, которые можно условно разделить на три группы:

1. Биты 0—8: *базовые права доступа*.
2. Биты 9—11: *расширенные права доступа*.
3. Биты 12—15: *тип файла*.

Эта цепочка называется *режимом файла* (file mode). Довольно часто режим файла отождествляют с *правами доступа* (permissions). Это не является грубой ошибкой, но и не совсем верно.

Обратите внимание на 3-ю группу. Здесь *тип файла* определяет не характер содержащейся в нем информации (текст, изображение, музыка и т. д.), а особенности низкоуровневого ввода-вывода. По этому критерию ядро разделяет все файлы на следующие типы:

- *Обычный файл* (regular file) — эти файлы предназначены для хранения данных. Бинарники и библиотеки также относятся к обычным файлам.

- *Символьное устройство* (character device) — об этих файлах речь пойдет в *разд. 13.5*.
- *Блочное устройство* (block device) — эти файлы также будут описаны в *разд. 13.5*.
- *Каталог* (directory) — каталоги также являются файлами, о них речь пойдет в *главе 15*.
- *Символическая ссылка* (symbolic link) — указывает на другие файлы. Она создается программой ln с флагом `-s`.
- *Канал FIFO* — именованные каналы FIFO (First In First Out) участвуют в межпроцессном взаимодействии. О них речь пойдет в *главе 24*.
- *Сокет* (socket) — сокеты — универсальное средство межпроцессного взаимодействия между разнородными системами. Именно при помощи сокетов компьютер под управлением Windows, например, может загружать Web-страницы с Linux-сервера. О сокетах будет рассказано в *главе 25*.

Linux — многопользовательская операционная система. Чтобы разграничить полномочия пользователей в отношении файлов, в ядре реализована концепция *прав доступа*, согласно которой каждый файл имеет *владельца* (user, owner) и *группу* (group). Владелец у файла всегда один, а в группе может состоять несколько пользователей. Кроме того, выделяют также понятие *другой пользователь* (others), т. е. тот, кто не обладает правами *суперпользователя* (root), не является владельцем и не состоит в указанной группе.

Для каждой из перечисленных категорий (владелец, группа, другие) устанавливаются индивидуальные права доступа исходя из трех перечисленных далее критериев.

1. *Право на чтение* (read) — дает пользователю возможность читать данные из файла. Если файл является каталогом, то право на чтение интерпретируется как возможность просматривать его содержимое.
2. *Право на запись* (write) — предоставляет пользователю возможность записывать данные в файл. Если же файл является каталогом, то право на запись интерпретируется как возможность создавать, удалять и переименовывать файлы, находящиеся в этом каталоге.
3. *Право на выполнение* (execute) — дает пользователю возможность запускать файл. Если файл является каталогом, то право на выполнение разрешает пользователю "заходить" в этот каталог, т. е. делать его *текущим* (current).

> ***Примечание***
>
> Понятие "текущий каталог" реализовано не в оболочке, а в ядре Linux. Об этом будет рассказано в *разд. 14.2*.

Биты из диапазона от 0 до 8 режима файла описывают перечисленные права доступа для владельца, группы и остальных пользователей. Для указания прав доступа программисты могут использовать специальные именованные *константы режима*, соединяя их операцией побитовой дизъюнкции (ИЛИ). Чтобы получить доступ к

константам режима, необходимо включить в программу заголовочный файл sys/stat.h.

Рассмотрим теперь, что означает каждый бит базовых прав доступа и какая константа предназначена для его установки (табл. 7.1).

Таблица 7.1. *Назначение битов прав доступа и соответствующие им константы*

Бит	Права	Константа
0	Выполнение для остальных пользователей	S_IXOTH
1	Запись для остальных пользователей	S_IWOTH
2	Чтение для остальных пользователей	S_IROTH
3	Выполнение для группы	S_IXGRP
4	Запись для группы	S_IWGRP
5	Чтение для группы	S_IRGRP
6	Выполнение для владельца	S_IXUSR
7	Запись для владельца	S_IWUSR
8	Чтение для владельца	S_IRUSR

Указать режим файла можно с помощью типа mode_t, который представляет собой целое число, размерность которого зависит от реализации. Для этого в программу необходимо включить заголовочный файл sys/types.h. Полный список констант режима приведен в *приложении 1*.

Примечание

В ранних Unix-подобных системах вместо mode_t использовался тип int.

Предположим, что в программе нужно выставить права "чтение и запись для владельца", а также "чтение для группы". Это делается так:

```
S_IRUSR | S_IWUSR | S_IRGRP
```

Теперь рассмотрим системный вызов creat(), который создает и открывает файл с указанным режимом, возвращая файловый дескриптор. Прототип функции creat() объявлен в заголовочном файле fcntl.h следующим образом:

```
int creat (const char * FILENAME, mode_t MODE);
```

Первый аргумент (FILENAME) — это имя файла. Второй аргумент (MODE) — режим файла. Следует отметить, что в таблице дескрипторов не может быть отрицательных значений. Если системный вызов creat() вернул –1, это символизирует о том, что файл не удалось создать и открыть.

Примечание

Некоторые особенности аргумента MODE системного вызова creat() будут описаны в *главе 16*.

Теперь напишем программу, которая создает файл с указанным именем и правами на чтение для владельца (листинг 7.1).

Листинг 7.1. Программа creat1.c

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

int main (int argc, char ** argv)
{
        int fd;
        mode_t fmode = S_IRUSR;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = creat (argv[1], fmode);
        if (fd == -1) {
                fprintf (stderr,
                "Cannot create file (%s)\n", argv[1]);
                return 1;
        }

        close (fd);
        return 0;
}
```

В программе сначала создается переменная целого типа, в которую системный вызов `creat()` при удачном завершении поместит дескриптор файла. В переменную `fmode` заносится единственная константа `S_IRUSR`, устанавливающая в единицу 8-й бит режима файла (чтение для владельца). Затем следует вызов `creat()`, который в случае неудачи возвращает –1. Системный вызов `close()` (объявленный в файле unistd.h) будет описан в *разд. 7.3*. Нетрудно догадаться, что он закрывает файл.

Приведенная программа довольно проста, но самое интересное еще впереди. При первом запуске все идет по плану:

```
$ gcc -o creat1 creat1.c
$ ./creat1 myfile
$ ls -l myfile
-r-------- 1 nnivanov nnivanov 0 2011-05-05 11:50 myfile
```

Действительно, был создан файл нулевой длины с требуемыми правами доступа. Но если запустить программу creat1 еще раз с тем же аргументом, получится сюрприз:

```
$ ./creat1 myfile
Cannot create file (myfile)
```

И дело не в том, что файл уже существует, а в том, что права доступа выставлены только на чтение. Если файл существует, `creat()` очищает его, не меняя текущих прав доступа. Но поскольку при первом вызове программы файл был создан с правами доступа "только для чтения", то вызов `creat()` не смог очистить его (даже если нечего было чистить) и завершился неудачей. Эту ситуацию легко исправить:

```
$ chmod u+w myfile
$ ls -l myfile
-rw------- 1 nnivanov nnivanov 0 2011-05-05 11:50 myfile
$ ./creat1 myfile
$ ls -l myfile
-rw------- 1 nnivanov nnivanov 0 2011-05-05 11:52 myfile
```

Обратите внимание, что права доступа у файла не изменились, как и ожидалось. Можно также проверить и тезис о том, что `creat()` очищает существующие файлы:

```
$ echo hello > myfile
$ cat myfile
hello
$ ./creat1 myfile
$ cat myfile
$
```

7.2. Открытие файла: *open()*

Мы уже знаем один способ открытия файла посредством системного вызова `creat()`. Но в этом случае файл всегда открывается в режиме "только запись" и, к тому же, если файл уже существует, то он попросту будет очищен. Кроме того, нет никаких адекватных механизмов проверки того, существовал ли файл вообще.

Открыть файл определенным образом можно через системный вызов `open()`. Приятная особенность `open()` — возможность установки *флагов открытия* файла, благодаря которым процесс открытия файла становится контролируемым.

Флаги открытия файла (как и флаги режима) представляют собой набор символических констант, которые могут соединяться операцией побитовой дизъюнкции (ИЛИ). Чтобы использовать эти флаги и системный вызов `open()`, в программу необходимо включить заголовочный файл fcntl.h.

> ***Примечание***
>
> Обычно имена заголовочных файлов перекликаются с механизмами, которые в них объявлены. Например, stdio.h расшифровывается как "STanDard Input/Output". Неблагозвучное имя fcntl.h образовано от словосочетания "File CoNTroL".

Результат объединения флагов открытия файла представляет собой обычное целое число. Полный перечень флагов приведен в *приложении 1*. В следующем списке указаны лишь самые "популярные" из них:

- `O_RDONLY` — открыть файл только для чтения;
- `O_WRONLY` — открыть файл только для записи;
- `O_RDWR` — открыть файл для чтения и записи;
- `O_CREAT` — создать файл, если не существует;
- `O_TRUNC` — очистить файл, если он существует;
- `O_APPEND` — дописать в существующий файл;
- `O_EXCL` — не открывать файл, если он существует (используется с флагом `O_CREAT`).

Системный вызов `open()`, объявленный в заголовочном файле fcntl.h, имеет два прототипа:

```
int open (const char * FILENAME, int FLAGS, mode_t MODE);
int open (const char * FILENAME, int FLAGS);
```

Как уже сообщалось в *главе 6*, подобная двойственность прототипа достигается благодаря механизму `va_arg` (он же `stdarg`), являющемуся частью стандарта языка C.

> ***Примечание***
>
> Механизм `stdarg` хорошо документирован и в данной книге отдельно не рассматривается.

Аргументы `FILENAME` и `MODE` полностью аналогичны соответствующим аргументам системного вызова `creat()`. Второй прототип `open()` обычно используется для открытия файлов в режиме "только чтение", когда указание режима является избыточным. Аргумент `FLAGS` представляет собой набор флагов открытия, объединенных операцией побитового *ИЛИ*. Следует отметить, что некоторые флаги не могут объединяться между собой. Здесь нужно рассуждать логически. Нельзя, например, объединить флаги `O_RDONLY` и `O_WRONLY`, поскольку они противоречат друг другу.

Очень часто в качестве аргумента `MODE` системных вызовов `creat()` и `open()` указывают не объединенные операцией побитовой дизъюнкции (ИЛИ) именованные константы, а числа в восьмеричной системе счисления. Такое представление называют *восьмеричным представлением прав доступа*. Возможно, вы никогда не будете применять такой подход, однако умение видеть права доступа в восьмеричной форме может пригодиться при чтении чужих программ. Кроме того, восьмеричная интерпретация прав доступа широко распространена в кругу системных администраторов.

> ***Примечание***
>
> В ранних Unix-подобных системах не было именованных констант из sys/stat.h. Для записи прав доступа использовались только восьмеричные числа.

Для записи прав доступа, которые занимают 9 бит режима файла, достаточно трех восьмеричных цифр. При этом следует учитывать, что биты отсчитываются справа налево (little endian). Восьмеричная запись битов прав доступа удобна тем, что каждой восьмеричной цифре соответствует последовательность из трех бит.

> ***Примечание***
>
> Некоторые дополнительные особенности аргумента `MODE` системного вызова `open()` будут описаны в *главе 16*.

Чтобы проверить, как это работает на практике, напишем небольшую программу, которая для начала выводит на экран права доступа в десятичном представлении (листинг 7.2).

Листинг 7.2. Программа mymask.c

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

int main (void)
{
       mode_t test_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
       printf ("%d\n", test_mode);
       return 0;
}
```

Проверим, что получилось:

```
$ gcc -o mymask mymask.c
$ ./mymask
420
```

Итак, правам доступа "чтение и запись для владельца" плюс "чтение для группы и остальных пользователей" соответствует число 420. Теперь переведем это число в восьмеричную систему счисления:

```
$ bc -q
obase=8
420
644
quit
```

Получилось 644. Переведем полученное число в двоичную систему счисления:

```
$ bc -q
ibase=8
obase=2
644
110100100
quit
```

Теперь мысленно пронумеруйте полученную цепочку от 0 до 8 справа налево. Это и есть биты прав доступа. Кроме того, если наложить эти биты на привычный нам шаблон `rwxrwxrwx`, который выводит программа ls, заменив все нули дефисами, то получим права доступа в удобном для чтения виде:

```
rw-r--r--
```

Теперь можно систематизировать знания, полученные в результате эксперимента. Итак, восьмеричные права доступа состоят из трех цифр:

- первая цифра — права доступа для владельца;
- вторая цифра — права доступа для группы;
- третья цифра — права доступа для остальных пользователей.

Если запомнить, что означает каждая цифра, то работа с восьмеричными правами не будет вызывать никаких затруднений (в скобках даны двоичные значения):

- 0 — отсутствие прав доступа (000);
- 1 — только выполнение (001);
- 2 — только запись (010);
- 3 — запись и выполнение (011);
- 4 — только чтение (100);
- 5 — чтение и выполнение (101);
- 6 — чтение и запись (110);
- 7 — чтение, запись и выполнение (111).

Восьмеричные права доступа также можно использовать с программой chmod:

```
$ touch myfile
$ ls -l myfile
-rw-r--r-- 1 nnivanov nnivanov 0 2011-05-05 12:02 myfile
$ chmod 600 myfile
$ ls -l myfile
-rw------- 1 nnivanov nnivanov 0 2011-05-05 12:02 myfile
```

Работая с таким представлением прав доступа в программах на языке C, следует помнить, что восьмеричное число должно начинаться с нуля (в противном случае оно будет интерпретировано компилятором как десятичное).

Рассмотрим теперь программу, которая создает файл при помощи системного вызова `open()` (листинг 7.3).

Листинг 7.3. Программа creatopen.c

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

```
int main (int argc, char ** argv)
{
        int fd;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0640);
        if (fd == -1) {
                fprintf (stderr, "Cannot create file (%s)\n",
                                argv[1]);
                return 1;
        }

        close (fd);
        return 0;
}
```

В первую очередь следует отметить, что системный вызов `creat()` полностью идентичен `open()`, вызванному с флагами открытия `O_WRONLY`, `O_CREAT` и `O_TRUNC`. Если не верите, посмотрите в исходниках ядра Linux файл fs/open.c.

В данную программу не включаются файлы sys/stat.h и sys/types.h, поскольку мы не пользуемся ни константами режима, ни типом `mode_t`. Заголовочный файл unistd.h нужен для системного вызова `close()`, о котором пойдет речь в следующем разделе.

7.3. Закрытие файла: *close()*

Системный вызов `close()` освобождает файловый дескриптор. Если использовалась буферизация записи данных, то `close()` посылает в ядро запрос на синхронизацию. Но это вовсе не означает, что данные будут сразу же записаны на носитель.

Системный вызов `close()` объявлен в файле unistd.h следующим образом:

```
int close (int FD);
```

В предыдущих примерах мы умышленно не замечали, что `close()` может что-то возвращать. Ошибки при закрытии файла — большая редкость, но все-таки они иногда случаются. Поэтому системный вызов `close()` возвращает 0 при удачном завершении или –1 в случае ошибки.

Теперь напишем программу, которая специально "провоцирует" `close()` на ошибочное завершение (листинг 7.4). Для этого попытаемся закрыть один и тот же файл дважды.

Листинг 7.4. Программа closetwice.c

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

```
int main (int argc, char ** argv)
{
        int fd;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);
        if (fd == -1) {
                fprintf (stderr, "Cannot create file (%s)\n",
                                argv[1]);
                return 2;
        }

        if (close (fd) == -1) {
                fprintf (stderr, "Cannot close file "
                                "with descriptor=%d\n", fd);
                return 3;
        }

        if (close (fd) == -1) {
                fprintf (stderr, "Cannot close file "
                                "with descriptor=%d\n", fd);
                return 4;
        }
        return 0;
}
```

Эта программа аварийная по определению. Обратите внимание, что все блоки обработки ошибок передают инструкции `return` различные числовые значения. Вы наверняка знаете, что функция `main()` возвращает статус завершения программы, называемый *кодом возврата*. Исторически сложилась практика, согласно которой нулевой код возврата символизирует об успешном завершении программы, а ненулевой код — об ошибке. Следующая команда позволяет узнать код возврата последней завершившейся программы:

```
$ echo $?
```

> ***Примечание***
>
> Команда `echo $?` поддерживается даже в оболочках семейства C Shell (csh, tcsh), которые вечно делают все "по-своему".

В нашем примере, оба блока проверки `close()` выводят одно и то же сообщение. Но мы теперь можем узнать, где именно произошла ошибка:

```
$ ./closetwice myfile
Cannot close file with descriptor=3
$ echo $?
4
```

В программе closetwice (см. листинг 7.4) есть еще один "подводный камень". Попробуйте запустить эту программу еще раз, указав то же имя файла, что и в предыдущем запуске:

```
$ ./closetwice myfile
Cannot create file (myfile)
$ echo $?
2
```

На этот раз до закрытия дело не дошло. Всему виной стал флаг `O_EXCL`, который не допускает повторного открытия уже существующего файла.

7.4. Чтение файла: *read()*

Для чтения файлов служит системный вызов `read()`, объявленный в файле unistd.h следующим образом:

```
ssize_t read (int FD, void * BUFFER, size_t SIZE);
```

Этот системный вызов пытается прочитать `SIZE` байт из файла с дескриптором `FD`. Прочитанная информация заносится в `BUFFER`. Возвращаемое значение — число реально прочитанных байтов.

Аргумент `size_t` — это беззнаковый целый тип, размерность которого зависит от реализации; `ssize_t` тоже является целым, но уже знаковым типом. Указание именно `ssize_t` обусловлено тем, что `read()` возвращает –1 в случае ошибки. Когда текущая позиция ввода-вывода достигает конца файла, `read()` возвращает 0.

В качестве буфера может выступать практически любой блок памяти известного размера. Но чаще всего в системном вызове `read()` используются строки. Рассмотрим пример программы, которая выводит на экран содержимое указанного файла, читая его посимвольно при помощи системного вызова `read()` (листинг 7.5).

Листинг 7.5. Программа read1.c

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

int main (int argc, char ** argv)
{
        int fd;
        char ch;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file (%s)\n",
                                argv[1]);
                return 1;
        }

        while (read (fd, &ch, 1) > 0) printf ("%c", ch);

        if (close (fd) == -1) {
                fprintf (stderr, "Cannot close file "
                                "with descriptor=%d\n", fd);
                return 1;
        }
        return 0;
}
```

Итак, в качестве буфера был передан адрес переменной типа `char`, и чтение осуществлялось по одному байту. Такая схема проста, но очень неэффективна. Обращение к системному вызову `read()` для чтения каждого символа значительно замедляет работу программы. Только представьте себе: чтобы прочитать файл размером 1 Кбайт, программа обращается к ядру 1024 раза! Поэтому файлы лучше читать достаточно большими блоками. Обычно для этих целей создается буфер размером в несколько килобайт.

Напишем теперь программу, которая делает то же, что и предыдущая, только более эффективным способом (листинг 7.6).

Листинг 7.6. Программа read2.c

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

#define BUF_SIZE       4096
char buffer[BUF_SIZE+1];

int main (int argc, char ** argv)
{
        ssize_t bytes;
        int fd;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file (%s)\n",
                                argv[1]);
                return 1;
        }

        while ((bytes = read (fd, buffer, BUF_SIZE)) > 0)
        {
                buffer [bytes] = 0;
                printf ("%s", buffer);
        }

        close (fd);
        return 0;
}
```

В этой версии программы появилась новая переменная `bytes`, в которую записывается размер прочитанной информации. Буфером является массив `buffer`. Обратите внимание, что каждый раз в конец этого массива дописывается нуль-терминатор. Это обусловлено тем, что вывод осуществляется функцией `printf()`, для которой признаком окончания вывода служит символ с кодом 0. Всегда следует помнить, что системный вызов `read()` просто читает байты, а ответственность за формирование строки возложена на программиста.

Функция `printf()` предназначена для вывода данных на экран терминала. Поэтому она не может воспринимать некоторые байты "как есть". Рассмотрим следующую команду:

```
$ ./read1 foo1 > foo2
```

В этом случае файл foo1 будет скопирован в foo2, если исходный файл не содержит определенные символы, которые `printf()` воспринимает как команды форматирования (звонок, символ с кодом 0 и т. д.). В связи с этим, программа read1 не может быть универсальным средством копирования файлов.

Напишем еще одну программу, упрощенную версию утилиты cp, которая будет осуществлять полноценное копирование файла (листинг 7.7).

Листинг 7.7. Программа копирования файлов mycp.c

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

#define BUF_SIZE       4096
char buffer [BUF_SIZE];
```

```
int main (int argc, char ** argv)
{
        int ifd, i;
        FILE * outfile;
        ssize_t bytes;

        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 0;
        }

        ifd = open (argv[1], O_RDONLY);
        if (ifd == -1) {
                fprintf (stderr, "Cannot open input file "
                                "(%s)\n", argv[1]);
                return 1;
        }

        outfile = fopen (argv[2], "w");
        if (outfile == NULL) {
                fprintf (stderr, "Cannot open output file "
                                "(%s)\n", argv[2]);
                return 1;
        }

        while ((bytes = read (ifd, buffer, BUF_SIZE)) > 0) {
                for (i = 0; i < bytes; i++)
                        fputc (buffer[i], outfile);
        }

        close (ifd);
        fclose (outfile);
        return 0;
}
```

Эта программа реализует вывод с помощью функции `fputc()`, которая безразлична к любым символам. В данном случае запись данных осуществляется посимвольно в цикле, поэтому нет необходимости устанавливать нуль-терминатор.

7.5. Запись файла: *write()*

Системный вызов `write()`, объявленный в заголовочном файле unistd.h, предназначен для записи файлов:

```
ssize_t write (int FD, const void * BUFFER, size_t SIZE);
```

Этот системный вызов пытается записать `SIZE` байтов из буфера `BUFFER` в файл с дескриптором `FD`. Возвращаемое значение — число реально записанных байтов. В случае ошибки возвращается –1.

В качестве примера перепишем программу mycp из предыдущего раздела таким образом, чтобы запись в файл осуществлялась посредством системного вызова `write()` (листинг 7.8).

Листинг 7.8. Модифицированная программа mycp1.c

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

#define BUF_SIZE       4096
char buffer [BUF_SIZE];

int main (int argc, char ** argv)
{
       int ifd, ofd;
       ssize_t bytes;

       if (argc < 3) {
              fprintf (stderr, "Too few arguments\n");
              return 0;
       }

       ifd = open (argv[1], O_RDONLY);
       if (ifd == -1) {
              fprintf (stderr, "Cannot open input file "
                           "(%s)\n", argv[1]);
              return 1;
       }

       ofd = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0640);
       if (ofd == -1) {
              fprintf (stderr, "Cannot open output file "
                           "(%s)\n", argv[2]);
              return 1;
       }

       while ((bytes = read (ifd, buffer, BUF_SIZE)) > 0)
                     write (ofd, buffer, bytes);

       close (ifd);
       close (ofd);
       return 0;
}
```

Как видно из примера, системный вызов `write()` очень похож на своего "собрата" `read()`. Рассмотрим теперь, как посредством этих системных вызовов осуществляется консольный ввод-вывод.

Вспомним ненадолго окружение. Мы знаем, что оно наследуется (копируется) от родительского процесса к дочернему. Примерно то же самое происходит и с файлами: таблица дескрипторов при запуске процесса заполняется открытыми файлами родителя. Но, в отличие от окружения, которое в каждом процессе представлено независимой копией, наследование дескрипторов обладает двумя особенностями: *файл не будет закрыт до тех пор, пока его не закроют все процессы, в которых он открыт*. Иначе говоря, если процесс A с открытым файлом F породил процесс B, то для закрытия файла F необходимо, чтобы оба процесса (A и B) закрыли этот файл. Обратите внимание, что это условие не является достаточным, поскольку файл F мог также достаться процессу A от его родителя; если процессы разделяют общий файловый дескриптор, то они разделяют также и текущую позицию ввода-вывода.

> ***Замечание***
>
> Указанные особенности распространяются не на файлы вообще, а только на разделяемые дескрипторы.

Как правило, если речь идет об обычных файлах, родитель "умалчивает" о том, что находится в таблице дескрипторов. В этом случае преемственность сохраняется лишь формально. Но если речь идет о консольном вводе-выводе, то здесь потомок гарантированно получает (и это ему известно) три дескриптора с фиксированными номерами:

- 0 — стандартный ввод (stdin);
- 1 — стандартный вывод (stdout);
- 2 — стандартный поток ошибок (stderr).

Таким образом, мы можем вывести на экран какое-нибудь сообщение при помощи системного вызова `write()`:

```
write (1, message, strlen (message));
```

Иногда вместо непосредственных номеров дескрипторов используют символические константы, объявленные в файле unistd.h:

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

Многие программисты по праву считают такие объявления избыточными и продолжают пользоваться обычными числами. Рассмотрим теперь программу, которая читает строку из стандартного ввода и выводит ее на экран в обратном направлении (листинг 7.9). Считывание происходит до тех пор, пока не встретится символ переноса строки.

Листинг 7.9. Программа reverse.c

```
#include <unistd.h>

#define BUF_SIZE       4096
char buffer [BUF_SIZE];
```

```
int main (void)
{
        int i;
        for (i = 0; (read (0, &buffer[i], 1) > 0) ||
                                (i < BUF_SIZE); i++) {
                        if (buffer[i] == '\n') {
                                i--;
                                break;
                        }
        }

        for (; i >= 0; i--) write (1, &buffer[i], 1);
        write (1, "\n", 1);
        return 0;
}
```

Получилась забавная вещь:

```
$ ./reverse
кашалот как компилятор
ротялипмок как толашак
```

Но главная особенность этой программы — возможность участвовать в *конвейере*:

```
$ uname | ./reverse
xuniL
```

7.6. Произвольный доступ: *lseek()*

Системный вызов `lseek()` позволяет произвольно перемещать текущую позицию ввода-вывода. `lseek()` объявлен в заголовочном файле unistd.h следующим образом:

```
off_t lseek (int FD, off_t OFFSET, int WHENCE);
```

Тип `off_t` является целым числом, размерность которого зависит от реализации. Как правило, это длинное целое (`long int`). `FD` — это файловый дескриптор, `OFFSET` — требуемое смещение относительно параметра `WHENCE`, который может быть представлен одной из трех констант:

- `SEEK_SET` — начало файла;
- `SEEK_CUR` — текущая позиция ввода-вывода;
- `SEEK_END` — конец файла.

Аргумент `OFFSET` может принимать отрицательные значения, смещающие текущую позицию назад от `WHENCE`.

Системный вызов `lseek()` возвращает установленную позицию ввода-вывода относительно начала файла. В случае ошибки возвращается –1.

В ядре Linux нет отдельного системного вызова, который сообщал бы текущую позицию в файле. Для этих целей применяется все тот же `lseek()`:

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

Рассмотрим теперь уже знакомый нам пример, который выводит на экран "хвост" указанного файла (листинг 7.10). Для реализации этой программы будем пользоваться низкоуровневыми механизмами ввода-вывода Linux.

Листинг 7.10. Программа mylseektail.c

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

#define BUF_SIZE       4096
char buffer[BUF_SIZE];

int main (int argc, char ** argv)
{
       int fd;
       off_t nback;
       ssize_t nbytes;
       char arg_emsg[] = "Too few arguments\n";
       char file_emsg[] = "Cannot open input file\n";
       char close_emsg[] = "Cannot close file\n";

       if (argc < 3) {
              write (2, arg_emsg, strlen (arg_emsg));
              return 1;
       }

       fd = open (argv[1], O_RDONLY);
       if (fd == -1) {
              write (2, file_emsg, strlen (file_emsg));
              return 1;
       }

       nback = abs (atoi (argv[2]));
       lseek (fd, 0, SEEK_END);
       if (nback > lseek (fd, 0, SEEK_CUR))
              lseek (fd, 0, SEEK_SET);
       else
              lseek (fd, -nback, SEEK_END);

       while ((nbytes = read (fd, buffer, BUF_SIZE)) > 0)
              write (1, buffer, nbytes);
```

```
        if (close (fd) == -1) {
                write (2, close_emsg, strlen (close_emsg));
                return 1;
        }
        return 0;
}
```

Итак, рассмотрим листинг 7.10 по порядку. В заголовочном файле fcntl.h объявлен системный вызов `open()`. Файл unistd.h нужен для `read()`, `write()`, `lseek()` и `close()`. В sys/types.h объявлены типы `off_t` и `ssize_t`, а в заголовочном файле string.h — функция `strlen()`.

Затем создается константа препроцессора `BUF_SIZE` и массив `buffer`, через который будут считываться и записываться данные. В языке C существует также другой подход для работы со статическими массивами. Для этого достаточно объявить следующий макрос:

```
#define ARRAY_SIZE(array) (sizeof(array)/sizeof(array[0]))
```

Этот макрос вычисляет на стадии компиляции размер статического массива:

```
...
#define ARRAY_SIZE(array) (sizeof(array)/sizeof(array[0]))
char buffer[4096];
...
while ((nbytes = read (fd, buffer, ARRAY_SIZE (buffer)))
        write (1, buffer, nbytes);
...
```

Следует учитывать, что такой "трюк" не работает с динамическими массивами.

В данной программе все операции ввода-вывода осуществляются при помощи системных вызовов `read()` и `write()`, включая вывод сообщений об ошибках. Для этого сначала создаются массивы `arg_emsg`, `file_emsg` и `close_emsg`. Вывод ошибок реализуется следующим образом:

```
write (2, arg_emsg, strlen (arg_emsg));
write (2, file_emsg, strlen (arg_emsg));
write (2, close_emsg, strlen (close_emsg));
```

Другими словами, мы просто записываем данные необходимого размера в файл с дескриптором 2, т. е. в стандартный поток ошибок. Учитывая то, что массивы `arg_emsg`, `file_emsg` и `close_emsg` статические, можно вообще не пользоваться функцией `strlen()`:

```
#define ARRAY_SIZE(array) (sizeof(array)/sizeof(array[0]))
write (2, arg_emsg, ARRAY_SIZE (arg_emsg));
write (2, file_emsg, ARRAY_SIZE (file_emsg));
write (2, close_emsg, ARRAY_SIZE (close_emsg));
```

Для закрепления материала главы рассмотрим еще одну программу, которая читает из файла заданное число байтов от указанной позиции. Чтобы программа была бо-

лее "продвинутой", научим ее обрабатывать опции функцией `getopt_long()`. Для начала определим набор опций и аргументов.

- Опция `-h` (`--help`) будет выводить краткую справочную информацию по работе с программой.
- Опция `-o` (`--output`) будет использоваться для того, чтобы перенаправить вывод в файл. Имя файла указывается в зависимом аргументе. Если опция `-o` не указывается, то выходная информация будет направляться в стандартный вывод.
- Опция `-p` (`--position`) передает программе позицию в файле (через зависимый аргумент), откуда будет производиться чтение информации. Если опция `-p` не указана, то по умолчанию чтение производится с начала файла.
- Опция `-c` (`--count`) через зависимый аргумент передает программе число байтов, которые требуется прочитать. Эта опция является обязательной, если не указан флаг `-h` (`--help`).
- Флаг `-n` (`--newline`) указывает программе, что вывод надо закончить переносом строки.
- Обязательный независимый аргумент передает программе имя файла, из которого будет происходить чтение.

Исходный код программы (листинг 7.11) достаточно велик, но он позволяет увидеть в действии практически все базовые принципы низкоуровневого ввода-вывода. Кроме того, данная программа является еще одной полезной иллюстрацией обработки длинных опций.

Листинг 7.11. Пример реализации низкоуровневого ввода-вывода readblock.c

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

#define ARR_SIZE(array) (sizeof(array)/sizeof(array[0]))

int main (int argc, char ** argv)
{
       int i, opt, ifd, ofd = 1, nflag = 0;
       char ch;
       off_t pos;
       size_t count;
       char help_str[] = "Usage: readblock OPTIONS FILENAME\n"
              "OPTIONS:\n"
              "-h, --help\n"
              "-o, --output <filename>\n"
              "-p, --position <number>\n"
```

```
        "-c, --count <number>\n"
        "-n, --newline\n";

char unkn_emsg[] = "Unknown error\n";
char ifile_emsg[] = "Cannot open input file\n";
char ofile_emsg[] = "Cannot open output file\n";
char close_emsg[] = "Cannot close file\n";
char lseek_emsg[] = "Cannot set I/O position\n";

char * ofname = NULL;
char * pos_str = NULL;
char * count_str = NULL;

const struct option long_opts[] = {
        { "help", no_argument, NULL, 'h' },
        { "output", required_argument, NULL, 'o' },
        { "position", required_argument, NULL, 'p' },
        { "count", required_argument, NULL, 'c' },
        { "newline", no_argument, NULL, 'n' },
        { NULL, 0, NULL, 0 }
};

while ((opt = getopt_long (argc, argv, "ho:p:c:n",
                    long_opts, NULL)) != -1) {
        switch (opt) {
                case 'h':
                write (1, help_str, ARR_SIZE (help_str));
                return 0;

                case 'o':
                ofname = optarg;
                break;

                case 'p':
                pos_str = optarg;
                break;

                case 'c':
                count_str = optarg;
                break;

                case 'n':
                nflag = 1;
                break;

                case '?':
                write (2, help_str, ARR_SIZE (help_str));
                return 1;
```

```
                default:
                write (2, unkn_emsg, ARR_SIZE (unkn_emsg));
                return 2;
        }
}

if (count_str == NULL) {
        write (2, help_str, ARR_SIZE (help_str));
        return 3;
}
count = abs(atoi(count_str));

if (pos_str != NULL) pos = abs (atoi (pos_str));
else pos = 0;

if (optind >= argc) {
        write (2, help_str, ARR_SIZE (help_str));
        return 4;
}

if (ofname != NULL) {
        ofd = open (ofname, O_WRONLY | O_CREAT | O_TRUNC,
                S_IRUSR | S_IWUSR | S_IRGRP); /* 0640 */
        if (ofd == -1) {
                write (2, ofile_emsg, ARR_SIZE (ofile_emsg));
                return 5;
        }
}

ifd = open (argv[optind], O_RDONLY);
if (ifd == -1) {
        write (2, ifile_emsg, ARR_SIZE (ifile_emsg));
        return 6;
}

if (pos > lseek (ifd, 0, SEEK_END)) {
        count = 0;
}
else if (lseek (ifd, pos, SEEK_SET) == -1) {
        write (2, lseek_emsg, ARR_SIZE (lseek_emsg));
        return 7;
}

for (i = 0; i < count; i++) {
        if (read (ifd, &ch, 1) <= 0) break;
        write (ofd, &ch, 1);
}
```

```
        if (nflag) write (ofd, "\n", 1);

        if (close (ifd) == -1) {
                write (2, close_emsg, ARR_SIZE (close_emsg));
                return 8;
        }

        if (ofd != 1) {
                if (close (ofd) == -1) {
                        write (2, close_emsg,
                                ARR_SIZE (close_emsg));
                        return 9;
                }
        }
        return 0;
}
```

Рассмотрим все по порядку. Сначала в программу включаются заголовочные файлы, которые объявляют следующее:

- unistd.h — объявляет системные вызовы `read()`, `write()`, `lseek()` и `close()`, а также константы `SEEK_SET`, `SEEK_CUR` и `SEEK_END`;
- fcntl.h — системный вызов `open()`, а также константы `O_RDONLY`, `O_WRONLY`, `O_CREAT` и `O_TRUNC`;
- sys/types.h — типы `off_t` и `size_t`;
- sys/stat.h — константы режима файла `S_IRUSR`, `S_IWUSR` и `S_IRGRP`;
- getopt.h — структуру `option`, переменные `optarg` и `optind`, а также функцию `getopt_long()`;
- stdlib.h — функции `abs()` и `atoi()`.

Макрос `ARR_SIZE()` вычисляет на стадии компиляции размер статического массива. В функции `main()` сначала объявляются различные вспомогательные переменные:

- целая переменная `i` — счетчик для цикла;
- целая переменная `opt` — для получения и обработки значения, возвращаемого функцией `getopt_long()`;
- целая переменная `ifd` — для хранения дескриптора входного файла;
- целая переменная `ofd` — для хранения дескриптора выходного файла, который по умолчанию (если не указана опция `-o`) связан со стандартным выводом;
- целая переменная `nflag` — показывает, нужно ли завершать вывод переносом строки;
- символьная переменная `ch` — для ввода-вывода;
- переменная `pos` — для вычисления и хранения начальной позиции чтения файла;

- переменная `count` — для вычисления и хранения числа байтов, подлежащих выводу.

Строки `help_str`, `unkn_emsg`, `ifile_emsg`, `ofile_emsg`, `close_emsg` и `lseek_emsg` служат для вывода информационных сообщений. Как правило, это сообщения об ошибках.

Указатели `ofname`, `pos_str` и `count_str` используются для получения зависимых аргументов опций `-o`, `-p` и `-c` соответственно. Изначально эти указатели инициализируются значением `NULL`. В этом есть определенный смысл: если, например, указатель `ofname` после цикла обработки опций будет по-прежнему содержать `NULL`, то это будет означать, что опция `-o` (`--output`) не указывалась.

Массив структур `long_opts` содержит описание длинных опций и их аргументов. Кроме того, этот массив устанавливает соответствия между длинными и короткими опциями. Так, например, длинной опции `--position` соответствует короткая опция `-p`.

После всех объявлений следует цикл обработки опций, который продолжается до тех пор, пока функция `getopt_long()` не вернет значение –1, символизирующее о том, что больше обрабатывать нечего. С каждым проходом цикла функция `getopt_long()` возвращает в переменную `opt` код следующей полученной опции. Если была получена неизвестная опция, то в `opt` помещается код символа "?" (вопросительный знак). Каждая программа может по-своему реагировать на появление неизвестной опции. Наша программа в этом случае выводит в стандартный поток ошибок краткую справку (`help_str`) и завершается с кодом возврата 1. Если `getopt_long()` возвращает что-то иное (блок `default` конструкции `switch`), то выводится сообщение об ошибке, и программа завершается с кодом возврата 2.

Обратите внимание, что в данной программе каждая ошибка завершает программу с индивидуальным кодом возврата. В нашем случае это не просто удобная, но и необходимая мера.

Все ошибки, связанные с неправильной передачей в программу опций или аргументов, выводят одну и ту же справочную информацию. Поэтому получение кода возврата — практически единственный способ узнать, где именно произошла ошибка.

В данной программе консольный ввод-вывод реализован через системный вызов `write()`. Таким образом здесь продемонстрировано, что для осуществления консольного ввода-вывода можно обойтись исключительно средствами ядра Linux. Хотя это не всегда удобно и целесообразно.

После цикла следует проверка того, была ли указана опция `-c`. В предварительном описании программы сообщалось, что эта опция является обязательной, поэтому если переменная `count_str` по-прежнему содержит значение `NULL`, то программа выводит краткую инструкцию и завершается с кодом возврата 3. Если проверка была удачно пройдена, то `count_str` переводится в числовое значение, которое заносится в переменную `count`. Функция получения модуля `abs()` в данной программе предна-

значена для уменьшения числа проверок и сохранения компактности исходного кода. Будет замечательно, если вы самостоятельно добавите блок обработки отрицательного значения для `count`.

Затем идет обработка `pos_str`. Если опция `-p` указывалась, то в переменную `pos` заносится свободный аргумент этой опции, приведенный функциями `abs()` и `atoi()` к целому неотрицательному значению. Если указатель `pos_str` по-прежнему содержит значение `NULL` (опция `-p` не указывалась), то переменная `pos` устанавливается в нулевое значение. Это говорит о том, что данные будут считываться с начала файла.

Далее идет проверка наличия обязательного свободного аргумента, в который заносится имя входного файла. Если аргумент отсутствует, то в стандартный поток ошибок направляется краткая справка (`help_str`), и программа завершается с кодом возврата 4.

После этого, если была указана опция `-o`, о чем свидетельствует отличное от `NULL` значение указателя `ofname`, предпринимается попытка открыть файл с флагами `O_WRONLY`, `O_CREAT` и `O_TRUNC` в режиме, соответствующем восьмеричным правам доступа `0640`. То же самое можно было сделать при помощи системного вызова `creat()`. Если файл не удалось открыть, то выводится сообщение об ошибке, и программа завершается с кодом возврата 5.

Потом предпринимается попытка открыть входной файл с флагом `O_RDONLY` (только чтение). В случае неудачи выводится сообщение об ошибке, и программа завершается с кодом возврата 6.

Если входной файл был успешно открыт, то программа переходит к установке текущей позиции ввода-вывода для этого файла относительно его начала. Здесь учитывается, что указанное значение `pos` может превышать размер файла. В этом случае переменная `count` обнуляется, как бы символизируя о том, что из указанной позиции нечего выводить. Если же значение переменной `pos` не превышает размера файла, то выполняется попытка установки текущей позиции ввода-вывода на указанное число байтов относительно начала файла (`SEEK_SET`). Если что-то пошло не так, то в стандартный поток ошибок выводится сообщение (`help_str`), и программа завершается с кодом возврата 7.

Наконец, когда все предварительные проверки успешно пройдены, программа посимвольно считывает указанный блок входного файла и записывает каждый символ в выходной файл. Естественно, это не самый оптимальный вариант. Попробуйте самостоятельно проделать то же самое, но с использованием механизма буферизации.

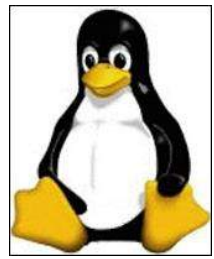
Если программе был передан флаг `-n`, то в выходной файл заносится символ переноса строки. Эта опция полезна, когда вывод осуществляется на экран: очередное приглашение командной строки в этом случае не прилипает к "хвосту" вывода программы, а появляется на привычном месте.

Теперь, когда все прочитано и записано, необходимо закрыть файлы. Если входной файл по каким-либо причинам не удалось закрыть, то выводится сообщение

об ошибке, и программа завершается с кодом возврата 8. Чтобы закрыть выходной файл, нужно сначала убедиться, что он не является стандартным выводом. Если произошла ошибка закрытия выходного файла, то выводится соответствующее сообщение, и программа завершается с кодом возврата 9.

Внимательный читатель заметит, что выходной файл можно и не проверять на соответствие значению по умолчанию, поскольку закрытие стандартного вывода перед выходом из программы не вызывает никаких негативных последствий. Вспомните, что говорилось ранее: чтобы реально закрыть файл, необходимо вызвать `close()` в *каждом* процессе, использующем этот файл.

ГЛАВА 8



Расширенные возможности ввода-вывода в Linux

В этой главе описываются дополнительные возможности низкоуровневого ввода-вывода в Linux: взаимодействие низкоуровневого и библиотечного ввода-вывода, векторное чтение, векторная запись, а также концепция "черных дыр".

8.1. Взаимодействие с библиотечными механизмами

Библиотечные механизмы ввода-вывода языка C реализованы на основе системных вызовов. Это легко проверить с помощью очень полезной утилиты strace, которая запускает переданную ей программу и выводит в стандартный поток ошибок отчет об использованных системных вызовах. Чтобы перенаправить вывод этой программы в отдельный файл, указывается опция `-o`. Итак, чтобы проверить механизм в действии, напишем простую программу, выводящую на экран знаменитое приветствие "Hello World" (листинг 8.1).

Листинг 8.1. Программа hw.c

```
#include <stdio.h>

int main (void)
{
       printf ("Hello World\n");
       return 0;
}
```

Теперь соберем программу и запустим ее с помощью утилиты strace:

```
$ gcc -o hw hw.c
$ strace -o report hw
Hello World
```

Если теперь посмотреть содержимое файла report, то можно обнаружить, что даже такая простая программа задействует много системных вызовов. Но нас сейчас интересует лишь ввод-вывод:

```
$ grep write report
write (1, "Hello World\n", 12)        = 12
```

Что и требовалось доказать: функция `printf()` действительно вывела приветствие при помощи системного вызова `write()`.

Возникает разумный вопрос: можно ли получить дескриптор файла из указателя `FILE`? Можно! Для этого предусмотрена функция `fileno()`, объявленная в заголовочном файле stdio.h следующим образом:

```
int fileno (FILE * FP);
```

Эта функция возвращает дескриптор, соответствующий файловому указателю `FP`. В случае ошибки возвращается –1.

Чтобы продемонстрировать работу функции `fileno()`, рассмотрим простой пример, который выводит содержимое указанного файла на экран (листинг 8.2).

Листинг 8.2. Пример filenodemo.c

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

#define ARR_SIZE(array) (sizeof(array)/sizeof(array[0]))
char buffer[4096];

int main (int argc, char ** argv)
{
       FILE * ifile;
       int fd;
       int nbytes;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       ifile = fopen (argv[1], "r");
       if (ifile == NULL) {
              fprintf (stderr, "Cannot open file "
                                  "(%s)\n", argv[1]);
              return 1;
       }

       if ((fd = fileno (ifile)) == -1) {
              fprintf (stderr, "Cannot get descriptor\n");
              return 1;
       }
```

```
        while ((nbytes = read (fd, buffer, ARR_SIZE (buffer))) > 0)
                        write (1, buffer, nbytes);
        fclose (ifile);
        return 0;
}
```

Здесь важно понимать, что функция `fclose()` освобождает дескриптор. Аналогичным образом, если вызвать системный вызов `close()` для дескриптора `fd`, то файловый указатель `ifile` будет считаться закрытым.

Иногда требуется провести обратную процедуру, т. е. получить из дескриптора файловый указатель. Это можно сделать при помощи функции `fdopen()`, объявленной в заголовочном файле stdio.h:

```
FILE * fdopen (int FD, const char * MODE);
```

Эта функция возвращает файловый указатель, соответствующий открытому файлу с дескриптором `FD`. В случае ошибки возвращается `NULL`. Аргумент `MODE` аналогичен соответствующему аргументу функции `fopen()`. Здесь важно следить, чтобы параметр `MODE` не противоречил флагам, соответствующим дескриптору `FD`. Например, если `FD` открыт только для чтения, то функция `fdopen()` не сможет создать файловый указатель, открытый только для записи. Следующая программа демонстрирует эту ошибку (листинг 8.3).

Листинг 8.3. Программа brokenfile.c

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

int main (int argc, char ** argv)
{
        int fd;
        FILE * filep;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                "(%s)\n", argv[1]);
                return 2;
        }

        filep = fdopen (fd, "w");
        if (filep == NULL) {
                fprintf (stderr, "Cannot create file pointer\n");
```

```
                return 3;
        }

        close (fd);
        return 0;
}
```

Эта программа завершается ошибкой. Рассмотрим теперь корректный пример, который выводит содержимое файла, используя библиотечные функции `fgetc()` и `fputc()` (листинг 8.4).

Листинг 8.4. Пример fdopendemo.c

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

int main (int argc, char ** argv)
{
        int fd;
        FILE * ifile;
        char ch;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                "(%s)\n", argv[1]);
                return 2;
        }

        ifile = fdopen (fd, "r");
        if (ifile == NULL) {
                fprintf (stderr, "Cannot create file pointer\n");
                return 3;
        }

        while ((ch = fgetc (ifile)) != EOF)
                fputc (ch, stdout);

        close (fd);
        return 0;
}
```

В этом примере ввод-вывод осуществляется посимвольно. Попробуйте теперь самостоятельно переделать эту программу так, чтобы данные читались и записывались через буфер.

8.2. Векторное чтение: *readv()*

Низкоуровневые механизмы ввода-вывода в Linux не ограничиваются только системными вызовами `creat()`, `open()`, `close()`, `read()`, `write()` и `lseek()`. В этом разделе будет описана еще одна концепция низкоуровневого ввода-вывода — *векторное чтение* файла.

Сначала файл (или его фрагмент) условно разделяется на блоки фиксированной длины, каждый из которых в процессе чтения будет помещаться в отдельный буфер.

Векторное чтение осуществляется через системный вызов `readv()`. В качестве абстракции файла здесь выступает обычный файловый дескриптор. Системный вызов `readv()` объявлен в файле sys/uio.h следующим образом:

```
ssize_t readv (int FD, struct iovec * VECTOR, int VSIZE);
```

Этот системный вызов записывает данные из файла с дескриптором `FD` в массив `VECTOR`, состоящий из `VSIZE` элементов. Возвращаемое значение — число прочитанных байтов. В случае ошибки возвращается –1.

Структура `iovec` объявлена в файле sys/uio.h следующим образом:

```
struct iovec
{
       void * iov_base;
       size_t iov_len;
};
```

> ***Примечание***
>
> На самом деле структура `iovec` объявлена в другом заголовочном файле (bits/uio.h), который включается в файл sys/uio.h. Но вас это никак не должно волновать.

Элемент `iov_base` — это буфер, в который читается блок информации, а `iov_len` — размер буфера.

В качестве примера рассмотрим программу (листинг 8.5), которая читает информацию о первом файле из архива, созданного программой tar. Для сохранения компактности будем считывать только следующую информацию:

- имя файла (каталога);
- режим файла (восьмеричное представление);
- идентификатор пользователя (UID);
- идентификатор группы (GID);
- размер файла.

Каждому пользователю в системе присваивается индивидуальное число, которое называется *идентификатором пользователя* (User IDentifier). Аналогичным образом каждая группа в системе имеет свой уникальный номер, называемый *идентификатором группы* (Group IDentifier). При помощи следующей команды пользователь может узнать свой UID:

```
$ id
uid=500(nn) gid=100(users) groups=6(disk),11(floppy)
```

Подробно об идентификаторах пользователей и групп будет рассказано в *главе 13*.

В tar-архивах перед каждым упакованным файлом расположен заголовок, содержащий информацию об этом файле. Заголовок состоит из полей фиксированной длины. Обычно в заголовке каждого заархивированного файла содержится 16 полей, но мы рассмотрим только 5 полей первого файла, общей протяженностью 136 байт:

1. Имя файла — 100 байт. Режим файла (восьмеричное представление) — 8 байт.
2. UID в восьмеричной системе счисления — 8 байт.
3. GID в восьмеричной системе счисления — 8 байт. Размер файла в восьмеричной системе счисления — 12 байт.

Полный список полей можно посмотреть в исходниках программы tar (файл src/tar.h, структура `posix_header`).

В нашей программе массив структур `iovec` будет статическим, а каждый буфер в этом массиве будет выделяться в *куче* (heap) при помощи функции `malloc()`.

Листинг 8.5. Программа readtar.c

```
#include <sys/uio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define ARR_SIZE(array) (sizeof(array)/sizeof(array[0]))

int main (int argc, char ** argv)
{
       int i, fd;
       int tfields[] = {    100,     /* Name */
                            8,       /* Mode */
                            8,       /* UID */
                            8,       /* GID */
                            12 };    /* Size (octal) */

       struct iovec tarhead [ARR_SIZE(tfields)];

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
```

```
                return 1;
        }

        for (i = 0; i < ARR_SIZE (tfields); i++)
        {
                tarhead[i].iov_base = (char *)
                        malloc (sizeof (char) * tfields[i]);

                if (tarhead[i].iov_base == NULL) {
                        fprintf (stderr, "Cannot allocate "
                                "memory for tarhead[%d]\n", i);
                        return 1;
                }

                tarhead[i].iov_len = tfields[i];
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                "(%s)\n", argv[1]);
                return 1;
        }

        if (readv (fd, tarhead, ARR_SIZE (tarhead)) <= 0) {
                fprintf (stderr, "Cannot read header\n");
                return 1;
        }

        printf ("name: %s\n", tarhead[0].iov_base);
        printf ("mode: %s\n", tarhead[1].iov_base);
        printf ("uid: %d\n",
                strtol (tarhead[2].iov_base, NULL, 8));
        printf ("gid: %d\n",
                strtol (tarhead[3].iov_base, NULL, 8));
        printf ("size: %d\n",
                strtol (tarhead[4].iov_base, NULL, 8));

        for (i = 0; i < ARR_SIZE (tarhead); i++)
                free (tarhead[i].iov_base);

        close (fd);
        return 0;
}
```

Рассмотрим программу по порядку. Сначала в нее включаются следующие заголовочные файлы:

- sys/uio.h объявляет системный вызов `readv()` и структуру `iovec`;
- fcntl.h объявляет системный вызов `open()`, а также флаг `O_RDONLY`;

- unistd.h объявляет системный вызов `close()`;
- stdio.h объявляет механизмы ввода-вывода стандартной библиотеки языка C;
- stdlib.h объявляет функции `malloc()`, `free()`, `strtol()` и `atoi()`.

Затем создается уже знакомый нам макрос `ARR_SIZE()`, вычисляющий на стадии компиляции размер статического массива. В функции `main()` сначала объявляются две целые переменные: `i` (счетчик цикла) и `fd` (файловый дескриптор). После этого создается вспомогательный статический массив `tfields`, состоящий из целых чисел. Эти числа показывают размеры полей tar-архива. Потом объявляется массив структур `iovec`, в который будут считываться данные.

После проверки наличия аргумента выделяется память для буферов массива `tarhead`. Если выделение памяти прошло успешно, файл открывается в режиме "только для чтения". Все данные читаются одной инструкцией:

```
readv (fd, tarhead, ARR_SIZE (tarhead));
```

Затем эти данные выводятся на экран. Восьмеричные числа переводятся в десятичные функцией `strtol()`. И, наконец, буферы освобождаются и файл закрывается.

8.3. Векторная запись: *writev()*

Векторная запись осуществляется практически так же, как и векторное чтение. Для этого предназначен системный вызов `writev()`, объявленный в файле sys/uio.h следующим образом:

```
ssize_t writev (int FD, const struct iovec * VECTOR, int VSIZE);
```

Все аргументы этого системного вызова несут тот же смысл, что и в `readv()`. Чтобы продемонстрировать работу системного вызова `writev()`, напишем несложную адресную книгу (листинг 8.6). Рабочие данные (имена, телефоны и адреса электронной почты) будут храниться в файле, который будет представлять собой небольшую базу данных с полями фиксированной длины.

Программа может работать с пользователем в двух режимах:

1. Добавление новой записи (аргумент `add`).
2. Поиск номера телефона и адреса электронной почты по имени их владельца (аргумент `find`).

Для сохранения компактности в программе умышленно опущены некоторые проверки (размер вводимых строк и т. п.). Кроме того, программа "не умеет" удалять записи из базы данных адресной книги, но эта функция будет добавлена в *главе 18*.

Листинг 8.6. Пример адресной книги abook.c

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

```
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define ABOOK_FNAME       "abook"
#define NAME_LENGTH       50
#define PHONE_LENGTH       30
#define EMAIL_LENGTH       30

struct iovec ab_entry[3]; /* 1) NAME; 2) PHONE and 3) E-MAIL */
char name_buffer [NAME_LENGTH];
char phone_buffer [PHONE_LENGTH];
char email_buffer [EMAIL_LENGTH];

void abook_failed (int retcode)
{
       fprintf (stderr, "Cannot open address book\n");
       exit (retcode);
}

void abook_add (void)
{
       int fd;
       printf ("Name: ");
       scanf ("%s", ab_entry[0].iov_base);
       printf ("Phone number: ");
       scanf ("%s", ab_entry[1].iov_base);
       printf ("E-mail: ");
       scanf ("%s", ab_entry[2].iov_base);

       fd = open (ABOOK_FNAME, O_WRONLY | O_CREAT | O_APPEND,
                     S_IRUSR | S_IWUSR | S_IRGRP);
       if (fd == -1) abook_failed (1);

       if (writev (fd, ab_entry, 3) <= 0) {
              fprintf (stderr, "Cannot write to address book\n");
              exit (1);
       }
       close (fd);
}

void abook_find (void)
{
       int fd;
       char find_buffer [NAME_LENGTH];
       printf ("Name: ");
```

```
        scanf ("%s", find_buffer);
        fd = open (ABOOK_FNAME, O_RDONLY);
        if (fd == -1) abook_failed (1);

        while (readv (fd, ab_entry, 3) > 0)
        {
                if (!strcmp (find_buffer, ab_entry[0].iov_base))
                {
                        printf ("Phone: %s\n",
                                ab_entry[1].iov_base);
                        printf ("E-mail: %s\n",
                                ab_entry[2].iov_base);
                        goto close;
                }
        }

        printf ("Name '%s' hasn't found\n", find_buffer);
close:
        close (fd);
}

int main (int argc, char ** argv)
{
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        ab_entry[0].iov_base = name_buffer;
        ab_entry[0].iov_len = NAME_LENGTH;
        ab_entry[1].iov_base = phone_buffer;
        ab_entry[1].iov_len = PHONE_LENGTH;
        ab_entry[2].iov_base = email_buffer;
        ab_entry[2].iov_len = EMAIL_LENGTH;

        if (!strcmp (argv[1], "add")) {
                abook_add ();
        } else if (!strcmp (argv[1], "find")) {
                abook_find ();
        } else {
                fprintf (stderr, "%s: unknown command\n"
                "Usage: abook { add , find }\n", argv[1]);
                return 1;
        }
        return 0;
}
```

Особого внимания заслуживает функция `abook_find()`. Многие программисты считают, что "вредоносный" оператор `goto` не имеет права на существование. Действительно, злоупотребление этим оператором может запутать программу и привести к многим скрытым ошибкам. Но в нашем случае использование `goto` *оправдано* и не несет в себе никаких негативных последствий.

В данной программе для чтения и записи предусмотрен общий массив структур `iovec`. Память для буферов `iov_base` здесь выделяется статически. Функция `abook_failed()` служит для обработки ошибок, связанных с невозможностью открытия файла адресной книги. Функция `abook_add()` добавляет запись в адресную книгу. Для этого файл открывается с флагом `O_APPEND` (добавить в конец файла). Функция `abook_find()` осуществляет поиск в адресной книге. Обратите внимание, что функция `readv()` работает так же, как и `read()`: считывает данные до тех пор, пока не будет достигнут конец файла. Отличие лишь в том, что читается не одиночный буфер, а структурированный набор данных фиксированной длины.

8.4. Концепция "черных дыр"

Начнем сразу с эксперимента. Для этого создадим программу, которая необычным образом использует системный вызов `lseek()` (листинг 8.7).

Листинг 8.7. Программа blackhole.c

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

#define HOLE_SIZE       5

int main (void)
{
       int fd = creat ("myfile", 0640);
       if (fd == -1) {
              fprintf (stderr, "Cannot open file\n");
              return 1;
       }

       write (fd, "AAA", 3);
       if (lseek (fd, HOLE_SIZE, SEEK_END) == -1) {
              fprintf (stderr, "Cannot set I/O position\n");
              return 1;
       }

       write (fd, "BBB", 3);
       close (fd);
       return 0;
}
```

Эта программа записывает в файл myfile три байта информации, затем выводит текущую позицию ввода-вывода за границы файла, после чего записывает еще три байта. В результате получился файл длиной в 11 байт:

```
$ ls -l myfile
-rw-r----- 1 nnivanov nnivanov 11 2011-05-06 12:48 myfile
```

Возникает вопрос: что же содержится в промежутке между первой и второй триадами записанных байтов? В этом случае каждый текстовый редактор будет отображать данный промежуток по-своему. Например, текстовый процессор OpenOffice.org Writer, в котором, кстати, написана эта книга, выстроил цепочку из пяти решеток (#). А текстовый редактор gedit вообще отказался открывать этот файл. Но если воспользоваться программой cat, то никаких промежуточных символов мы не обнаружим:

```
$ cat myfile
AAABBB
```

Таким образом, пользуясь привычными программами, мы не сможем выяснить, что на самом деле находится между "AAA" и "BBB". Выход только один — написать еще одну программу, которая выведет ASCII-код каждого символа, содержащегося в myfile (листинг 8.8).

Листинг 8.8. Программа showhole.c

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

int main (int argc, char ** argv)
{
        int fd;
        char ch;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file\n");
                return 1;
        }

        while (read (fd, &ch, 1) > 0)
                printf ("code: %d\n", ch);

        close (fd);
        return 0;
}
```

Оказалось, что наш файл содержит в промежутке нулевые байты:

```
$ ./showhole myfile
code: 65
code: 65
code: 65
code: 0
code: 0
code: 0
code: 0
code: 0
code: 66
code: 66
code: 66
```

> ***Примечание***
>
> Можно было бы воспользоваться утилитой hexdump, которая всегда выводит информацию "как есть", не занимаясь интерпретацией полученных данных. Но сделать что-то самостоятельно всегда намного интереснее!

На первый взгляд — ничего особенного. Но давайте попробуем увеличить нашу "черную дыру" до 20 Мбайт. Для этого изменим значение константы `HOLE_SIZE` в файле blackhole.c:

```
#define HOLE_SIZE (1024*1024*20)
```

Не торопитесь запускать полученную программу. Давайте сначала удалим прежний файл myfile и проверим, сколько мегабайт памяти занято в файловой системе:

```
$ rm myfile
$ df -mT .
Filesystem    Type   1M-blocks      Used Available Use% Mounted on
/dev/sda2     ext4      230888     38059    181101  18% /
```

> ***Примечание***
>
> Утилита df служит для просмотра информации об использовании файловой системы и дисков.

Теперь запустим программу и проверим, что получилось:

```
$ ./blackhole
$ ls -lh myfile
-rw-r----- 1 nnivanov nnivanov 21M 2011-05-06 13:04 myfile
$ df -mT .
Filesystem    Type   1M-blocks      Used Available Use% Mounted on
/dev/sda2     ext4      230888     38059    181101  18% /
```

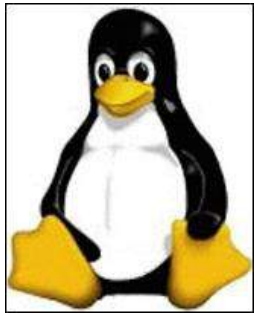
Итак, несмотря на то, что в файловой системе появился файл размером более 20 Мбайт, это никак не отразилось на статистике использования дискового пространства.

В приведенном примере опция `-T` программы df была указана не просто так. Оказывается, "продвинутые" файловые системы, наподобие ext3/ext4, реагируют на

"черные дыры" в файлах особенным образом: нули, которыми заполнена "черная дыра", на самом деле не записываются в файл. Каждый раз, когда программа посредством системного вызова `read()` обращается к ядру с запросом на чтение данных из "пустого" промежутка, *файловая система просто генерирует нулевые байты*.

> ***Примечание***
>
> Флаг `-T` заставляет утилиту df выводить тип файловой системы. Опция `-m` указывает на то, чтобы размеры выводились в мегабайтах, однако эта опция почему-то отсутствует во многих описаниях утилиты df.

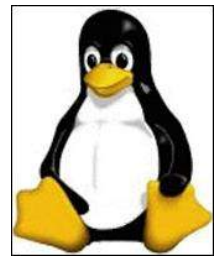
Следует помнить, что такое поведение характерно не для всех типов файловых систем. Если, например, поместить файл myfile в файловую систему ОС Windows, то диск реально заполнится 20-ю мегабайтами нулей.



ЧАСТЬ III

Многозадачность

ГЛАВА 9



Основы многозадачности в Linux

Вы уже знаете, что многозадачность в Linux построена на процессах, которые образуют иерархию, называемую деревом процессов, а во главе этой иерархии находится процесс init. Также упоминалось, что процесс, порождающий другой процесс, называется родителем или родительским процессом, а порожденный процесс называется потомком. Теперь пришла пора опробовать все это на практике.

В этой главе будут рассмотрены следующие темы:

- Реализация многозадачности с использованием библиотечной функции `system()`.
- Древовидная иерархия процессов в Linux.
- Способы получения информации о процессе.

9.1. Библиотечный подход: *system()*

Самый простой способ породить в Linux новый процесс — вызвать библиотечную функцию `system()`, которая просто передает команду в оболочку, на которую ссылается файл /bin/sh. Функция `system()` объявлена в заголовочном файле stdlib.h следующим образом:

```
int system (const char * COMMAND);
```

Рассмотрим пример (листинг 9.1), демонстрирующий работу функции `system()`.

Листинг 9.1. Пример system1.c

```
#include <stdlib.h>

int main (void)
{
        system ("uname");
        return 0;
}
```

Вот что получилось:

```
$ gcc -o system1 system1.c
$ ./system1
Linux
```

Примечание

Утилита uname выводит начальную информацию о системе. По умолчанию эта программа сообщает название ядра системы (в нашем случае — Linux). Дело в том, что uname является частью пакета GNU coreutils. Руководитель проекта GNU, Ричард Столлман, настаивает на правильности и чистоте использования имени Linux. Согласно постулатам проекта GNU, Linux — это название ядра операционной системы GNU/Linux. Дополнительные опции программы uname позволяют получить больше информации.

Рассмотрим еще один пример вызова функции `system()` (листинг 9.2).

Листинг 9.2. Пример system2.c

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

int main (void)
{
       system ("sleep 5");
       fprintf (stderr, "end-of-program\n");
       return 0;
}
```

Получилось следующее:

```
$ gcc -o system2 system2.c
$ ./system2
end-of-program
```

Эта программа раскрывает следующие две особенности функции `system()`:

- `system()` ожидает завершения переданной ей команды. Поэтому сообщение "end-of-program" выводится на экран примерно через 5 с;
- в `system()` можно передавать не только имя программы, но и аргументы.

Предыдущий пример можно изменить таким образом, что функция `system()` не будет дожидаться выполнения программы (листинг 9.3).

Листинг 9.3. Пример system3.c

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

int main (void)
```

```
{
        system ("sleep 5 &");
        fprintf (stderr, "end-of-program\n");
        return 0;
}
```

В этом примере функция `system()` передает в оболочку /bin/sh команду запуска программы в фоновом режиме. Поэтому сообщение "end-of-program" выводится без 5-секундной паузы.

Наконец, можно удостовериться, что аргумент функции `system()` действительно передается в оболочку. Для этого спровоцируем /bin/sh на вывод ошибки:

```
system ("cd abrakadabra");
```

Запустив программу, мы увидим примерно следующее:

```
$ ./system3
sh: line 0: cd: abrakadabra: No such file or directory
end-of-program
```

Итак, с аргументом `COMMAND` все понятно. Но что возвращает сама функция `system()`? Это число называется *статусом завершения потомка*, о котором речь пойдет в *главе 10*.

Как вы вскоре узнаете, организация многозадачности в Linux посредством функции `system()` — плохая идея. Безусловно, в ряде случаев эта функция бывает полезной, однако полноценное управление процессами в Linux должно осуществляться при помощи системных вызовов, которые будут описаны в *главе 10*.

9.2. Процессы в Linux

В Linux каждому работающему процессу соответствует уникальное положительное целое число, которое называется *идентификатором процесса* и часто обозначается аббревиатурой PID (Process IDentifier).

К процессу также привязано еще одно положительное целое число — *идентификатор родительского процесса*, которое часто обозначается аббревиатурой PPID (Parent Process IDentifier).

Мы уже говорили о том, что на вершине дерева процессов находится процесс init. Идентификатор этого процесса всегда равен 1. Родителем init условно считается процесс с идентификатором 0, но фактически init не имеет родителя.

Для получения информации о процессах предназначена программа ps, поддерживающая большое количество опций. Некоторые из этих опций являются стандартными для Unix-подобных систем, другие зависят от конкретной реализации ps. В Linux обычно установлена программа ps из пакета procps.

Рассмотрим некоторые примеры вызова программы ps. Если вызвать ps без аргументов, то на экране появится список процессов, запущенных под текущим терминалом:

```
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26310 pts/0    00:00:00 ps
```

Замечание

Терминал и командная оболочка — разные понятия. В современных Unix-подобных системах под терминалом обычно понимают программу эмуляции терминала. Но это довольно сложная тема, заслуживающая написания отдельной книги.

Появилась таблица, состоящая из четырех столбцов. Первый столбец (PID) — это, как вы наверняка догадались, — идентификатор процесса. Второй и третий столбцы (TTY и TIME) мы пока рассматривать не будем. В четвертом столбце (CMD) записывается имя исполняемого файла программы, запущенной внутри процесса.

В нашем случае под текущим терминалом работают два процесса: командная оболочка и, собственно, программа ps. Если выполнить приведенную команду еще раз, то можно увидеть некоторые изменения:

```
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26444 pts/0    00:00:00 ps
```

В данном случае изменился PID процесса для программы ps. Всякий раз, когда в системе рождается новый процесс, ядро Linux автоматически выделяет для него уникальный идентификатор.

Примечание

Идентификаторы процессов могут через какое-то время повторяться, но два одновременно работающих процесса не могут иметь одинаковый идентификатор.

Если теперь запустить под оболочкой какую-нибудь программу, то она будет связана с текущим терминалом и появится в выводе программы ps. Чтобы продемонстрировать это, запустим программу yes в фоновом режиме, перенаправив ее стандартный вывод на устройство /dev/null:

```
$ yes > /dev/null &
[1] 26600
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26600 pts/0    00:00:05 yes
26618 pts/0    00:00:00 ps
```

Если теперь ничего не трогать, то даже через год процесс yes будет усердно пересылать поток символов "y" в "никуда". Чтобы предотвратить эту бессмыслицу, выведем процесс из фонового режима и завершим его комбинацией клавиш <Ctrl>+<C>:

```
$ fg yes
yes > /dev/null
^C
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26729 pts/0    00:00:00 ps
```

Обратите внимание, что в этом примере запись `^C` означает нажатие клавиатурной комбинации <Ctrl>+<C>.

Если вызвать программу ps с опцией `-f`, то вывод пополнится несколькими новыми столбцами:

```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
nnivanov   25164 25161  0 13:43 pts/0    00:00:00 bash
nnivanov   26790 25164  0 13:59 pts/0    00:00:00 ps -f
```

Особый интерес здесь представляют столбцы UID и PPID. Нетрудно догадаться, что в столбце PPID выводится идентификатор родительского процесса. Однако столбец UID требует особого рассмотрения.

Предположим, что в системе есть какой-то файл foo с правами доступа 0600 (восьмеричное представление), принадлежащий не вам. Но вам вдруг захотелось посмотреть его содержимое при помощи программы cat:

```
$ cat foo
cat: foo: Permission denied
```

Все тривиально: программа cat не смогла открыть файл и вывела сообщение о том, что доступ запрещен. Но если владелец файла введет ту же самую команду, то ошибки не произойдет. Возникает резонный вопрос: как одна и та же программа cat отличает "своих" от "чужих"? Ответ прост: каждый процесс в Linux работает от лица одного из пользователей системы. Здесь есть свои тонкости, но о них речь пойдет в *главе 13*.

Итак, столбец UID (User IDentifier) в расширенном выводе программы ps содержит имя пользователя, от лица которого запущен процесс.

Если запустить программу ps с флагом `-e`, то будет выведен полный список работающих в системе процессов. Многие флаги программы ps можно комбинировать. Например, следующие две команды эквивалентны и выводят полный список процессов в расширенном виде:

```
$ ps -e -f
$ ps -ef
```

Чтобы получить сводку опций программы ps, просто введите следующую команду:

```
$ ps --help
```

Подробное описание программы ps можно найти на соответствующей странице справочного руководства (man).

9.3. Дерево процессов

Если вызывать программу ps с флагами `-e` и `-H`, то можно увидеть дерево процессов в наглядной форме.

> ***Примечание***
>
> При вызове программы ps можно объединять короткие (односимвольные) флаги. Таким образом, команды `ps -e -H` и `ps -eH` эквивалентны.

Если ваша терминальная программа использует пропорциональный шрифт, то следующая команда выведет дерево процессов с соединительными ветвями:

```
$ ps -e --forest
```

> ***Примечание***
>
> Пропорциональным называется шрифт, каждый символ которого занимает на экране прямоугольник фиксированного размера. Чтобы выяснить, работает ли ваша терминальная программа с пропорциональным шрифтом, просто запустите Midnight Commander; если меню и панели этой программы отображаются нормально (в виде ровных прямоугольников), то терминальная программа задействует пропорциональный шрифт.

Если в вашей системе установлен пакет программ psmisc, то дерево процессов можно увидеть при помощи программы pstree. Существуют также графические утилиты для просмотра дерева процессов: ksysguard, gnome-system-monitor и т. д.

Возникает вопрос: что произойдет с работающим потомком при завершении родительского процесса? В этом случае возможны следующие варианты:

- дочерний процесс завершается (например, из-за оборванной связи с управляющим терминалом);
- родительским процессом для "осиротевшего" потомка становится процесс init.

Первый вариант будет рассматриваться в *главе 20*, а второй можно легко проверить:

```
$ bash
$ yes > /dev/null &
[1] 28316
$ ps -f
```

В результате будет выведено:

```
UID         PID  PPID  C STIME TTY          TIME CMD
nnivanov  28229 28226  0 14:13 pts/0    00:00:00 bash
nnivanov  28276 28229  0 14:13 pts/0    00:00:00 bash
nnivanov  28316 28276 90 14:14 pts/0    00:00:05 yes
nnivanov  28326 28276  0 14:14 pts/0    00:00:00 ps -f
$ exit
exit
$ ps -f
```

В результате получим:

```
UID          PID  PPID  C STIME TTY          TIME CMD
nnivanov   28229 28226  0 14:13 pts/0    00:00:00 bash
nnivanov   28316     1 94 14:14 pts/0    00:00:19 yes
nnivanov   28356 28229  0 14:14 pts/0    00:00:00 ps -f
```

Рассмотрим внимательно приведенный пример. Сначала мы запускаем еще один экземпляр оболочки bash. Под этим новым экземпляром оболочки запускаем в фоновом режиме программу yes, перенаправив ее вывод на устройство /dev/null. Команда `ps -f` выводит информацию о том, что родителем yes является второй экземпляр оболочки с идентификатором 28276. После этого команда exit завершает второй экземпляр оболочки. Наконец, команда `ps -f`, выполненная уже под исходным экземпляром оболочки, показывает, что процесс 28316 не завершился, но его PPID стал равным 1, т. е. процесс init стал родителем "осиротевшего" процесса.

9.4. Получение информации о процессе

Итак, мы выяснили, что к каждому процессу привязаны следующие важные данные:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- идентификатор пользователя (UID).

Есть и другие данные, которыми располагает процесс, но мы пока о них говорить не будем.

Получить PID, PPID и UID текущего процесса можно с помощью следующих системных вызовов, объявленных в заголовочном файле unistd.h:

- `pid_t getpid (void);`
- `pid_t getppid (void);`
- `uid_t getuid (void);`

Типы данных `pid_t` и `uid_t` являются целыми числами, размерность которых зависит от реализации. Чтобы использовать эти типы, нужно включить в программу заголовочный файл sys/types.h. Системный вызов `getpid()` возвращает идентификатор текущего процесса, `getppid()` — родительского, а `getuid()` — идентификатор пользователя, от лица которого выполняется процесс. Но возникает вопрос: почему последний из этих вызовов возвращает целое число?

Загляните в файл /etc/passwd, где хранится информация обо всех пользователях системы. В современных дистрибутивах Linux файл /etc/passwd может содержать довольно много "странных" записей. Не пугайтесь, в основном это служебные учетные записи *псевдопользователей* (pseudousers).

Каждая строка файла /etc/passwd — это запись, соответствующая конкретному пользователю системы. В свою очередь каждая запись разделяется двоеточиями на семь полей.

- Первое поле — имя пользователя. Это то, что вы вводите в приглашении "login:" после загрузки системы.
- В ранних Unix-подобных системах второе поле файла /etc/passwd содержало зашифрованный пароль. В современных системах это поле пустое или содержит символ "x", а зашифрованные пароли хранятся в других файлах.
- Третье поле — идентификатор пользователя. Это то самое уникальное для каждого пользователя системы число, которое возвращает системный вызов `getuid()`.
- Четвертое поле — идентификатор группы по умолчанию.
- В пятом поле /etc/passwd может быть что угодно. Это поле содержит описание пользователя, но чаще всего в нем находится либо полное имя пользователя, либо информация о том, для чего в системе "прописался" псевдопользователь.
- Следующее поле содержит рабочий (домашний) каталог пользователя.
- В последнем поле /etc/passwd указывается путь к командной оболочке пользователя.

Итак, теперь вы знаете, что каждому пользователю системы соответствует уникальное целое неотрицательное число, которое называется идентификатором пользователя и часто обозначается аббревиатурой UID (User IDentifier).

> ***Примечание***
>
> В Unix-подобных системах каждому имени пользователя соответствует один (и только один) числовой идентификатор. В широком смысле под идентификатором пользователя (UID) может пониматься не только число, но и имя, однозначно соответствующее этому числу. Так, например, программа ps в столбце UID выводит имя пользователя вместо числа. Получение числового идентификатора из имени пользователя называется *разрешением имени пользователя* (username resolution).

Чтобы узнать числовой UID текущего пользователя, можно выполнить следующую команду:

```
$ id -u
```

Можно также узнать UID любого пользователя системы:

```
$ id -u USERNAME
```

UID суперпользователя (с именем root в подавляющем большинстве случаев) всегда равен 0:

```
$ id -u root
0
```

Для получения имени пользователя из числового UID применяется библиотечная функция `getpwuid()`, объявленная в заголовочном файле pwd.h следующим образом:

```
struct passwd * getpwuid (uid_t UID);
```

В структуре `passwd` нас интересует только одно поле, которое содержит имя пользователя:

```
struct passwd
{
       /* ... */
       char * pw_name;
};
```

Если UID не соответствует ни одному пользователю системы, то `getpwuid()` возвращает `NULL`.

Теперь можно написать программу, которая выводит некоторую информацию о текущем процессе (листинг 9.4).

Листинг 9.4. Программа getpinfo.c

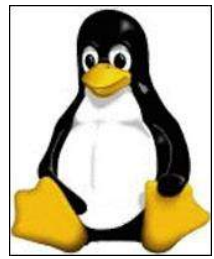
```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
       int uid = getuid ();
       struct passwd * pwdp = getpwuid (uid);
       if (pwdp == NULL)
       {
              fprintf (stderr, "Bad username\n");
              return 1;
       }

       printf ("PID: %d\n", getpid ());
       printf ("PPID: %d\n", getppid ());
       printf ("UID: %d\n", uid);
       printf ("Username: %s\n", pwdp->pw_name);

       sleep (15);
       return 0;
}
```

Инструкция `sleep (15)` необходима для того, чтобы мы могли в течение 15 с проверить достоверность значения PID, выводимого программой. Итак, проверяем:

```
$ gcc -o getpinfo getpinfo.c
$ ps -f
UID         PID  PPID  C STIME TTY          TIME CMD
nnivanov  28229 28226  0 14:13 pts/0    00:00:00 bash
nnivanov  28316     1 98 14:14 pts/0    00:10:55 yes
nnivanov  29414 28229  0 14:24 pts/0    00:00:00 ./getpinfo
```

```
nnivanov  29421 28229  0 14:25 pts/0    00:00:00 ps -f
$ cat output
PID: 29414
PPID: 28229
UID: 500
Username: nnivanov
[1]+  Done                    ./getpinfo > output
```

Команду `cat output` лучше выполнять после завершения getpinfo. Не забывайте, что стандартный вывод использует механизм буферизации. Данные, перенаправляемые в файл output, могут попросту оставаться в буфере стандартного вывода до тех пор, пока программа getpinfo не завершится.



ГЛАВА 10

Базовая многозадачность

В этой главе будут рассмотрены следующие темы:

- Порождение нового процесса при помощи системного вызова `fork()`.
- Передача управления другой программе через системный вызов `execve()`.
- Использование различных реализаций `execve()`.
- Ожидание завершения процесса при помощи системного вызова `wait()`.

10.1. Концепция развилки: *fork()*

Для порождения нового процесса предназначен системный вызов `fork()`, объявленный в заголовочном файле unistd.h следующим образом:

```
pid_t fork (void);
```

Системный вызов `fork()` порождает процесс методом "клонирования". Это значит, что новый процесс является точной копией своего родителя и выполняет ту же самую программу. Это звучит странно, но попробуем разобраться во всем по порядку. Рассмотрим сначала небольшой пример (листинг 10.1).

Листинг 10.1. Пример fork1.c

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

int main (void)
{
        fork ();
        printf ("Hello World\n");
        sleep (15);
        return 0;
}
```

Вот что получилось:

```
$ gcc -o fork1 fork1.c
$ ./fork1 > output &
[1] 4272
$ ps
 PID TTY          TIME CMD
3325 pts/1    00:00:00 bash
4272 pts/1    00:00:00 fork1
4273 pts/1    00:00:00 fork1
4274 pts/1    00:00:00 ps
$ cat output
Hello World
Hello World
[1]+  Done                    ./fork1 >output
```

Программа fork1 породила новый процесс, о чем свидетельствует вывод программы ps. Сразу после вызова `fork()` каждый процесс продолжил самостоятельно выполнять одну и ту же программу fork1. Этим и объясняется наличие двух приветствий "Hello World" в файле output.

> ***Примечание***
>
> Если вы решите самостоятельно добавить в программу (листинг 10.1) вывод какого-нибудь сообщения перед вызовом `fork()`, то, скорее всего, найдете в файле output две копии вашего сообщения. Это издержки механизма буферизации стандартного вывода. Дело в том, что ваше сообщение будет выведено не сразу, а после вызова `fork()`, когда программу будут выполнять уже два процесса.

Чтобы в контексте программы отделить один процесс от другого, достаточно знать, что системный вызов `fork()` возвращает в текущем процессе PID порожденного потомка (или –1 в случае ошибки). А в новый процесс `fork()` возвращает 0.

Рассмотрим пример (листинг 10.2), в котором родительский и дочерний процессы выполняют разные действия.

Листинг 10.2. Пример fork2.c

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

int main (void)
{
       pid_t result = fork();
       if (result == -1) {
              fprintf (stderr, "Error\n");
              return 1;
       }
```

```
        if (result == 0)
                printf ("I'm child with PID=%d\n", getpid ());
        else
                printf ("I'm parent with PID=%d\n", getpid ());

        return 0;
}
```

Получилось следующее:

```
$ ./fork2
I'm child with PID=19414
I'm parent with PID=19413
```

В приведенном примере сообщение родителя могло бы появиться первым. Дело в том, что процессы в Linux работают на самом деле не одновременно. Ядро периодически дает возможность каждому процессу передать свой исполняемый код процессору (или процессорам, если их несколько). Эти переключения обычно происходят настолько быстро, что у нас создается иллюзия одновременной работы нескольких программ.

За переключение процессов отвечают специальные алгоритмы, находящиеся в ядре Linux. Считается, что пользователи и программисты должны условно считать алгоритмы переключения процессов непредсказуемыми. Иными словами, процессы в Linux работают независимо друг от друга. Подобная договоренность позволяет ядру Linux "интеллектуально" распределять системные ресурсы между процессами, повышая производительность системы в целом.

Рассмотрим пример (листинг 10.3), демонстрирующий "непредсказуемость" переключения процессов в Linux.

Листинг 10.3. Пример race1.c

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

#define WORKTIME        3

int main (void)
{
        unsigned long parents = 0;
        unsigned long children = 0;
        pid_t result;
        time_t nowtime = time (NULL);

        result = fork ();
```

```
        if (!result) {
                while (time (NULL) < nowtime+WORKTIME) children++;
                printf ("children: %ld\n", children);
        } else {
                while (time (NULL) < nowtime+WORKTIME) parents++;
                printf ("parents: %ld\n", parents);
        }

        return 0;
}
```

Приведенная в листинге 10.3 программа в каждом из процессов запускает практически одинаковые циклы, которые в течение трех секунд "накручивают" счетчики `children` и `parents`. По завершении каждого цикла выводится отчет о состоянии счетчика. Итак, проверяем:

```
$ gcc -o race1 race1.c
$ ./race1
children: 470136
parents: 830649
$ ./race1
children: 586457
parents: 979719
$ ./race1
children: 447814
parents: 810994
```

Обратите внимание, что при выполнении приведенного примера возможен вариант, при котором "родитель" завершится чуть раньше и оболочка "выкинет" приглашение командной строки перед отчетом процесса-потомка. Если такое произойдет, то это будет лишь очередным доказательством того, что процессы работают независимо.

10.2. Передача управления: *execve()*

Системный вызов `execve()` загружает в процесс другую программу и передает ей безвозвратное управление. Этот системный вызов объявлен в заголовочном файле unistd.h следующим образом:

```
int execve (const char * PATH, const char ** ARGV,
                    const char ** ENVP);
```

Итак, системный вызов `execve()` принимает три аргумента:

- `PATH` — это путь к исполняемому файлу программы, которая будет запускаться внутри процесса. Здесь следует учитывать, что одноименная переменная окружения в системном вызове `execve()` не используется. В результате ответственность за поиск программы возложена только на программиста.

- ARGV — это уже знакомый нам массив аргументов программы. Здесь важно помнить, что первый аргумент (ARGV[0]) этого массива является именем программы или чем-то другим (на ваше усмотрение), но не фактическим аргументом. Последним элементом ARGV должен быть NULL.
- ENVP — это тоже уже знакомый нам массив, содержащий окружение запускаемой программы. Этот массив также должен заканчиваться элементом NULL.

> ***Примечание***
>
> В *разд. 10.3* будут описаны надстройки над execve(), позволяющие использовать переменную окружения PATH для запуска программы.

Выполняющийся внутри процесса код называется *образом процесса* (process image). Важно понимать, что системный вызов execve() заменяет текущий образ процесса на новый. Следовательно, возврата в исходную программу не происходит. В случае ошибки execve() возвращает –1, но если новая программа начала выполняться, то execve() уже ничего не вернет, поскольку работа исходной программы в текущем процессе на этом заканчивается.

Исходя из сказанного, можно сделать вывод, что возврат из системного вызова execve() происходит только в том случае, если произошла ошибка. Учитывая также то, что execve() не может возвращать ничего кроме –1, следующая проверка будет явно избыточной:

```
if (execve (binary_file, argv, environ) == -1) {
       /* обработка ошибки */
}
```

Целесообразнее более простая форма обнаружения ошибки:

```
execve (binary_file, argv, environ);
/* обработка ошибки */
```

Рассмотрим теперь программу (листинг 10.4), демонстрирующую работу системного вызова execve().

Листинг 10.4. Программа execve1.c

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

extern char ** environ;

int main (void)
{
       char * uname_args[] = {
              "uname",
              "-a",
              NULL
       };
```

```
        execve ("/bin/uname", uname_args, environ);
        fprintf (stderr, "Error\n");
        return 0;
}
```

Итак, в приведенной программе мы заменили образ текущего процесса программой /bin/uname с опцией `-a`. Если программа была успешно вызвана (независимо от того, как она завершилась), то сообщение "Error" не выводится. В качестве окружения мы воспользовались массивом `environ`, который был описан в *разд. 3.2*.

Рассмотрим еще один пример, демонстрирующий все важные особенности работы системного вызова `execve()`. Для этого создадим в одном каталоге две программы. Первая (листинг 10.5) будет при помощи `execve()` запускать вторую программу (листинг 10.6).

Листинг 10.5. Программа execve2.c

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

int main (void)
{
        char * newprog_args[] = {
                "Tee-hee!",
                "newprog_arg1",
                "newprog_arg2",
                NULL
        };

        char * newprog_envp[] = {
                "USER=abrakadabra",
                "HOME=/home/abrakadabra",
                NULL
        };

        printf ("Old PID: %d\n", getpid ());
        execve ("./newprog", newprog_args, newprog_envp);
        fprintf (stderr, "Error\n");

        return 0;
}
```

Листинг 10.6. Программа newprog.c

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

extern char ** environ;
int main (int argc, char ** argv)
```

```
{
        int i;

        printf ("ENVIRONMENT:\n");
        for (i = 0; environ[i] != NULL; i++)
                printf ("environ[%d]=%s\n", i, environ[i]);

        printf ("ARGUMENTS:\n");
        for (i = 0; i < argc; i++)
                printf ("argv[%d]=%s\n", i, argv[i]);

        printf ("New PID: %d\n", getpid ());
        return 0;
}
```

Ну и чтобы освежить в памяти материал из *главы 2*, создадим небольшой make-файл (листинг 10.7).

Листинг 10.7. Файл Makefile

```
execve2: execve2.c newprog
        gcc -o $@ execve2.c

newprog: newprog.c
        gcc -o $@ $^

clean:
        rm -f newprog execve2
```

Примечание

Не забывайте перед каждой инструкцией в make-файлах ставить символы табуляции. Помните также, что некоторые текстовые редакторы оказывают программисту "медвежью услугу" и заменяют табуляции группой пробелов.

Вот что получилось:

```
$ make
gcc -o newprog newprog.c
gcc -o execve2 execve2.c
$ ./execve2
Old PID: 4040
ENVIRONMENT:
environ[0]=USER=abrakadabra
environ[1]=HOME=/home/abrakadabra
ARGUMENTS:
argv[0]=Tee-hee!
argv[1]=newprog_arg1
argv[2]=newprog_arg2
New PID: 4040
```

Данный пример демонстрирует сразу три аспекта работы системного вызова `execve()`:

- обе программы выполнялись в одном и том же процессе;
- при помощи системного вызова `execve()` программе можно "подсунуть" любое окружение;
- в элементе `argv[0]` действительно может быть все, что угодно.

> ***Примечание***
>
> Теперь вы наверняка поняли, почему предпочтительнее говорить "аргументы программы", а не "аргументы командной строки".

Запускаемой программе можно передавать пустое окружение. Для этого достаточно указать `NULL` в третьем аргументе системного вызова `execve()`. Следующая небольшая программа демонстрирует это (листинг 10.8).

Листинг 10.8. Программа execve3.c

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

int main (void)
{
       char * env_args[] = {
              "env",
              NULL
       };

       execve ("/usr/bin/env", env_args, NULL);
       fprintf (stderr, "Error\n");

       return 0;
}
```

> ***Замечание***
>
> В некоторых дистрибутивах Linux программа env может располагаться не в /usr/bin, а, например, в /bin или в /usr/local/bin.

Итак, мы постепенно подошли к тому, чтобы опробовать многозадачность "во всей красе". Совместное использование системных вызовов `fork()` и `execve()` позволяет запускать программы в отдельных процессах, что чаще всего и требуется от программиста.

Рассмотрим пример (листинг 10.9), показывающий, как системные вызовы `fork()` и `execve()` работают вместе.

Листинг 10.9. Пример forkexec1.c

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

```
extern char ** environ;

int main (void)
{
       pid_t result;
       char * sleep_args[] = {
              "sleep",
              "5",
              NULL
       };

       result = fork ();
       if (result == -1) {
              fprintf (stderr, "fork error\n");
              return 1;
       }

       if (result == 0) {
              execve ("/bin/sleep", sleep_args, environ);
              fprintf (stderr, "execve error\n");
              return 1;
       } else {
              fprintf (stderr, "I'm parent with PID=%d\n",
                                  getpid());
       }

       return 0;
}
```

Проверяем:

```
$ gcc -o forkexec1 forkexec1.c
$ ./forkexec1
I'm parent with PID=4185
$ ps
 PID TTY          TIME CMD
3239 pts/1    00:00:00 bash
4186 pts/1    00:00:00 sleep
4187 pts/1    00:00:00 ps
```

Итак, программа forkexec1 породила новый процесс и запустила в нем программу /bin/sleep, которая дает нам возможность в течение 15 с набрать команду ps и насладиться наличием в системе отдельного процесса.

Сообщение "I'm parent with PID=..." выводится в стандартный поток ошибок (stderr), хотя фактически не является ошибкой. Это искусственный прием, позволяющий выводить сообщение на экран немедленно, не задумываясь о возможных последствиях буферизации стандартного вывода.

Если внимательно разобраться в приведенной в листинге 10.9 программе, то станет очевидным, что конструкция `else` является избыточной, поскольку дочерний процесс может только безвозвратно запустить другую программу или завершиться ошибкой. Все инструкции, находящиеся далее вызова `execve()`, могут выполняться только родительским процессом.

10.3. Семейство *exec()*

В стандартной библиотеке языка C есть пять дополнительных функций, которые реализованы с использованием `execve()` и вместе называются *семейством exec()*. Все функции семейства `exec()` объявлены в заголовочном файле unistd.h следующим образом:

```
int execl (const char * PATH, const char * ARG, ...);
int execle (const char * PATH, const char * ARG, ...,
                    const char ** ENVP);
int execlp (const char * FILE, const char * ARG, ...);
int execv (const char * PATH, const char ** ARGV);
int execvp (const char * FILE, const char ** ARGV);
```

Системный вызов `execve()` является "лидером" и полноправным членом семейства `exec()`. Напомним еще раз его прототип:

```
int execve (const char * PATH, const char ** ARGV,
                    const char ** ENVP);
```

Рассмотрев внимательно объявления всех функций семейства `exec()`, можно составить следующий универсальный шаблон:

```
execX[Y] (...);
```

Таким образом, имя функции формируется добавлением к префиксу `exec` обязательного `X` и произвольного `Y`. В качестве `X` могут выступать символы `l` и `v` (без кавычек), а `Y` может принимать значения `e` и `p`. Осталось только узнать, что означают эти символы. Их назначение приведено в табл. 10.1.

> ***Примечание***
>
> Будем далее под функцией `exec()` понимать одноименное семейство функций (включая системный вызов `execve()`), а также любой из членов этого семейства.

Таблица 10.1. *Назначение символов в имени* `exec()`

Символ	Наличие или отсутствие	Комментарий
`l`	Есть	Аргументы программы передаются не в виде массива `ARGV`, а в виде отдельных аргументов функции `exec()`, список которых заканчивается аргументом `NULL`
	Нет	Набор аргументов запускаемой программы передается в виде массива строк `ARGV`

Таблица 10.1 *(окончание)*

Символ	Наличие или отсутствие	Комментарий
v	Есть	Аргументы программы передаются в виде единого массива строк ARGV, последним элементом которого является NULL
	Нет	Вместо "v" присутствует символ "l", который задает список программы в виде отдельных аргументов-строк, список которых заканчивается аргументом NULL
e	Есть	В качестве последнего аргумента используется массив строк ENVP (окружение будущей программы)
	Нет	Начальным окружением запускаемой программы будет окружение текущего процесса
p	Есть	В качестве первого аргумента используется имя исполняемого файла программы, поиск которого будет производиться в каталогах, перечисленных в переменной окружения PATH
	Нет	В качестве первого аргумента выступает полный путь к исполняемому файлу программы (относительно текущего и корневого каталога)

Рассмотрим поочередно каждую функцию семейства exec(), реализуя фактически одну и ту же программу, которая будет запускать программу ls для подробного просмотра корневого каталога. В командной оболочке мы запустили бы такую команду следующим образом:

```
$ ls -l /
total 136
-rw-r--r--   1 root root     0 2010-12-28 21:11 acpid
-rw-r--r--   1 root root     0 2010-12-28 21:11 apmd
-rw-r--r--   1 root root     0 2010-12-28 21:14 atieventsd
drwxr-xr-x   2 root root  4096 2011-03-14 10:54 bin/
drwxr-xr-x   3 root root  4096 2011-05-06 10:53 boot/
-rw-r--r--   1 root root     0 2010-12-28 21:09 bpalogin
-rw-r--r--   1 root root     0 2010-12-28 21:11 capi4linux
-rw-r--r--   1 root root     0 2010-12-28 21:09 cpufreq
-rw-r--r--   1 root root     0 2010-12-28 21:09 crond
-rw-------   1 root root 22665 2011-03-13 14:26 dead.letter
drwxr-xr-x  18 root root  3840 2011-05-06 10:56 dev/
drwxr-xr-x 134 root root 12288 2011-05-06 11:00 etc/
-rw-r--r--   1 root root     0 2011-03-08 06:27 hddtemp
drwxr-xr-x   6 root root  4096 2011-03-08 05:41 home/
-rw-r--r--   1 root root     0 2011-03-08 10:03 httpd
-rw-r--r--   1 root root     0 2010-12-28 21:09 ibod
drwxr-xr-x   2 root root  4096 2011-03-07 17:49 initrd/
-rw-r--r--   1 root root     0 2010-12-28 21:14 ip6tables
```

```
-rw-r--r--   1 root root     0 2010-12-28 21:14 iptables
-rw-r--r--   1 root root     0 2010-12-28 21:09 irqbalance
-rw-r--r--   1 root root     0 2010-12-28 21:09 isdn4linux
-rw-r--r--   1 root root     0 2010-12-28 21:09 isdnlog
-rw-r--r--   1 root root     0 2010-12-28 21:09 laptop-mode
drwxr-xr-x  17 root root 12288 2011-04-13 09:15 lib/
drwxr-xr-x   5 root root  4096 2011-03-07 17:23 live/
-rw-r--r--   1 root root     0 2011-03-08 06:05 lm_sensors
drwx------   2 root root 16384 2011-03-07 17:29 lost+found/
-rw-r--r--   1 root root     0 2010-12-28 21:15 mandi
-rw-r--r--   1 root root     0 2010-12-28 21:09 mdadm
drwxr-xr-x   2 root root  4096 2011-05-03 21:38 media/
drwxr-xr-x   3 root root  4096 2011-03-07 17:28 mnt/
-rw-r--r--   1 root root     0 2011-05-06 10:55 Module.symvers
-rw-r--r--   1 root root     0 2010-12-28 21:12 msec
-rw-r--r--   1 root root     0 2011-03-14 08:16 networkmanager
-rw-r--r--   1 root root     0 2010-12-28 21:10 nfs-common
-rw-r--r--   1 root root     0 2010-12-28 21:09 ntpd
-rw-r--r--   1 root root     0 2011-03-07 17:38 null
-rw-r--r--   1 root root     0 2010-12-28 21:09 numlock
drwxr-xr-x   7 root root  4096 2011-05-06 11:00 opt/
-rw-r--r--   1 root root     0 2010-12-28 21:09 preload
dr-xr-xr-x 163 root root     0 2011-05-06 10:53 proc/
drwxr-x---  19 root root  4096 2011-05-06 15:13 root/
-rw-r--r--   1 root root     0 2010-12-28 21:09 rpcbind
-rw-r--r--   1 root root     0 2010-12-28 21:09 rsyslog
drwxr-xr-x   2 root root 12288 2011-04-13 09:19 sbin/
-rw-r--r--   1 root root     0 2010-12-28 21:14 shorewall
-rw-r--r--   1 root root     0 2010-12-28 21:13 slmodemd
drwxr-xr-x   2 root root  4096 2010-03-17 09:31 srv/
-rw-r--r--   1 root root     0 2010-12-28 21:12 sshd
drwxr-xr-x  12 root root     0 2011-05-06 10:53 sys/
drwxrwxrwt  64 root root 12288 2011-05-06 15:27 tmp/
drwxr-xr-x  15 root root  4096 2011-05-06 11:00 usr/
drwxrwxr-x  18 root root  4096 2011-03-14 14:18 var/
-rw-r--r--   1 root root     0 2010-12-28 21:12 vboxadd-timesync
```

Начнем с уже изученного нами системного вызова `execve()`, который является "главой семейства".

> ***Примечание***
>
> Для вас не должно иметь значения, является ли конкретный представитель семейства `exec()` библиотечной функцией или системным вызовом. Выбор конкретной формы `exec()` должен зависеть не от особенностей реализации, а от поставленной задачи.

Напишем программу (листинг 10.10), которая при помощи `execve()` запускает ls с указанными ранее аргументами в отдельном процессе.

Листинг 10.10. Программа execvels.c

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

extern char ** environ;

int main (void)
{
       pid_t result;
       char * ls_args[] = {
              "ls",
              "-l",
              "/",
              NULL
       };

       result = fork ();
       if (result == -1) {
              fprintf (stderr, "Fork error\n");
              return 1;
       }

       /* Child */
       if (result == 0) {
              execve ("/bin/ls", ls_args, environ);
              fprintf (stderr, "Exec error\n");
              return 1;
       }

       /* Parent */
       return 0;
}
```

Теперь реализуем ту же программу на основе `execl()` (листинг 10.11).

Листинг 10.11. Программа execlls.c

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

int main (void)
{
       pid_t result;
```

```
        result = fork ();
        if (result == -1) {
                fprintf (stderr, "Fork error\n");
                return 1;
        }

        /* Child */
        if (result == 0) {
                execl ("/bin/ls", "ls", "-l", "/", NULL);
                fprintf (stderr, "Exec error\n");
                return 1;
        }

        /* Parent */
        return 0;
}
```

Из листинга 10.11 видно, что использование `execl()` в нашем случае значительно упростило программу. Но иногда требуется тот же `execl()`, но с возможностью передачи окружения в вызываемую программу. Для этого существует функция `execle()`, работа которой продемонстрирована в следующем примере (листинг 10.12).

Листинг 10.12. Пример execlels.c

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

extern char ** environ;

int main (void)
{
        pid_t result;

        result = fork ();
        if (result == -1) {
                fprintf (stderr, "Fork error\n");
                return 1;
        }

        /* Child */
        if (result == 0) {
                execle ("/bin/ls", "ls", "-l", "/", NULL, environ);
                fprintf (stderr, "Exec error\n");
                return 1;
        }
```

```
        /* Parent */
        return 0;
}
```

Функции `execve()`, `execl()` и `execle()` вызывают программу, указывая абсолютный путь к ее исполняемому файлу. Но для нас зачастую более привычен вариант использования переменной окружения `PATH` для поиска указанного исполняемого файла. Такую возможность предоставляет функция `execlp()`, что иллюстрирует листинг 10.13.

Листинг 10.13. Пример execlpls.c

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

int main (void)
{
        pid_t result;

        result = fork ();
        if (result == -1) {
                fprintf (stderr, "Fork error\n");
                return 1;
        }

        /* Child */
        if (result == 0) {
                execlp ("ls", "ls", "-l", "/", NULL);
                fprintf (stderr, "Exec error\n");
                return 1;
        }

        /* Parent */
        return 0;
}
```

Функция `execv()`, рассмотренная далее (листинг 10.14), работает так же, как и `execve()`, только без возможности передачи "особенного" окружения.

Листинг 10.14. Пример execvls.c

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

```
int main (void)
{
       pid_t result;
       char * ls_args[] = {
              "ls",
              "-l",
              "/",
              NULL
       };

       result = fork ();
       if (result == -1) {
              fprintf (stderr, "Fork error\n");
              return 1;
       }

       /* Child */
       if (result == 0) {
              execv ("/bin/ls", ls_args);
              fprintf (stderr, "Exec error\n");
              return 1;
       }

       /* Parent */
       return 0;
}
```

И наконец, функция `execvp()` делает то же, что и `execv()`, но с возможностью поиска исполняемого файла программы в переменной окружения `PATH`. Следующий пример (листинг 10.15) демонстрирует работу `execvp()`.

Листинг 10.15. Пример execvpls.c

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

int main (void)
{
       pid_t result;
       char * ls_args[] = {
              "ls",
              "-l",
              "/",
              NULL
       };
```

```
        result = fork ();
        if (result == -1) {
                fprintf (stderr, "Fork error\n");
                return 1;
        }

        /* Child */
        if (result == 0) {
                execvp ("ls", ls_args);
                fprintf (stderr, "Exec error\n");
                return 1;
        }

        /* Parent */
        return 0;
}
```

Итак, мы рассмотрели примеры работы функций семейства `exec()` "на все случаи жизни". Если что-то со временем забудется, держите закладку на этом разделе: хороший пример освежит вашу память лучше любого справочника.

10.4. Ожидание процесса: *wait()*

Мы уже говорили о том, что процессы в Linux работают независимо друг от друга. Но иногда перед программистом встает задача организации последовательного выполнения процессов.

Типичный пример реализации последовательного выполнения процессов — ожидание родителем завершения своего потомка. Возьмем, например, командную оболочку. Если мы запускаем программу не в фоновом режиме, то оболочка не выдаст приглашение командной строки до тех пор, пока данная программа не завершится.

Другой пример — это последовательный запуск родительским процессом двух и более потомков. Например, многие командные оболочки позволяют разделять несколько команд точкой с запятой, организовывая тем самым их последовательное выполнение. Рассмотрим пример:

```
$ sleep 5 ; ls /
```

Эта команда "ждет" 5 с, а потом выводит содержимое корневого каталога вашей системы.

> ***Примечание***
>
> Важно понимать, что при организации показанной "эстафеты" синхронизация происходит не между ls и sleep, а между bash и sleep.

Самый "варварский" способ заставить родительский процесс дождаться завершения своего потомка — выждать некоторое время. Рассмотрим пример (листинг 10.16).

Листинг 10.16. Пример lsroot.c

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

int main (void)
{
       pid_t result = fork ();

       if (result == -1) {
              fprintf (stderr, "Fork error\n");
              return 1;
       }

       /* Child */
       if (result == 0) {
              execlp ("ls", "ls", "/", NULL);
              fprintf (stderr, "Exec error\n");
              return 1;
       }
       /* Parent */
       sleep (3);
       fprintf (stderr, "I'm parent\n");

       return 0;
}
```

Никогда не делайте так, как предложено в листинге 10.16! Очевидно, что программе ls с лихвой хватит трех секунд, чтобы вывести небогатое содержимое корневого каталога. Но если в этом каталоге случайно окажется миллион файлов, то предыдущее утверждение будет весьма спорным. Есть и еще один немаловажный нюанс: если, например, программа ls выведет содержимое корневого каталога за одну десятую секунды, зачем заставлять пользователя ждать "впустую" еще 2,9 с?

Существует системный вызов `wait()`, который позволяет избежать создания таких нелепых программ (листинг 10.16). `wait()` *блокирует* родительский процесс до тех пор, пока не завершится один из его потомков. Этот системный вызов объявлен в заголовочном файле wait.h следующим образом:

```
pid_t wait (int * EXIT_STATUS);
```

> ***Примечание***
>
> На самом деле в Linux `wait()` объявлен в файле sys/wait.h, который включается в заголовочный файл wait.h. Но вас это никак не должно волновать.

По типу возвращаемого значения нетрудно догадаться, что `wait()` возвращает идентификатор завершившегося потомка. Однако указатель `EXIT_STATUS` требует особого рассмотрения.

При изучении библиотечной функции `system()` в *главе 9* упоминалось, что она возвращает *статус завершившегося потомка*. Пришла пора разобраться в том, что это такое. Итак, статус завершившегося потомка — это целое число, содержащее код возврата и некоторую другую информацию о том, как завершился процесс. Для извлечения этой информации существуют специальные макросы:

- `WIFEXITED()` — возвращает ненулевое значение, если потомок завершится посредством возврата из функции `main()` или через вызов `exit()`.
- `WEXITSTATUS()` — возвращает код возврата завершившегося процесса. Этот макрос вызывается в том случае, если `WIFEXITED()` вернул ненулевое значение.
- `WIFSIGNALED()` — возвращает ненулевое значение, если процесс был завершен посредством получения *сигнала*. О сигналах речь пойдет в *главе 20*.
- `WTERMSIG()`, `WCOREDUMP()`, `WIFSTOPPED()`, `WSTOPSIG()` и `WCONTINUED()` — относятся к сигналам и будут рассматриваться в *главе 20*.

Рассмотрим теперь пример использования системного вызова `wait()` (листинг 10.17).

Листинг 10.17. Пример lsstatus.c

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

int main (int argc, char ** argv)
{
       pid_t status, childpid;
       int exit_status;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       status = fork ();
       if (status == -1) {
              fprintf (stderr, "Fork error\n");
              return 1;
       }

       /* Child */
       if (status == 0) {
              execlp ("ls", "ls", argv[1], NULL);
              fprintf (stderr, "Exec error\n");
              return 1;
       }
```

```
        /* Parent */
        childpid = wait (&exit_status);

        if (WIFEXITED (exit_status)) {
                printf ("Process with PID=%d "
                        "has exited with code=%d\n", childpid,
                        WEXITSTATUS (exit_status));
        }

        return 0;
}
```

Вот что получилось в результате выполнения этого примера:

```
$ gcc -o lsstatus lsstatus.c
$ ./lsstatus .
lsstatus  lsstatus.c
Process with PID=10454 exited with code=0
$ ./lsstatus abrakadabra
ls: abrakadabra: No such file or directory
Process with PID=10456 exited with code=2
```

Забежим немного вперед и определим понятие сигнала. Сигнал — это системное сообщение, которое один процесс направляет другому. При этом процесс-приемник может либо проигнорировать сигнал и продолжить свое выполнение, либо прервать все свои "дела" и немедленно начать *обрабатывать* (handle) полученный сигнал. Необходимо заметить, что существуют различные типы сигналов, некоторые из которых подразумевают немедленное завершение процесса.

Вы наверняка знаете, что при помощи команды `kill` можно "убить" процесс, над которым потерян контроль. На самом деле `kill` посылает процессу сигнал на завершение (если явно не указано другое). Вот как это делается:

```
$ yes > /dev/null &
[1] 10493
$ ps
  PID TTY          TIME CMD
 9481 pts/1    00:00:00 bash
10493 pts/1    00:00:00 yes
10494 pts/1    00:00:00 ps
$ kill 10493
$ ps
  PID TTY          TIME CMD
 9481 pts/1    00:00:00 bash
10495 pts/1    00:00:00 ps
[1]+  Terminated              yes >/dev/null
```

Примечание

В Linux есть программа /bin/kill, однако оболочка bash использует собственную внутреннюю команду `kill`.

Рассмотрим еще один пример (листинг 10.18), в котором родительский процесс обрабатывает статус завершения потомка на предмет получения сигнала.

Листинг 10.18. Пример killedchild.c

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

int main (void)
{
       pid_t status, childpid;
       int exit_status;

       status = fork ();
       if (status == -1) {
              fprintf (stderr, "Fork error\n");
              return 1;
       }

       /* Child */
       if (status == 0) {
              execlp ("sleep", "sleep", "30", NULL);
              fprintf (stderr, "Exec error\n");
              return 1;
       }

       /* Parent */
       childpid = wait (&exit_status);

       if (WIFEXITED (exit_status)) {
              printf ("Process with PID=%d "
                     "has exited with code=%d\n", childpid,
                     WEXITSTATUS (exit_status));
       }

       if (WIFSIGNALED (exit_status)) {
              printf ("Process with PID=%d "
                     "has exited with signal.\n", childpid);
       }

       return 0;
}
```

Итак, если запустить эту программу и подождать 30 с, то на экран будет выведено примерно следующее сообщение:

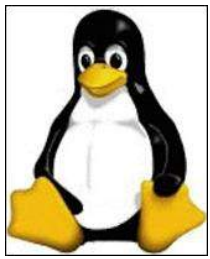
```
Process with PID=10205 has exited with code=0
```

Но если в течение 30 с узнать PID процесса sleep и "убить" его командой `kill`, то результат будет совсем другой:

```
$ ./killedchild &
[1] 10597
$ ps
  PID TTY          TIME CMD
 9481 pts/1    00:00:00 bash
10597 pts/1    00:00:00 killedchild
10598 pts/1    00:00:00 sleep
10599 pts/1    00:00:00 ps
$ kill 10598
Process with PID=10598 has exited with signal.
[1]+  Done                    ./killedchild
```

Системный вызов `wait()` имеет много интересных особенностей, которые будут подробнее рассматриваться в *главе 12*.

ГЛАВА 11



Потоки

Потоки (threads) позволяют одновременно выполнять разные действия в контексте одной программы. Механизмы работы с потоками реализованы в Linux в отдельной библиотеке Pthread.

Потоки в Linux — довольно сложная тема, заслуживающая написания отдельной книги. В этой главе описываются лишь основы работы с потоками. Дополнительную информацию по этой теме можно получить из источников, перечисленных в последнем разделе данной главы.

В рамках этой главы будут рассмотрены следующие темы:

- Понятие потоков и их особенности.
- Создание и завершение потока.
- Синхронизация потоков.
- Получение информации о потоках.
- Обмен данными между потоками.

11.1. Концепция потоков в Linux

Любой процесс — это выполняющийся код плюс данные, которыми он манипулирует. Этот закон "вступил в силу" с момента изобретения первого компьютера и актуален по сей день.

Вернемся к Linux. Когда один процесс вызывает `fork()`, в системе появляется другой процесс, который выполняет тот же код. Но данные, которыми манипулирует этот код, являются независимой копией данных процесса-родителя. Чтобы продемонстрировать это, рассмотрим небольшой пример (листинг 11.1).

Листинг 11.1. Пример pdata.c

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

```
int main (void)
{
        int a = 10;
        pid_t status = fork ();

        /* Child */
        if (!status) {
                a++;
                printf ("Child's a=%d\n", a);
                return 0;
        }

        /* Parent */
        wait ();
        printf ("Parent's a=%d\n", a);

        return 0;
}
```

Пример показывает, что переменная `a` в дочернем процессе не имеет ничего общего (кроме имени) с переменной `a` в родительском процессе.

> ***Примечание***
>
> Если в приведенном примере (листинг 11.1) вместо значений переменной `a` вы захотите посмотреть адреса, то будете удивлены, что они численно совпадают. Это происходит из-за того, что каждый процесс в Linux использует собственное адресное пространство. Поскольку сразу после вызова `fork()` процессы практически идентичны, то локальные адреса переменной `a` совпадают.

Процессы могут иметь общие данные, но для этого программисты обращаются к методам *межпроцессного взаимодействия* (interprocess communication), которые будут рассматриваться далее в этой книге. Отметим лишь то, что межпроцессное взаимодействие подчас требует сложных манипуляций со стороны программиста. Кроме того, в некоторых случаях плохо реализованное межпроцессное взаимодействие может стать мишенью для злоумышленников.

Потоки позволяют в рамках одной программы выполнять одновременно несколько действий, используя при этом общие данные. В Linux потоки выполняются так же, как и процессы, т. е. независимо.

Потоки в Linux реализуются очень просто:

1. Создается функция, которая называется *потоковой функцией*.
2. При помощи функции `pthread_create()` создается поток, в котором начинает параллельно остальной программе выполняться потоковая функция.
3. Вызывающая сторона продолжает выполнять какие-то действия, не дожидаясь завершения потоковой функции.

В самом начале этой главы сообщалось, что потоки реализованы в библиотеке Pthread. Чтобы подключить эту библиотеку к программе, нужно передать компоновщику опцию `-lpthread`.

11.2. Создание потока: *pthread_create()*

Потоки подобно процессам работают с идентификаторами. Только эти идентификаторы существуют локально в рамках текущего процесса. Для их хранения предусмотрен специальный тип `pthread_t`, который становится доступным при включении в программу заголовочного файла pthread.h.

Для создания потока и запуска потоковой функции используется функция `pthread_create()`, объявленная в заголовочном файле pthread.h следующим образом:

```
int pthread_create (pthread_t * THREAD_ID, void * ATTR,
              void *(*THREAD_FUNC) (void*), void * ARG);
```

Итак, `pthread_create()` принимает четыре аргумента:

- По адресу в `THREAD_ID` помещается идентификатор нового потока (если таковой был создан).
- Бестиповый указатель `ATTR` служит для указания *атрибутов потока*. Если этот аргумент равен `NULL`, то поток создается с атрибутами по умолчанию. В рамках данной книги мы не будем передавать потокам специальные атрибуты.
- Пугающий своим видом аргумент `PTHREAD_FUNC` является указателем на потоковую функцию. Это обычная функция, возвращающая бестиповый указатель (`void*`) и принимающая бестиповый указатель в качестве единственного аргумента.
- Аргумент `ARG` — это бестиповый указатель, содержащий аргументы потока. Если потоковая функция не требует наличия аргументов, то в качестве `ARG` можно указать `NULL`.

Чтобы лучше понять концепцию потоков в Linux, нужно представить себе, что программа при запуске создает один поток, а функция `pthread_create()` позволяет создавать дополнительные потоки. Это означает, что основную программу следует трактовать как "родительский поток".

Следует также понимать, что если один из потоков завершает программу, то все остальные потоки тут же завершаются. Это еще одно важное отличие потоков от процессов.

Рассмотрим теперь небольшой пример (листинг 11.2).

Листинг 11.2. Пример nothread.c

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

```
void * any_func (void * args)
{
        fprintf (stderr, "Hello World\n");
        sleep (5);
        return NULL;
}

int main (void)
{
        any_func (NULL);
        fprintf (stderr, "Goodbye World\n");
        while (1);

        return 0;
}
```

Эта программа выводит сообщение "Hello World", ждет 5 с, выводит второе сообщение "Goodbye World" и уходит в бесконечный цикл. Чтобы принудительно завершить программу, нажмите комбинацию клавиш <Ctrl>+<C>. Итак, пятисекундная пауза между сообщениями обусловлена тем, что программа ждет завершения функции `any_func()`.

Давайте теперь модернизируем программу таким образом, чтобы `any_func()` была потоковой функцией (листинг 11.3).

Листинг 11.3. Программа thread1.c

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

void * any_func (void * args)
{
        fprintf (stderr, "Hello World\n");
        sleep (5);
        return NULL;
}

int main (void)
{
        pthread_t thread;
        int result;

        result = pthread_create
                (&thread, NULL, &any_func, NULL);

        if (result != 0) {
                fprintf (stderr, "Error\n");
```

```
                return 1;
        }

        fprintf (stderr, "Goodbye World\n");
        while (1);

        return 0;
}
```

Для удобства добавим также make-файл (листинг 11.4).

Листинг 11.4. Файл Makefile

```
thread1: thread1.c
        gcc -o $@ $^ -lpthread

clean:
        rm -f thread1
```

Теперь функции `main()` и `any_func()` работают параллельно, поэтому никакой видимой задержки между выводом двух сообщений не происходит. Бесконечный цикл в этой программе выполняет особую роль. Мы уже говорили, что при завершении программы одним из потоков все остальные потоки немедленно завершаются. Если бы в нашей программе не было бесконечного цикла, то возврат из функции `main()` мог бы произойти раньше, чем вывод сообщения "Hello World".

Для передачи данных в поток используется четвертый аргумент функции `pthread_create()`. Этот указатель автоматически становится аргументом потоковой функции. Следующий пример демонстрирует эту возможность (листинг 11.5).

Листинг 11.5. Пример threadarg.c

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

void * any_func (void * arg)
{
        int a = *(int*) arg;
        fprintf (stderr, "Hello World "
                "with argument=%d\n", a);
        return NULL;
}

int main (int argc, char ** argv)
{
        pthread_t thread;
        int arg, result;
```

```
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        arg = atoi (argv[1]);
        result = pthread_create (&thread, NULL,
                        &any_func, &arg);

        fprintf (stderr, "Goodbye World\n");
        while (1);

        return 0;
}
```

В приведенном примере первый аргумент программы (`argv[1]`) преобразуется в целое число, которое затем передается в потоковую функцию. Если требуется передать в поток несколько аргументов, то их можно разместить в структуре, указатель на которую также передается в потоковую функцию. Этот подход продемонстрирован в следующем примере (листинг 11.6).

Листинг 11.6. Пример threadstruct.c

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

struct thread_arg
{
        char * str;
        int num;
};

void * any_func (void * arg)
{
        struct thread_arg targ =
                *(struct thread_arg *) arg;
        fprintf (stderr, "str=%s\n", targ.str);
        fprintf (stderr, "num=%d\n", targ.num);
        return NULL;
}

int main (void)
{
        pthread_t thread;
        int result;
```

```
        struct thread_arg targ;
        targ.str = "Hello World";
        targ.num = 2007;

        result = pthread_create (&thread, NULL,
                    &any_func, &targ);

        while (1);
        return 0;
}
```

В этом примере потоковую функцию `any_func()` можно было бы реализовать следующим образом:

```
void * any_func (void * arg)
{
        struct thread_arg * targ =
               (struct thread_arg *) arg;
        fprintf (stderr, "str=%s\n", targ->str);
        fprintf (stderr, "num=%d\n", targ->num);
        return NULL;
}
```

Один процесс может создать сколько угодно потоков (в пределах разумного). Следующая программа (листинг 11.7) демонстрирует создание двух независимо работающих потоков.

Листинг 11.7. Программа multithread.c

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

void * thread_func1 (void * arg)
{
        fprintf (stderr, "thread1: %s\n", (char*) arg);
        sleep (5);
        return NULL;
}

void * thread_func2 (void * arg)
{
        fprintf (stderr, "thread2: %s\n", (char*) arg);
        sleep (5);
        return NULL;
}
```

```
int main (void)
{
        pthread_t thread1, thread2;
        char * thread1_str = "Thread1";
        char * thread2_str = "Thread2";

        if (pthread_create (&thread1, NULL,
                        &thread_func1, thread1_str) != 0) {
                fprintf (stderr, "Error (thread1)\n");
                return 1;
        }

        if (pthread_create (&thread2, NULL,
                        &thread_func2, thread2_str) != 0) {
                fprintf (stderr, "Error (thread2)\n");
                return 1;
        }

        fprintf (stderr, "Hello World\n");
        while (1);
        return 0;
}
```

11.3. Завершение потока: *pthread_exit()*

Известно, что программа обычно завершается посредством возврата из функции `main()` или через вызов `exit()`. Аналогичным образом работают и потоки, которые могут завершаться возвратом из потоковой функции или вызовом специальной функции `pthread_exit()`, которая объявлена в заголовочном файле pthread.h следующим образом:

```
void pthread_exit (void * RESULT);
```

Если потоковая функция вызывает другие функции, то `pthread_exit()` в таких случаях бывает очень полезной. Аргумент `RESULT` будет рассмотрен позже. Следующий пример (листинг 11.8) демонстрирует работу функции `pthread_exit()`.

Листинг 11.8. Пример threadexit.c

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

void print_msg (void)
{
        fprintf (stderr, "Hello World\n");
        pthread_exit (NULL);
}
```

```
void * any_func (void * arg)
{
       print_msg ();
       fprintf (stderr, "End of any_func()\n");
       return 0;
}

int main (void)
{
       pthread_t thread;
       if (pthread_create (&thread, NULL,
                     &any_func, NULL) != 0) {
              fprintf (stderr, "Error\n");
              return 1;
       }

       while (1);
       return 0;
}
```

11.4. Ожидание потока: *pthread_join()*

Функция `pthread_join()` позволяет синхронизировать потоки. Она объявлена в заголовочном файле pthread.h следующим образом:

```
int pthread_join (pthread_t THREAD_ID, void ** DATA);
```

Эта функция блокирует вызывающий поток до тех пор, пока не завершится поток с идентификатором `THREAD_ID`. По адресу `DATA` помещаются данные, возвращаемые потоком через функцию `pthread_exit()` или через инструкцию `return` потоковой функции.

При удачном завершении `pthread_join()` возвращает 0, любое другое значение сигнализирует об ошибке.

Возникает вопрос: зачем блокировать вызывающий поток в ожидании другого потока, если можно с тем же успехом использовать простую функцию вместо поточной? Дело в том, что информация о завершении потока продолжает храниться в текущем процессе до тех пор, пока не будет вызвана `pthread_join()`. Например, после вызова `pthread_create()` родительский поток может начать делать что-то "свое" и после завершения этой работы вызвать `pthread_join()`. Если дочерний поток к тому времени завершится, то вызывающая сторона получит результат и продолжит выполнение. Если же поток-потомок будет еще работать, то родителю придется подождать, но уже с "гордым чувством" выполненной работы.

Следующий пример (листинг 11.9) показывает, как работает функция `pthread_join()`.

Листинг 11.9. Пример join1.c

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

#define A_COUNT              15
#define B_COUNT              10

void * print_b (void * arg)
{
       int i;
       for (i = 0; i < B_COUNT; i++) {
              fprintf (stderr, "B");
              sleep (1);
       }
       fprintf (stderr, "C");

       return NULL;
}

int main (void)
{
       pthread_t thread;
       int i;

       if (pthread_create (&thread, NULL,
                     &print_b, NULL) != 0) {
              fprintf (stderr, "Error\n");
              return 1;
       }

       for (i = 0; i < A_COUNT; i++) {
              fprintf (stderr, "A");
              sleep (1);
       }

       if (pthread_join (thread, NULL) != 0) {
              fprintf (stderr, "Join error\n");
              return 1;
       }

       fprintf (stderr, "D\n");
       return 0;
}
```

Эта программа в течение некоторого времени выводит примерно такой "шифр":

```
$ ./join1
ABABABABABABABABABACAAAAD
```

Символы A печатает родительский поток. Символы B выводятся порожденным потоком. Когда этот поток завершается, выводится символ C. А когда завершается программа, печатается символ D. Изменяя значения A_COUNT и B_COUNT, вы можете визуально наблюдать, как работают потоки в различных ситуациях.

Кроме того, применение функции pthread_join() наконец-то избавило нас от необходимости каждый раз запускать бесконечный цикл в конце программы.

Завершение функции pthread_join() с кодом 0 гарантирует завершение потока. После этого можно читать возвращенное потоком значение. Приведенный в листинге 11.10 пример показывает, как это делается.

Листинг 11.10. Пример join2.c

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

void * any_func (void * arg)
{
        int a = *(int *) arg;
        a++;
        return (void *) a;
}

int main (void)
{
        pthread_t thread;
        int parg = 2007, pdata;

        if (pthread_create (&thread, NULL,
                        &any_func, &parg) != 0) {
                fprintf (stderr, "Error\n");
                return 1;
        }

        pthread_join (thread, (void *) &pdata);
        printf ("%d\n", pdata);

        return 0;
}
```

Функцию pthread_join() может вызывать любой поток, работающий внутри текущего процесса. Рассмотрим пример, иллюстрирующий это (листинг 11.11).

Листинг 11.11. Пример join3.c

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

#define A_COUNT              10
#define B_COUNT              25
#define C_COUNT              10

void * print_b (void * arg)
{
       int i;
       for (i = 0; i < B_COUNT; i++) {
              fprintf (stderr, "B");
              sleep (1);
       }
       fprintf (stderr, "(end-of-B)\n");

       return NULL;
}

void * print_c (void * arg)
{
       pthread_t thread = * (pthread_t *) arg;
       int i;

       for (i = 0; i < C_COUNT; i++) {
              fprintf (stderr, "C");
              sleep (1);
       }
       fprintf (stderr, "(end-of-C)\n");
       pthread_join (thread, NULL);

       return NULL;
}

int main (void)
{
       pthread_t thread1, thread2;
       int i;

       if (pthread_create (&thread1, NULL,
                     &print_b, NULL) != 0) {
              fprintf (stderr, "Error (thread1)\n");
              return 1;
       }
```

```
        if (pthread_create (&thread2, NULL,
                    &print_c, &thread1) != 0) {
                fprintf (stderr, "Error (thread2)\n");
                return 1;
        }

        for (i = 0; i < A_COUNT; i++) {
                fprintf (stderr, "A");
                sleep (1);
        }
        fprintf (stderr, "(end-of-A)\n");

        pthread_join (thread2, NULL);
        fprintf (stderr, "(end-of-all)\n");

        return 0;
}
```

Вот что получилось:

```
$ gcc -o join3 join3.c -lpthread
$ ./join3
ABCABCABCABCABCABCABCABCABC(end-of-A)
B(end-of-C)
BBBBBBBBBBBBBB(end-of-B)
(end-of-all)
```

В этой программе основной поток создает два новых потока. При этом поток, выводящий символы `C`, вызывает функцию `pthread_join()` для потока, печатающего символы `B`.

11.5. Получение информации о потоке: *pthread_self()*, *pthread_equal()*

При изучении процессов мы говорили, что системный вызов `getpid()` возвращает идентификатор текущего процесса. Подобная функция существует и для потоков. Это функция `pthread_self()`, имеющая следующий прототип:

```
pthread_t pthread_self (void);
```

Для сравнения идентификаторов двух процессов используется оператор `==`. Тип `pthread_t` является целым числом, но к идентификаторам потоков не рекомендуется применять математические операции. Для сравнения идентификаторов двух потоков служит функция `pthread_equal()`, объявленная в заголовочном файле pthread.h следующим образом:

```
int pthread_equal (pthread_t THREAD1, pthread_t THREAD2);
```

Если идентификаторы относятся к одному потоку, то эта функция возвращает ненулевое значение. Если THREAD1 и THREAD2 являются идентификаторами разных потоков, то pthread_equal() возвращает 0.

Возникает резонный вопрос: зачем потоку знать свой идентификатор? Чаще всего это значение необходимо для того, чтобы уберечь поток от вызова phtread_join() для самого себя. Рассмотрим пример (листинг 11.12), демонстрирующий работу с pthread_self() и pthread_equal().

Листинг 11.12. Пример join4.c

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

void * any_func (void * arg)
{
       pthread_t thread = * (pthread_t *) arg;

       if (pthread_equal
              (pthread_self(), thread) != 0)
                     fprintf (stderr, "1\n");

       return NULL;
}

int main (void)
{
       pthread_t thread;
       if (pthread_create (&thread, NULL,
                     &any_func, &thread) != 0) {
              fprintf (stderr, "Error\n");
              return 1;
       }

       if (pthread_equal
              (pthread_self(), thread) != 0)
                     fprintf (stderr, "2\n");

       pthread_join (thread, NULL);
       return 0;
}
```

Очевидно, что эта программа всегда выводит на экран единицу и никогда не выведет двойку. Известный закон гласит: если что-то плохое может произойти, оно обязательно произойдет. Поэтому следующая проверка почти всегда будет признаком хорошего стиля программирования:

```
if (!pthread_equal (pthread_self (), any_thread))
       pthread_join (any_thread, NULL);
```

11.6. Отмена потока: *pthread_cancel()*

Любой поток может послать другому потоку запрос на завершение. Такой запрос называют *отменой потока*. Для этого предусмотрена функция `pthread_cancel()`, имеющая следующий прототип:

```
int pthread_cancel (pthread_t THREAD_ID);
```

Функция `pthread_cancel()` возвращает 0 при удачном завершении, ненулевое значение сигнализирует об ошибке.

Важно понимать, что несмотря на то, что `pthread_cancel()` возвращается сразу, это вовсе не означает немедленного завершения потока. Основная задача рассматриваемой функции — доставка потоку запроса на удаление, а не фактическое его удаление. В связи с этим, если для вас важно, чтобы поток был удален, нужно дождаться его завершения функцией `pthread_join()`.

> ***ЗАМЕЧАНИЕ***
>
> Существуют так называемые неотменяемые потоки. Эта тема выходит за рамки данной книги, но следует отметить, что некоторые потоки делают себя неотменяемыми. Такие потоки игнорируют любые запросы от `pthread_cancel()`, как если бы их вообще не было. Кроме этого потоки могут временно откладывать запросы на удаление, помещая их в специальную очередь.

Листинг 11.13 содержит пример, демонстрирующий отмену потока.

Листинг 11.13. Пример pcancel1.c

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

void * any_func (void * arg)
{
       while (1) {
              fprintf (stderr, ".");
              sleep (1);
       }

       return NULL;
}

int main (void)
{
       pthread_t thread;
       if (pthread_create (&thread, NULL,
                     &any_func, NULL) != 0) {
              fprintf (stderr, "Error\n");
              return 1;
       }
```

```
        sleep (5);
        pthread_cancel (thread);

        if (!pthread_equal (pthread_self (), thread))
                pthread_join (thread, NULL);

        fprintf (stderr, "\n");
        return 0;
}
```

Эта программа с интервалом в одну секунду выводит на экран точки до тех пор, пока не происходит вызов `pthread_cancel()`. Но только после вызова `pthread_join()` мы можем быть уверены, что поток действительно завершился.

Функцию `pthread_cancel()` можно вызывать из любого места программы. Поэтому на момент возврата из `pthread_join()` необходимо знать, как завершился поток: своими силами или был отменен. Эта информация становится очень важной, когда вызывающая сторона ждет от потока возвращаемых данных.

Итак, чтобы узнать, был ли поток отменен, нужно прочитать возвращаемое этим потоком значение и сравнить его с константой `PTHREAD_CANCELED`. Эта константа является не целым числом, как вы могли бы подумать, а бестиповым указателем. В листинге 11.14 показано, как выполняется проверка потока на предмет совершенной отмены.

Листинг 11.14. Пример pcancel2.c

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

void * any_func (void * arg)
{
        while (1) {
                fprintf (stderr, ".");
                sleep (1);
        }

        return NULL;
}

int main (void)
{
        pthread_t thread;
        void * result;
        if (pthread_create (&thread, NULL,
                        &any_func, NULL) != 0) {
                fprintf (stderr, "Error\n");
```

```
                return 1;
        }

        sleep (5);
        pthread_cancel (thread);

        if (!pthread_equal (pthread_self (), thread))
                pthread_join (thread, &result);

        if (result == PTHREAD_CANCELED)
                fprintf (stderr, "Canceled\n");

        return 0;
}
```

Если опасения (или надежды) по поводу возможной отмены процесса не оправдались, то `result` можно смело использовать для получения возвращенного потоком значения.

Примечание

Некоторые источники утверждают, что возвращаемое значение отмененного потока не устанавливается. Предыдущий пример (листинг 11.14) доказывает обратное.

11.7. Получение дополнительной информации

В этой главе были рассмотрены лишь основы работы с потоками POSIX. Дополнительную информацию по этой теме можно получить на перечисленных далее страницах руководства man. Обычно эти страницы находятся в секции 3p руководства:

- man pthreads;
- man pthread;
- man pthread_create;
- man pthread_exit;
- man pthread_join;
- man pthread_self;
- man pthread_equal;
- man pthread_cancel.

Не рассмотренные в этой главе темы можно изучить на следующих страницах справочного руководства man:

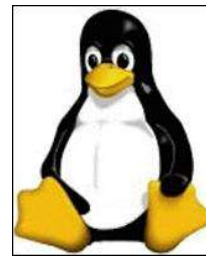
- man pthread_atfork;
- man pthread_kill;
- man pthread_once;

- man pthread_mutex_lock;
- man pthread_mutex_unlock;
- man pthread_detach;
- man pthread_setcancelstate;
- man pthread_setcanceltype;
- man pthread_testcancel;
- man pthread_setspecific.

По приведенным далее ссылкам размещены наиболее полные руководства и источники дополнительной информации о потоках:

- **http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html**;
- **http://www.llnl.gov/computing/tutorials/pthreads/**;
- **http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/**;
- **http://www.humanfactor.com/pthreads/pthread-tutorials.html**.

ГЛАВА 12



Расширенная многозадачность

В этой главе будут рассмотрены некоторые дополнительные вопросы, связанные с многозадачностью в Linux. Мы научимся менять приоритет выполнения процесса, узнаем, что системный вызов `wait()` не одинок в Linux, а также разберемся в том, насколько "страшны" *процессы-зомби*.

12.1. Уступчивость процесса: *nice()*

Мы уже говорили, что в ядре Linux работают специальные алгоритмы, которые управляют переключением процессов и распределяют между ними системные ресурсы.

Мы можем помочь ядру в этом нелегком деле. Для этого определим понятие *уступчивости процесса*. Итак, при распределении системных ресурсов ядро по умолчанию рассматривает процессы как равные. Но можно сообщить ядру, что некоторый процесс желает уступить часть своего права на системные ресурсы. В таком случае говорят, что уступчивость процесса увеличивается. Реже встречается понятие *приоритет процесса*, характеризующий величину, обратную уступчивости.

В Linux уступчивость процесса измеряется целыми числами в диапазоне от –20 до 19. По умолчанию процессы выполняются с уступчивостью 0. Запрос на получение пониженной уступчивости из отрицательного диапазона (от –20 до –1) может выполнять только суперпользователь. Диапазон от 0 до 19 доступен всем пользователям.

Получается, что обычный пользователь может только повышать уступчивость процессов. Для изменения уступчивости запускаемого процесса предназначена команда `nice`. Если вы работаете с оболочкой bash, то для запуска какой-нибудь программы с указанной уступчивостью используется внешняя утилита /bin/nice. Для этого указывается следующий простой шаблон:

```
$ nice -n NICE_VALUE PROGRAM ARGUMENTS
```

Здесь NICE_VALUE — это целое число от –20 до 19 либо от 0 до 19, если вы не имеете соответствующих прав. PROGRAM — это команда, запускающая программу, а ARGUMENTS — список аргументов запускаемой программы. Например, чтобы запустить программу ls для вывода содержимого корневого каталога с наибольшей уступчивостью, вводится следующая команда:

```
$ nice -n 19 ls /
```

Если обычный пользователь попытается "ускорить" выполнение программы ls, то nice не даст этого сделать:

```
$ nice -n -20 ls /
nice: cannot set niceness: Permission denied
```

В оболочках семейства C Shell (csh, tcsh) имеется своя внутренняя команда nice. Вот пример ее запуска:

```
$ nice +19 ls /
```

А вот что выведет оболочка csh, если обычный пользователь решит вызвать nice с отрицательной уступчивостью:

```
$ nice -20 ls /
setpriority: Permission denied.
```

Вернемся в "родную" для рядового линуксоида оболочку bash. Если вызвать программу nice без аргументов, то будет выведено значение текущей уступчивости оболочки:

```
$ nice
0
```

Заметим, что значение уступчивости наследуется дочерним процессом от родительского. Проверим это, запустив несколько вложенных сеансов оболочки bash:

```
$ nice
0
$ nice -n 10 bash
$ nice
10
$ bash
$ nice
10
$ exit
exit
$ exit
exit
```

> ***Примечание***
>
> В реальности зависимый аргумент опции -n программы nice — это не значение уступчивости запускаемой программы, а приращение к текущему значению. Например, если текущая уступчивость оболочки равна 10, то команда nice -n 1 ls запустит программу ls с уступчивостью 11, а не 1, как вы могли бы подумать.

Программист может изменять текущее значение уступчивости текущего процесса при помощи системного вызова `nice()`, который объявлен в заголовочном файле unistd.h следующим образом:

```
int nice (int INCREMENT);
```

Здесь нет ничего сложного: системный вызов пытается прибавить к текущей уступчивости значение `INCREMENT` и возвращает установленное значение. Если что-то пошло не так, возвращается –1.

Следующий простой пример (листинг 12.1) иллюстрирует работу системного вызова `nice()`.

Листинг 12.1. Пример nice1.c

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

int main (void)
{
        int newnice = nice (10);
        printf ("New nice: %d\n", newnice);
        return 0;
}
```

Вот что получилось:

```
$ gcc -o nice1 nice1.c
$ ./nice1
New nice: 10
```

Но нам это ни о чем не говорит. Попробуем сделать что-нибудь посложнее. Например, запустим оболочку под процессом с измененной уступчивостью (листинг 12.2).

Листинг 12.2. Пример nice2.c

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

int main (void)
{
        if (nice (5) == -1) {
                fprintf (stderr, "Nice error\n");
                return 1;
        }

        execlp ("bash", "bash", NULL);
        fprintf (stderr, "Exec error\n");
        return 0;
}
```

Эта программа сначала увеличивает уступчивость текущего процесса до 5, а затем заменяет образ текущего процесса командной оболочкой bash. Получилось следующее:

```
$ gcc -o nice2 nice2.c
$ ./nice2
$ ps
  PID TTY          TIME CMD
 3174 pts/1    00:00:00 bash
 3367 pts/1    00:00:00 bash
 3382 pts/1    00:00:00 ps
$ nice
5
$ exit
exit
$ nice
0
```

Важно понимать, что изменение уступчивости процесса вовсе не означает, что приложение будет работать быстрее или медленнее. Системный вызов `nice()` следует рассматривать исключительно как средство отправки запроса к ядру на повышение или понижение конкурентоспособности процесса в гонке за ресурсами. Ядро при этом пытается сделать все возможное, но не дает никаких гарантий.

12.2. Семейство *wait()*

В *разд. 10.4* сообщалось, что системный вызов `wait()` блокирует родительский процесс до тех пор, пока не завершится один из его потомков. Подобно системному вызову `execve()` у `wait()` тоже есть свое небольшое семейство:

```
pid_t waitpid (pid_t * PID, int EXIT_STATUS, int OPTIONS);
pid_t wait3 (int * EXIT_STATUS, int OPTIONS,
             struct rusage * RUSAGE);
pid_t wait4 (pid_t PID, int * EXIT_STATUS, int OPTIONS,
             struct rusage * RUSAGE);
```

Системный вызов `waitpid()` позволяет родительскому процессу ожидать конкретного потомка, а не всех сразу, как это делает `wait()`. Кроме этого `waitpid()` обладает некоторыми другими полезными свойствами, о которых будет рассказано немного позже.

Функции `wait3()` и `wait4()` "перекочевали" в Linux от BSD Unix и встречаются очень редко. В рамках данной книги мы не будем их рассматривать. Отметим только, что они работают аналогично `wait()` и `waitpid()`, но с возможностью сбора статистики о работе дочернего процесса.

Вернемся к `waitpid()`. Этот системный вызов принимает в качестве первого аргумента идентификатор процесса, окончание работы которого следует ожидать. Аргумент `EXIT_STATUS` имеет тот же смысл, что и в `wait()`. Третий аргумент позво-

ляет передавать в `waitpid()` опции, объединяемые операцией побитовой дизъюнкции. Мы будем использовать только одну из возможных опций — `WNOHANG`, которая позволяет не блокировать родительский процесс, если потомок еще работает. Если не требуется передача опций, то в третьем аргументе `waitpid()` пишут 0.

Системный вызов `waitpid()` возвращает идентификатор завершившегося потомка или –1 в случае ошибки. Если указан флаг `WNOHANG`, а ожидаемый дочерний процесс еще не завершился, то `waitpid()` возвращает 0.

> ***Примечание***
>
> Если в качестве первого аргумента `waitpid()` указать –1, то родительский процесс будет ожидать завершения любого потомка. Вызов `waitpid()` с первым аргументом, равным –1, и третьим аргументом, равным 0, будет полностью аналогичен вызову `wait()`.

Рассмотрим сначала программу, демонстрирующую работу `waitpid()` без указания дополнительных опций (листинг 12.3).

Листинг 12.3. Программа waitpid1.c

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

int main (int argc, char ** argv)
{
       pid_t child;
       int status;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       /* Child */
       if (!(child = fork ())) {
              execlp ("ls", "ls", argv[1], NULL);
              fprintf (stderr, "Exec error\n");
       }

       /* Parent */
       waitpid (child, &status, 0);

       if (WIFEXITED (status)) {
              printf ("Exitcode=%d\n",
                            WEXITSTATUS (status));
       }

       return 0;
}
```

Как видите, ничего сложного в использовании системного вызова `waitpid()` нет. Рассмотрим еще одну программу (листинг 12.4).

Листинг 12.4. Программа waitpid2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

#define DOT_COUNT       15

int main (void)
{
        int i, status;

        /* Child */
        if (!fork ()) {
                for (i = 0; i < DOT_COUNT; i++) {
                        fprintf (stderr, ".");
                        sleep (1);
                }

                exit (5);
        }

        /* Parent */
        while (1) {
                if (!waitpid (-1, &status, WNOHANG)) {
                        fprintf (stderr, "*");
                } else {
                        fprintf (stderr, "(exit)\n");
                        break;
                }

                sleep (2);
        }

        if (WIFEXITED (status)) {
                fprintf (stderr, "Exited with code="
                        "%d\n", WEXITSTATUS (status));
        }
        else if (WIFSIGNALED (status)) {
                fprintf (stderr, "Exited by signal\n");
        }

        return 0;
}
```

В этой программе сосредоточено все, что вы должны знать о `waitpid()`. Рассмотрим листинг 12.4 по порядку. Системный вызов `fork()` порождает новый процесс, в котором выполняется цикл ежесекундного вывода точек на экран. После завершения этого цикла вызывается функция `exit()`. Но вы уже знаете, что `exit()` завершает не программу, а текущий процесс. Аргумент 5 здесь выбран произвольно.

В это время родитель каждые 2 с вызывает `waitpid()`. В первом аргументе этого системного вызова стоит –1, что означает ожидание завершения любого дочернего процесса. Со вторым аргументом все понятно — это статус завершившегося потомка. Особого внимания требует флаг `WNOHANG` в третьем аргументе `waitpid()`. Благодаря его использованию родительский процесс не блокируется в ожидании завершения дочернего процесса. Если потомок еще не завершился, то `waitpid()` просто возвращает 0. И так каждые 2 с до тех пор, пока дочерний процесс не завершится. После этого происходит уже знакомое нам исследование переменной `status`.

Программа работает примерно так:

```
$ gcc -o waitpid2 waitpid2.c
$ ./waitpid2
.*.*..*..*..*..*..*..*.(exit)
Exited with code=5
```

Но давайте попробуем остановить дочерний процесс по сигналу. Для этого сначала увеличим значение `DOT_COUNT`:

```
#define DOT_COUNT       40
```

Теперь соберем программу заново и воспользуемся командой `kill`:

```
$ gcc -o waitpid2 waitpid2.c
$ ./waitpid2 2>anyfile &
[1] 4646
$ ps
  PID TTY          TIME CMD
 3174 pts/1    00:00:00 bash
 4646 pts/1    00:00:00 waitpid2
 4647 pts/1    00:00:00 waitpid2
 4648 pts/1    00:00:00 ps
$ kill 4647
$ cat anyfile
.*.*..*..*..*..*..(exit)
Exited by signal
[1]+  Done                    ./waitpid2 2>anyfile
```

Запись `2>anyfile` в оболочке bash перенаправляет стандартный поток ошибок в файл. 2 — это дескриптор стандартного потока ошибок.

12.3. Зомби

Возникает вопрос: как скоро после порождения нового процесса мы должны вызывать `wait()` или `waitpid()`, чтобы успеть получить статус завершившегося потомка? Ответ прост: когда угодно.

Мы уже неоднократно говорили, что процессы работают асинхронно. Поэтому было бы глупо надеяться на то, что функция семейства `wait()` будет вызвана раньше, чем завершится дочерний процесс. Вместо этого в ядре Linux реализован хитрый механизм, который присваивает каждому завершившемуся процессу статус *зомби* (zombie). Иначе говоря, процесс продолжает существовать в системе в форме "живого трупа" до тех пор, пока родительский процесс не завершится или не вызовет функцию семейства `wait()`. Таким образом гарантируется сохранность информации о завершившемся процессе.

Рассмотрим пример (листинг 12.5), демонстрирующий то, как в системе появляется процесс-зомби.

Листинг 12.5. Пример zombie1.c

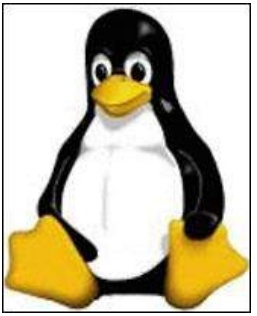
```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main (void)
{
        int status;

        /* Child */
        if (!fork ()) {
                execlp ("ls", "ls", NULL);
                fprintf (stderr, "Exec error\n");
        }

        /* Parent */
        sleep (40);
        wait (&status);

        if (WIFEXITED (status))
                printf ("Code=%d\n",
                        WEXITSTATUS (status));

        return 0;
}
```

В приведенном примере дочерний процесс вызывает программу ls и завершается, а родитель "засыпает" на 40 с. В промежуток времени между завершением дочернего процесса и вызовом функции `wait()` в системе будет существовать зомби. Далее показано, как все это можно увидеть своими глазами:

```
$ gcc -o zombie1 zombie1.c
$ ./zombie1 > anyfile &
[1] 4778
$ ps -l
F S   UID   PID  PPID ... TTY          TIME CMD
0 S  1000  3174  3173 ... pts/1    00:00:00 bash
0 S  1000  4778  3174 ... pts/1    00:00:00 zombie1
0 Z  1000  4779  4778 ... pts/1    00:00:00 ls <defunct>
0 R  1000  4780  3174 ... pts/1    00:00:00 ps
```

Итак, программа ps, вызванная с флагом `-l`, позволяет вывести очень подробную информацию о процессах. В приведенном ранее примере для компактности несколько столбцов посередине заменены многоточием. Нас интересует второй столбец, который показывает статус процесса. Символ `Z` означает, что процесс имеет статус зомби. Если подождать "пробуждения" родительского процесса и вызвать ps еще раз, то наш зомби исчезнет из списка процессов, а в файле anyfile будет находиться информация о его завершении:

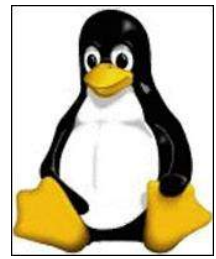
```
$ ps -l
F S   UID   PID  PPID ... TTY          TIME CMD
0 S  1000  3174  3173 ... pts/1    00:00:00 bash
0 R  1000  4782  3174 ... pts/1    00:00:00 ps
[1]+  Done                    ./zombie1 >anyfile
$ cat anyfile
anyfile
zombie1
zombie1.c
Code=0
```



ЧАСТЬ IV

Файловая система

ГЛАВА 13



Обзор файловой системы в Linux

Мы уже убедились в том, что файлы играют в Linux особую роль. В этой главе приводится общее описание файловой системы Linux как средства систематизации, представления и хранения файлов.

13.1. Аксиоматика файловой системы в Linux

Типичную модель файловой системы Linux можно представить в виде трех уровней ее реализации.

- На *нижнем уровне* находятся драйверы устройств и файловых систем. Например, когда мы говорим "файловая система ext4", то имеем в виду нижний уровень. Этот уровень полностью реализован в ядре Linux.
- На *среднем уровне* находятся системные вызовы, работающие с файловыми системами и осуществляющие низкоуровневый ввод-вывод. Этот уровень предоставляет программисту унифицированный интерфейс файловой системы, который позволяет не задумываться над техническими аспектами. Здесь все файлы делятся на семь типов: каталоги, символьные устройства, блочные устройства, обычные файлы, каналы FIFO, символические ссылки и сокеты. Средний уровень реализован в ядре Linux, но доступен рядовому программисту за счет представления системных вызовов в виде обычных функций.
- *Верхний уровень* включает в себя библиотеки и приложения, которые создают для нас полноценную иллюзию единого дерева файловой системы. На этом уровне тип файла обычно определяется его содержимым (рисунок PNG, исполняемый файл программы, видеоролик AVI, документ OpenOffice.org и т. д.).

> ***Примечание***
>
> Почти все современные аппаратные архитектуры реализуют концепцию концентрических (вложенных одно в другое по принципу матрешки) колец безопасности. Согласно этой концепции разные программы могут иметь различные привилегии. Ядро Linux работает на внутреннем кольце, т. е. имеет все привилегии (доступ к устройствам, памяти, аппаратным прерываниям и т. д.). Обычные программы работают на внешних кольцах, а для доступа к аппаратным ресурсам обращаются к ядру при помощи системных вызовов, которые реализованы в Linux в виде обычных функций.

Иногда нижний уровень называют *уровнем драйверов*. Средний уровень называют *уровнем системных вызовов*, а верхний — *уровнем приложений*. Существуют также другие модели, которые делят файловую систему на пять и более уровней, но нам это сейчас не потребуется.

13.2. Типы файлов

Мы выяснили, что на среднем уровне реализации файловой системы все файлы разделены на семь типов, краткий обзор которых был приведен в *разд. 7.1*. Любой пользователь может узнать тип того или иного файла при помощи программы ls, вызванной с флагом `-l`.

Итак, давайте создадим какой-нибудь пустой файл при помощи программы touch:

```
$ touch anyfile
```

Теперь наберем следующую команду:

```
$ ls -l anyfile
-rw-r--r-- 1 nnivanov nnivanov 0 2011-05-06 20:11 anyfile
```

Обратите внимание на поле прав доступа. Мы уже знаем значение последовательности `rw-r--r--` (у вас может быть что-то другое). Но что означает прочерк в самом начале? Это и есть поле типа файла с точки зрения среднего уровня реализации файловой системы. Если здесь стоит минус (как в нашем случае), то это обычный файл.

Давайте теперь создадим каталог:

```
$ mkdir anydir
$ ls -l
drwxr-xr-x 2 nnivanov nnivanov 4096 2011-05-06 20:13 anydir/
-rw-r--r-- 1 nnivanov nnivanov    0 2011-05-06 20:11 anyfile
```

В поле типа файла для anydir находится символ `d`, показывающий, что это каталог (directory).

Вы наверняка знаете, что символические ссылки — это указатели на другие файлы. Они создаются программой ln с флагом `-s`:

```
$ ln -s anyfile anylink
$ ls -l
drwxr-xr-x 2 nnivanov nnivanov 4096 2011-05-06 20:13 anydir/
-rw-r--r-- 1 nnivanov nnivanov    0 2011-05-06 20:11 anyfile
lrwxrwxrwx 1 nnivanov nnivanov    7 2011-05-06 20:14 anylink -> anyfile
```

Итак, символические ссылки в выводе ls обозначаются символом `l` (link).

Устройства в Linux могут быть представлены в виде файлов. На среднем уровне реализации файловой системы под устройствами понимают файлы двух типов:

- символьные устройства, обозначаемые в выводе программы ls символом `c` (character);
- блочные устройства, обозначаемые символом `b` (block).

Введите следующую команду:

```
$ ls -l /dev/null
crw-rw-rw- 1 root root 1, 3 2011-05-06 20:16 /dev/null
```

Вывод программы ls показывает, что воспетый в анекдотах /dev/null является символьным устройством.

> ***Примечание***
>
> Файлы устройств не обязательно соответствуют реально существующим аппаратным устройствам.

Дисковые накопители обычно представлены в файловой системе в виде блочных устройств. Еще сравнительно недавно для IDE-накопителей использовались устройства /dev/hda, /dev/hdb и аналогичные, а для разделов, созданных на этих накопителях, использовались численные обозначения /dev/hda1, /dev/hda2, /dev/hdb1, /dev/hdb2, /dev/hdc1. Для SCSI-накопителей точно так же использовались имена /dev/sda, /dev/sdb, /dev/sdc. Но потом было решено унифицировать обозначения устройств, содержащих файловые системы, и обозначать все жесткие диски, USB-накопители, CD/DVD-приводы так, как если бы это были SCSI-устройства. Таким образом, теперь /dev/sda может оказаться винчестером вашего компьютера (соответственно /dev/sda1 — его первый раздел), /dev/sdb может ссылаться на CD/DVD-привод, а /dev/sdc1 будет разделом на вставленной в USB-порт флэш-карте. Если вы используете дистрибутив Linux, выпущенный за последние несколько лет (по состоянию на 2011 г.), то у вас дела обстоят именно так. Вот так может выглядеть список блочных устройств:

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8,  0 2011-05-06 18:41 /dev/sda
brw-rw---- 1 root disk 8,  1 2011-05-06 18:41 /dev/sda1
brw-rw---- 1 root disk 8,  2 2011-05-06 18:41 /dev/sda2
brw-rw---- 1 root disk 8, 16 2011-05-06 18:53 /dev/sdb
brw-rw---- 1 root disk 8, 17 2011-05-06 18:53 /dev/sdb1
```

О символьных и блочных устройствах будет подробно рассказано в *разд. 13.5*.

Для взаимодействия процессов существуют особые файлы, которые называются каналами FIFO (First In First Out) или просто FIFO. Эти файлы создаются командой `mkfifo` и обозначаются в выводе программы ls символом `p` (pipe):

```
$ mkfifo anyfifo
$ ls -l anyfifo
prw-r--r-- 1 nnivanov nnivanov 0 2011-05-06 20:33 anyfifo
```

Об этих файлах речь пойдет в *главе 24*.

Еще один тип файлов — сокеты. Они динамически создаются программами при помощи системного вызова `socket()`. Сокеты обозначаются в выводе команды `ls` символом `s` (socket). О сокетах речь пойдет в *главе 25*.

13.3. Права доступа

В *главе 7* мы отмечали, что режим файла состоит из трех компонентов: базовые права доступа, расширенные права доступа и тип файла. Первый и третий компоненты нами уже рассмотрены. Осталось узнать, что представляют собой расширенные права доступа.

Вы уже знаете, что системный вызов `getuid()` возвращает идентификатор пользователя, от лица которого работает текущий процесс. Аналогичным образом работает системный вызов `getgid()`, который возвращает идентификатор группы текущего процесса.

Когда процесс пытается получить доступ к какому-нибудь файлу, ядро Linux использует UID текущего процесса для проверки прав доступа. Если проверка не удалась, в ход идет GID (Group ID, идентификатор группы). Если и здесь доступ закрыт, проверяются права доступа для остальных пользователей.

Во многих современных Linux-дистрибутивах данные о пользователях хранятся в файле /etc/passwd, который открыт для всех, а зашифрованные пароли находятся в файле /etc/shadow, доступ к которому разрешен только суперпользователю и представителям какой-нибудь группы, в которой редко состоят обычные пользователи:

```
$ ls -l shadow
-r--r----- 1 root shadow 1012 2011-03-08 10:03 /etc/shadow
```

> ***Примечание***
>
> Некоторые Linux-системы могут хранить зашифрованные пароли в файлах с другими именами (например, /etc/pwd.db). Однако важно лишь то, что в современных системах данные о пользователях отделены от базы данных паролей. Ранние Unix-подобные системы хранили пароли в /etc/passwd.

Теперь вспомним программу su, которая позволяет временно входить в систему на правах другого пользователя.

> ***Примечание***
>
> Многие пользователи думают, что имя su расшифровывается как "SUperuser". На самом деле имя этой программы произошло от словосочетания "Switch UID".

Подумайте, как эта программа "умудряется" сравнивать введенные вами пароли с содержимым закрытого от посторонних глаз файла /etc/shadow. Давайте проведем эксперимент. Для этого запустите еще один сеанс оболочки в отдельном терминале. В первом окне введите команду `su`:

```
$ su
Password:
```

Теперь перейдите в другое окно и введите там следующую команду:

```
$ ps -ef
...
nnivanov  17952  9299  0 20:37 pts/1    00:00:00 bash
root      17989  9302  0 20:37 pts/0    00:00:00 su
nnivanov  18008 17952  0 20:37 pts/1    00:00:00 ps -ef
```

Обратите внимание на то, что содержится в столбце UID для процесса, в котором работает программа su. Вы не ошиблись: программа выполняется от лица пользователя root!

Введите теперь следующую команду:

```
$ ls -l /bin/su
-rwsr-xr-x 1 root root 34500 2010-04-29 19:17 /bin/su
```

Триада, показывающая базовые права доступа для пользователя, содержит символ `s` на месте бита исполнения. Этот символ означает, что для исполняемого файла установлен *бит SUID* (Set User IDentifier), который позволяет запускать данную программу от лица владельца ее исполняемого файла. В результате программа su может читать зашифрованные пароли из файла /etc/shadow.

Существует также *бит SGID* (Set Group IDentifier), который позволяет запускать программу от имени группы, которой принадлежит этот файл.

Чтобы установить бит SUID для исполняемого файла, вызывается следующая команда:

```
$ chmod u+s file
```

А эта команда устанавливает для исполняемого файла бит SGID:

```
$ chmod g+s file
```

На самом деле к процессу привязано два идентификатора пользователя:

- *реальный UID* (Real UID, RUID или просто UID);
- *эффективный UID* (Effective UID или просто EUID).

В большинстве случаев эти идентификаторы равны. Но если для исполняемого файла программы установлен бит SUID, то ситуация меняется. Например, когда пользователь anyuser запускает программу su, то реальный UID процесса равен идентификатору пользователя anyuser, а эффективный UID равен идентификатору пользователя root.

При проверке прав доступа к файлам ядро пользуется эффективным UID. Однако разделение идентификаторов на реальные и эффективные может быть очень полезным. Например, некоторые приложения могут не разрешать "работу по доверенности" с использованием SUID.

Следует отметить, что системный вызов `getuid()` возвращает реальный UID пользователя. Для получения эффективного UID предусмотрен системный вызов `geteuid()`, объявленный в заголовочном файле unistd.h следующим образом:

```
uid_t geteuid (void);
```

Аналогичным образом обстоят дела с идентификаторами групп, которые также делятся на реальные и эффективные. Для получения эффективного GID используется системный вызов `getegid()`, имеющий следующий прототип:

```
gid_t getegid (void);
```

Расширенные права доступа позволяют задействовать еще один бит, который называется *битом SVTX* или *липким битом* (sticky). В ранних Unix-подобных системах липкий бит устанавливался для исполняемых файлов программ. После завершения такая программа оставалась в памяти компьютера (по возможности) и на повторный запуск требовалось гораздо меньше времени. В настоящее время липкий бит применяется только для каталогов и с совсем иной целью.

Представьте себе, что нужно иметь каталог, в котором каждый может создавать, удалять и переименовывать файлы, не мешая другим пользователям делать то же самое. Если просто установить для этого каталога право на запись для всех, то начнется "беспредел". Для этого и существует липкий бит, который позволяет пользователю манипулировать в каталоге только своими файлами. Следует отметить, что в каждой Linux-системе есть такой каталог. Это каталог /tmp, который доступен всем для хранения временных файлов. Вот как он фигурирует в выводе программы ls:

```
$ ls -l / | grep tmp
drwxrwxrwt  63 root root 12288 2011-05-06 20:42 tmp/
```

> ***Замечание***
>
> Мы будем подробно изучать временные файлы в *главе 18*.

Обратите внимание, что на месте бита исполнения "для всех" находится символ `t`, указывающий на то, что этому каталогу присвоен липкий бит.

> ***Примечание***
>
> Странное имя SVTX расшифровывается как "SaVe TeXt" (сохранить текст). Под словом "текст" здесь понимается сегмент кода программы.

Назначить некоторому каталогу directory липкий бит можно с помощью следующей команды:

```
$ chmod +t directory
```

Необходимо также отметить, что в отношении обычных файлов липкий бит игнорируется.

> ***Примечание***
>
> Как вы могли заметить, программа ls при выводе прав доступа закрывает битами SUID, SGID и SVTX поля прав на выполнение. На самом деле, если права на выполнение не были установлены, то символы `s` и `t` будут отображаться в верхнем регистре. Например, если файл не является исполняемым, но ему присваивается бит SUID, то его права доступа будут отображаться примерно так: `-rwSr--r--`.

13.4. Служебные файловые системы

Иногда под файловой системой в Linux понимается какой-нибудь каталог со всем его содержимым. Например, под рабочей файловой системой пользователя anyuser, вероятно, понимается дерево каталогов, берущее начало в /home/anyuser. В рамках данного

определения можно выделить несколько файловых систем, которые носят в Linux служебный характер.

- Файловая система /dev — хранилище файлов символьных и блочных устройств. В *разд. 13.5* вы узнаете, что файлы устройств могут храниться практически в любом месте файловой системы Linux. Но на практике каталог /dev является самым подходящим для них местом.
- Файловая система /proc — изначально была создана для предоставления пользователю информации о работающих процессах. Затем разработчики компонентов ядра Linux стали помещать туда всю "полезную" информацию (версия Linux, объем оперативной памяти, сообщения ядра и т. п.). Кроме того, в некоторые файлы дерева /proc можно писать, обеспечивая тем самым динамическое конфигурирование ядра. Файловая система /proc виртуальная, т. е. не базируется на блочном устройстве, а находится прямо в оперативной памяти. Файловая система /proc постепенно вытесняется более "продвинутой" файловой системой /sys, выполняющей примерно те же функции.
- Файловая система /tmp — предназначена для хранения временных файлов. Эта файловая система может располагаться как на диске, так и в памяти. При каждой перезагрузке системы эта файловая система автоматически очищается. Таким образом, даже если некоторое приложение записало большой временный файл и не позаботилось о его удалении, это произойдет автоматически во время перезагрузки системы. О временных файлах более подробно речь пойдет в *главе 18*.

> ***Примечание***
>
> Следует отметить, что разные дистрибутивы Linux используют различные политики ведения "хозяйства" /tmp. Утверждение о том, что /tmp очищается при каждой перезагрузке — это не всеобъемлющий факт, а наиболее часто встречающийся подход.

13.5. Устройства

Мы уже говорили о том, что на среднем уровне реализации файловой системы все устройства представляются двумя типами файлов:

- *символьные устройства* (character devices);
- *блочные устройства* (block devices).

Давайте разберемся, в чем их различие. Особенность символьных устройств — механизм последовательного ввода-вывода. В отличие от обычных файлов символьные устройства рассматриваются не в виде линейного массива данных, а как поток информации.

> ***Примечание***
>
> В Linux файлы устройств обычно связаны с драйверами устройств. Ядро представляет разработчикам драйверов унифицированный интерфейс файловых операций, благодаря которому разработчик драйвера самостоятельно реализует поведение системных вызовов `read()`, `write()`, `lseek()` и т. д. Иными словами, если вы работаете с устройством через файл, то поведение системных вызовов ввода-вывода будет зависеть от конкретного драйвера.

Блочные устройства читают и записывают данные блоками фиксированного размера с использованием механизма буферизации. Файлы устройств этого типа чаще всего применяются для дисковых накопителей и для дисковых разделов. Некоторые сетевые устройства также могут быть представлены файлами этого типа.

Файлы устройств обычно создаются в Linux программой mknod. Эта программа имеет следующий формат вызова:

```
$ mknod [-m MODE] NAME TYPE MAJOR MINOR
```

> ***Примечание***
>
> Программа mknod может потребовать наличия привилегий суперпользователя для создания файла устройства.

Рассмотрим по порядку аргументы этой программы.

- Необязательная опция `-m` принимает независимый аргумент `MODE`, который является правами доступа создаваемого файла в восьмеричном представлении.
- Аргумент `NAME` — это имя файла устройства.
- Аргумент `TYPE` может быть представлен одним из следующих символов: `c`, `b`, `u` или `p`. Символ `c` предполагает, что будет создано символьное устройство. Символ `b` инициирует создание блочного устройства. Символ `u` также означает блочное устройство, но с выключенным механизмом буферизации (unbuffered). Символ `p` используется для создания именованных каналов FIFO (mkfifo является синонимом mknod с символом `p` в качестве аргумента `TYPE`).
- Аргумент `MAJOR` — это *старший номер устройства* (см. далее).
- Аргумент `MINOR` — *младший номер устройства* (см. далее).

Особый интерес здесь представляют аргументы `MAJOR` и `MINOR`, означающие соответственно старший и младший номера устройства. Условно можно сказать, что старший номер устройства — это номер драйвера, отвечающего за данное устройство. Младший номер — это конкретная форма применения драйвера. Например, драйвер, контролирующий SCSI-накопители по соглашению, имеет старший номер 8. А младший номер здесь будет означать либо конкретное устройство (первый жесткий диск, второй жесткий диск, CD/DVD-привод, USB-накопитель и т. п.), либо конкретный раздел этого устройства. Это позволяет создавать идентичные по своему применению файлы устройств в любом месте файловой системы. Например, пользователь root может ввести в своем домашнем каталоге такую команду:

```
$ mknod first b 8 1
```

В результате будет создан файл блочного устройства с именем first, старшим номером 8 и младшим — 1. Через этот файл будет осуществляться доступ к первому разделу первого SCSI-диска компьютера. Таким образом, файл /root/first будет делать то же, что и файл /dev/sda1.

В Linux существуют соглашения, определяющие то, какие старшие и младшие номера должны иметь конкретные устройства, а также то, как эти устройства будут именоваться в файловой системе /dev. Эти соглашения доступны в файле Documentation/devices.txt в исходниках ядра Linux.

Например, в соответствии с этими соглашениями псевдоустройство /dev/null должно иметь старший номер 1 и младший номер 3. Следует отметить, что для символьных и блочных устройств задаются отдельные системы нумерации.

13.6. Монтирование файловых систем

Монтирование позволяет объединять различные файловые системы в одно дерево. Программа mount, вызванная без аргументов, позволяет увидеть список смонтированных файловых систем:

```
$ mount
/dev/sda2 on / type ext4 (rw)
none on /proc type proc (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
/dev/sdb1 on /media/4779-86EA type vfat
(rw,nosuid,nodev,uhelper=udisks,uid=500,gid=500,shortname=mixed,dmask=0077,
utf8=1,flush)
```

Вывод этой команды для каждой смонтированной файловой системы обычно имеет следующий формат:

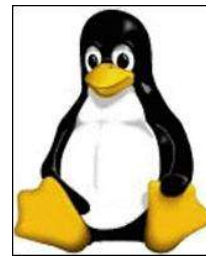
```
DEVICE on MOUNTPOINT type FSTYPE (FLAGS)
```

Рассмотрим теперь, что означают эти поля.

- `DEVICE` — это имя устройства.
- `MOUNTPOINT` — точка монтирования. Точкой монтирования называют каталог, являющийся как бы "порталом" для смонтированной файловой системы. Через этот каталог будет доступно содержимое этой файловой системы.
- `FSTYPE` — это тип файловой системы с точки зрения нижнего уровня ее реализации.
- Поле `FLAGS` — это опции монтирования. Здесь "rw" (Read/Write) означает, что файловая система смонтирована в режиме для чтения и записи. Полный список опций монтирования можно получить на man-странице программы mount.

Операция отсоединения файловой системы от точки монтирования называется *размонтированием* файловой системы. После этого точка монтирования становится обычным пустым каталогом.

ГЛАВА 14



Чтение информации о файловой системе

В этой главе будут рассмотрены следующие темы:

- Чтение информации о файловой системе посредством библиотечных функций из семейства `statvfs()`.
- Понятие текущего каталога. Получение информации о текущем каталоге.
- Перечень источников дополнительной информации по данной теме.

14.1. Семейство *statvfs()*

В *главе 13* мы выяснили, что файловая система Linux в подавляющем большинстве случаев состоит из нескольких файловых систем, смонтированных в одно дерево. Информация о смонтированных файловых системах обычно хранится в файле /etc/mtab:

```
$ cat /etc/mtab
/dev/sda2 / ext4 rw 0 0
none /proc proc rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
```

> ***Примечание***
>
> В Linux информацию о смонтированных файловых системах можно получить также из файла /proc/mounts.

Этот файл имеет достаточно простой синтаксис. Каждая строка /etc/mtab соответствует одной смонтированной файловой системе и содержит шесть полей:

- Первое поле — это *носитель файловой системы*. Если носитель соответствует блочному устройству, то файловая система называется реальной. В противном случае файловая система называется виртуальной, а ее носитель называют сервером файловой системы. Например, /dev/sda2 — это носитель реальной файловой системы, а proc — сервер виртуальной файловой системы.
- Второе поле — *точка монтирования*.

- Третье поле — *тип файловой системы*.
- Четвертое поле — *флаги монтирования*.
- Пятое поле используется утилитой суперпользователя dump.
- Шестое поле необходимо для утилиты fsck.

> ***Примечание***
>
> Хранение списка смонтированных файловых систем в файле /etc/mtab продиктовано стандартом LSB (Linux Standard Base). Но некоторые дистрибутивы Linux могут отступать от этого стандарта.

Программисту нет необходимости выполнять самостоятельный синтаксический разбор файла /etc/mtab. Для этих целей предназначены следующие функции, объявленные в заголовочном файле mntent.h:

```
FILE * setmntent (const char * FILENAME, const char * MODE);
struct mntent * getmntent (FILE * FILEP);
int endmntent (FILE * FILEP);
```

Функция `setmntent()` открывает файл, в котором содержится список смонтированных файловых систем (обычно это /etc/mtab). Аргумент `MODE` определяет режим открытия файла и может принимать те же значения, что и второй аргумент функции `fopen()`. Если произошла ошибка, то `setmntent()` возвращает `NULL`.

Функция `getmntent()` извлекает следующую запись о файловой системе. Если `getmntent()` вернула `NULL`, то это значит, что все записи прочитаны. Структура `mntent` объявлена в заголовочном файле mntent.h следующим образом:

```
struct mntent
{
       char *mnt_fsname;
       char *mnt_dir;
       char *mnt_type;
       char *mnt_opts;
       int mnt_freq;
       int mnt_passno;
};
```

Поля этой структуры следуют в том же порядке, что и поля записей /etc/mtab. Функция `endmntent()` вызывается по окончании чтения записей о файловых системах.

Более подробную информацию о файловых системах позволяют получить следующие функции, объявленные в заголовочном файле sys/statvfs.h:

```
int statvfs (const char * PATH, struct statvfs * FS);
int fstatvfs (int FD, struct statvfs * FS);
```

Эти функции часто называют *семейством statvfs()*. Они заносят в структуру `statvfs` подробную информацию о файловой системе, в которой находится файл (или каталог), указанный в первом аргументе. Для `statvfs()` первым аргументом является имя файла, а `fstatvfs()` идентифицирует файл по его дескриптору.

Структура `statvfs` содержит более десятка полей, но мы рассмотрим только некоторые из них.

- Поле `f_bsize` — целое число, содержащее размер блока файловой системы в байтах. Мы уже говорили, что реальные файловые системы базируются на блочных устройствах, которые адресуют данные не по одному байту, а блоками фиксированного размера.
- Поле `f_blocks` — целое число, показывающее количество блоков в файловой системе.
- Поле `f_bfree` — целое число, содержащее количество свободных блоков в файловой системе.
- Поле `f_bavail` содержит число доступных блоков, без учета резервных блоков, которые некоторые файловые системы выделяют для специального использования.
- Поле `f_namemax` — это целое число, показывающее максимально возможную длину имени файла в данной файловой системе. Под именем файла здесь понимается не полный путь к файлу, а одиночный элемент пути.

> ***Замечание***
>
> В *главе 16* будут также описаны поля `f_files`, `f_ffree` и `f_favail` структуры `statvfs`.

Рассмотрим теперь пример (листинг 14.1), демонстрирующий работу функции `statvfs()`.

Листинг 14.1. Программа statvfsdemo.c

```
#include <sys/statvfs.h>
#include <mntent.h>
#include <stdio.h>

int main (void)
{
       struct mntent * entry;
       struct statvfs fs;
       FILE * file;

       file = setmntent ("/etc/mtab", "r");
       if (file == NULL) {
              fprintf (stderr, "Cannot open /etc/mtab\n");
              return 1;
       }

       while ((entry = getmntent (file)) != NULL) {
              if (statvfs (entry->mnt_dir, &fs) == -1) {
                     fprintf (stderr, "statvfs() error\n");
```

```
                return 1;
        }

        printf ("Filesystem: %s\n", entry->mnt_fsname);
        printf ("Mountpoint: %s\n", entry->mnt_dir);
        printf ("Block size: %ld\n", (unsigned long int)
                fs.f_bsize);
        printf ("Blocks: %ld\n", (unsigned long int)
                fs.f_blocks);
        printf ("Blocks free: %ld\n", (unsigned long int)
                fs.f_bfree);
        printf ("Blocks available: %ld\n",
                (unsigned long int) fs.f_bavail);
        printf ("Max. filename length: %ld\n",
                (unsigned long int) fs.f_namemax);

        printf ("---\n");
    }

    endmntent (file);
    return 0;
}
```

В представленной программе функция `setmntent()` открывает файл /etc/mtab для чтения. Затем функция `getmntent()` вызывается в цикле для последовательного считывания всех записей /etc/mtab. Для каждой файловой системы вызывается функция `statvfs()`. И, наконец, функция `endmntent()` закрывает файл.

Обратите внимание, что точка монтирования является элементом соответствующей файловой системы и может быть указана первым аргументом `statvfs()`. Элементы структуры `statvfs` приводятся к беззнаковому длинному целому типу. Это обусловлено тем, что современные файловые системы могут содержать в данных полях достаточно большие значения. В *главе 16* мы вернемся к изучению этой программы.

Следующий пример (листинг 14.2) показывает, как работает функция `fstatvfs()`.

Листинг 14.2. Программа fstatvfsdemo.c

```
#include <sys/statvfs.h>
#include <fcntl.h>
#include <stdio.h>

int main (int argc, char ** argv)
{
    struct statvfs fs;
    int fd;
```

```
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file (%s)\n", argv[1]);
                return 1;
        }

        if (fstatvfs (fd, &fs) == -1) {
                fprintf (stderr, "fstatvfs() error\n");
                return 1;
        }

        printf ("Block size: %ld\n", (unsigned long int)
                                fs.f_bsize);
        printf ("Blocks: %ld\n", (unsigned long int)
                                fs.f_blocks);
        printf ("Blocks free: %ld\n", (unsigned long int)
                                fs.f_bfree);
        printf ("Blocks available: %ld\n",
                                (unsigned long int) fs.f_bavail);
        printf ("Max. filename length: %ld\n",
                                (unsigned long int) fs.f_namemax);
        close (fd);
        return 0;
}
```

14.2. Текущий каталог: *getcwd()*

В *разд. 7.1* уже упоминалось, что понятие текущего каталога заложено не в командную оболочку, а в само ядро Linux. Ядро хранит определенный набор параметров для каждого процесса, и одним из таких параметров является текущий каталог.

Каждый дочерний процесс наследует от своего родителя независимую копию текущего каталога. В этом можно легко убедиться:

```
$ cd /etc
$ bash
$ pwd
/etc
$ exit
exit
```

Для получения значения текущего каталога используется системный вызов `getcwd()`, объявленный в заголовочном файле unistd.h следующим образом:

```
char * getcwd (char * BUFFER, size_t SIZE);
```

> ***Примечание***
>
> Функция `getcwd()` стала системным вызовом в ядре Linux версии 2.1.92. До этого `getcwd()` была библиотечной функцией, которая читала данные о текущем каталоге из /proc/self/cwd.

Если в качестве аргумента `BUFFER` указан `NULL`, то `getcwd()` динамически выделяет блок памяти размером `SIZE` байт и возвращает адрес этого блока, предварительно поместив туда значение текущего каталога.

Можно также указать в первом аргументе собственный буфер, а во второй аргумент записать его размер. В этом случае `getcwd()` поместит в данный буфер значение текущего каталога.

Следует отметить, что `getcwd()` помещает в буфер полный путь к текущему каталогу. Чтобы получить только базовое имя каталога, можно воспользоваться функцией `basename()`, объявленной в заголовочном файле string.h следующим образом:

```
char * basename (const char * PATH);
```

Эта функция "отсекает" путь и возвращает только базовое имя файла или каталога.

> ***Примечание***
>
> Если первый аргумент `getcwd()` равен `NULL`, то полученный буфер после использования желательно освободить функцией `free()`. Это тот редкий случай, когда вызов `free()` для "кота в мешке" не является дурным стилем программирования.

Рассмотрим пример (листинг 14.3), демонстрирующий работу системного вызова `getcwd()`.

Листинг 14.3. Пример cwd.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PWD_BUF_SIZE       1024

int main (void)
{
       char * buf = (char*) malloc (PWD_BUF_SIZE * sizeof (char));
       if (buf == NULL) {
              fprintf (stderr, "buf is NULL\n");
              return 1;
       }
```

```
        buf = getcwd (buf, PWD_BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "getcwd() error\n");
                return 1;
        }

        printf ("Current directory: %s\n", buf);
        printf ("Basename: %s\n", basename (buf));

        free (buf);
        return 0;
}
```

Листинг 14.4 содержит другой пример, который тоже выводит значение текущего каталога, но возлагает обязанности выделения памяти на системный вызов `getcwd()`.

Листинг 14.4. Пример cwdauto.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PWD_BUF_SIZE       1024

int main (void)
{
        char * buf;
        if (buf == NULL) {
                fprintf (stderr, "buf is NULL\n");
                return 1;
        }

        buf = getcwd (NULL, PWD_BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "getcwd() error\n");
                return 1;
        }

        printf ("Current directory: %s\n", buf);
        printf ("Basename: %s\n", basename (buf));

        free (buf);
        return 0;
}
```

Обратите внимание, что подобное автоматическое выделение памяти системным вызовом `getcwd()` гарантированно работает только в Linux. В других Unix-подобных системах эта возможность может отсутствовать.

14.3. Получение дополнительной информации

В этой главе были описаны лишь самые основные приемы чтения информации о файловой системе. Например, аналогом семейства функций `statvfs()` является семейство системных вызовов `statfs()`. Информацию о них можно получить на соответствующих страницах руководства man:

- man 2 statfs;
- man 2 fstatfs.

Эти системные вызовы очень похожи на семейство функций `statvfs()`, однако поддерживаются только некоторыми Unix-подобными системами. Поэтому в этой главе было описано именно семейство `statvfs()`.

Рассмотренный в *разд. 14.2* системный вызов `getcwd()` — не единственный способ получения текущего каталога. Следующие man-страницы содержат информацию об аналогах `getcwd()`:

- man 3 getwd;
- man 3 get_current_dir_name.

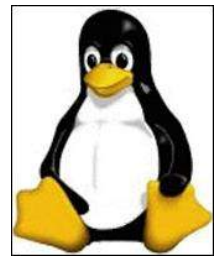
Поскольку не каждому пользователю Linux разрешено монтировать и размонтировать файловые системы, в этой книге не рассматриваются системные вызовы `mount()` и `umount()`, осуществляющие эти операции. Однако информацию о них можно получить на следующих страницах руководства man:

- man 2 mount;
- man 2 umount;
- man 2 umount2.

Программа df используется в Unix-совместимых системах для сбора информации о файловых системах. Изучая исходники этой программы, можно узнать много нового. df — это часть свободно распространяемого пакета программ GNU coreutils, исходные коды которого можно получить по адресу **ftp://ftp.gnu.org/pub/gnu/coreutils/**.

Исходный код программы df находится в файле src/df.c.

ГЛАВА 15



Чтение каталогов

В этой главе будут рассмотрены следующие темы:

- Смена текущего каталога при помощи системных вызовов `chdir()` и `fchdir()`.
- Открытие и закрытие каталога.
- Чтение и повторное чтение содержимого каталога.
- Получение данных о файлах посредством системных вызовов семейства `stat()`, `fstat()` и `lstat()`.
- Чтение ссылок через системный вызов `readlink()`.

15.1. Смена текущего каталога: *chdir()*

Любой процесс может сменить свой текущий каталог. Это можно сделать при помощи одного из приведенных далее системных вызовов, объявленных в заголовочном файле unistd.h:

```
int chdir (const char * PATH);
int fchdir (int FD);
```

Обе функции возвращают 0 при успешном завершении и –1, если произошла ошибка. Системный вызов `chdir()` изменяет текущий каталог, используя его имя (путь), а `fchdir()` вместо имени читает файловый дескриптор каталога.

> ***Примечание***
>
> На среднем уровне реализации файловой системы Linux каталоги рассматриваются как файлы и могут быть открыты при помощи системного вызова `open()`.

Листинг 15.1 иллюстрирует пример работы системного вызова `chdir()`.

Листинг 15.1. Пример chdirdemo.c

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

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

#define PWD_BUF_SIZE       1024
#define FILE_BUF_SIZE       4096

char file_buf [FILE_BUF_SIZE];

int main (void)
{
        int fd;
        ssize_t count;

        char * buf = getcwd (NULL, PWD_BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "getcwd() error\n");
                return 1;
        }

        printf ("Old dir: %s\n", buf);
        free (buf);

        if (chdir ("/etc") == -1) {
                fprintf (stderr, "chdir() error\n");
                return 1;
        }

        buf = getcwd (NULL, PWD_BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "getcwd() error\n");
                return 1;
        }

        printf ("New dir: %s\n", buf);
        free (buf);

        fd = open ("fstab", O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open fstab\n");
                return 1;
        }

        printf ("FSTAB:\n");
        while ((count = read (fd, file_buf, FILE_BUF_SIZE)) > 0)
                write (1, file_buf, count);

        close (fd);
        return 0;
}
```

Эта программа сначала выводит на экран начальное значение текущего каталога. Затем текущий каталог сменяется на /etc. После этого мы заново вызываем `getcwd()`, чтобы убедиться, что текущим каталогом теперь действительно является /etc. Наконец, чтобы окончательно удостовериться в том, что текущий каталог успешно изменен, мы читаем файл /etc/fstab, указав системному вызову `open()` только имя fstab.

Обратите внимание, что функция `free()` для указателя `buf` вызывается дважды. В *главе 14* мы уже говорили, что если системному вызову `getcwd()` передать `NULL` в качестве первого аргумента, то возвращенный указатель будет ссылаться на динамически выделенную память. Если бы мы не вызвали `free()` после первого чтения текущего каталога, то повторный вызов `getcwd()` привел бы к потере предыдущего указателя.

Давайте теперь переделаем нашу программу (листинг 15.1), заменив системный вызов `chdir()` на `fchdir()` (листинг 15.2).

Листинг 15.2. Пример fchdirdemo.c

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

#define PWD_BUF_SIZE       1024
#define FILE_BUF_SIZE       4096

char file_buf [FILE_BUF_SIZE];

int main (void)
{
       int fd, dirfd;
       ssize_t count;

       char * buf = getcwd (NULL, PWD_BUF_SIZE);
       if (buf == NULL) {
              fprintf (stderr, "getcwd() error\n");
              return 1;
       }

       printf ("Old dir: %s\n", buf);
       free (buf);

       dirfd = open ("/etc", O_RDONLY);
       if (dirfd == -1) {
              fprintf (stderr, "Cannot open /etc\n");
              return 1;
       }
```

```
        if (fchdir (dirfd) == -1) {
                fprintf (stderr, "fchdir() error\n");
                return 1;
        }

        buf = getcwd (NULL, PWD_BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "getcwd() error\n");
                return 1;
        }

        printf ("New dir: %s\n", buf);
        free (buf);

        fd = open ("fstab", O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open fstab\n");
                return 1;
        }

        printf ("FSTAB:\n");
        while ((count = read (fd, file_buf, FILE_BUF_SIZE)) > 0)
                write (1, file_buf, count);

        close (fd);
        close (dirfd);
        return 0;
}
```

Этот пример показывает, что для смены каталога не обязательно знать его имя.

15.2. Открытие и закрытие каталога: *opendir()*, *closedir()*

В рамках этой книги мы будем учиться читать содержимое каталогов при помощи библиотечных механизмов стандартной библиотеки языка C, которые (в большинстве случаев) более удобны, чем аналогичные системные вызовы.

Поскольку пользовательские библиотеки располагаются на верхнем уровне реализации файловой системы, то каталог здесь рассматривается как самостоятельная сущность, не являющаяся особым файловым типом. Поэтому библиотечная абстракция файла — указатель типа `FILE` — не может применяться к каталогам.

> ***Примечание***
>
> Вообще говоря, можно создать библиотеку, которая использовала бы одну и ту же абстракцию для файлов и каталогов. Но это привело бы к тому, что профессиональные программисты называют *избыточной унификацией*.

В стандартной библиотеке языка C абстракцией каталога является указатель типа DIR. Прежде чем читать содержимое каталога, его нужно открыть, а прочитанный каталог полагается закрывать. Для этого существуют следующие функции, объявленные в заголовочном файле dirent.h:

```
DIR * opendir (const char * NAME);
int closedir (DIR * DIRP);
```

Первая функция открывает каталог с именем NAME и возвращает указатель типа DIR. В случае ошибки возвращается NULL. Функция closedir() закрывает каталог и возвращает 0 при успешном завершении. В случае ошибки closedir() возвращает –1.

Каталог можно также открыть при помощи функции fdopendir(), которая вместо имени каталога использует файловый дескриптор:

```
DIR * fdopendir (int FD);
```

По своему поведению fdopendir() напоминает функцию fdopen(), которая описывалась в *разд. 8.1*.

15.3. Чтение каталога: *readdir()*

Для чтения содержимого каталога применяется функция readdir(), объявленная в заголовочном файле dirent.h следующим образом:

```
struct dirent * readdir (DIR * DIRP);
```

Структура dirent содержит несколько полей, но нам понадобится только одно из них:

```
struct dirent {
       /* ... */
       char * d_name;
};
```

Итак, функция readdir() заносит в поле d_name структуры dirent имя очередного элемента просматриваемого каталога. Если каталог полностью просмотрен, то readdir() возвращает NULL.

> ***Примечание***
>
> На самом деле указатель d_name объявлен в виде статического массива d_name[256], но вас это не должно волновать.

Следующая программа (листинг 15.3) выводит на экран содержимое указанного каталога примерно так же, как это делает программа ls с набором флагов -1fa (в одну колонку, не сортируя, все файлы).

Листинг 15.3. Программа readdir1.c

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

```
int main (int argc, char ** argv)
{
        DIR * dir;
        struct dirent * entry;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        dir = opendir (argv[1]);
        if (dir == NULL) {
                fprintf (stderr, "opendir() error\n");
                return 1;
        }

        while ((entry = readdir (dir)) != NULL)
                printf ("%s\n", entry->d_name);

        closedir (dir);
        return 0;
}
```

15.4. Повторное чтение каталога: *rewinddir()*

Бывают ситуации, когда требуется повторно прочитать каталог. Для этого, конечно, можно закрыть, а потом заново открыть этот каталог. Но есть более простой способ установки текущей позиции чтения каталога на первый элемент. Это делает функция `rewinddir()`, объявленная в файле dirent.h следующим образом:

```
void rewinddir (DIR * DIRP);
```

Рассмотрим сразу пример (листинг 15.4), который демонстрирует работу этой функции.

Листинг 15.4. Пример readdir2.c

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

void read_directory (DIR * dir)
{
        struct dirent * entry;
        while ((entry = readdir (dir)) != NULL)
                printf ("%s\n", entry->d_name);
}
```

```
int main (int argc, char ** argv)
{
       DIR * dir;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       dir = opendir (argv[1]);
       if (dir == NULL) {
              fprintf (stderr, "opendir() error\n");
              return 1;
       }

       read_directory (dir);

       printf ("-- RE-READ --\n");

       rewinddir (dir);
       read_directory (dir);

       closedir (dir);
       return 0;
}
```

15.5. Получение данных о файлах: семейство *stat()*

В большинстве случаев при просмотре каталога мало знать только имя файла. Дополнительную информацию о файлах позволяют получить системные вызовы *семейства stat()*, объявленные в заголовочном файле sys/stat.h:

```
int stat (const char * FNAME, struct stat * STATISTICS);
int fstat (int FD, struct stat * STATISTICS);
int lstat (const char * FNAME, struct stat * STATISTICS);
```

Системный вызов `stat()` читает информацию о файле с именем `FNAME` и записывает эту информацию в структуру `stat` по адресу `STATISTICS`. Вызов `fstat()` также читает информацию о файле, который представлен дескриптором `FD`. И, наконец, `lstat()` работает аналогично `stat()`. Но при обнаружении символической ссылки `lstat()` читает информацию *о ней*, а не о файле, на который эта ссылка указывает.

Структура `stat` также объявлена в файле sys/stat.h. Для нас наибольший интерес представляют следующие поля этой структуры:

```
struct stat
{

       mode_t st_mode;
       uid_t st_uid;
       gid_t st_gid;
       off_t st_size;
       time_t st_atime;
       time_t st_mtime;
};
```

Примечание

На самом деле структура `stat` объявлена в файле bits/stat.h, который включается в файл sys/types.h, но вас это никак не должно тревожить.

Перечислим назначение полей структуры `stat`:

- `st_mode` — это режим файла. Позже в этом разделе мы узнаем секрет его использования;
- `st_uid` — это числовой идентификатор владельца файла;
- `st_gid` — идентификатор группы;
- `st_size` — это размер файла в байтах;
- `st_atime` — содержит дату и время последнего обращения к файлу;
- `st_mtime` — содержит дату и время последней модификации файла.

Замечание

В *главе 16* мы познакомимся также с полем `st_ino` структуры `stat`.

Все системные вызовы семейства `stat()` возвращают 0 при успешном завершении. В случае ошибки возвращается –1. Следующий пример (листинг 15.5) демонстрирует использование системного вызова `stat()`.

Листинг 15.5. Пример stat1.c

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

int main (int argc, char ** argv)
{
       struct stat st;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }
```

```
        if (stat (argv[1], &st) == -1) {
                fprintf (stderr, "stat() error\n");
                return 1;
        }

        printf ("FILE:\t\t%s\n", argv[1]);
        printf ("UID:\t\t%d\n", (int) st.st_uid);
        printf ("GID:\t\t%d\n", (int) st.st_gid);
        printf ("SIZE:\t\t%ld\n", (long int) st.st_size);
        printf ("AT:\t\t%s", ctime (&st.st_atime));
        printf ("MT:\t\t%s", ctime (&st.st_mtime));

        return 0;
}
```

В *главе 16* мы вернемся к изучению этой программы.

Рассмотрим теперь пример системного вызова `fstat()` (листинг 15.6).

Листинг 15.6. Пример stat2.c

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

int main (int argc, char ** argv)
{
        struct stat st;
        int fd;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                "(%s)\n", argv[1]);
                return 1;
        }

        if (fstat (fd, &st) == -1) {
                fprintf (stderr, "stat() error\n");
```

```
                return 1;
        }

        printf ("FILE:\t\t%s\n", argv[1]);
        printf ("UID:\t\t%d\n", (int) st.st_uid);
        printf ("GID:\t\t%d\n", (int) st.st_gid);
        printf ("SIZE:\t\t%ld\n", (long int) st.st_size);
        printf ("AT:\t\t%s", ctime (&st.st_atime));
        printf ("MT:\t\t%s", ctime (&st.st_mtime));

        close (fd);
        return 0;
}
```

На первый взгляд обе программы работают абсолютно одинаково. Но давайте проведем небольшой эксперимент:

```
$ touch myfile
$ chmod 000 myfile
$ ls -l myfile
---------- 1 nnivanov nnivanov 0 2011-05-07 09:24 myfile
$ ./stat1 myfile
FILE:          myfile
UID:           500
GID:           500
SIZE:          0
AT:            Sat May  7 09:24:17 2011
MT:            Sat May  7 09:24:17 2011
$ ./stat2 myfile
Cannot open file (myfile)
```

Итак, программа stat1 (см. листинг 15.5) оказалась в преимущественном положении, поскольку системный вызов `stat()` может читать информацию о файлах с любыми правами доступа.

Рассмотрим теперь пример системного вызова `lstat()` (листинг 15.7).

Листинг 15.7. Пример stat3.c

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

void print_statistics (const char * fname, struct stat * st)
{
        printf ("FILE:\t\t%s\n", fname);
        printf ("UID:\t\t%d\n", (int) st->st_uid);
        printf ("GID:\t\t%d\n", (int) st->st_gid);
```

```
        printf ("SIZE:\t\t%ld\n", (long int) st->st_size);
        printf ("AT:\t\t%s", ctime (&st->st_atime));
        printf ("MT:\t\t%s", ctime (&st->st_mtime));
}

int main (int argc, char ** argv)
{
        struct stat st;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (stat (argv[1], &st) == -1) {
                fprintf (stderr, "stat() error\n");
                return 1;
        }

        printf ("-- STAT() --\n");
        print_statistics (argv[1], &st);

        if (lstat (argv[1], &st) == -1) {
                fprintf (stderr, "lstat() error\n");
                return 1;
        }

        printf ("-- LSTAT() --\n");
        print_statistics (argv[1], &st);

        return 0;
}
```

Вот что получилось в результате:

```
$ touch myfile
$ ln -s myfile mylink
$ ./stat3 myfile
-- STAT() --
FILE:           myfile
UID:            500
GID:            500
SIZE:           0
AT:             Sat May  7 09:59:14 2011
MT:             Sat May  7 09:59:14 2011
-- LSTAT() --
FILE:           myfile
```

```
UID:            500
GID:            500
SIZE:           0
AT:             Sat May  7 09:59:14 2011
MT:             Sat May  7 09:59:14 2011
$ ./stat3 mylink
-- STAT() --
FILE:           mylink
UID:            500
GID:            500
SIZE:           0
AT:             Sat May  7 09:59:14 2011
MT:             Sat May  7 09:59:14 2011
-- LSTAT() --
FILE:           mylink
UID:            500
GID:            500
SIZE:           6
AT:             Sat May  7 09:59:52 2011
MT:             Sat May  7 09:59:24 2011
```

Приведенный эксперимент доказывает, что системные вызовы `stat()` и `lstat()` поразному интерпретируют символические ссылки, но ведут себя одинаково по отношению к другим типам файлов.

Вернемся теперь к полю `st_mode` структуры `stat`, которое мы до сих пор игнорировали. Вы уже знаете, что биты режима файла делятся на три группы:

- базовые права доступа (см. *разд. 7.1*);
- расширенные права доступа (см. *разд. 13.3*);
- тип файла (см. *разд. 13.2*).

Для проверки базовых прав доступа достаточно сравнить режим файла с одной из констант из sys/types.h (`S_IRUSR`, `S_IWUSR` и т. п.) с использованием операции побитовой конъюнкции (И). Например, если владельцу разрешено писать в файл, то следующее выражение будет истинным:

```
mode & S_IWUSR
```

Аналогичным образом проверяются расширенные права доступа:

- константа `S_ISUID` используется для установки и проверки бита SUID;
- константа `S_ISGID` служит для установки и проверки бита SGID;
- константа `S_ISVTX` предназначена для установки и проверки sticky-бита.

Таким образом, если, например, для исполняемого файла с режимом `mode` установлен бит SUID, то следующее выражение будет истинным:

```
mode & S_ISUID
```

А для определения типа файла применяются следующие макросы:

- `S_ISDIR(mode)` — каталог;
- `S_ISCHR(mode)` — символьное устройство;
- `S_ISBLK(mode)` — блочное устройство;
- `S_ISREG(mode)` — обычный файл;
- `S_ISFIFO(mode)` — именованный канал FIFO;
- `S_ISLNK(mode)` — символическая ссылка;
- `S_ISSOCK(mode)` — сокет.

Давайте теперь напишем программу, которая читает содержимое каталога и выводит тип и размер каждого файла (листинг 15.8).

Листинг 15.8. Программа readdir3.c

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <sys/stat.h>

void print_stat (const char * fname)
{
       struct stat st;
       if (lstat (fname, &st) == -1) {
              fprintf (stderr, "lstat() error\n");
              exit (1);
       }

       if (S_ISDIR (st.st_mode))
              printf ("dir");
       else if (S_ISCHR (st.st_mode))
              printf ("chrdev");
       else if (S_ISBLK (st.st_mode))
              printf ("blkdev");
       else if (S_ISREG (st.st_mode))
              printf ("regfile");
       else if (S_ISFIFO (st.st_mode))
              printf ("fifo");
       else if (S_ISLNK (st.st_mode))
              printf ("symlink");
       else if (S_ISSOCK (st.st_mode))
              printf ("socket");
       else
              printf ("unknown");
```

```
        printf (", %d\n", st.st_size);
}

int main (int argc, char ** argv)
{
        DIR * dir;
        struct dirent * entry;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        dir = opendir (argv[1]);
        if (dir == NULL) {
                fprintf (stderr, "opendir() error\n");
                return 1;
        }

        while ((entry = readdir (dir)) != NULL) {
                printf ("%s\t\t", entry->d_name);

                if (chdir (argv[1]) == -1) {
                        fprintf (stderr, "chdir() error\n");
                        return 1;
                }

                print_stat (entry->d_name);
        }

        closedir (dir);
        return 0;
}
```

Обратите внимание, что в приведенной программе чтение информации о файлах реализует системный вызов `lstat()`. Если бы мы использовали `stat()`, то символические ссылки заменялись бы файлами, на которые они указывают, и макрос `S_ISLNK()` никогда не принимал бы истинные значения.

15.6. Чтение ссылок: *readlink()*

Программа ls, вызванная с флагом `-l`, позволяет узнать, на какой файл указывает символическая ссылка:

```
$ touch myfile
$ ln -s myfile mylink
$ ls -l mylink
lrwxrwxrwx 1 nnivanov nnivanov 6 2011-05-07 09:59 mylink -> myfile
```

Такая операция называется *разыменованием символической ссылки* (symlink dereferencing) или *разрешением имени файла* (filename resolution). Разыменование символических ссылок осуществляется при помощи системного вызова `readlink()`, который объявлен в заголовочном файле unistd.h следующим образом:

```
ssize_t readlink (const char * SYMLNK, char * BUF, size_t SIZE);
```

Этот системный вызов помещает в буфер `BUF` размера `SIZE` путь к файлу, на который указывает символическая ссылка `SYMLNK`. Возвращаемое значение — число байтов, записанных в буфер `BUF`. В случае ошибки возвращается –1.

Следует отметить, что системный вызов `readlink()` не завершает буфер нуль-терминатором. Об этом программист должен заботиться самостоятельно.

Давайте теперь изменим программу readdir3 из листинга 15.8 так, чтобы она разыменовывала символические ссылки (листинг 15.9).

Листинг 15.9. Программа readdir4.c

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define READLNK_BUF_SIZE       1024
char readlnk_buf  [READLNK_BUF_SIZE+1];

void print_link_info (const char * fname)
{
       int count = readlink (fname, readlnk_buf,
                     READLNK_BUF_SIZE);
       if (count == -1) {
              fprintf (stderr, "readlink() error\n");
              exit (1);
       }

       readlnk_buf [count] = '\0';
       printf ("%s --> %s", fname, readlnk_buf);
}

void print_stat (const char * fname)
{
       struct stat st;
       if (lstat (fname, &st) == -1) {
              fprintf (stderr, "lstat() error\n");
              exit (1);
       }
```

```
        if (S_ISDIR (st.st_mode))
                printf ("dir");
        else if (S_ISCHR (st.st_mode))
                printf ("chrdev");
        else if (S_ISBLK (st.st_mode))
                printf ("blkdev");
        else if (S_ISREG (st.st_mode))
                printf ("regfile");
        else if (S_ISFIFO (st.st_mode))
                printf ("fifo");
        else if (S_ISLNK (st.st_mode))
                print_link_info (fname);
        else if (S_ISSOCK (st.st_mode))
                printf ("socket");
        else
                printf ("unknown");

        printf (", %d\n", st.st_size);
}

int main (int argc, char ** argv)
{
        DIR * dir;
        struct dirent * entry;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        dir = opendir (argv[1]);
        if (dir == NULL) {
                fprintf (stderr, "opendir() error\n");
                return 1;
        }

        while ((entry = readdir (dir)) != NULL) {
                printf ("%s\t\t", entry->d_name);

                if (chdir (argv[1]) == -1) {
                        fprintf (stderr, "chdir() error\n");
                        return 1;
                }

                print_stat (entry->d_name);
        }
```

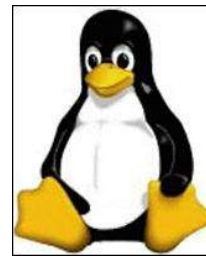
```
        closedir (dir);
        return 0;
}
```

Вот что получилось в результате выполнения:

```
$ touch myfile
$ ln -s myfile mylink
$ ./readdir4 .
.               dir, 4096
..              dir, 4096
myfile          regfile, 0
readdir4                regfile, 8404
readdir4.c              regfile, 1529
mylink          mylink --> myfile, 6
```

Обратите внимание, что размер символической ссылки `mylink` составляет 6 байтов. Это длина пути к файлу, на который указывает ссылка. Проверим это:

```
$ ln -s /etc/fstab mynewlink
$ ./readdir4 .
.               dir, 4096
..              dir, 4096
myfile          regfile, 0
readdir4                regfile, 8404
readdir4.c              regfile, 1529
mylink          mylink --> myfile, 6
mynewlink               mynewlink --> /etc/fstab, 10
```



ГЛАВА 16

Операции над файлами

В этой главе будут рассмотрены следующие темы:

- Удаление файла при помощи системного вызова `unlink()`.
- Файловые индексы, каталоги, ссылки.
- Переименование и перемещение файла с использованием системного вызова `rename()`.
- Создание жестких и символических ссылок.
- Создание и удаление каталогов.
- Маска прав доступа и системный вызов `umask()`.

16.1. Удаление файла: *unlink()*

Для удаления файла служит системный вызов `unlink()`, объявленный в заголовочном файле unistd.h следующим образом:

```
int unlink (const char * FNAME);
```

Аргумент `FNAME` — это имя удаляемого файла. `unlink()` возвращает 0 при успешном завершении. В случае ошибки возвращается –1.

> ***Примечание***
>
> На среднем уровне реализации файловой системы каталоги рассматриваются как файлы, однако для их удаления используется системный вызов `rmdir()`, о котором речь пойдет в *разд. 16.5*.

Следующая программа (листинг 16.1) показывает, как работает системный вызов `unlink()`.

Листинг 16.1. Программа unlink1.c

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

```
int main (int argc, char ** argv)
{
       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       if (unlink (argv[1]) == -1) {
              fprintf (stderr, "Cannot unlink file "
                           "(%s)\n", argv[1]);
              return 1;
       }

       return 0;
}
```

На первый взгляд все кажется простым и понятным. Но возникает вопрос: почему создатели UNIX назвали этот системный вызов странным именем `unlink()`, а не `remove()` или `delete()`, например? А все дело в том, что файл — это комплексное понятие, состоящее из следующих компонентов:

- *данные* (data);
- *индексы* (inodes);
- *ссылки* (links).

Реальные файловые системы обычно располагаются на блочных устройствах. Данные на таких устройствах в действительности адресуются блоками фиксированного размера, а не отдельными байтами.

Индексы — это специальные ячейки памяти, зарезервированные файловой системой для разделения смешанных "в одну кучу" данных на отдельные структурные единицы, которые мы называем файлами. Индексы содержат информацию о том, в каких блоках файловой системы хранятся данные конкретного файла. В индексах также содержатся сведения о дате и времени открытия и модификации файла. Иногда индексы называют *индексными узлами*. Каждый индекс имеет уникальный в рамках данной файловой системы номер.

Ссылки — это привычное для нас отображение файлов в файловой системе. Можно условно сказать, что ссылки являются именами индексных узлов файловой системы. Каталоги можно рассматривать как особые файлы, предназначенные для хранения ссылок на индексные узлы.

Программа ls, вызванная с флагом `-i`, позволяет увидеть номер индексного узла, на который указывает ссылка. Проведем небольшой эксперимент:

```
$ mkdir idemo
$ cd idemo
$ touch file1
$ touch file2
```

```
$ ls -i
952139 file1  952141 file2
```

Итак, приведенный пример показывает, что `file1` — это ссылка на индекс с номером 952139 (у вас он, скорее всего, будет другим). А ссылка `file2` указывает на другой индексный узел с номером 952141. Продолжим эксперимент:

```
$ ln -s file1 symlnkf1
$ ls -i
952139 file1  952141 file2  952190 symlnkf1
```

Вывод программы ls показывает, что символическая ссылка является также ссылкой на индекс с номером 952190. Важно понимать, что символические ссылки указывают не на индекс, а на имя файла.

Теперь воспользуемся командой `ln` без флага `-s`:

```
$ ln file1 hardfile1
$ ls -i
952139 file1  952141 file2  952139 hardfile1  952190 symlnkf1
```

Вы, наверное, знаете, что наряду с символическими существуют также жесткие ссылки. `hardfile1` является жесткой ссылкой на файл file1. Вывод команды `ls` показывает, что `file1` и `hardfile1` указывают на один и тот же индекс с номером 952139. На самом деле жесткие ссылки (в отличие от символических) не носят подчиненный характер. Следовательно, `file1` и `hardfile1` — это полноценные ссылки на один и тот же индексный узел.

Индексы являются промежуточным звеном, связывающим ссылку с данными на блочном устройстве. Если две ссылки указывают на один и тот же индекс, то можно сказать, что они имеют доступ к одним и тем же данным. Это легко проверить:

```
$ echo hello > file1
$ cat hardfile1
hello
```

Теперь мы можем сказать, что программа rm работает следующим образом:

- если удаляемый файл является последней ссылкой на соответствующий индексный узел в файловой системе, то данные и индекс освобождаются;
- если в файловой системе еще остались ссылки на соответствующий индексный узел, то удаляется только ссылка.

Проверим это:

```
$ rm file1
$ cat hardfile1
hello
```

Теперь обратите внимание на вывод программы ls с флагом `-l`:

```
$ ls -l
-rw-r--r-- 1 nnivanov nnivanov 0 2011-05-07 10:00 file2
-rw-r--r-- 1 nnivanov nnivanov 6 2011-05-07 10:00 hardfile1
lrwxrwxrwx 1 nnivanov nnivanov 5 2011-05-07 10:00 symlnkf1 -> file1
```

Символическая ссылка `symlnkf1` по-прежнему указывает на файл file1, которого уже не существует:

```
$ cat symlnkf1
cat: symlnkf1: No such file or directory
```

Обратите внимание на числа во втором столбце вывода программы ls. Это счетчики ссылок на соответствующие индексные узлы. В нашем случае, например, на индексный узел 952141 указывает всего одна ссылка `file2`. Поэтому удаление `file2` командой `rm` приведет к освобождению индекса 952141. Но если мы создадим еще одну ссылку, то счетчик индексов увеличится:

```
$ ls -l
-rw-r--r-- 2 nnivanov nnivanov 0 2011-05-07 10:00 file2
-rw-r--r-- 1 nnivanov nnivanov 6 2011-05-07 10:00 hardfile1
-rw-r--r-- 2 nnivanov nnivanov 0 2011-05-07 10:00 hardfile2
lrwxrwxrwx 1 nnivanov nnivanov 5 2011-05-07 10:00 symlnkf1 -> file1
```

Теперь во втором столбце вывода ls напротив `file2` и `hardfile2` находится число 2, символизирующее о том, что на соответствующий индексный узел указывают две ссылки.

Если вызвать команду `df` с флагом `-i`, то на экран будет выведена информация по индексным узлам смонтированных файловых систем:

```
$ df -i
Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda6       1311552  264492 1047060   21% /
udev              96028     487   95541    1% /dev
/dev/sda1         66264      58   66206    1% /boot
/dev/sda7       3407872  149600 3258272    5% /home
```

В файловых системах может присутствовать ограниченное число индексов (столбец Inodes). Столбец IUsed показывает число используемых индексов, а в столбце IFree содержится число свободных индексных узлов файловой системы.

Итак, каждый раз при создании файла в файловой системе выделяется индекс:

```
$ df -i .
Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda7       3407872  149586 3258286    5% /home
$ touch file3
$ df -i .
Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda7       3407872  149587 3258285    5% /home
```

А создание жесткой ссылки не приводит к появлению в файловой системе нового индекса:

```
$ df -i .
Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda7       3407872  149587 3258285    5% /home
$ ln file3 hardfile3
```

```
$ df -i .
Filesystem       Inodes   IUsed   IFree IUse% Mounted on
/dev/sda7       3407872  149587 3258285    5% /home
```

Примечание

Обратите внимание, что приведенные примеры могут работать некорректно, если какая-нибудь программа в вашей системе параллельно занимается созданием или удалением файлов.

Вернемся теперь к программе statvfsdemo из *разд. 14.1* (листинг 14.1). Эта программа использовала функцию `statvfs()` для вывода информации о смонтированных файловых системах. Структура `statvfs` содержит также следующие поля, которые мы не рассматривали в *главе 14*:

- `f_files` — общее число индексов для данной файловой системы;
- `f_free` — число свободных индексов файловой системы;
- `f_favail` — число доступных индексов в файловой системе.

Листинг 16.2 содержит дополненную версию программы statvfsdemo, выводящую также информацию об индексах для каждой файловой системы.

Листинг 16.2. Программа statvfsinode.c

```
#include <sys/statvfs.h>
#include <mntent.h>
#include <stdio.h>

int main (void)
{
       struct mntent * entry;
       struct statvfs fs;
       FILE * file;

       file = setmntent ("/etc/mtab", "r");
       if (file == NULL) {
              fprintf (stderr, "Cannot open /etc/mtab\n");
              return 1;
       }

       while ((entry = getmntent (file)) != NULL) {
              if (statvfs (entry->mnt_dir, &fs) == -1) {
                     fprintf (stderr, "statvfs() error\n");
                     return 1;
              }

              printf ("Filesystem: %s\n", entry->mnt_fsname);
              printf ("Mountpoint: %s\n", entry->mnt_dir);
              printf ("Block size: %ld\n", (unsigned long int)
                            fs.f_bsize);
```

```
                printf ("Blocks: %ld\n", (unsigned long int)
                        fs.f_blocks);
                printf ("Blocks free: %ld\n", (unsigned long int)
                        fs.f_bfree);
                printf ("Blocks available: %ld\n",
                        (unsigned long int) fs.f_bavail);
                printf ("Max. filename length: %ld\n",
                        (unsigned long int) fs.f_namemax);

                printf ("Inodes: %ld\n",
                        (unsigned long int) fs.f_files);
                printf ("Inodes free: %ld\n",
                        (unsigned long int) fs.f_ffree);
                printf ("Inodes available: %ld\n",
                        (unsigned long int) fs.f_favail);

                printf ("---\n");
        }

        endmntent (file);
        return 0;
}
```

Рассмотрим также программу stat1 из *разд. 15.5* (листинг 15.5). Структура `stat` содержит еще одно поле `st_ino`, в котором находится номер индексного узла, на который ссылается файл. Листинг 16.3 содержит дополненную версию программы stat1.

Листинг 16.3. Программа statinode.c

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

int main (int argc, char ** argv)
{
        struct stat st;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (stat (argv[1], &st) == -1) {
                fprintf (stderr, "stat() error\n");
```

```
                return 1;
        }

        printf ("FILE:\t\t%s\n", argv[1]);
        printf ("UID:\t\t%d\n", (int) st.st_uid);
        printf ("GID:\t\t%d\n", (int) st.st_gid);
        printf ("SIZE:\t\t%ld\n", (long int) st.st_size);
        printf ("AT:\t\t%s", ctime (&st.st_atime));
        printf ("MT:\t\t%s", ctime (&st.st_mtime));

        printf ("INODE:\t\t%ld\n", (long int) st.st_ino);

        return 0;
}
```

Вернемся к программе unlink1 (см. листинг 16.1). Теперь мы можем сказать, что системный вызов `unlink()` удаляет ссылку на индексный узел. Если эта ссылка была последней, то индекс освобождается. Рассмотрим еще один пример, демонстрирующий эту концепцию (листинг 16.4).

Листинг 16.4. Пример unlink2.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define HELLO_MSG       "Hello World\n"

int main (int argc, char ** argv)
{
        int fd;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_WRONLY | O_TRUNC, 0644);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                "(%s)\n", argv[1]);
                return 1;
        }

        if (unlink (argv[1]) == -1) {
                fprintf (stderr, "Cannot unlink file "
                                "(%s)\n", argv[1]);
```

```
                return 1;
        }

        if (write (fd, HELLO_MSG,
                        strlen (HELLO_MSG)) == -1) {
                fprintf (stderr, "write() error\n");
                return 1;
        }

        if (close (fd) == -1) {
                fprintf (stderr, "close() error\n");
                return 1;
        }

        return 0;
}
```

Вот что получилось после выполнения этого примера:

```
$ touch file1
$ ln file1 file2
$ ls -i file1 file2
1048740 file1  1048740 file2
$ ./unlink2 file1
$ ls file1
/bin/ls: file1: No such file or directory
$ cat file2
Hello World
```

Проведем еще один эксперимент:

```
$ ./unlink2 file2
$ ls file2
/bin/ls: file2: No such file or directory
```

Эта программа показывает, что над открытым файлом можно успешно осуществлять операции ввода-вывода, даже если последняя ссылка на этот файл удалена системным вызовом `unlink()`. Мы вернемся к рассмотрению этой концепции в *главе 18*.

16.2. Перемещение файла: *rename()*

Системный вызов `rename()` позволяет переименовывать или перемещать файл в пределах одной файловой системы. В заголовочном файле stdio.h он объявлен следующим образом:

```
int rename (const char * OLDF, const char * NEWF);
```

При успешном завершении `rename()` возвращает 0. В случае ошибки возвращается –1.

Примечание

Функция `rename()` является в Linux системным вызовом, хотя и объявлена в файле stdio.h, где обычно располагаются библиотечные механизмы. Реализацию `rename()` можно найти в исходниках ядра Linux в файле fs/namei.c.

Следующий пример (листинг 16.5) демонстрирует работу системного вызова `rename()`.

Листинг 16.5. Пример rename1.c

```
#include <stdio.h>

int main (int argc, char ** argv)
{
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (rename (argv[1], argv[2]) == -1) {
                fprintf (stderr, "rename() error\n");
                return 1;
        }

        return 0;
}
```

Посмотрим теперь, что будет делать `rename()`, если перемещаемый файл открыт. Для этого создадим еще одну программу (листинг 16.6).

Листинг 16.6. Программа rename2.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define HELLO_MSG       "Hello World\n"

int main (int argc, char ** argv)
{
        int fd;

        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        fd = open (argv[1], O_WRONLY | O_TRUNC, 0644);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file "
                                 "(%s)\n", argv[1]);
                return 1;
        }

        if (rename (argv[1], argv[2]) == -1) {
                fprintf (stderr, "rename() error\n");
                return 1;
        }

        if (write (fd, HELLO_MSG,
                        strlen (HELLO_MSG)) == -1) {
                fprintf (stderr, "write() error\n");
                return 1;
        }

        close (fd);
        return 0;
}
```

Теперь проведем небольшой эксперимент:

```
$ touch file1
$ cat file1
$ ./rename2 file1 file2
$ ls file1
/bin/ls: file1: No such file or directory
$ cat file2
Hello World
```

Итак, мы убедились, что перемещение открытого файла никак не отражается на операциях ввода-вывода.

16.3. Создание ссылок: *link()*

Мы выяснили, что ссылки в файловой системе Linux бывают двух типов:

- символические ссылки (symbolic links);
- жесткие (прямые) ссылки (hard links).

Пользователь может создавать ссылки при помощи программы ln. А в распоряжении программиста имеются системные вызовы `link()` и `symlink()`. Первый создает прямые ссылки, второй — символические. Эти системные вызовы объявлены в заголовочном файле unistd.h следующим образом:

```
int link (const char * FROM, const char * TO);
int symlink (const char * FROM, const char * TO);
```

Оба возвращают 0 при успешном завершении и –1, если произошла ошибка. Аргументы FROM (откуда) и TO (куда) аналогичны соответствующим свободным аргументам программы ln.

Следующая программа (листинг 16.7) показывает, как работает системный вызов `link()`.

Листинг 16.7. Программа linkdemo.c

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

int main (int argc, char ** argv)
{
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (link (argv[1], argv[2]) == -1) {
                fprintf (stderr, "link() error\n");
                return 1;
        }

        return 0;
}
```

Проверяем:

```
$ touch file1
$ ./linkdemo file1 file2
$ ls -i file1 file2
1048776 file1  1048776 file2
```

Теперь создадим еще одну программу (листинг 16.8), которая демонстрирует работу системного вызова `symlink()`.

Листинг 16.8. Программа symlinkdemo.c

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

int main (int argc, char ** argv)
{
        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        if (symlink (argv[1], argv[2]) == -1) {
                fprintf (stderr, "link() error\n");
                return 1;
        }

        return 0;
}
```

Вот что получилось в результате:

```
$ touch file1
$ ./symlinkdemo file1 file2
$ ls -li file1 file2
1048779 -rw-r--r-- 1 nnivanov nnivanov 0 2011-05-07 10:03 file1
1048780 lrwxrwxrwx 1 nnivanov nnivanov 5 2011-05-07 10:03 file2 -> file1
```

16.4. Создание каталога: *mkdir()*

Для создания каталога используется системный вызов `mkdir()`, который объявлен в заголовочном файле sys/stat.h следующим образом:

```
int mkdir (const char * NAME, mode_t MODE);
```

Системный вызов `mkdir()` создает каталог с именем `NAME` и режимом `MODE`. При успешном завершении `mkdir()` возвращает 0. В случае ошибки возвращается –1.

Следующий пример (листинг 16.9) показывает, как работает системный вызов `mkdir()`.

Листинг 16.9. Пример mkdir1.c

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

int main (int argc, char ** argv)
{
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (mkdir (argv[1], 0755) == -1) {
                fprintf (stderr, "mkdir() error\n");
                return 0;
        }

        return 0;
}
```

Приведенный пример — это упрощенная версия программы mkdir:

```
$ ./mkdir1 mydir
$ ls -l | grep mydir
drwx------ 2 nnivanov nnivanov 4096 2011-05-07 10:05 mydir
```

Рассмотрим еще один пример (листинг 16.10).

Листинг 16.10. Пример mkdir2.c

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

int main (int argc, char ** argv)
{
        mode_t mode = 0777;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (mkdir (argv[1], mode) == -1) {
                fprintf (stderr, "mkdir() error\n");
                return 0;
        }

        return 0;
}
```

Эта программа работает не так, как мы ожидали:

```
$ ./mkdir2 mydir
$ ls -l | grep mydir
drwxr-xr-x 2 nnivanov nnivanov 4096 2011-05-07 10:09 mydir
```

В системный вызов `mkdir()` передавался аргумент `mode`, в котором все биты базовых прав доступа установлены в единицу. Но вывод программы ls показывает, что созданный каталог имеет права доступа 0755 (у вас может быть что-то другое). Давайте разберемся, почему так получилось.

К каждому процессу в Linux привязана *маска прав доступа*, которая наследуется потомком от родительского процесса (аналогично текущему каталогу, окружению и т. п.). Маска прав доступа — это число, представляющее собой набор битов прав доступа, которые никогда не будут устанавливаться для создаваемых процессом файлов или каталогов. Команда `umask` позволяет узнать текущую маску прав доступа командной оболочки:

```
$ umask
0022
```

Итак, маска прав доступа 0022 разрешает при создании файлов или каталогов устанавливать любые права доступа для владельца, но не разрешает права на запись для группы и остальных пользователей.

> ***Примечание***
>
> umask — внутренняя команда оболочки bash. Многие другие оболочки (csh, ksh и проч.) также содержат в своем арсенале эту команду.

Текущий процесс вправе изменять свою копию маски прав доступа. Например, оболочка bash делает это при помощи той же самой команды umask:

```
$ umask 0044
$ umask
0044
```

Дочерние процессы наследуют копию маски прав доступа родительского процесса. Это легко проверить:

```
$ umask 000
$ umask
0000
$ bash
$ umask
0000
$ exit
exit
```

Давайте теперь вернемся к нашему примеру (листинг 16.10). Попробуем запустить программу mkdir2 с измененной маской прав доступа оболочки:

```
$ umask 0000
$ umask
0000
$ ./mkdir2 mydir
$ ls -l | grep mydir
drwxrwxrwx 2 nnivanov nnivanov 4096 2011-05-07 10:15 mydir
```

Программа может изменить маску прав доступа текущего процесса при помощи системного вызова umask(), который объявлен в заголовочном файле /sys/stat.h следующим образом:

```
mode_t umask (mode_t MASK);
```

Этот системный вызов изменяет текущую маску прав доступа и возвращает предыдущее значение маски. Рассмотрим теперь пример использования системного вызова umask() для создания каталога с правами доступа 0777 (листинг 16.11).

Листинг 16.11. Пример mkdir3.c

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

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

int main (int argc, char ** argv)
{
       mode_t mode = 0777;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       umask (0000);
       if (mkdir (argv[1], mode) == -1) {
              fprintf (stderr, "mkdir() error\n");
              return 0;
       }

       return 0;
}
```

Вот что получилось в результате выполнения этого примера:

```
$ umask 0022
$ umask
0022
$ ./mkdir3 mydir
$ ls -l | grep mydir
drwxrwxrwx 2 nnivanov nnivanov 4096 2011-05-07 10:17 mydir
```

Примечание

Обратите внимание, что в выводе команды `ls` для каталога mydir счетчик ссылок (второй столбец) равен 2. Эта вторая ссылка находится в самом каталоге mydir, обозначается точкой (.) и указывает на текущий каталог. Вот и весь секрет!

Расширенные права доступа также могут использоваться при создании каталогов. Следующий пример (листинг 16.12) демонстрирует создание каталога с липким битом.

Листинг 16.12. Пример mkdir4.c

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

int main (int argc, char ** argv)
{
       mode_t mode = 0777 | S_ISVTX;
```

```
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        umask (0000);
        if (mkdir (argv[1], mode) == -1) {
                fprintf (stderr, "mkdir() error\n");
                return 0;
        }

        return 0;
}
```

Проверяем:

```
$ ./mkdir4 mydir
$ ls -l | grep mydir
drwxrwxrwt 2 nnivanov nnivanov 4096 2011-05-07 10:20 mydir
```

16.5. Удаление каталога: *rmdir()*

Для удаления каталога служит системный вызов `rmdir()`, который объявлен в заголовочном файле unistd.h следующим образом:

```
int rmdir (const char * DIR);
```

При успешном завершении `rmdir()` возвращает 0. В случае ошибки возвращается –1. Аргумент `DIR` — это имя (путь) к каталогу, который следует удалить.

Следующий пример (листинг 16.13) демонстрирует работу системного вызова `rmdir()`.

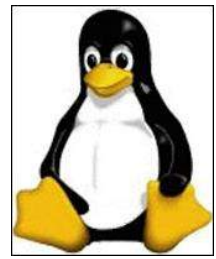
Листинг 16.13. Пример rmdirdemo.c

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

int main (int argc, char ** argv)
{
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (rmdir (argv[1]) == -1) {
                fprintf (stderr, "rmdir() error\n");
                return 1;
        }

        return 0;
}
```

Следует отметить, что системный вызов `rmdir()` удаляет только пустые каталоги. Если каталог не пуст, то `rmdir()` завершится неудачей. Проверим это:

```
$ mkdir mydir
$ touch mydir/myfile
$ ./rmdirdemo mydir
rmdir() error
$ rm mydir/myfile
$ ./rmdirdemo mydir
$ ls mydir
/bin/ls: mydir: No such file or directory
```

ГЛАВА 17



Права доступа

В этой главе будут рассмотрены следующие темы:

- Смена владельца файла при помощи системных вызовов семейства `chown()`.
- Смена прав доступа к файлу при помощи системных вызовов семейства `chmod()`.

17.1. Смена владельца: *chown()*

Для изменения владельца и группы файла предназначены системные вызовы семейства `chown()`, объявленные в unistd.h следующим образом:

```
int chown (const char * FNAME, uid_t USER, gid_t GROUP);
int fchown (int FD, uid_t USER, gid_t GROUP);
int lchown (const char * FNAME, uid_t USER, gid_t GROUP);
```

Все они возвращают 0 при успешном завершении и –1, если произошла ошибка. Системный вызов `chown()` изменяет владельца (на `USER`) и группу (на `GROUP`) файла с именем `FNAME`. `fchown()` делает то же самое, но использует не имя файла, а дескриптор `FD`. Системный вызов `lchown()` аналогичен `chown()`, но при обнаружении ссылки изменяет ее саму, а не файл, на который эта ссылка указывает.

Следует заметить, что если в качестве `USER` указывается –1, то владелец файла не изменяется. Аналогично, если аргумент `GROUP` равен –1, то группа не изменяется.

Здесь мы не сможем привести универсальный пример использования семейства `chown()`, поскольку у обычного пользователя Linux должные полномочия для выполнения этих системных вызовов могут отсутствовать.

17.2. Смена прав доступа: семейство *chmod()*

Для смены прав доступа файла применяются системные вызовы семейства `chmod()`, объявленные в заголовочном файле sys/stat.h следующим образом:

```
int chmod (const char * FNAME, mode_t MODE);
int fchmod (int FD, mode_t MODE);
```

При успешном завершении эти системные вызовы возвращают 0. В случае ошибки возвращается –1. `chmod()` принимает в качестве первого аргумента (`FNAME`) имя файла. Системный вызов `fchmod()` использует вместо имени файла дескриптор `FD`.

Листинг 17.1 содержит пример, демонстрирующий работу системного вызова `chmod()`. Возможно, эта программа покажется вам слишком громоздкой, однако она позволяет проводить различные эксперименты, направленные на выявление скрытых особенностей рассматриваемого системного вызова.

Листинг 17.1. Программа chmoddemo.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>

#define ANSWER_SIZE       256

void add_mode (mode_t * mode, const char * answer, mode_t bit)
{
       if (answer[0] == 'y') {
              *mode |= bit;
              printf ("set\n");
       } else {
              printf ("not set\n");
       }
}

int main (int argc, char ** argv)
{
       char * answer;
       mode_t mode = 0000;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       answer = (char *) malloc (sizeof (char) *
                     ANSWER_SIZE);
       if (answer == NULL) {
              fprintf (stderr, "malloc() error\n");
              return 1;
       }

       printf ("User can read: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IRUSR);
```

```
printf ("User can write: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IWUSR);

printf ("User can execute: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IXUSR);

printf ("Group can read: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IRGRP);

printf ("Group can write: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IWGRP);

printf ("Group can execute: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IXGRP);

printf ("World can read: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IROTH);

printf ("World can write: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IWOTH);

printf ("World can execute: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_IXOTH);

printf ("SUID: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_ISUID);

printf ("SGID: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_ISGID);

printf ("Sticky: ");
scanf ("%s", answer);
add_mode (&mode, answer, S_ISVTX);

if (chmod (argv[1], mode) == -1) {
       fprintf (stderr, "chmod() error\n");
```

```
                return 1;
        }

        free (answer);
        return 0;
}
```

Данная программа последовательно запрашивает у пользователя подтверждение установки каждого бита базовых и расширенных прав доступа. Функция `add_mode()` обрабатывает строку, введенную пользователем. Если строка начинается с символа "y", то к текущему режиму добавляется соответствующий бит прав доступа.

> ***Примечание***
>
> Приведенная программа называет остальных пользователей словом "world" (мир). Это довольно распространенный программистский жаргон, вероятно происходящий от фразеологизма "делать что-либо всем миром".

Вот как работает эта программа:

```
$ touch myfile
$ ls -l myfile
-rw-r--r-- 1 nn nn 0 2011-05-07 11:11 myfile
$ ./chmoddemo myfile
User can read: y
set
User can write: y
set
User can execute: n
not set
Group can read: y
set
Group can write: y
set
Group can execute: n
not set
World can read: y
set
World can write: y
set
World can execute: n
not set
SUID: n
not set
SGID: n
not set
Sticky: n
not set
$ ls -l myfile
-rw-rw-rw- 1 nn nn 0 2011-05-07 11:12 myfile
```

Системный вызов `chmod()` можно также применять к каталогам:

```
$ umask
0022
$ mkdir mydir
$ ls -l | grep mydir
drwxr-xr-x 2 nn nn 4096 2011-05-07 11:12 mydir
$ ./chmoddemo mydir
User can read: y
set
User can write: y
set
User can execute: y
set
Group can read: y
set
Group can write: y
set
Group can execute: y
set
World can read: y
set
World can write: y
set
World can execute: y
set
SUID: n
not set
SGID: n
not set
Sticky: y
set
$ ls -l | grep mydir
drwxrwxrwt 2 nn nn 4096 2011-05-07 11:13 mydir
```

Обратите внимание, что `chmod()` не зависит от текущей маски прав доступа. Рассмотрим теперь аналогичный пример, демонстрирующий работу системного вызова `fchmod()` (листинг 17.2).

Листинг 17.2. Программа fchmoddemo.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
```

```
#define ANSWER_SIZE       256

void add_mode (mode_t * mode, const char * answer, mode_t bit)
{
       if (answer[0] == 'y') {
              *mode |= bit;
              printf ("set\n");
       } else {
              printf ("not set\n");
       }
}

int main (int argc, char ** argv)
{
       int fd;
       char * answer;
       mode_t mode = 0000;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       answer = (char *) malloc (sizeof (char) *
                     ANSWER_SIZE);
       if (answer == NULL) {
              fprintf (stderr, "malloc() error\n");
              return 1;
       }

       printf ("User can read: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IRUSR);

       printf ("User can write: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IWUSR);

       printf ("User can execute: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IXUSR);

       printf ("Group can read: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IRGRP);

       printf ("Group can write: ");
       scanf ("%s", answer);
       add_mode (&mode, answer, S_IWGRP);
```

```
        printf ("Group can execute: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_IXGRP);

        printf ("World can read: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_IROTH);

        printf ("World can write: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_IWOTH);

        printf ("World can execute: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_IXOTH);

        printf ("SUID: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_ISUID);

        printf ("SGID: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_ISGID);

        printf ("Sticky: ");
        scanf ("%s", answer);
        add_mode (&mode, answer, S_ISVTX);

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file\n");
                return 1;
        }

        if (fchmod (fd, mode) == -1) {
                fprintf (stderr, "fchmod() error\n");
                return 1;
        }

        close (fd);
        free (answer);
        return 0;
}
```

Эта программа по принципу своей работы не идентична той, что была представлена в листинге 17.1. Необходимое условие успешной работы данной программы — наличие у файла начальных прав на чтение:

```
$ touch myfile
$ chmod 0000 myfile
$ ls -l myfile
---------- 1 nn nn 0 2011-05-07 11:15 myfile
$ ./fchmoddemo myfile
User can read: y
set
User can write: y
set
User can execute: y
set
Group can read: y
set
Group can write: y
set
Group can execute: y
set
World can read: y
set
World can write: y
set
World can execute: y
set
SUID: n
not set
SGID: n
not set
Sticky: n
not set
Cannot open file
$ ls -l myfile
---------- 1 nn nn 0 2011-05-07 11:15 myfile
```

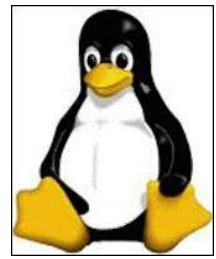
Если теперь присвоить файлу myfile право на чтение для владельца, то программа будет работать корректно:

```
$ chmod u+r myfile
$ ls -l myfile
-r-------- 1 nn nn 0 2011-05-07 11:15 myfile
$ ./fchmoddemo myfile
User can read: y
set
User can write: y
set
User can execute: n
not set
Group can read: y
set
```

```
Group can write: n
not set
Group can execute: y
set
World can read: y
set
World can write: n
not set
World can execute: n
not set
SUID: y
set
SGID: y
set
Sticky: n
not set
$ ls -l myfile
-rwSr-sr-- 1 nn nn 0 2011-05-07 11:15 myfile
```

Обратите внимание, что в последнем выводе программы ls бит SUID показан заглавным символом "S", а бит SGID отображается строчной литерой "s". Это означает, что бит выполнения не установлен для владельца (заглавная "S"), но установлен для группы (строчная "s"). Аналогичным образом программа ls показывает наличие бита выполнения для остальных пользователей, когда данное поле закрывает липкий бит.

ГЛАВА 18



Временные файлы

В этой главе будут рассмотрены следующие темы:

- Использование временных файлов и безопасность приложений.
- Создание временного файла с помощью библиотечной функции `mkstemp()`.
- Закрытие и удаление временного файла.

18.1. Концепция использования временных файлов

Иногда программе приходится работать с большими (или потенциально большими) объемами данных. Чтобы не расходовать драгоценную оперативную память, программист может поместить данные во временный файл.

При этом следует заранее задуматься о последствиях. Наиболее значимым "побочным эффектом" создания временных файлов является то, что вышедшая из-под контроля программа может попросту аварийно завершиться, не удалив временный файл. А это приведет к заполнению файловой системы ненужными данными.

Предусмотрительный программист способен решить эту проблему очень просто. В *главе 16* мы выяснили, что системный вызов `unlink()` можно использовать для открытых файлов, не нарушая целостности операций ввода-вывода. Поэтому сразу после открытия временного файла желательно вызвать `unlink()`, чтобы даже неожиданный аварийный выход из программы мог бы гарантировать удаление данного файла.

Временные файлы приводят также к некоторым проблемам, связанным с безопасностью. Если имя временного файла известно заранее, то злоумышленник может извлечь из этого файла какие-нибудь конфиденциальные данные, с которыми работает программа. Злоумышленник может также "подсунуть" программе собственную версию временного файла. Из-за этого программа может начать вести себя неадекватно.

Если программе нужно обезопасить себя, то весьма желательно каждый раз присваивать временному файлу непредсказуемое имя и адекватные права доступа.

Временные файлы желательно помещать в каталог /tmp. Этот каталог гарантированно присутствует в любой Linux-системе. Обычно /tmp имеет права доступа 0777 и использует липкий бит для защиты пользователей друг от друга:

```
$ ls -l / | grep tmp
drwxrwxrwt  63 root root 12288 2011-05-07 11:47 tmp/
```

Каталог /tmp может, например, являться точкой монтирования для отдельной файловой системы, служащей для хранения временных файлов. Это может быть нежурналируемая высокопроизводительная файловая система или даже RAM-диск (виртуальное блочное устройство в оперативной памяти). Поэтому создание каталога /tmp позволяет "приобщить" вашу программу к общей политике использования временных файлов в конкретной Linux-системе.

18.2. Создание временного файла: *mkstemp()*

Для создания временных файлов предусмотрена специальная библиотечная функция `mkstemp()`, которая объявлена в заголовочном файле stdlib.h следующим образом:

```
int mkstemp (char * TEMPLATE);
```

Эта функция возвращает дескриптор созданного временного файла или –1, если произошла ошибка. Аргумент `TEMPLATE` — это шаблон имени временного файла, который должен заканчиваться последовательностью из шести символов 'X' (XXXXXX). Эта последовательность автоматически заменится произвольным набором символов, а результат будет возвращен по адресу в `TEMPLATE`. После успешного завершения `mkstemp()` указатель `TEMPLATE` будет содержать реальное имя временного файла.

Следует также отметить, что `mkstemp()` открывает файл в режиме для чтения и записи, как если бы мы использовали системный вызов `open()` с флагом `O_RDWR`. Права доступа временного файла устанавливаются в 0600.

> ***Примечание***
>
> Ранние версии `mkstemp()` создавали временный файл с правами доступа 0666.

18.3. Закрытие и удаление временного файла

Теперь мы можем составить краткий алгоритм работы с временными файлами.

1. Временный файл открывается при помощи функции `mkstemp()`, которая добавляет произвольный набор символов в имя файла, делая его труднодоступным для

потенциальных злоумышленников. Сам файл желательно располагать в дереве каталогов /tmp.

2. Сразу после проверки дескриптора вызывается `unlink()`. При этом дескриптор временного файла остается работоспособным до тех пор, пока все процессы, разделяющие этот дескриптор, не запросят у ядра закрытие файла.
3. Запись во временный файл осуществляется обычным образом. Иными словами, `mkstemp()` при успешном завершении возвращает полноценный файловый дескриптор.
4. Перед чтением временного файла необходимо сместить назад текущую позицию ввода-вывода. Это обычно делается при помощи системного вызова `lseek()`.
5. Чтение из файла также осуществляется обычным образом.
6. После всех манипуляций файл закрывается при помощи системного вызова `close()`. Если процесс завершается аварийно, не вызвав `close()` непосредственно, то в ядро все равно поступает запрос на закрытие файла. Поэтому, однажды вызвав `unlink()`, программист может более не заботиться об удалении временного файла.

> ***Примечание***
>
> Следует отметить, что `mkstemp()` открывает файл с флагом `O_EXCL`. Это означает, что функция `mkstemp()` из текущей или из другой программы не сможет "случайно" открыть временный файл с таким же именем.

Рассмотрим теперь небольшой пример использования временного файла (листинг 18.1).

Листинг 18.1. Пример mkstempdemo.c

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

int main (void)
{
       int fd;
       char tfile[] = "/tmp/mydemo-XXXXXX";
       char tmsg[] = "Hello World\n";
       char ch;

       fd = mkstemp (tfile);
       if (fd == -1) {
              fprintf (stderr, "mkstemp() error\n");
              return 1;
       }

       printf ("Filename: %s\n", tfile);
```

```
        if (write (fd, tmsg, strlen (tmsg)) == -1) {
                fprintf (stderr, "write() error\n");
                return 1;
        }

        if (unlink (tfile) == -1) {
                fprintf (stderr, "unlink() error\n");
                return 1;
        }

        if (lseek (fd, 0, SEEK_SET) == -1) {
                fprintf (stderr, "lseek() error\n");
                return 1;
        }

        while (read (fd, &ch, 1) > 0) {
                write (1, &ch, 1);
        }

        close (fd);
        return 0;
}
```

При выполнении получилось примерно следующее:

```
$ ./mkstempdemo
Filename: /tmp/mydemo-RydBF6
Hello World
```

Давайте теперь вспомним простую адресную книгу, которую мы рассматривали в *главе 8* (листинг 8.6). Тогда сообщалось, что поддержка удаления записей из адресной книги будет реализована позже. Настала пора сделать это.

Итак, программа не требует существенной переделки. Единственное, что нам нужно сделать, — это добавить функцию удаления записи (`abook_delete()`, например) и реализовать обработку аргумента "delete" командной строки, который инициирует диалог удаления учетной записи.

Как вы, наверное, догадались, мы будем удалять запись из адресной книги с использованием временного файла. Системный вызов `lseek()` позволяет произвольно перемещаться по файлу. Однако удаление блоков данных из файла — достаточно сложная операция. В *главе 22* будет рассматриваться системный вызов `mmap()`, который упрощает подобные задачи. Но сейчас мы будем действовать согласно приведенному далее алгоритму.

1. Просматриваем в цикле каждую запись в файле адресной книги.
2. Каждая запись, кроме той, что следует удалить, переписывается во временный файл.

3. Закрываем исходную адресную книгу и открываем вновь, но уже с флагами `O_WRONLY` (только запись) и `O_TRUNC` (очистить).
4. Копируем информацию из временного файла в чистую адресную книгу.

В листинге 18.2 представлен исходный код адресной книги с возможностью удаления записей.

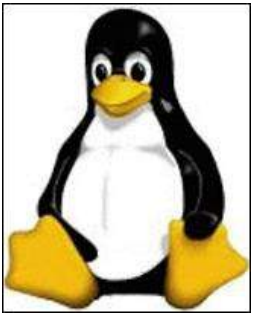
Листинг 18.2. Адресная книга abook1.c

```
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define ABOOK_FNAME       "abook"
#define NAME_LENGTH       50
#define PHONE_LENGTH       30
#define EMAIL_LENGTH       30

struct iovec ab_entry[3];
char name_buffer [NAME_LENGTH];
char phone_buffer [PHONE_LENGTH];
char email_buffer [EMAIL_LENGTH];

void abook_failed (int retcode)
{
       fprintf (stderr, "Cannot open address book\n");
       exit (retcode);
}

void abook_delete (void)
{
       int fd, tfd;
       char find_buffer [NAME_LENGTH];
       char tfile[] = "/tmp/abook-XXXXXX";
       char ch;

       tfd = mkstemp (tfile);
       if (tfd == -1) {
              fprintf (stderr, "mkstemp() error\n");
              exit (1);
       }

       unlink (tfile);
       printf ("Name: ");
```

```
        scanf ("%s", find_buffer);
        fd = open (ABOOK_FNAME, O_RDONLY);
        if (fd == -1) abook_failed (1);

        while (readv (fd, ab_entry, 3) > 0)
        {
                if (!strcmp (find_buffer,
                             ab_entry[0].iov_base)) {
                        printf ("Delete: %s\n",
                                     ab_entry[0].iov_base);
                        continue;
                }
                writev (tfd, ab_entry, 3);
        }

        lseek (tfd, 0, SEEK_SET);
        close (fd);

        fd = open (ABOOK_FNAME, O_WRONLY | O_TRUNC);
        if (fd == -1) abook_failed (1);

        while (read (tfd, &ch, 1) > 0) write (fd, &ch, 1);

        close (fd);
        close (tfd);
}

void abook_add (void)
{
        int fd;
        printf ("Name: ");
        scanf ("%s", ab_entry[0].iov_base);
        printf ("Phone number: ");
        scanf ("%s", ab_entry[1].iov_base);
        printf ("E-mail: ");
        scanf ("%s", ab_entry[2].iov_base);

        fd = open (ABOOK_FNAME, O_WRONLY | O_CREAT | O_APPEND,
                        S_IRUSR | S_IWUSR | S_IRGRP);
        if (fd == -1) abook_failed (1);

        if (writev (fd, ab_entry, 3) <= 0) {
                fprintf (stderr, "Cannot write to address book\n");
                exit (1);
        }
        close (fd);
}
```

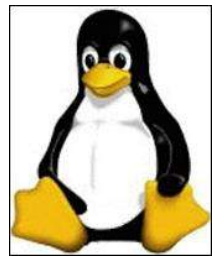
```
void abook_find (void)
{
       int fd;
       char find_buffer [NAME_LENGTH];
       printf ("Name: ");
       scanf ("%s", find_buffer);
       fd = open (ABOOK_FNAME, O_RDONLY);
       if (fd == -1) abook_failed (1);

       while (readv (fd, ab_entry, 3) > 0)
       {
              if (!strcmp (find_buffer, ab_entry[0].iov_base))
              {
                     printf ("Phone: %s\n",
                            ab_entry[1].iov_base);
                     printf ("E-mail: %s\n",
                            ab_entry[2].iov_base);
                     goto close;
              }
       }

       printf ("Name '%s' hasn't found\n", find_buffer);
close:
       close (fd);
}

int main (int argc, char ** argv)
{
       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       ab_entry[0].iov_base = name_buffer;
       ab_entry[0].iov_len = NAME_LENGTH;
       ab_entry[1].iov_base = phone_buffer;
       ab_entry[1].iov_len = PHONE_LENGTH;
       ab_entry[2].iov_base = email_buffer;
       ab_entry[2].iov_len = EMAIL_LENGTH;

       if (!strcmp (argv[1], "add")) {
              abook_add ();
       } else if (!strcmp (argv[1], "find")) {
              abook_find ();
       } else if (!strcmp (argv[1], "delete")) {
              abook_delete ();
```

```
        } else {
                fprintf (stderr, "%s: unknown command\n"
                "Usage: abook { add , find, delete }\n",
                                argv[1]);
                return 1;
        }
        return 0;
}
```



ЧАСТЬ V

Межпроцессное взаимодействие



ГЛАВА 19

Обзор методов межпроцессного взаимодействия в Linux

В этой главе будут рассмотрены следующие темы:

- Основные понятия и виды межпроцессного взаимодействия в Linux.
- Характеристика локальных методов межпроцессного взаимодействия.
- Характеристика методов удаленного межпроцессного взаимодействия.

19.1. Общие сведения о межпроцессном взаимодействии в Linux

В основе межпроцессного взаимодействия лежит обмен данными между работающими процессами. Межпроцессное взаимодействие обозначают также аббревиатурой IPC (InterProcess Communication).

Не существует единого универсального метода взаимодействия процессов. Выбор того или иного способа взаимодействия зависит от поставленных задач.

Межпроцессное взаимодействие в Linux можно классифицировать по трем критериям.

- По *широте охвата* взаимодействие бывает *локальным* и *удаленным*. Локальное взаимодействие подразделяется на *родственное* и *неродственное*.
- По *направлению передачи данных* межпроцессное взаимодействие бывает *однонаправленным* и *двунаправленным*.
- По *характеру доступа* взаимодействие бывает *открытым* и *закрытым*.

> ***Примечание***
>
> Существует также *групповое взаимодействие* между процессами, работающими под управлением одного терминала. Это достаточно обширная тема, выходящая за рамки данной книги.

Локальное взаимодействие отвечает за обмен данными между процессами, которые работают в одной Linux-системе. Если обмен данными осуществляется между ро-

дительским и дочерним процессами, то такое взаимодействие называется родственным.

Удаленное взаимодействие — это обмен данными между двумя процессами, которые работают в разных системах. Данный тип взаимодействия осуществляется по сети. Например, путешествуя по Интернету, вы постоянно инициируете межпроцессное взаимодействие между вашим браузером и процессами-серверами, которые работают на других компьютерах.

Однонаправленное межпроцессное взаимодействие характеризуется тем, что один процесс является отправителем данных, другой — приемником этих данных. Двунаправленное взаимодействие позволяет процессам общаться на равных. Но в этом случае процессы должны "договориться" об очередности и синхронизации приема и передачи информации.

Если взаимодействие осуществляется только между двумя процессами, то такое взаимодействие называется закрытым. Если какой-нибудь другой процесс может присоединиться к обмену данными, то такое взаимодействие называется *открытым*.

Существование различных способов межпроцессного взаимодействия обусловлено тем, что при их реализации возникают две проблемы:

1. *Проблема синхронизации*. Уже неоднократно говорилось, что процессы работают независимо друг от друга. В ходе обмена данными между процессами должна учитываться эта "независимость".
2. *Проблема безопасности*. Если один процесс направляет данные другому процессу, то возникает опасность того, что потенциальный злоумышленник может перехватить эти данные. Проблема усложняется еще и тем, что злоумышленник может "подсунуть" процессу-приемнику собственные данные, если последние не защищены. Чем шире зона охвата межпроцессного взаимодействия, тем более уязвимым (с точки зрения безопасности) оно является.

19.2. Локальные методы межпроцессного взаимодействия

Простейший способ межпроцессного взаимодействия — локальное родственное двунаправленное закрытое взаимодействие родителя и потомка. Родительский процесс может передавать дочернему некоторые данные посредством аргументов через одну из функций семейства `exec()`. В свою очередь дочерний процесс при нормальном завершении сообщает родителю свой код возврата. Рассмотрим пример, в котором осуществляется такое взаимодействие.

Сначала реализуем программу (листинг 19.1), которая будет работать в дочернем процессе. Данная программа принимает единственный аргумент — календарный год. Если год является високосным, то программа завершается с кодом возврата 1. Если год не является високосным, то кодом возврата будет 0.

Листинг 19.1. Программа kinsfolk-child1.c

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

int main (int argc, char ** argv)
{
        int year;
        if (argc < 2) {
                fprintf (stderr, "child: too few arguments\n");
                return 2;
        }

        year = atoi (argv[1]);

        if (year <= 0)
                return 2;

        if ( ((year%4 == 0) && (year%100 != 0)) ||
                        (year%400 == 0) )
                return 1;
        else
                return 0;

        return 0;
}
```

Примечание

Как известно, стандартной единицей измерения времени в физике является секунда. В одном часе 3600 секунд, а в сутках 24 часа. Мы также знаем, что Земля обращается вокруг своей оси за сутки. Оборот Земли вокруг Солнца мы называем годом, в котором 365 суток. Но в эту простую схему вкралась одна неточность. Если разделить точное время одного оборота Земли вокруг Солнца на 365, то выяснится, что календарные сутки (в которых 24 часа) короче астрономических почти на 4 минуты. Григорианский календарь, принятый в большинстве стран мира, решает эту проблему введением високосного года. Один раз в 4 года разница между календарным и астрономическим годом становится равной примерно 24 часам. И эти 24 часа просто "набрасываются" на календарь (29 февраля). Но даже при такой схеме погрешность сохраняется. Было подсчитано, что можно почти полностью ликвидировать данную неточность, если один раз в 100 лет отменять високосный год, но за исключением тех случаев, когда год кратен 400.

Приведенная в листинге 19.1 программа может, например, участвовать в родственном межпроцессном взаимодействии с командной оболочкой. Вот как это происходит:

```
$ gcc -o kinsfolk-child1 kinsfolk-child1.c
$ ./kinsfolk-child1 2007
$ echo $?
0
```

```
$ ./kinsfolk-child1 2008
$ echo $?
1
$ ./kinsfolk-child1 2300
$ echo $?
0
$ ./kinsfolk-child1 2400
$ echo $?
1
```

Показанное взаимодействие с командной оболочкой не позволяет увидеть механизм работы родителя. Поэтому попробуем теперь реализовать собственного "родителя" (листинг 19.2).

Листинг 19.2. Программа kinsfolk-parent1.c

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

int main (int argc, char ** argv)
{
       pid_t cpid;
       int status;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       cpid = fork();
       if (!cpid) {
              execl ("./kinsfolk-child1", "Child", argv[1], NULL);
              fprintf (stderr, "execl() error\n");
              return 1;
       }

       waitpid (cpid, &status, 0);

       if (WIFEXITED (status)) {
              if (WEXITSTATUS (status) == 1)
                     printf ("%s: leap year\n", argv[1]);
              else if (WEXITSTATUS (status) == 0)
                     printf ("%s: not leap year\n", argv[1]);
              else {
                     fprintf (stderr, "error: unexpected "
                                   "return code\n");
```

```
                    return 1;
                }
        }

        return 0;
}
```

Вот что получилось в результате:

```
$ gcc -o kinsfolk-parent1 kinsfolk-parent1.c
$ ./kinsfolk-parent1 2000
2000: leap year
$ ./kinsfolk-parent1 2007
2007: not leap year
```

Приведенный пример локального межпроцессного взаимодействия описывает примитивную и грубую (для нашей задачи) методику обмена данными между процессами. Далее в этой книге будут описываться более тонкие подходы к локальному межпроцессному взаимодействию. Приведем краткую характеристику этих методов.

- *Сигналы* — это атомарные сообщения, которые один процесс может посылать другому процессу или самому себе. Процесс-приемник регистрирует факт получения сигнала и его тип (другую информацию сигналы не передают). В зависимости от типа сигнала процесс-получатель может завершиться, обработать сигнал или просто проигнорировать его. Уникальность межпроцессного взаимодействия посредством сигналов состоит в том, что процесс-получатель немедленно реагирует на факт отправки сигнала. Подробно о сигналах будет рассказано в *главе 20*.

- *Общая память*. Каждый процесс в Linux работает в собственном адресном пространстве. Но при помощи специальных механизмов можно выделить блок (сегмент) общей памяти и открыть к нему доступ для других процессов. Особенность данного способа взаимодействия — быстрая скорость обмена данными. Но здесь от программиста требуется осторожность. Нужно "научить" процессы работать с памятью поочередно. Наиболее эффективное решение данной задачи — использование семафоров. *Семафоры* — это специальные механизмы, которые позволяют контролировать доступ к общим ресурсам (совместно используемая память, файлы и др.). Подробно об общей памяти и о семафорах будет рассказано в *главе 22*.

- *Общие файлы*. В Linux есть системный вызов `mmap()`, который позволяет отобразить блок файла в памяти. Если два процесса вызовут `mmap()` для одного и того же файла, то налицо еще один способ межпроцессного взаимодействия. Стандартные механизмы файлового ввода-вывода также реализуют взаимодействие процессов через общие файлы. Однако проблемы произвольного доступа и синхронизации данных делают использование `mmap()` более выгодным. Дело в том, что в случае `mmap()` работа с файлом сводится к операциям с блоком памяти, а вызов-побратим `msync()` позволяет гарантированно синхронизировать измененные данные. Подробно об общих файлах будет рассказано в *главе 23*.

- *Каналы* — очень простой и надежный способ родственного межпроцессного взаимодействия. Условно канал можно представить в виде однонаправленного потока информации. Интерфейс канала — это два файловых дескриптора (концы канала). Через первый дескриптор осуществляется запись в канал, через второй — чтение. Важной особенностью каналов является то, что данный способ межпроцессного взаимодействия не требует от процессов контроля очередности доступа к данным. Подробно о каналах будет рассказано в *главе 24*.
- *FIFO*. Именованные каналы FIFO (First In, First Out) во многом напоминают обычные каналы, но позволяют осуществлять обмен данными не только между родственными процессами. Вы уже знаете, что FIFO — это один из семи типов файлов в Linux. Подробно о FIFO будет рассказано в *главе 25*.

19.3. Удаленное межпроцессное взаимодействие

Удаленное межпроцессное взаимодействие осуществляется через *сеть*. Очень важно понимать, что обмен данными между компьютерами не всегда носит характер межпроцессного взаимодействия.

Рассмотрим наглядный пример. Предположим, что некоторый компьютер под управлением Linux соединен через USB со смартфоном, также работающим под управлением Linux. Предположим также, что пользователь компьютера монтирует файловую систему flash-памяти смартфона и закачивает оттуда какие-то данные. В приведенном примере межпроцессного взаимодействия не происходит, хотя обе системы имеют физический доступ к одним и тем же данным.

Рассмотрим теперь другой пример. Предположим, что на компьютере установлен Web-сервер. Если пользователь смартфона при помощи встроенного браузера обменивается данными с этим сервером (закачивает картинки, например), то такой обмен можно смело назвать удаленным межпроцессным взаимодействием.

Оба примера осуществляют одну и ту же операцию — перемещение данных с одного компьютера на другой. Однако в первом случае реализуется просто физический доступ к общему накопительному устройству, а во втором случае передача данных осуществляется через сеть.

> ***Примечание***
>
> Элементами сети могут быть не только компьютеры. Типичным примером этому могут послужить специальные сетевые принтеры, аппаратные маршрутизаторы и т. п. Поэтому элементы сети принято называть *узлами* (hosts).

Удаленное межпроцессное взаимодействие осуществляется посредством сокетов. *Сокет* — это комплексное понятие, которое условно можно назвать "точкой соединения процессов". Полное описание сокетов и методов работы с ними требует написания отдельной книги. Здесь будут рассмотрены лишь основы работы с сокетами.

Следует отметить, что существуют различные типы сокетов. В рамках данной книги мы будем рассматривать два типа:

- *Unix-сокеты* для локального взаимодействия процессов;
- *Интернет-сокеты* для удаленного взаимодействия процессов.

В программах сокеты фигурируют в виде файловых дескрипторов, над которыми (во многих случаях) можно осуществлять обычные операции чтения-записи (`read()`, `write()` и т. д.). Но набор действий, подготавливающих файловый дескриптор к использованию, несколько сложнее, чем в случае с обычными файлами.

Обычные методы взаимодействия рассматривают процессы по принципу "источник — приемник". При взаимодействии посредством сокетов процессы рассматриваются по схеме "клиент — сервер".

Процесс-сервер устанавливает "правила общения" и предлагает всем желающим межпроцессное взаимодействие. Обычно работа с сокетами на стороне сервера осуществляется при помощи следующих операций:

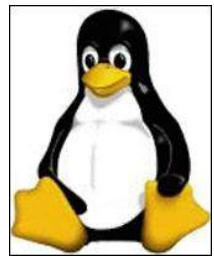
- *Создание сокета*. На этом этапе процесс-сервер определяет тип сокета и способ взаимодействия.
- *Назначение сокету адреса*. В зависимости от типа сокета адресом может быть файл (для локальных сокетов) или нечто другое, не имеющее представления в файловой системе (для удаленного взаимодействия).
- *Прослушивание сокета*. Эта операция официально предлагает всем желающим начать межпроцессное взаимодействие.
- *Принятие запроса*. Как только на прослушиваемый сокет поступил запрос от клиента, сервер выходит из режима ожидания (прослушивания) и начинает обрабатывать этот запрос. Операция приема запроса создает на стороне сервера новый сокет, через который будет осуществляться передача данных.
- *Закрытие (удаление) сокета*. Любая из сторон взаимодействия может инициировать разрыв связи и прекращение взаимодействия. Для этого используется хорошо знакомый нам системный вызов `close()`.

Работа с сокетами на стороне клиента осуществляется посредством следующих операций:

- *Создание сокета*. Клиентский процесс создает сокет нужного типа, чтобы осуществлять через него взаимодействие с сервером.
- *Соединение с сервером*. Эта операция посылает на сервер запрос на соединение. Прием запроса создает на серверной стороне дополнительный сокет, связанный с текущим сокетом клиента. С этого момента сервер и клиент могут обмениваться информацией. Здесь важно понимать, что сервер осуществляет прием запроса через один сокет, а передачу данных — через другой. Но клиент использует один и тот же сокет для подключения и для передачи данных.
- *Отсоединение от сервера*. Как только клиент вызывает `close()`, соединение с сервером разрывается.

Следует отметить, что приведенный набор операций может быть совсем иным. Это зависит от выбранного способа взаимодействия сокетов. Подробно о сокетах будет рассказано в *главе 25*.

ГЛАВА 20



Сигналы

В этой главе будут рассмотрены следующие темы:

- Понятие сигнала.
- Отправка сигнала.
- Перехватывание и обработка сигнала.
- Реализация межпроцессного взаимодействия при помощи сигналов.

20.1. Понятие сигнала в Linux

Уже отмечалось, что сигнал — это сообщение, которое один процесс-отправитель посылает другому процессу или самому себе. В распоряжении процессов находится стандартный набор сигналов, заранее определенных ядром Linux, который приведен в *приложении 3*.

Каждый сигнал имеет свой уникальный номер, а также символическую константу, соответствующую этому номеру. Соответствие между номерами сигналов и символическими константами определено в заголовочном файле bits/signum.h.

> ***Примечание***
>
> Не следует включать в обычные программы заголовочный файл bits/signum.h. Используйте вместо этого файл signal.h.

Процесс-получатель может отреагировать на сигнал одним из следующих трех способов:

- *Принятие сигнала*. Обычно это приводит к немедленному завершению процесса. Принятие сигнала означает, что процесс не подготовился к получению данного типа сигнала и принимает "политику сигналов по умолчанию".
- *Игнорирование сигнала*. Процесс может установить для себя политику игнорирования определенных сигналов. Это означает, что при поступлении сигнала процесс будет продолжать свою работу, как если бы этого сигнала вообще не было. Следует отметить, что не все сигналы можно проигнорировать.

- *Перехватывание и обработка сигнала*. Процесс может задать собственное поведение в отношении конкретного сигнала. В этом случае поступление сигнала мгновенно инициирует вызов функции-обработчика, после завершения которой процесс продолжает выполнение с прежнего места. Нужно помнить, что не каждый сигнал можно перехватить и обработать.

Для отправки сигнала используется команда `kill`. В упрощенном виде синтаксис этой команды можно представить следующим образом:

```
$ kill -s SIGNAL PID
```

Здесь `SIGNAL` — это символическая константа посылаемого сигнала, `PID` — идентификатор процесса-получателя сигнала. Вот пример использования команды `kill`:

```
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5851 pts/2    00:00:00 ps
$ yes > /dev/null &
[1] 5852
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5852 pts/2    00:00:04 yes
 5853 pts/2    00:00:00 ps
$ kill -s SIGINT 5852
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5854 pts/2    00:00:00 ps
[1]+  Interrupt               yes >/dev/null
```

В приведенном примере процессу с идентификатором 5852 отсылается сигнал `SIGINT` (прерывание). Это тот же самый сигнал, который посылается процессу, когда пользователь нажимает комбинацию клавиш <Ctrl>+<C>. Рассмотрим еще один пример.

```
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5857 pts/2    00:00:00 ps
$ yes > /dev/null &
[1] 5858
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5858 pts/2    00:00:01 yes
 5859 pts/2    00:00:00 ps
$ kill -s SIGSEGV 5858
```

```
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 5860 pts/2    00:00:00 ps
[1]+  Segmentation fault      yes >/dev/null
```

Обратите внимание, что по завершении процесса было выведено сообщение "Segmentation fault" (нарушение сегментации). На самом деле ошибки сегментации не было. Но в обычном случае, если процесс пытается получить доступ к "чужой" памяти, ядро немедленно посылает этому процессу сигнал `SIGSEGV`, который, кстати, нельзя перехватить или проигнорировать.

Примечание

Ядро Linux также может быть источником сигналов. Обычно это те сигналы, которые не поддаются обработке и приводят к немедленному завершению процесса.

20.2. Отправка сигнала: *kill()*

Для отправки сигнала предусмотрен системный вызов `kill()`, который объявлен в заголовочном файле signal.h следующим образом:

```
int kill (pid_t PID, int SIGNAL);
```

Этот системный вызов посылает процессу с идентификатором `PID` сигнал `SIGNAL`. Возвращаемое значение — 0 (при успешном завершении) или –1 (в случае ошибки).

Листинг 20.1 иллюстрирует пример использования системного вызова `kill()`.

Листинг 20.1. Программа kill1.c

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

int main (int argc, char ** argv)
{
       pid_t dpid;
       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       dpid = atoi (argv[1]);

       if (kill (dpid, SIGKILL) == -1) {
              fprintf (stderr, "Cannot send signal\n");
              return 1;
       }

       return 0;
}
```

Данная программа посылает процессу сигнал SIGKILL, который приводит к немедленному завершению процесса. Вот как работает данная программа:

```
$ gcc -o kill1 kill1.c
$ yes > /dev/null &
[1] 6032
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 6032 pts/2    00:00:01 yes
 6033 pts/2    00:00:00 ps
$ ./kill1 6032
$ ps
  PID TTY          TIME CMD
 5481 pts/2    00:00:00 bash
 6035 pts/2    00:00:00 ps
[1]+  Killed                  yes >/dev/null
```

Системный вызов kill() позволяет также посылать сигналы целым группам процессов. Но эта тема выходит за рамки данной книги.

Если в аргументе PID системного вызова kill() указать текущий идентификатор, то процесс пошлет сигнал сам себе. Следующий короткий пример демонстрирует это (листинг 20.2).

Листинг 20.2. Пример kill2.c

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

int main (void)
{
       pid_t dpid = getpid ();

       if (kill (dpid, SIGABRT) == -1) {
              fprintf (stderr, "Cannot send signal\n");
              return 1;
       }

       return 0;
}
```

Проверяем:

```
$ gcc -o kill2 kill2.c
$ ./kill2
Aborted
```

В данном случае процесс послал сам себе сигнал SIGABRT, что привело к немедленному аварийному завершению программы.

20.3. Обработка сигнала: *sigaction()*

Системный вызов sigaction() позволяет задавать поведение процесса по отношению к конкретным сигналам. Он объявлен в заголовочном файле signal.h следующим образом:

```
int sigaction (int SIGNAL, const struct sigaction * ACTION,
                      struct sigaction * OLDACTION);
```

Данный системный вызов устанавливает политику реагирования процесса на сигнал SIGNAL. Политика определяется указателем ACTION на структуру типа sigaction. При удачном завершении функции по адресу OLDACTION заносится прежняя политика в отношении сигнала. Если эта информация вам не нужна, то в качестве третьего аргумента можно просто указать NULL. При успешном завершении sigaction() возвращает 0. В случае ошибки возвращается –1.

Структура sigaction содержит различные поля. Нас интересуют только три:

- sa_handler — адрес функции-обработчика сигнала. Если не вдаваться в подробности, то можно сказать, что это обычная функция без возвращаемого значения и принимающая один целый аргумент;
- sa_flags — набор флагов. Для простой обработки сигнала достаточно обнулить это поле;
- sa_mask — это так называемая *маска сигналов*. Данное поле может содержать список сигналов, которые будут заблокированы во время работы функции-обработчика. Чтобы не "возиться" с полем sa_mask, его можно просто обнулить при помощи функции sigemptyset().

Рассмотрим теперь пример перехватывания и обработки сигнала SIGINT, который обычно посылается процессу, когда пользователь нажимает комбинацию клавиш <Ctrl>+<C>. Программа представлена в листинге 20.3.

Листинг 20.3. Программа sigaction1.c

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

void sig_handler (int snum)
{
       fprintf (stderr, "signal...\n");
}

int main (void)
{
       struct sigaction act;
```

```
        sigemptyset (&act.sa_mask);
        act.sa_handler = &sig_handler;

        act.sa_flags = 0;

        if (sigaction (SIGINT, &act, NULL) == -1) {
                fprintf (stderr, "sigaction() error\n");
                return 1;
        }

        while (1);
        return 0;
}
```

Просто запустите программу и нажимайте комбинацию клавиш <Ctrl>+<C> до тех пор, пока не надоест:

```
$ gcc -o sigaction1 sigaction1.c
$ ./sigaction1
signal...
signal...
signal...
signal...
```

Обратите внимание, что каждый раз при нажатии клавиш <Ctrl>+<C> процесс реагирует мгновенно. Сигналы работают в Linux как прерывания. Однако при использовании сигналов всегда следует помнить, что на момент получения сигнала программа может выполнять что-то важное.

Чтобы завершить программу, просто откройте другой терминал и "убейте" соответствующий процесс при помощи команды `kill`:

```
$ ps -e | grep sigaction1
 4883 pts/1    00:44:27 sigaction1
$ kill 4883
```

Если команде `kill` явно не указать конкретный сигнал, то по умолчанию устанавливается `SIGKILL`. Обратите также внимание, что если из другого терминала посылать процессу `sigaction1` сигнал `SIGINT`, то программа будет выводить сообщения "signal..." до тех пор, пока мы не отправим какой-нибудь другой сигнал.

20.4. Сигналы и многозадачность

Итак, мы выяснили, что при получении сигнала процесс немедленно оставляет все свои "дела" и приступает к обработке этого сигнала. Если при помощи `sigaction()` сигнал перехватывается, то вызывается функция-обработчик. Но что будет, если во время выполнения функции-обработчика процесс получает еще один сигнал? В этом случае программа может начать вести себя неадекватно.

Один из способов решения этой проблемы — установка маски сигналов (поле sa_mask структуры sigaction). Но это не будет полноценным разрешением проблемы, поскольку некоторые сигналы немаскируемые.

Единственный действенный способ избежать описанной выше ситуации — включить в обработчик сигнала минимальное число инструкций. Лучше, если это будет одна инструкция, которая просто устанавливает флаг, регистрирующий получение сигнала. Программа может при необходимости (или периодически) проверять значение этого флага и реагировать соответствующим образом.

Следует отметить, что даже одна инструкция присваивания значения флагу может быть прервана. Например, если на программу "обрушится" сплошной поток сигналов, то такая ситуация вполне может стать реальностью. Чтобы избежать подобных случаев, применяется специальный тип данных sig_atomic_t. Это обычное целое число. Но ядро Linux гарантирует, что математические операции над таким числом являются атомарными и не могут быть прерваны.

В листинге 20.4 представлена переработанная версия предыдущей программы (листинг 20.3), реализующая правильный подход к реализации обработчика сигнала.

Листинг 20.4. Программа sigaction2.c

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

sig_atomic_t sig_occured = 0;

void sig_handler (int snum)
{
       sig_occured = 1;
}

int main (void)
{
       struct sigaction act;
       sigemptyset (&act.sa_mask);
       act.sa_handler = &sig_handler;
       act.sa_flags = 0;

       if (sigaction (SIGINT, &act, NULL) == -1) {
              fprintf (stderr, "sigaction() error\n");
              return 1;
       }

       while (1) {
              if (sig_occured) {
                     fprintf (stderr, "signal...\n");
```

```
                        sig_occured = 0;
                }
        }

        return 0;
}
```

Рассмотрим теперь пример межпроцессного взаимодействия на основе сигналов. Для этого переделаем программу kinsfolk1 из *главы 19* (листинги 19.1 и 19.2).

Сначала реализуем "потомка" (листинг 20.5), который будет в зависимости от переданных данных посылать родительскому процессу сигнал `SIGUSR1`, если год високосный, и `SIGUSR2`, если не високосный.

Листинг 20.5. Программа-потомок kinsfolk-child2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main (int argc, char ** argv)
{
        int year;
        if (argc < 2) {
                fprintf (stderr, "child: too few arguments\n");
                return 2;
        }

        year = atoi (argv[1]);

        if (year <= 0)
                return 2;

        if ( ((year%4 == 0) && (year%100 != 0)) ||
                        (year%400 == 0) )
                kill (getppid (), SIGUSR1);
        else
                kill (getppid (), SIGUSR2);

        return 0;
}
```

Теперь реализуем "родителя" (листинг 20.6).

Листинг 20.6. Программа-родитель kinsfolk-parent2.c

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

```
#include <sys/types.h>
#include <wait.h>
#include <signal.h>

/* 0 — no signal, 1 — SIGUSR1, 2 — SIGUSR2 */
sig_atomic_t sig_status = 0;

void handle_usr1 (int s_num)
{
       sig_status = 1;
}

void handle_usr2 (int s_num)
{
       sig_status = 2;
}

int main (int argc, char ** argv)
{
       struct sigaction act_usr1, act_usr2;
       sigemptyset (&act_usr1.sa_mask);
       sigemptyset (&act_usr2.sa_mask);
       act_usr1.sa_flags = 0;
       act_usr2.sa_flags = 0;
       act_usr1.sa_handler = &handle_usr1;
       act_usr2.sa_handler = &handle_usr2;

       if (sigaction (SIGUSR1, &act_usr1, NULL) == -1) {
              fprintf (stderr, "sigaction (act_usr1) error\n");
              return 1;
       }

       if (sigaction (SIGUSR2, &act_usr2, NULL) == -1) {
              fprintf (stderr, "sigaction (act_usr2) error\n");
              return 1;
       }

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       if (!fork()) {
              execl ("./kinsfolk-child2", "Child", argv[1], NULL);
              fprintf (stderr, "execl() error\n");
              return 1;
       }
```

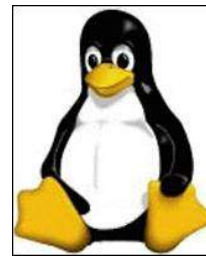
```
        while (1) {
                if (sig_status == 1) {
                        printf ("%s: leap year\n", argv[1]);
                        return 0;
                }

                if (sig_status == 2) {
                        printf ("%s: not leap year\n", argv[1]);
                        return 0;
                }
        }

        return 0;
}
```

20.5. Получение дополнительной информации

В этой главе были рассмотрены лишь основные методы межпроцессного взаимодействия с использованием сигналов. Дополнительную информацию можно получить на следующих man-страницах:

- man 2 kill;
- man 2 sigaction;
- man 2 signal;
- man 2 sigpending;
- man 2 sigprocmask;
- man 2 sigsuspend;
- man 2 sigwaitinfo.

ГЛАВА 21



Использование общей памяти

В этой главе будут рассмотрены следующие темы:

- Выделение совместно используемой (общей) памяти.
- Активация и отключение совместного доступа к сегментам памяти.
- Контроль над общей памятью.
- Реализация семафоров.
- Управление семафорами.

21.1. Выделение памяти: *shmget()*

Реализация межпроцессного взаимодействия посредством совместно используемой памяти осуществляется очень просто. Сначала один из процессов выделяет некоторый объем памяти, который называется сегментом. К общему сегменту памяти привязаны два числа:

- *Идентификатор сегмента* — служит для доступа к сегменту внутри процесса.
- *Ключ сегмента* — идентифицирует сегмент для других процессов. Ключ может быть выделен автоматически во время создания сегмента. Процессы могут также заранее договориться о применении определенного статического ключа.

После выделения совместно используемого сегмента другие процессы могут задействовать его. Эта операция называется *подключением сегмента*, о ней речь пойдет в следующем разделе.

Выделение общей памяти осуществляется при помощи системного вызова `shmget()`, который объявлен в заголовочном файле sys/shm.h следующим образом:

```
int shmget (key_t KEY, size_t SIZE, int FLAGS);
```

Этот системный вызов возвращает идентификатор сегмента или –1, если произошла ошибка. Рассмотрим по порядку аргументы `shmget()`.

- KEY — это ключ сегмента. Если взаимодействующие программы не договорились заранее о применении статического ключа, то в данном поле можно указать константу IPC_PRIVATE, которая инициирует динамическое выделение ключа.
- SIZE — размер сегмента. Здесь следует учитывать, что данный аргумент является запрашиваемым числом байтов для сегмента. Реально выделенное число байтов может отличаться от SIZE.
- Аргумент FLAGS определяет права доступа к совместно используемому сегменту, а также дополнительные флаги: IPC_CREAT (создать), IPC_EXCL ("выругаться", если сегмент уже создан). Права доступа объединяются с флагами при помощи операций побитовой дизъюнкции.

Процесс может получить доступ к созданному сегменту двумя способами:

1. Использовать идентификатор сегмента, полученный от другого процесса.
2. Вызвать shmget(), применяя для этого заранее известный ключ.

Программа ipcs позволяет, кроме прочего, получить информацию о совместно используемых сегментах памяти и их идентификаторах.

21.2. Активизация совместного доступа: *shmat()*

Для работы с общим сегментом памяти нужно, чтобы каждый из взаимодействующих процессов обратился к системному вызову shmat(), который объявлен в заголовочном файле sys/shm.h следующим образом:

```
void * shmat (int ID, void * ADDRESS, int FLAGS);
```

Данный системный вызов возвращает адрес совместно используемого сегмента памяти. Первый аргумент (ID) — это идентификатор сегмента. Аргумент ADDRESS позволяет указать адрес, по которому будет доступен общий сегмент памяти. Если этот аргумент равен NULL, то shmat() самостоятельно выберет область памяти для отображения сегмента. Аргумент FLAGS позволяет передать дополнительные флаги. Обычно этот аргумент содержит 0 (отсутствие флагов).

В случае ошибки системный вызов shmat() возвращает указатель (void *) -1.

21.3. Отключение совместного доступа: *shmdt()*

Системный вызов shmdt() вызывается в каждом процессе после того, как тот закончил работу с общей памятью. Этот системный вызов объявлен в заголовочном файле sys/shm.h следующим образом:

```
int shmdt (void * ADDRESS);
```

Единственный аргумент ADDRESS — это адрес совместно используемой памяти, который возвращает shmat() при подключении сегмента. При успешном завершении shmdt() возвращает 0. В случае ошибки возвращается –1.

21.4. Контроль использования памяти: *shmctl()*

Системный вызов `shmctl()` позволяет осуществлять различные операции над общим сегментом памяти. Этот системный вызов объявлен в заголовочном файле sys/shm.h следующим образом:

```
int shmctl (int ID, int COMMAND, struct shmid_ds * DESC);
```

Рассмотрим по порядку аргументы этого системного вызова.

- `ID` — идентификатор совместно используемого сегмента памяти.
- `COMMAND` — команда, которую требуется выполнить в отношении сегмента общей памяти с идентификатором `ID`. Часто употребляются две команды: `IPC_STAT` (получить данные о сегменте) и `IPC_RMID` (удалить сегмент).
- `DESC` — указатель на структуру, в которую (для команды `IPC_STAT`) заносятся данные о сегменте. Для команды `IPC_RMID` в этом аргументе можно указать `NULL`.

При успешном завершении `shmctl()` возвращает 0. В случае ошибки возвращается –1.

Рассмотрим теперь простейший пример межпроцессного взаимодействия на основе совместно используемых сегментов памяти. Сначала создадим процесс, который создает общий сегмент и открывает доступ к нему для всех желающих (листинг 21.1).

Листинг 21.1. Программа shm1-owner.c

```
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>

#define SHMEM_SIZE 4096
#define SH_MESSAGE "Hello World!\n"

int main (void)
{
        int shm_id;
        char * shm_buf;
        int shm_size;
        struct shmid_ds ds;

        shm_id = shmget (IPC_PRIVATE, SHMEM_SIZE,
                        IPC_CREAT | IPC_EXCL | 0600);

        if (shm_id == -1) {
                fprintf (stderr, "shmget() error\n");
                return 1;
        }
```

```
        shm_buf = (char *) shmat (shm_id, NULL, 0);
        if (shm_buf == (char *) -1) {
                fprintf (stderr, "shmat() error\n");
                return 1;
        }

        shmctl (shm_id, IPC_STAT, &ds);

        shm_size = ds.shm_segsz;
        if (shm_size < strlen (SH_MESSAGE)) {
                fprintf (stderr, "error: segsize=%d\n", shm_size);
                return 1;
        }

        strcpy (shm_buf, SH_MESSAGE);

        printf ("ID: %d\n", shm_id);
        printf ("Press <Enter> to exit...");
        fgetc (stdin);

        shmdt (shm_buf);
        shmctl (shm_id, IPC_RMID, NULL);

        return 0;
}
```

Эта программа создает общий сегмент с использованием "динамического ключа" (`IPC_PRIVATE`). После успешного создания и подключения сегмента программа выводит на экран его идентификатор, заносит в сегмент данные и останавливается до тех пор, пока пользователь не нажмет клавишу <Enter>. Эта задержка позволяет другому процессу подключить общий сегмент памяти и прочитать оттуда данные.

Рассмотрим теперь программу, которая читает данные из совместно используемого сегмента (листинг 21.2).

Листинг 21.2. Программа shm1-user.c

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

int main (int argc, char ** argv)
{
        int shm_id;
        char * shm_buf;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
```

```
                return 1;
        }

        shm_id = atoi (argv[1]);
        shm_buf = (char *) shmat (shm_id, 0, 0);
        if (shm_buf == (char *) -1) {
                fprintf (stderr, "shmat() error\n");
                return 1;
        }

        printf ("Message: %s\n", shm_buf);
        shmdt (shm_buf);

        return 0;
}
```

Поскольку процессы "не договорились" заранее о выделении общего ключа для сегмента памяти, то единственным адекватным способом взаимодействия будет непосредственная передача программе-клиенту идентификатора сегмента через аргумент.

Для проверки работы этой программы желательно иметь два терминальных окна. В первом окне запускаем процесс-сервер:

```
$ gcc -o shm1-owner shm1-owner.c
$ ./shm1-owner
ID: 31391750
Press <Enter> to exit...
```

Теперь, не нажимая клавиши <Enter>, переходим в другое терминальное окно и запускаем процесс-клиент:

```
$ gcc -o shm1-user shm1-user.c
$ ./shm1-user 31391750
Message: Hello World!
```

Но не торопитесь переходить в исходное окно и нажимать клавишу <Enter>. Эксперимент еще не закончен! Наберите в клиентском окне команду `ipcs -m`:

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner  perms  bytes   nattch  status
0x00005d8b 13565953   root   777    316     1
0x00000000 13795330   nn     600    393216  2       dest
0x00000000 13828099   nn     600    393216  2       dest
0x00000000 29589508   nn     666    112320  1       dest
0x00000000 14811141   nn     777    393216  2       dest
0x00000000 31391750   nn     600    4096    1
```

Эта команда выводит на экран список совместно используемых сегментов памяти с их ключами и идентификаторами. В приведенном примере последняя строка со-

держит сведения о нашем сегменте. Теперь нажмите клавишу <Enter> в исходном терминале и вызовите `ipcs -m` еще раз:

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner  perms  bytes   nattch  status
0x00005d8b 13565953   root   777    316     1
0x00000000 13795330   nn     600    393216  2       dest
0x00000000 13828099   nn     600    393216  2       dest
0x00000000 29589508   nn     666    112320  1       dest
0x00000000 14811141   nn     777    393216  2       dest
```

Как видите, сегмента больше нет. Но обратите внимание на то, что если программа завершится, не вызвав `shmctl()` с командой `IPC_RMID`, то сегмент продолжит существовать в памяти. Однако пользователь, которому принадлежит сегмент (см. колонку `owner`), может самостоятельно удалить его при помощи команды `ipcrm`.

21.5. Использование семафоров

Семафоры позволяют процессам задействовать общие ресурсы организованно, не допуская конфликтов. Типичный пример — организация межпроцессного взаимодействия с использованием общей памяти. В предыдущем примере нам приходилось изобретать собственные не слишком надежные средства контроля доступа к общему сегменту памяти. В большинстве случаев приложения требуют более надежных подходов к решению данной задачи. Одна из возможностей — применение семафоров.

Семафоры работают подобно дорожным светофорам. Представьте себе две дороги A и B, которые пересекаются на одном уровне и образуют X-образный перекресток. Можно условно сказать, что перекресток в данном случае является общим ресурсом. Если все автомобили будут одновременно использовать его, то может случиться ДТП. Но если на перекрестке установлен светофор, то автомобилям достаточно просто руководствоваться его сигналами, чтобы в скором времени безопасно продолжить свой путь.

Вернемся к программированию. Предположим теперь, что два процесса делят какой-нибудь общий ресурс. Чтобы не произошло конфликтов, процессы договариваются об использовании дополнительного разделяемого ресурса — семафора. Каждый из процессов перед доступом к общему ресурсу пытается установить для себя семафор (занять место). Если семафор уже установлен кем-то другим, то процесс блокируется до тех пор, пока другой процесс не освободит ресурс. Это самая простая схема взаимодействия. На самом деле семафоры в Linux представляют собой очень гибкий механизм, который может контролировать доступ к совместным ресурсам именно так, как нужно вам.

Применение семафоров в Linux осуществляется на основе следующих механизмов, объявленных в заголовочном файле sys/sem.h:

```
struct sembuf
{
       unsigned short int sem_num;
       short int sem_op;
       short int sem_flg;
};

union semnum
{
       int val;
       struct semid_ds * buf;
       unsigned short * array;
};

int semget (key_t KEY, int SEMS, int FLAGS);
int semop (int ID, struct sembuf * SB, size_t SIZE);
int semctl (int ID, int SNUM, int COMMAND, ...);
```

Семафоры, как и совместно используемые сегменты памяти, также задействуют ключи, идентификаторы и флаги. Системный вызов `semget()` создает набор семафоров в количестве `SEMS` с ключом `KEY` и флагами `FLAGS`.

Системный вызов `semop()` позволяет осуществлять рядовые операции над набором семафоров с идентификатором `ID` (который возвращает `semget()`). Сами операции осуществляются заданием полей массива `SB`. `SIZE` — это размер массива `SB` или его части.

Рассмотрим теперь элементы структуры `sembuf`. Поле `sem_num` — это номер семафора. Поле `sem_flg` устанавливает флаги для операции над семафорами. Особый интерес здесь представляет флаг `SEM_UNDO`, который отменяет операцию над семафором, если процесс неожиданно завершается.

Поле `sem_op` структуры `sembuf` имеет ключевое значение при использовании семафоров. Следует признать, что механизм семафоров в Linux умышленно усложнен за счет включения в него множества редко реализуемых возможностей. Если не погружаться в эти подробности, то значение поля `sem_op` можно описать понятным языком.

- Если поле `sem_op` структуры `sembuf` равно –1, то процесс как бы говорит: "Далее я собираюсь использовать общий ресурс. Остановите меня, если ресурс занят".
- Если поле `sem_op` структуры `sembuf` равно 1, то процесс как бы говорит: "Я закончил работу. Ресурс свободен".

Дополнительные возможности структуры `sembuf` можно изучить на man-странице системного вызова `semop()`.

21.6. Контроль за семафорами: *semctl()*

Администрирование семафоров осуществляется при помощи системного вызова `semctl()`, который объявлен в заголовочном файле sys/sem.h следующим образом:

```
int semctl (int ID, int SNUM, int COMMAND, ...);
```

Этот системный вызов обычно задает две команды: инициализацию и удаление семафоров. Здесь `ID` — это идентификатор группы семафоров, `SNUM` — номер семафора. Для инициализации семафоров в качестве третьего аргумента передается константа `SETALL`, а для удаления — `IPC_RMID`. Обе команды требуют наличия четвертого аргумента в виде объединения типа `semnum`. Опять же, данный механизм имеет много дополнительных возможностей, поэтому сильно усложнен. Для одного семафора достаточно создать объединение `semnum` и присвоить ему несколько необходимых значений. Вот как это делается:

```
union semnum {
       int val;
       struct semid_ds * buf;
       unsigned short * array;
} sem_arg;

/* Где-то в программе перед первым вызовом semctl()... */
unsigned short sem_vals[1];
sem_vals[0] = 1;
sem_arg.array = sem_vals;
```

Теперь мы можем модернизировать программы из *разд. 21.4* таким образом, чтобы они использовали сегменты разделяемой памяти с применением семафора (листинги 21.3 и 21.4).

Листинг 21.3. Программа shm2-owner.c

```
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMEM_SIZE       4096
#define SH_MESSAGE       "Hello World!\n"

#define SEM_KEY              2007
#define SHM_KEY              2007

union semnum {
       int val;
       struct semid_ds * buf;
       unsigned short * array;
} sem_arg;
```

```
int main (void)
{
        int shm_id, sem_id;
        char * shm_buf;
        int shm_size;
        struct shmid_ds ds;
        struct sembuf sb[1];
        unsigned short sem_vals[1];

        shm_id = shmget (SHM_KEY, SHMEM_SIZE,
                        IPC_CREAT | IPC_EXCL | 0600);

        if (shm_id == -1) {
                fprintf (stderr, "shmget() error\n");
                return 1;
        }

        sem_id = semget (SEM_KEY, 1,
                        0600 | IPC_CREAT | IPC_EXCL);

        if (sem_id == -1) {
                fprintf (stderr, "semget() error\n");
                return 1;
        }

        printf ("Semaphore: %d\n", sem_id);
        sem_vals[0] = 1;
        sem_arg.array = sem_vals;

        if (semctl (sem_id, 0, SETALL, sem_arg) == -1) {
                fprintf (stderr, "semctl() error\n");
                return 1;
        }

        shm_buf = (char *) shmat (shm_id, NULL, 0);
        if (shm_buf == (char *) -1) {
                fprintf (stderr, "shmat() error\n");
                return 1;
        }

        shmctl (shm_id, IPC_STAT, &ds);

        shm_size = ds.shm_segsz;
        if (shm_size < strlen (SH_MESSAGE)) {
                fprintf (stderr, "error: segsize=%d\n", shm_size);
                return 1;
        }
```

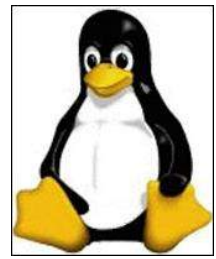
```
        strcpy (shm_buf, SH_MESSAGE);

        printf ("ID: %d\n", shm_id);

        sb[0].sem_num = 0;
        sb[0].sem_flg = SEM_UNDO;

        sb[0].sem_op = -1;
        semop (sem_id, sb, 1);

        sb[0].sem_op = -1;
        semop (sem_id, sb, 1);

        semctl (sem_id, 1, IPC_RMID, sem_arg);
        shmdt (shm_buf);
        shmctl (shm_id, IPC_RMID, NULL);

        return 0;
}
```

Листинг 21.4. Программа shm2-user.c

```
#include <sys/shm.h>
#include <stdio.h>
#include <sys/sem.h>

#define SEM_KEY       2007
#define SHM_KEY       2007

int main (int argc, char ** argv)
{
        int shm_id, sem_id;
        char * shm_buf;
        struct sembuf sb[1];

        shm_id = shmget (SHM_KEY, 1, 0600);
        if (shm_id == -1) {
                fprintf (stderr, "shmget() error\n");
                return 1;
        }

        sem_id = semget (SEM_KEY, 1, 0600);
        if (sem_id == -1) {
                fprintf (stderr, "semget() error\n");
                return 1;
        }
```

```
        shm_buf = (char *) shmat (shm_id, 0, 0);
        if (shm_buf == (char *) -1) {
                fprintf (stderr, "shmat() error\n");
                return 1;
        }

        printf ("Message: %s\n", shm_buf);

        sb[0].sem_num = 0;
        sb[0].sem_flg = SEM_UNDO;

        sb[0].sem_op = 1;
        semop (sem_id, sb, 1);

        shmdt (shm_buf);
        return 0;
}
```

ГЛАВА 22



Использование общих файлов

В этой главе будут рассмотрены следующие темы:

- Отображение файла в оперативной памяти.
- Освобождение памяти, выделенной для отображения файла.
- Синхронизация данных при использовании отображения файлов в оперативной памяти.

22.1. Размещение файла в памяти: *mmap()*

Системный вызов `mmap()` позволяет частично или целиком отображать в оперативной памяти содержимое файла и может предназначаться в Linux для различных целей, однако применение его для межпроцессного взаимодействия представляет особый интерес.

Системный вызов `mmap()` объявлен в заголовочном файле sys/mman.h следующим образом:

```
void * mmap (void * ADDRESS, size_t LEN, int PROT,
                      int FLAGS, int FD, off_t OFFSET);
```

Рассмотрим по порядку аргументы `mmap()`.

- `ADDRESS` — это адрес, по которому будет отображаться файл. Если указать здесь `NULL`, то адрес будет выбран автоматически.
- `LEN` — размер отображаемой области.
- `PROT` — уровень защиты отображаемой области. Этот аргумент может состоять из произвольной побитовой дизъюнкции флагов `PROT_READ` (чтение), `PROT_WRITE` (запись) и `PROT_EXEC` (выполнение).
- `FLAGS` — при реализации межпроцессного взаимодействия этот аргумент обычно принимает значение `MAP_SHARED`.
- `FD` — дескриптор открытого файла, который будет отображаться в памяти.
- `OFFSET` — смещение, от которого будет отображаться файл.

При успешном завершении mmap() возвращает адрес отображаемой области памяти. В случае ошибки возвращается указатель MAP_FAILED.

22.2. Освобождение памяти: *munmap()*

Системный вызов munmap() освобождает отображаемую область памяти. Он объявлен в заголовочном файле sys/mman.h следующим образом:

```
int munmap (void * ADDRESS, size_t LEN);
```

Данный системный вызов освобождает буфер отображаемой памяти, расположенный по адресу ADDRESS и имеющий размер LEN. При успешном завершении munmap() возвращает 0. В случае ошибки возвращается –1.

Рассмотрим пример небольшой программы (листинг 22.1), которая демонстрирует использование системного вызова mmap(), перезаписывая содержимое файла "задом наперед".

Листинг 22.1. Программа mmap1.c

```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define FLENGTH       256

void reverse (char * buf, int size)
{
       int i;
       char ch;
       for (i = 0; i < (size/2); i++)
       {
              ch = buf[i];
              buf[i] = buf[size-i-1];
              buf[size-i-1] = ch;
       }
}

int main (int argc, char ** argv)
{
       int fd;
       char * buf;
       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }
```

```
        fd = open (argv[1], O_RDWR);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file (%s)\n",
                                argv[1]);
                return 1;
        }

        buf = mmap (0, FLENGTH, PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);
        if (buf == MAP_FAILED) {
                fprintf (stderr, "mmap() error\n");
                return 1;
        }

        close (fd);
        reverse (buf, strlen (buf));

        munmap (buf, FLENGTH);
        return 0;
}
```

Проверяем:

```
$ gcc -o mmap1 mmap1.c
$ echo "" > myfile
$ echo -n LINUX >> myfile
$ ./mmap1 myfile
$ cat myfile
XUNIL
```

22.3. Синхронизация: *msync()*

Данные файла, отображаемые в памяти, на самом деле существуют отдельно от самого файла. Если отображаемая область изменяется, то сброс данных файла на носитель осуществляется только при вызове `munmap()`. Однако использование отображаемых в памяти файлов при реализации межпроцессного взаимодействия зачастую требует осуществить сброс данных на носитель (синхронизацию) без отключения отображаемой области. Это позволяет делать системный вызов `msync()`, объявленный в заголовочном файле sys/mman.h следующим образом:

```
int msync (void * ADDRESS, size_t LEN, int FLAGS);
```

`ADDRESS` — буфер отображаемого файла размера `LEN`. При реализации межпроцессного взаимодействия в третьем аргументе `msync()` обычно фигурирует флаг `MS_SYNC`, который не дает системному вызову завершиться до тех пор, пока данные не будут синхронизированы.

Рассмотрим теперь пример межпроцессного взаимодействия (листинги 22.2 и 22.3) на основе механизма отображения файлов в памяти. Для разделения доступа к общему файлу применяется семафор.

Листинг 22.2. Пример mmap2-owner.c

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define SEM_KEY       2007
#define MM_FILENAME       "message"

union semnum {
       int val;
       struct semid_ds * buf;
       unsigned short * array;
} sem_arg;

int main (int argc, char ** argv)
{
       int fd, sem_id, msg_len;
       struct sembuf sb[1];
       unsigned short sem_vals[1];
       char * mbuf;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       msg_len = strlen (argv[1]);
       fd = open (MM_FILENAME, O_RDWR | O_CREAT |
                     O_TRUNC, 0600);
       if (fd == -1) {
              fprintf (stderr, "Cannot open file\n");
              return 1;
       }

       mbuf = (char *) malloc (msg_len);
       if (mbuf == NULL) {
              fprintf (stderr, "malloc() error\n");
              return 1;
       }

       mbuf = memset (mbuf, 0, msg_len);
       write (fd, mbuf, msg_len);
       lseek (fd, 0, SEEK_SET);
       free (mbuf);
```

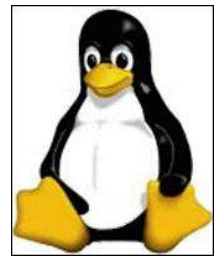
```
        mbuf = mmap (0, msg_len, PROT_WRITE | PROT_READ,
                     MAP_SHARED, fd, 0);
        if (mbuf == MAP_FAILED) {
                fprintf (stderr, "mmap() error\n");
                return 1;
        }

        close (fd);
        strncpy (mbuf, argv[1], msg_len);
        msync (mbuf, msg_len, MS_SYNC);

        sem_id = semget (SEM_KEY, 1,
                     0600 | IPC_CREAT | IPC_EXCL);
        if (sem_id == -1) {
                fprintf (stderr, "semget() error\n");
                return 1;
        }

        sem_vals[0] = 1;
        sem_arg.array = sem_vals;

        if (semctl (sem_id, 0, SETALL, sem_arg) == -1) {
                fprintf (stderr, "semctl() error\n");
                return 1;
        }

        sb[0].sem_num = 0;
        sb[0].sem_flg = SEM_UNDO;

        sb[0].sem_op = -1;
        semop (sem_id, sb, 1);

        sb[0].sem_op = -1;
        semop (sem_id, sb, 1);

        semctl (sem_id, 1, IPC_RMID, sem_arg);
        munmap (mbuf, msg_len);
        unlink (MM_FILENAME);
        return 0;
}
```

Листинг 22.3. Пример mmap2-user.c

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#define SEM_KEY       2007
#define MM_FILENAME       "message"

int main (void)
{
       int fd, sem_id, buf_len;
       struct sembuf sb[1];
       char * mbuf;
       struct stat st;

       sem_id = semget (SEM_KEY, 1, 0600);
       if (sem_id == -1) {
              fprintf (stderr, "semget() error\n");
              return 1;
       }

       fd = open (MM_FILENAME, O_RDONLY);
       if (fd == -1) {
              fprintf (stderr, "Cannot open file\n");
              return 1;
       }

       if (fstat (fd, &st) == -1) {
              fprintf (stderr, "fstat() error\n");
              return 1;
       }

       buf_len = st.st_size;

       mbuf = mmap (0, buf_len, PROT_READ,
                     MAP_SHARED, fd, 0);
       if (mbuf == MAP_FAILED) {
              fprintf (stderr, "mmap() error\n");
              return 1;
       }

       close (fd);
       write (1, mbuf, buf_len);
       printf ("\n");
       munmap (mbuf, buf_len);

       sb[0].sem_num = 0;
       sb[0].sem_flg = SEM_UNDO;

       sb[0].sem_op = 1;
       semop (sem_id, sb, 1);

       return 0;
}
```

ГЛАВА 23



Каналы

В этой главе будут рассмотрены следующие темы:

- Создание и использование каналов.
- Перенаправление ввода-вывода.

23.1. Создание канала: *pipe()*

Каналы осуществляют в Linux родственное локальное межпроцессное взаимодействие. Интерфейс канала представляет собой два связанных файловых дескриптора, один из которых предназначен для записи данных, другой — для чтения. Возможность родственного межпроцессного взаимодействия через канал обусловлена тем, что дочерние процессы наследуют от родителей открытые файловые дескрипторы.

Для создания канала служит системный вызов `pipe()`, который объявлен в заголовочном файле unistd.h следующим образом:

```
int pipe (int PFDS[2]);
```

При успешном завершении системного вызова `pipe()` в текущем процессе появляется канал, запись в который осуществляется через дескриптор `PFDS[0]`, а чтение — через `PFDS[1]`. При удачном завершении `pipe()` возвращает 0. В случае ошибки возвращается –1.

Данные, передаваемые через канал, практически не нуждаются в разделении доступа. Канал автоматически блокирует читающий или пишущий процесс, когда это необходимо. Если читающий процесс запрашивает больше данных, чем есть в настоящий момент в канале, то процесс блокируется до тех пор, пока в канале не появятся новые данные.

Закрытие "концов" канала осуществляется при помощи системного вызова `close()`. Следует отметить, что снять блокировку с читающего процесса можно, закрыв входной "конец" канала. И наоборот: если пишущий процесс заблокирован операцией `write()`, например, то закрытие читающего конца канала снимет блокировку.

Здесь нужно помнить то, о чем неоднократно говорилось ранее: закрытие дескриптора происходит только тогда, когда все процессы, разделяющие этот дескриптор, вызовут `close()`.

Рассмотрим теперь пример взаимодействия двух процессов-братьев через канал, созданный в родительском процессе (листинги 23.1—23.3).

Листинг 23.1. Пример pipe1-parent.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

#define STR_SIZE       32

int main (void)
{
       int pf[2];
       int pid1, pid2;
       char spf [2][STR_SIZE];

       if (pipe (pf) == -1) {
              fprintf (stderr, "pipe() error\n");
              return 1;
       }

       sprintf (spf[0], "%d", pf[0]);
       sprintf (spf[1], "%d", pf[1]);

       if ((pid1 = fork ()) == 0) {
              close (pf[0]);
              execl ("./pipe1-src", "pipe1-src",
                                    spf[1], NULL);
              fprintf (stderr, "exec() [src] error\n");
              return 1;
       }

       if ((pid2 = fork ()) == 0) {
              close (pf[1]);
              execl ("./pipe1-dst", "pipe1-dst",
                                    spf[0], NULL);
              fprintf (stderr, "exec() [dst] error\n");
              return 1;
       }

       waitpid (pid1, NULL, 0);
       close (pf[0]);
```

```
        close (pf[1]);
        waitpid (pid2, NULL, 0);

        return 0;
}
```

Листинг 23.2. Пример pipe1-src.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PP_MESSAGE       "Hello World\n"
#define WAIT_SECS       5

int main (int argc, char ** argv)
{
        int i, fd;
        if (argc < 2) {
                fprintf (stderr, "src: Too few arguments\n");
                return 1;
        }

        fd = atoi (argv[1]);

        fprintf (stderr, "Wait please");
        for (i = 0; i < WAIT_SECS; i++, sleep (1))
                fprintf (stderr, ".");
        fprintf (stderr, "\n");

        if (write (fd, PP_MESSAGE,
                        strlen (PP_MESSAGE)) == -1) {
                fprintf (stderr, "src: write() error\n");
                return 1;
        }

        close (fd);
        return 0;
}
```

Листинг 23.3. Пример pipe1-dst.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
int main (int argc, char ** argv)
{
       int fd;
       char ch;
       if (argc < 2) {
              fprintf (stderr, "dst: Too few arguments\n");
              return 1;
       }

       fd = atoi (argv[1]);
       while (read (fd, &ch, 1) > 0)
              write (1, &ch, 1);

       close (fd);
       return 0;
}
```

Вот как все это работает:

```
$ gcc -o pipe1-parent pipe1-parent.c
$ gcc -o pipe1-src pipe1-src.c
$ gcc -o pipe1-dst pipe1-dst.c
$ ./pipe1-parent
Wait please.....
Hello World
```

23.2. Перенаправление ввода-вывода: *dup2()*

Системный вызов `dup2()` позволяет перенаправлять ввод-вывод. Он объявлен в заголовочном файле unistd.h следующим образом:

```
int dup2 (int FD1, int FD2);
```

Следует признать, что man-страница этого простого системного вызова описывает его слишком сложным языком. Итак, если смотреть проще, то можно сказать так: системный вызов `dup2()` перенаправляет ввод-вывод с дескриптора `FD2` на дескриптор `FD1`. Или так: системный вызов `dup2()` связывает дескриптор `FD2` с ресурсом (файлом, потоком, устройством и т. п.) дескриптора `FD1`.

> ***Примечание***
>
> Существует также системный вызов `dup()`, который функционирует по аналогии с `dup2()`, но, как правило, менее полезен при реализации межпроцессного взаимодействия.

При успешном завершении `dup2()` возвращает 0. В случае ошибки возвращается –1. Листинг 23.4 содержит пример перенаправления стандартного вывода в файл.

Листинг 23.4. Программа dup01.c

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

#define FILENAME       "myfile"

int main (void)
{
       char ch;
       int fd = open (FILENAME, O_WRONLY | O_CREAT |
                           O_TRUNC, 0640);
       if (fd == -1) {
              fprintf (stderr, "open() error\n");
              return 1;
       }

       if (dup2 (fd, 1) == -1) {
              fprintf (stderr, "dup2() error\n");
              return 1;
       }

       printf ("Hello World!\n");

       close (fd);
       return 0;
}
```

Эта программа выводит сообщение "Hello World!" посредством функции `printf()`. Но системный вызов `dup2()` перенаправил стандартный вывод (дескриптор 1) в файл. Поэтому после работы программы текст "Hello World!" окажется не на экране терминала, а в файле myfile:

```
$ gcc -o dup01 dup01.c
$ ./dup01
$ cat myfile
Hello World!
```

Следующая программа (листинг 23.5) осуществляет перенаправление стандартного ввода.

Листинг 23.5. Программа dup02.c

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

```
#define FILENAME       "myfile"

int main (void)
{
       char ch;
       int fd = open (FILENAME, O_RDONLY);
       if (fd == -1) {
              fprintf (stderr, "open() error\n");
              return 1;
       }

       if (dup2 (fd, 0) == -1) {
              fprintf (stderr, "dup2() error\n");
              return 1;
       }

       while (read (0, &ch, 1) > 0)
              write (1, &ch, 1);

       close (fd);
       return 0;
}
```

При попытке прочитать что-нибудь из стандартного ввода программа выведет на экран содержимое файла myfile:

```
$ gcc -o dup02 dup02.c
$ echo "Hello World" > myfile
$ ./dup02
Hello World
```

Аргументами `dup2()` могут быть дескрипторы канала. Это позволяет налаживать взаимодействие любых процессов, которые используют в своей работе консольный ввод-вывод. Следующий пример (листинг 23.6) принимает два аргумента: имя каталога и произвольную строку. Затем программа вызывает команду `ls` для первого аргумента и `grep -i` для второго. При этом стандартный вывод `ls` связывается со стандартным вводом `grep`. В итоге на экран выводятся все элементы указанного каталога, в которых содержится строка (без учета регистра символов) из второго аргумента.

Листинг 23.6. Программа pipe2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
```

```
int main (int argc, char ** argv)
{
        int pf[2];
        int pid1, pid2;

        if (argc < 3) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        if (pipe (pf) == -1) {
                fprintf (stderr, "pipe() error\n");
                return 1;
        }

        if ((pid1 = fork ()) == 0) {
                dup2 (pf[1], 1);
                close (pf[0]);
                execlp ("ls", "ls", argv[1], NULL);
                fprintf (stderr, "exec() [1] error\n");
                return 1;
        }

        if ((pid1 = fork ()) == 0) {
                dup2 (pf[0], 0);
                close (pf[1]);
                execlp ("grep", "grep", "-i", argv[2], NULL);
                fprintf (stderr, "exec() [2] error\n");
                return 1;
        }

        close (pf[1]);
        waitpid (pid1, NULL, 0);
        close (pf[0]);
        waitpid (pid2, NULL, 0);
        return 0;
}
```

Проверяем:

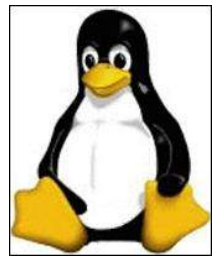
```
$ gcc -o pipe2 pipe2.c
$ ./pipe2 / bin
bin
sbin
```

23.3. Получение дополнительной информации

Дополнительную информацию по использованию каналов и перенаправлению ввода-вывода можно получить на следующих man-страницах:

- man 2 pipe;
- man 2 dup;
- man 2 dup2;
- man 2 fcntl.

ГЛАВА 24



Именованные каналы FIFO

В этой главе будут рассмотрены следующие темы:

- Создание и удаление именованного канала FIFO.
- Межпроцессное взаимодействие через FIFO.

24.1. Создание именованного канала

Именованные каналы FIFO работают аналогично обычным. Особенность FIFO в том, что они представлены файлами специального типа и "видны" в файловой системе. Работа с этими файлами осуществляется обычным образом при помощи системных вызовов `open()`, `close()`, `read()`, `write()` и т. д. Это, как правило, позволяет использовать их вместо обычных файлов. Следует также отметить, что через FIFO могут взаимодействовать процессы, не являющиеся ближайшими родственниками.

Именованные каналы создаются командой `mkfifo` и удаляются командой `rm`:

```
$ mkfifo myfifo
$ ls -l myfifo
prw-r--r-- 1 nn nn 0 2011-05-11 09:57 myfifo
$ rm myfifo
```

Обратите внимание, что в расширенном выводе программы ls каналы FIFO обозначаются символом `p` (от англ. pipe). Работать с FIFO можно даже без написания специальных программ. Проведем небольшой эксперимент:

```
$ mkfifo myfifo
$ cat myfifo
```

Поскольку канал FIFO пуст, то программа cat оказалась заблокированной. Для снятия блокировки откроем новое окно терминала и введем следующую команду:

```
$ echo "Hello World" > myfifo
```

В первом окне будет написано сообщение "Hello World" и программа cat завершится.

Программа может самостоятельно создавать именованные каналы FIFO. Для этого предусмотрен системный вызов `mkfifo()`, который объявлен в заголовочном файле sys/stat.h следующим образом:

```
int mkfifo (const char * FILENAME, mode_t MODE);
```

`FILENAME` — это имя файла, `MODE` — права доступа. При успешном завершении `mkfifo()` возвращает 0. В случае ошибки возвращается –1. Удаление FIFO осуществляется системным вызовом `unlink()`.

Рассмотрим небольшую программу (листинг 24.1), которая создает именованный канал FIFO, имя которого передается через аргумент.

Листинг 24.1. Программа mkfifo1.c

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

int main (int argc, char ** argv)
{
       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       if (mkfifo (argv[1], 0640) == -1) {
              fprintf (stderr, "Can't make fifo\n");
              return 1;
       }

       return 0;
}
```

24.2. Чтение, запись и закрытие FIFO

Как уже упоминалось, каналы FIFO функционируют аналогично обычным каналам. Если канал опустел, то читающий процесс блокируется до тех пор, пока не будет прочитано заданное количество данных или пишущий процесс не вызовет `close()`. И наоборот, пишущий процесс "засыпает", если данные не успевают считываться из FIFO. Листинги 24.2 и 24.3 содержат две программы, которые взаимодействуют через FIFO.

Листинг 24.2. Программа fifo2-server.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <fcntl.h>

#define FIFO_NAME       "myfifo"
#define BUF_SIZE       512

int main (void)
{
       FILE * fifo;
       char * buf;

       if (mkfifo ("myfifo", 0640) == -1) {
              fprintf (stderr, "Can't create fifo\n");
              return 1;
       }

       fifo = fopen (FIFO_NAME, "r");
       if (fifo == NULL) {
              fprintf (stderr, "Cannot open fifo\n");
              return 1;
       }

       buf = (char *) malloc (BUF_SIZE);
       if (buf == NULL) {
              fprintf (stderr, "malloc () error\n");
              return 1;
       }

       fscanf (fifo, "%s", buf);
       printf ("%s\n", buf);

       fclose (fifo);
       free (buf);
       unlink (FIFO_NAME);
       return 0;
}
```

Листинг 24.3. Программа fifo2-client.c

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

#define FIFO_NAME       "myfifo"

int main (int argc, char ** argv)
{
       int fifo;
```

```
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fifo = open (FIFO_NAME, O_WRONLY);
        if (fifo == -1) {
                fprintf (stderr, "Cannot open fifo\n");
                return 1;
        }

        if (write (fifo, argv[1], strlen (argv[1])) == -1) {
                fprintf (stderr, "write() error\n");
                return 1;
        }

        close (fifo);
        return 0;
}
```

Если запустить "сервер", то он создаст FIFO и "замрет" в ожидании поступления данных:

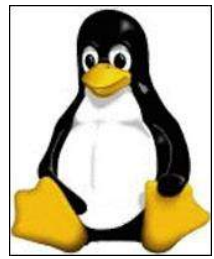
```
$ gcc -o fifo2-server fifo2-server.c
$ ./fifo2-server
```

Запустим теперь в другом терминальном окне клиентскую программу:

```
$ gcc -o fifo2-client fifo2-client.c
$ ./fifo2-client Hello
```

Серверный процесс тут же "проснется", выведет сообщение "Hello" и завершится.

ГЛАВА 25



Сокеты

В *главе 19* мы уже определили общие понятия, связанные с сокетами. В этой главе будут описаны основные методы работы с сокетами. Список источников дополнительной информации приведен в *разд. 25.8*.

25.1. Типы сокетов

Сокеты — это универсальный способ взаимодействия процессов. Они могут использоваться как для локального взаимодействия, так и для обмена данными между удаленными разнородными системами. Классификацию сокетов можно провести по двум критериям: *семейству протоколов* и *способу взаимодействия*.

> ***Примечание***
>
> Семейство протоколов иногда называют *доменом сокета* или *пространством имен сокета*.

Ядро Linux поддерживает несколько десятков семейств протоколов для сокетов. Некоторые из них встречаются повсеместно, другие — очень редко. В рамках данной книги мы будем рассматривать два семейства: *Unix-сокеты* и *интернет-сокеты*. Поскольку Unix-сокеты обеспечивают взаимодействие процессов, работающих в одной системе, то их часто называют *локальными сокетами*.

По способу взаимодействия сокеты также делятся на несколько типов. Мы будем рассматривать два типа: *потоковые* и *дейтаграммные сокеты*. Первые служат для организации надежной передачи данных в сети через протоколы, обеспечивающие контроль целостности доставки и приема информации. Наиболее распространенный протокол такого типа — TCP (Transmission Control Protocol). Дейтаграммные сокеты, напротив, ориентированы не на надежную, а на быструю передачу информации. В сети такие данные чаще всего передаются по протоколу UDP (User Datagram Protocol), который, в отличие от TCP, не контролирует целостность доставки информации и не создает предварительных соединений между клиентом и сервером.

25.2. Создание и удаление сокетов

Для создания сокетов предназначен системный вызов `socket()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
int socket (int PF, int SOCK_TYPE, int PROTOCOL);
```

Данный системный вызов при успешном завершении возвращает дескриптор сокета. Если произошла ошибка, то возвращается –1. Аргумент `PF` — это семейство протоколов сокета, `SOCK_TYPE` — способ взаимодействия. Аргумент `PROTOCOL` необходим для задания конкретного протокола передачи данных. Если здесь указать 0, то выбор протокола произойдет автоматически.

Семейство протоколов определяется одной из символических констант, которые объявлены в файле bits/socket.h. Вот некоторые из них:

- `PF_LOCAL` — локальные сокеты (Unix-сокеты);
- `PF_UNIX` — синоним `PF_LOCAL` (для совместимости с BSD-системами);
- `PF_FILE` — нестандартный синоним `PF_LOCAL`;
- `PF_INET` — интернет-сокеты, основанные на IP версии 4;
- `PF_INET6` — интернет-сокеты, основанные на IP версии 6;
- `PF_BLUETOOTH` — bluetooth-сокеты.

> ***Примечание***
>
> Заголовочный файл bits/socket.h не предназначен для непосредственного включения в пользовательские программы. Для доступа к константам семейств протоколов используйте файл socket.h.

Способ взаимодействия также определяется одной из констант, находящихся в заголовочном файле bits/socket.h. Нам понадобятся только две константы:

- `SOCK_STREAM` — потоковые сокеты;
- `SOCK_DGRAM` — дейтаграммные сокеты.

В качестве третьего аргумента системного вызова `socket()` в подавляющем большинстве случаев достаточно указать 0 (выбор по умолчанию).

> ***Примечание***
>
> Основное техническое отличие потоковых и дейтаграммных сокетов состоит в том, что первые требуют от взаимодействующих процессов установки соединения. Один из процессов (клиент) посылает другому процессу (серверу) запрос на подключение. Если сервер доступен и готов взаимодействовать, то он принимает запрос, и с этого момента процессы могут обмениваться данными. Дейтаграммные сокеты, напротив, не устанавливают соединения. Это обеспечивает быстрый обмен данными, но без контроля доставки. Следует также отметить, что оба типа сокетов пригодны как для локального, так и для удаленного межпроцессного взаимодействия.

25.3. Назначение адреса: *bind()*

Чтобы сервер и клиент могли взаимодействовать, сокету нужно назначить адрес (имя). В зависимости от типа сокета адресом может являться:

- имя файла (локальное взаимодействие);
- сетевой адрес и *порт* (удаленное взаимодействие).

Адрес сокета назначается на стороне сервера. Клиентский процесс может использовать этот адрес для подсоединения к серверу или для передачи данных.

Назначение адреса сокету осуществляется системным вызовом `bind()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
int bind (int FD, const struct sockaddr * ADDRESS,
                              socklen_t LEN);
```

Здесь `FD` — это дескриптор сокета. Расположенная по адресу `ADDRESS` структура типа `sockaddr` задает адресное пространство и, собственно, сам адрес. Аргумент `LEN` указывает размер адресной структуры во втором аргументе.

Обычно во втором аргументе `bind()` указывается (через механизм приведения типов языка C) адрес структуры одного из следующих типов:

- `struct sockaddr_un` — для локальных сокетов;
- `struct sockaddr_in` — для интернет-сокетов.

Структура `sockaddr_un` содержит два поля, которые нас интересуют:

- `sun_family` — семейство адресов (см. далее);
- `sun_path` — имя (путь) файла сокета.

В структуре `sockaddr_in` обычно устанавливаются следующие поля:

- `sin_family` — семейство адресов (см. далее);
- `sin_addr` — адрес сокета;
- `sin_port` — порт сокета (см. далее).

Размер структуры `sockaddr_un` можно вычислить при помощи функции `sizeof()` или специального макроса `SUN_LEN()`. Размер `sockaddr_in` вычисляется только через `sizeof()`.

При локальном взаимодействии на основе Unix-сокетов в качестве второго аргумента `bind()` используется указатель на структуру `sockaddr_un`. В качестве `sun_family` в этом случае обычно устанавливается константа `AF_LOCAL` или ее синоним `AF_UNIX`. Поле `sun_path` — это обычная строка, содержащая путь к файлу сокета.

> ***Примечание***
>
> Поле `sun_path` структуры `sockaddr_un` — это не просто указатель, а статический массив символов. Запись данных в этот элемент выполняйте функциями наподобие `strcpy()` или `sprintf()`.

При взаимодействии через интернет-сокеты вторым аргументом `bind()` является указатель на структуру `sockaddr_in`. В поле `sin_family` здесь обычно указывается константа `AF_INET`, соответствующая семейству адресов Интернета.

Поле `sin_addr` — это приведенный к целому числу IP-адрес серверного узла. Обычно адреса IP записываются в виде "четверки с точками" (например, 192.168.0.1). Кроме того, в Интернете распространена система доменных имен (DNS, Domain Name System), которая сопоставляет IP-адресам удобочитаемые имена (домены). Для преобразования "четверки с точками" или доменного имени в числовой адрес существует библиотечная функция `gethostbyname()`, которая объявлена в заголовочном файле netdb.h следующим образом:

```
struct hostent * gethostbyname (const char * NAME);
```

Адрес узла обычно находится в элементе `host->h_addr_list[0]` структуры `hostent`. Аргумент `NAME` — это доменное имя (адрес) в виде "четверки с точками".

> ***Примечание***
>
> Не следует путать доменные имена и адреса URL (Uniform Resource Locator). Адреса URL служат для определения местонахождения конкретных ресурсов в сети Интернет, а домены — это просто имена IP-адресов в системе DNS. Например, **http://www.bhv.ru/** — это URL-адрес главной страницы сайта издательства "БХВ-Петербург". Однако доменным именем сервера будет просто **bhv.ru**.

При удаленном взаимодействии одних адресов недостаточно. Дело в том, что сервер может одновременно обмениваться данными с несколькими клиентами. Кроме того, на одном узле может работать несколько различных серверов. Для решения данной проблемы используют порты. Порт — это число, идентифицирующее сеанс межпроцессного взаимодействия. Чтобы лучше понять концепцию портов, рассмотрим пример получения Web-страницы с сервера.

1. Предположим, что Web-сервер работает на узле с адресом 192.168.0.1. Сначала сервер создает сокет и назначает ему адрес 192.168.0.1 и порт 80 при помощи вызова `bind()`. Затем сервер выполняет системный вызов `listen()`, который инициирует "прослушивание порта" (см. *разд. 25.5*).
2. Браузер удаленного пользователя "знает", что Web-серверы обычно прослушивают порт 80 на предмет получения запросов. Поэтому клиент создает сокет и вызывает системный вызов `connect()` для адреса 192.168.0.1 и порта 80 (см. *разд. 25.4*).
3. Сервер получает запрос и вызывает `accept()`. В результате создается новый сокет с отдельным номером порта (например, 13574). Возвращаемым значением `accept()` является дескриптор этого нового сокета (см. *разд. 25.6*). Для "старого" сокета снова вызывается `listen()`, чтобы дать другим пользователям Интернета возможность получать Web-страницы с сервера. Следует отметить, что для обслуживания нового сокета запускается отдельная копия Web-сервера (в виде отдельного процесса либо в отдельном потоке).
4. Браузер записывает в свой сокет HTTP-запрос "GET /" (получить главную страницу).

5. Сервер читает запрос "GET /" из "нового" сокета, которому соответствует порт 13574. После обработки запроса в этот же сокет записывается содержимое главной Web-страницы сервера.
6. Браузер получает Web-страницу и закрывает сокет.

Некоторые серверы работают проще. Но, в любом случае, каждый сеанс межпроцессного взаимодействия с сервером идентифицируется отдельным портом.

В поле `sin_port` структуры `sockaddr_in` содержится 16-разрядный номер порта. Дело в том, что порядок записи байтов и битов на конкретном компьютере может отличаться от той, что принята в сети общего пользования. Поэтому для преобразования локального числа в 16-разрядный "сетевой" формат необходима специальная функция `htons()`.

Рассмотрим пример (листинг 25.1), который демонстрирует создание локального сокета и назначения ему локального адреса.

Листинг 25.1. Пример socket1.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <sys/un.h>

int main (int argc, char ** argv)
{
        int sock;
        struct sockaddr_un saddr;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        sock = socket (PF_UNIX, SOCK_STREAM, 0);
        if (sock == -1) {
                fprintf (stderr, "socket() error\n");
                return 1;
        }

        saddr.sun_family = AF_UNIX;
        strcpy (saddr.sun_path, argv[1]);
        if (bind (sock, (struct sockaddr *) &saddr,
                        SUN_LEN (&saddr)) == -1) {
                fprintf (stderr, "bind() error\n");
                return 1;
        }
```

```
        fprintf (stderr, "Press <Enter> to continue...");
        fgetc (stdin);

        close (sock);
        unlink (argv[1]);
        return 0;
}
```

Проверяем:

```
$ gcc -o socket1 socket1.c
$ ./socket1 mysocket
Press <Enter> to continue...
```

Теперь откроем другое терминальное окно и посмотрим на наш сокет:

```
$ ls -l mysocket
srwxr-xr-x 1 nn nn 0 2011-05-11 10:18 mysocket
```

Обратите внимание, что в расширенном выводе программы ls сокет обозначается символом `s`. После нажатия клавиши <Enter> в первом терминале сокет исчезнет.

25.4. Соединение сокетов: *connect()*

При использовании потоковых сокетов между взаимодействующими процессами должно сначала установиться соединение. Для этого клиент вызывает системный вызов `connect()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
int connect (int FD, const struct sockaddr * SADDR,
                     socklen_t LEN);
```

Здесь `FD` — дескриптор сокета, `SADDR` — указатель на структуру, содержащую сведения об адресе сервера, `LEN` — размер адресной структуры. При успешном завершении `connect()` возвращает 0. В случае ошибки возвращается –1.

Следующая программа (листинг 25.2) читает главную страницу из указанного Web-сервера.

Листинг 25.2. Программа getwwwpage.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

#define BUF_LEN       4096
#define HTTP_PORT       80
```

```
int main (int argc, char ** argv)
{
        int sock, count;
        char * buf;
        struct hostent * host;
        struct sockaddr_in addr;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        buf = (char *) malloc (BUF_LEN);
        if (buf == NULL) {
                fprintf (stderr, "malloc() error\n");
                return 1;
        }

        sock = socket (PF_INET, SOCK_STREAM, 0);
        if (sock == -1) {
                fprintf (stderr, "socket() error\n");
                return 1;
        }

        addr.sin_family = AF_INET;

        host = gethostbyname (argv[1]);
        if (host == NULL) {
                fprintf (stderr, "Unknown server\n");
                return 1;
        }

        addr.sin_addr = * (struct in_addr*)
                            host->h_addr_list[0];

        addr.sin_port = htons (HTTP_PORT);

        if (connect (sock, (struct sockaddr*) &addr,
                            sizeof (addr)) == -1) {
                fprintf (stderr, "connect() error\n");
                return 1;
        }

        strcpy (buf, "GET /\n");
        write (sock, buf, strlen (buf));

        while ((count = read (sock, buf, BUF_LEN)) > 0)
                        write (1, buf, count);
```

```
        close (sock);
        free (buf);
        return 0;
}
```

Если у вас есть подключение к Интернету, то можно начать эксперимент:

```
$ gcc -o getwwwpage getwwwpage.c
$ ./getwwwpage bhv.ru > index.html
```

По завершении эксперимента в файле index.html будет находиться содержимое главной Web-страницы издательства "БХВ-Петербург".

> ***Замечание***
>
> Представленная программа не сохраняет изображения и прочий контент Web-страницы, как если бы мы выполняли сохранение при помощи Интернет-браузера. Таким образом, в файле index.html окажется только HTML-код страницы.

25.5. Прослушивание сокета: *listen()*

При взаимодействии через потоковые сокеты сервер должен включить прослушивание, т. е. перейти в режим ожидания запросов на подключение. Это делается при помощи системного вызова `listen()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
int listen (int FD, int QUEUE_LEN);
```

Здесь `FD` — дескриптор сокета. Аргумент `QUEUE_LEN` определяет максимальный размер очереди запросов на подключение. При успешном завершении `listen()` возвращает 0. В случае ошибки возвращается –1.

25.6. Принятие запроса на подключение: *accept()*

Системный вызов `listen()` блокирует сервер до тех пор, пока какой-нибудь клиент не выдаст запрос на подключение. Как только запрос поступил, сервер "просыпается". Если есть возможность обслужить запрос, то сервер вызывает системный вызов `accept()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
int accept (int FD, struct sockaddr * ADDRESS,
                    socklen_t * LENP);
```

Здесь `FD` — это дескриптор сокета. По адресу `ADDRESS` расположена структура, в которую помещаются адресные данные созданного соединения (в частности — порт). Если вам не нужны эти данные, можете просто указать `NULL`. `LENP` — адрес переменной, в которую помещается размер адресной структуры.

При успешном завершении `accept()` возвращает дескриптор нового сокета, который будет обслуживать созданное соединение. В случае ошибки возвращается –1.

Следующие две программы (листинги 25.3 и 25.4) демонстрируют межпроцессное взаимодействие с использованием локальных потоковых сокетов.

Листинг 25.3. Программа socket2-server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <sys/un.h>
#include <stdlib.h>

#define QUEUE_LENGTH       10
#define BUF_LEN              256
#define SOCK_NAME       "mysocket"

int main (void)
{
       int sock, client_sock;
       struct sockaddr_un saddr;
       char * buf;
       int count;

       sock = socket (PF_UNIX, SOCK_STREAM, 0);
       if (sock == -1) {
              fprintf (stderr, "socket() error\n");
              return 1;
       }

       buf = (char *) malloc (BUF_LEN);
       if (buf == NULL) {
              fprintf (stderr, "malloc() error\n");
              return 1;
       }

       saddr.sun_family = AF_UNIX;
       strcpy (saddr.sun_path, SOCK_NAME);
       if (bind (sock, (struct sockaddr *) &saddr,
                     SUN_LEN (&saddr)) == -1) {
              fprintf (stderr, "bind() error\n");
              return 1;
       }

       if (listen (sock, QUEUE_LENGTH) == -1) {
              fprintf (stderr, "listen() error\n");
```

```
                return 0;
        }

        while (1) {
                client_sock = accept (sock, NULL, NULL);

                if (client_sock == -1) {
                        fprintf (stderr, "accept() error\n");
                        return 1;
                }

                if ((count = read (client_sock,
                                buf, BUF_LEN-1)) == -1) {
                        fprintf (stderr, "read() error\n");
                        return 1;
                }

                buf[count] = '\0';
                printf (">> %s\n", buf);
                close (client_sock);

                if (!strcmp (buf, "exit")) break;
        }

        free (buf);
        close (sock);
        unlink (SOCK_NAME);
        return 0;
}
```

Листинг 25.4. Программа socket2-client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <string.h>

#define SOCK_NAME       "mysocket"

int main (int argc, char ** argv)
{
        int sock;
        struct sockaddr_un addr;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
```

```
                return 1;
        }

        sock = socket (PF_UNIX, SOCK_STREAM, 0);
        if (sock == -1) {
                fprintf (stderr, "socket() error\n");
                return 1;
        }

        addr.sun_family = AF_UNIX;
        strcpy (addr.sun_path, SOCK_NAME);

        if (connect (sock, (struct sockaddr *) &addr,
                     SUN_LEN (&addr)) == -1) {
                fprintf (stderr, "connect() error\n");
                return 1;
        }

        if (write (sock, argv[1], strlen (argv[1])) == -1) {
                fprintf (stderr, "write() error\n");
                return 1;
        }

        close (sock);
        return 0;
}
```

Вскоре после запуска сервер переходит в режим прослушивания сокета:

```
$ gcc -o socket2-server socket2-server.c
$ ./socket2-server
```

Откроем теперь другое терминальное окно и начнем передавать серверу запросы:

```
$ gcc -o socket2-client socket2-client.c
$ ./socket2-client Hello
$ ./socket2-client World
$ ./socket2-client Linux
```

В исходном терминальном окне сервера будут выводиться соответствующие сообщения:

```
>> Hello
>> World
>> Linux
```

Сервер будет "покорно" выводить сообщения до тех пор, пока клиент не пошлет строку "exit".

25.7. Прием и передача данных через сокеты

Из предыдущих примеров мы увидели, что при работе с потоковыми сокетами применяются обычные операции ввода-вывода. Однако в случае дейтаграммных сокетов такой подход невозможен. Дело в том, что дейтаграммные сокеты не предназначены для установки соединений посредством системных вызовов `connect()`, `listen()` и `accept()`. Иными словами, для передачи информации без использования соединений нужны системные вызовы, которые посылают и принимают данные через непосредственные адреса.

Для передачи данных на сервер через дейтаграммный сокет предусмотрен системный вызов `sendto()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
ssize_t sendto (int FD, const void * BUFFER,
                size_t BUF_SIZE, int FLAGS,
                const struct sockaddr * SADDR,
                socklen_t * LEN);
```

Возвращаемое значение и первые три аргумента полностью идентичны тем, что используются в системном вызове `write()`. Аргумент `FLAGS` позволяет передавать дополнительные флаги; если таковых не имеется, пишите 0. Через `SADDR` передается адресная структура сервера, на который посылаются данные. `LEN` — это размер этой адресной структуры.

На серверной стороне данные принимаются при помощи системного вызова `recvfrom()`, который объявлен в заголовочном файле sys/socket.h следующим образом:

```
ssize_t recvfrom (int FD, void * BUFFER,
                size_t BUF_SIZE, int FLAGS,
                struct sockaddr * CADDR,
                socklen_t * LENP);
```

Этот системный вызов, кроме прочего, позволяет получить адрес отправителя, что важно в тех случаях, когда клиент ожидает ответ от сервера. Возвращаемое значение и первые три аргумента идентичны тем, что используются в системном вызове `read()`. Аргумент `FLAGS` позволяет передавать дополнительные флаги. Через `CADDR` сервер может получить адресную структуру клиента, а через `LENP` — размер этой структуры.

Следующий пример (листинги 25.5 и 25.6) функционирует аналогично предыдущему (листинги 25.3 и 25.4), но использует дейтаграммные Unix-сокеты.

Листинг 25.5. Программа socket3-server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
```

```
#include <sys/un.h>
#include <stdlib.h>

#define QUEUE_LENGTH       10
#define BUF_LEN       256
#define SOCK_NAME       "mysocket"

int main (void)
{
       int sock;
       struct sockaddr_un saddr;
       char * buf;
       int count;

       sock = socket (PF_UNIX, SOCK_DGRAM, 0);
       if (sock == -1) {
              fprintf (stderr, "socket() error\n");
              return 1;
       }

       buf = (char *) malloc (BUF_LEN);
       if (buf == NULL) {
              fprintf (stderr, "malloc() error\n");
              return 1;
       }

       saddr.sun_family = AF_UNIX;
       strcpy (saddr.sun_path, SOCK_NAME);
       if (bind (sock, (struct sockaddr *) &saddr,
                     SUN_LEN (&saddr)) == -1) {
              fprintf (stderr, "bind() error\n");
              return 1;
       }

       while (1) {
              if ((count = recvfrom (sock, buf, BUF_LEN-1,
                     0, NULL, NULL)) == -1)
              {
                     fprintf (stderr, "recvfrom() error\n");
                     return 1;
              }

              buf[count] = '\0';
              printf (">> %s\n", buf);

              if (!strcmp (buf, "exit")) break;
       }
```

```
        free (buf);
        close (sock);
        unlink (SOCK_NAME);
        return 0;
}
```

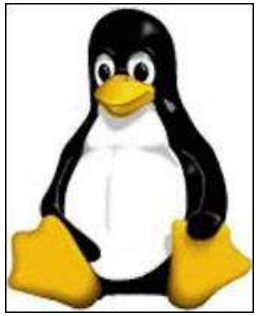
Листинг 25.6. Программа socket3-client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <string.h>

#define SOCK_NAME       "mysocket"

int main (int argc, char ** argv)
{
        int sock;
        struct sockaddr_un addr;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        sock = socket (PF_UNIX, SOCK_DGRAM, 0);
        if (sock == -1) {
                fprintf (stderr, "socket() error\n");
                return 1;
        }

        addr.sun_family = AF_UNIX;
        strcpy (addr.sun_path, SOCK_NAME);

        if (sendto (sock, argv[1], strlen (argv[1]), 0,
                        (struct sockaddr *) &addr,
                        SUN_LEN (&addr)) == -1) {
                fprintf (stderr, "sendto() error\n");
                return 1;
        }

        close (sock);
        return 0;
}
```

25.8. Получение дополнительной информации

Как уже говорилось, сокеты — это очень объемная тема, которая заслуживает написания отдельной книги. В этой главе были приведены лишь основы работы с сокетами. Дополнительную информацию по этой теме можно получить на следующих man-страницах:

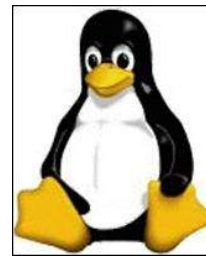
- man 2 socket;
- man 2 bind;
- man 2 listen;
- man 2 accept;
- man 2 connect;
- man 2 send;
- man 2 recv;
- man 3 gethostbyname;
- man 3 htonl;
- man 3 getservent;
- man 3 getnetent;
- man 2 getpeername;
- man 2 getsockname;
- man 2 socketpair;
- man 2 getsockopt;
- man 2 getprotoent.



ЧАСТЬ VI

Работа над ошибками и отладка

ГЛАВА 26



Выявление и обработка ошибок

В этой главе будут рассмотрены следующие темы:

- Классификация ошибок.
- Общие правила вывода сообщений об ошибках.
- Использование макроса `assert()` для защиты от случайных ошибок.

26.1. Типы ошибок

Существуют различные классификации ошибок, которые могут появляться в программах. Обычно ошибки делят на следующие типы:

- ошибки входных данных;
- непредвиденные ошибки;
- скрытые ошибки;
- унаследованные ошибки.

Ошибки входных данных — это нормальная ситуация, когда программа получает от пользователя то, что не может быть адекватно обработано. Основная проблема здесь заключается именно в выявлении ошибки: программист должен предусмотреть все возможные варианты. Но это теория, на практике все обстоит иначе. Например, если программа принимает от пользователя только два символа (char), то число возможных комбинаций входных данных составляет 65536.

Если программа заранее не выявила ошибку входных данных, то это может сказаться на безопасности. Злоумышленник может подобрать такие входные данные, которые способны значительно изменить поведение программы. Обычно такие ошибки устраняются по мере их появления.

Иногда даже те программы, которые мы считаем абсолютно надежными и проверенными, начинают вести себя неадекватно при работе с некоторыми входными данными. Рассмотрим небольшой пример из серии первоапрельских розыгрышей (листинг 26.1).

Листинг 26.1. Пример joke1.c

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

int main (void)
{
        int fd = creat ("--version", 0640);
        if (fd == -1) {
                fprintf (stderr, "creat() error\n");
                return 1;
        }

        close (fd);
        return 0;
}
```

Итак, проверяем:

```
$ gcc -o joke1 joke1.c
$ ./joke1
$ ls
joke1  joke1.c  --version
```

В текущем каталоге появился файл --version. Попробуйте теперь удалить этот файл при помощи программы rm в оболочке bash:

```
$ rm --version
rm (GNU coreutils) 8.5
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Written by Paul Rubin, David MacKenzie, Richard M. Stallman,
and Jim Meyering.
$ ls
joke1  joke1.c  --version
```

Не помогают даже кавычки и шаблоны wildcard:

```
$ rm "--version"

rm (GNU coreutils) 8.5
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

```
Written by Paul Rubin, David MacKenzie, Richard M. Stallman,
and Jim Meyering.
$ rm *version
rm (GNU coreutils) 8.5
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Written by Paul Rubin, David MacKenzie, Richard M. Stallman,
and Jim Meyering.
$ ls
joke1  joke1.c  --version
```

Это просто невинная шутка. Но подумайте, что будет, если имя файла содержит что-нибудь наподобие $HOME. Не экспериментируйте!

Но здесь важно понимать, что если программа будет слишком "усердно" проверять входные данные, то это наверняка скажется на производительности. Поэтому иногда стоит просто отказаться от проверки в угоду скорости, если не возникает явных проблем с безопасностью.

Непредвиденные ошибки возникают тогда, когда программа попадает в экстремальные ситуации: нехватка памяти, мало свободного места на диске и т. д. Следует понимать, что такие ошибки являются штатными, и программа должна быть готовой к их появлению. Есть несколько общих правил, которые позволяют избежать многих непредвиденных ошибок.

- Всегда проверяйте результат выделения динамической памяти.
- Проверяйте файловые дескрипторы после операций `open()`, `creat()`, `socket()` и т. п.
- Проверяйте результат выполнения системных вызовов, включая `close()`.
- Проверяйте `argc` перед использованием `argv`.
- Проверяйте указатели, полученные от других функций.

Некоторые ситуации сложно воспроизвести при тестировании. Но, например, нехватку дискового пространства можно симулировать при помощи специального устройства /dev/full. Для проверки работы программы с большими файлами можно использовать /dev/zero.

Если непредвиденные ошибки можно сравнить с "неисправным огнетушителем", то скрытые ошибки больше напоминают "пехотные мины": наступить на нее — дело случая. Вот распространенный пример такой ошибки:

```
s = (char*) malloc (BUF_SIZE);
...
count = read (fd, s, BUF_SIZE);
s [count] = '\0';
```

Итак, сначала в программе выделяется динамическая память размером `BUF_SIZE`, и адрес заносится в указатель `s`. Затем системный вызов `read()` читает не более чем

BUF_SIZE байтов из некоторого файла и заносит данные в память, обозначенную указателем s. Следующая строка — это и есть скрытая ошибка. Когда вызов read() прочитает BUF_SIZE байтов, нуль-терминатор будет занесен в элемент s[BUF_SIZE], т. е. за пределы выделенной памяти. Эта программа может работать без сбоев несколько лет. Но ничто не помешает ей "рухнуть" в самый ответственный момент.

Унаследованные ошибки появляются в программах из библиотек или даже из ядра ОС. Эти ошибки обычно исчезают после исправления библиотеки. Но здесь есть один побочный эффект. Часто бывает, что динамическая библиотека содержит ошибку, а программист пренебрегает документацией и добивается корректной работы своей программы "методом научного тыка", используя заведомо неправильное поведение библиотеки. Когда появляется исправленная версия библиотеки, то уже сама программа перестает работать правильно. Чтобы избежать подобного, не стоит "выбивать" из программ правильное поведение. Работа любой сторонней функции должна быть вам понятна.

26.2. Сообщения об ошибках

В зависимости от типа ошибки программой могут выводиться различные сообщения. Ошибки входных данных — это штатная ситуация, подразумевающая диалог с пользователем. Например, если пользователь передал в программу имя несуществующего файла, достаточно просто сказать "файл не существует" и указать имя этого файла.

Непредвиденные ошибки могут быть фатальными и обрабатываемыми. Фатальные ошибки возникают тогда, когда программа не может справиться с возникшей ситуацией. В подавляющем большинстве случаев эти ошибки заканчиваются завершением программы. Идеальный вариант сообщения о фатальной ошибке — вывести отладочную информацию и предложить пользователю отправить эту информацию разработчикам программы. Обрабатываемые сообщения об ошибках обычно подразумевают продолжение работы программы. В этом случае пользователю выводится сообщение о том, что операция не удалась.

Скрытые ошибки оправдывают свое название. Наиболее приемлемый способ защититься от них — использовать макрос assert() (см. *разд. 26.3*), который позволяет контролировать правильность выполнения программы в различных ее местах. Если что-то пошло не так, как было задумано, assert() завершает программу и выводит информацию о "нарушении". При помощи assert() можно защититься и от унаследованных ошибок.

Важно понимать, что при возникновении любой ошибки программа не должна "молчать". Если ошибка не связана напрямую с входными данными и подразумевает завершение программы, то лучше вывести как можно больше информации в надежде на то, что пользователь предоставит эту информацию разработчикам.

26.3. Макрос *assert()*

Макрос `assert()` — это простой, но очень эффективный способ защиты от скрытых и наследуемых ошибок. `assert()` объявлен в заголовочном файле assert.h следующим образом:

```
void assert (EXPRESSION);
```

Единственный аргумент `EXPRESSION` — это логическое выражение. Макрос `assert()` работает очень просто: если выражение `EXPRESSION` ложное, то программа аварийно завершается с выводом соответствующего сообщения.

Нужно понимать, что `assert()` — это не упрощенный способ вывода пользователю сообщений об ошибках. В качестве `EXPRESSION` должно фигурировать выражение, которое является истинным по определению. Например, перед использованием массива полезно будет выполнить такую проверку:

```
assert (array != NULL);
array_sort (array);
```

Но следующая запись говорит о дурном стиле программирования:

```
array = (char *) malloc (ARR_SIZE);
assert (array != NULL);
```

А эта запись вообще никуда не годится:

```
int fd = open (FILENAME, O_RDONLY);
assert (fd != -1);
```

В двух последних случаях выражение внутри `assert()` вовсе не обязано быть истинным по определению. Функция `malloc()` может вернуть нулевой указатель, если исчерпана свободная память, а отсутствие открываемого файла (второй случай) — обычное дело.

Рассмотрим пример (листинг 26.2), который демонстрирует работу макроса `assert()`.

Листинг 26.2. Пример assert1.c

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

int main (int argc, char ** argv)
{
        int month;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }
```

```
        month = atoi (argv[1]);
        assert ((month >= 1) && (month <= 12));

        return 0;
}
```

Добавим заодно к программе make-файл (листинг 26.3), который нам понадобится в дальнейшем.

Листинг 26.3. Файл Makefile

```
FLAGS=

assert1: assert1.c
        gcc $(FLAGS) -o $@ $^

clean:
        rm -f assert1
```

Итак, данная программа получает от пользователя календарный месяц в виде числа. Предположим, что проверка вхождения этого значения в диапазоне от 1 до 12 уже выполнена. В каком-то месте программы появляется макрос `assert()`, чтобы подтвердить нормальный ход программы. В нашей программе нет реальной проверки входных данных, поэтому мы легко сможем увидеть действие `assert()`:

```
$ make
gcc  -o assert1 assert1.c
$ ./assert1 10
$ ./assert1 13
assert1: assert1.c:15: main: Assertion `(month >= 1) &&
(month <= 12)' failed.
Aborted
```

Макрос `assert()` обладает еще одной полезной особенностью. Если скомпилировать программу с включением макроконстанты `NDEBUG`, то все вхождения `assert()` исчезнут. Если в программе `assert()` встречается часто, то применение `NDEBUG` может "одним махом" увеличить производительность приложения, а также уменьшить размер исполняемого кода. Обычно это делается тогда, когда программа завершает тестирование и выпускается для официального использования.

Макроконстанту `NDEBUG` можно включить непосредственно в программу:

```
#define NDEBUG
```

В этом случае программист может устанавливать политику использования `assert()` для каждого исходного файла в отдельности. Но это требует изменения исходного кода программы, что не всегда приемлемо.

Можно объявить `NDEBUG` на стадии компиляции через аргумент `-D` компилятора gcc. Наш make-файл содержит константу `FLAGS`, которую можно указать прямо во время вызова make.

Вот как это делается:

```
$ make clean
rm -f assert1
$ make FLAGS=-DNDEBUG
gcc -DNDEBUG -o assert1 assert1.c
$ ./assert1 15
$
```

Рассмотрим еще один немного странный, но показательный пример (листинг 26.4).

Листинг 26.4. Пример assert2.c

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

#define FAIL_SUM       10

int myfunc (int a, int b)
{
       printf ("Hello\n");
       return (a + b);
}

int main (int argc, char ** argv)
{
       int a, b;
       if (argc < 3) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       a = atoi (argv[1]);
       b = atoi (argv[2]);

       assert (myfunc (a, b) != FAIL_SUM);

       return 0;
}
```

Эта программа принимает два аргумента, переводит их в числовое значение, а потом сравнивает сумму этих чисел со значением `FAIL_SUM`. Предполагается, что введенные числа в сумме не должны давать `FAIL_SUM`.

Функция `myfunc()` выводит на экран сообщение "Hello" и возвращает сумму введенных пользователем чисел. Затем макрос `assert()` проверяет возвращенное функцией `myfunc()` значение на равенство `FAIL_SUM`. Все довольно просто и работает без нареканий:

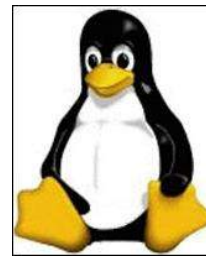
```
$ gcc -o assert2 assert2.c
$ ./assert2 7 4
Hello
$ ./assert2 7 3
Hello
assert2: assert2.c:24: main: Assertion
'myfunc (a, b) != 10' failed.
Aborted
```

Но все изменится, если мы соберем программу с макроконстантой `NDEBUG`:

```
$ gcc -DNDEBUG -o assert1 assert1.c
$ ./assert1 7 4
$ ./assert1 7 3
```

Отключение вхождений `assert()` привело к тому, что функция `myfunc()` теперь вообще не вызывается. "Мораль басни" такова: не нужно забывать, что `assert()` является отключаемым макросом, а не функцией.

ГЛАВА 27



Ошибки системных вызовов

В случае ошибки системные вызовы обычно возвращают –1. Примеры, которые мы рассматривали в предыдущих главах, просто констатировали факт ошибочного завершения системных вызовов и завершались с ненулевым кодом возврата. Однако в Linux есть средства, позволяющие конкретизировать ошибки системных вызовов и обрабатывать их.

27.1. Чтение ошибки: *errno*

Рассмотрим системный вызов `open()`. В случае ошибки он возвращает –1. Но этого часто бывает недостаточно даже для вывода конкретного сообщения об ошибке. Например, ошибка `open()` может означать отсутствие файла, отсутствие прав доступа или еще что-нибудь. Для диагностики ошибок всех системных вызовов предусмотрена внешняя переменная `errno`, которая объявлена в заголовочном файле errno.h следующим образом:

```
extern int errno;
```

Механизм работает очень просто: если какой-нибудь системный вызов завершился с ошибкой, то код ошибки заносится в `errno`. Но здесь надо учитывать, что каждый ошибочный системный вызов перезаписывает общую для всех в контексте одной программы переменную `errno`. Это зачастую требует своевременного сохранения значения `errno` в другой переменной.

Вот наиболее распространенный пример неправильного использования `errno`:

```
if (write (fd, buf, buf_size) == -1) {
       fprintf (stderr, "write() problem:\n");
       if (errno == EBADF) {
              ...
       }
              ...
}
```

Это скрытая ошибка, которая может проявиться не сразу. Действительно, вывод сообщения в поток ошибок `stderr` редко заканчивается ошибкой. Но мы должны

помнить, что функция `fprintf()` работает с системными вызовами. Если один из этих системных вызовов завершится неудачей, то значение `errno` будет перезаписано и обработка исходной ошибки будет неправильной.

Вот еще одна ошибка такого рода:

```
if (write (fd, buf, buf_size) == -1) {
        switch (errno) {
                ...
        }
}
```

Коды ошибок системных вызовов обозначаются символическими константами, которые становятся доступными при включении в программу заголовочного файла errno.h. Каждый системный вызов располагает собственным набором таких констант. В *приложении 2* приведен список этих констант с описаниями для каждого системного вызова.

Рассмотрим теперь пример обработки ошибок системных вызовов с использованием `errno` (листинг 27.1).

Листинг 27.1. Программа errno1.c

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

int main (int argc, char ** argv)
{
        int fd, errno_local;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDWR, 0600);
        if (fd == -1) {
                errno_local = errno;
                switch (errno_local)
                {
                        case (ENOENT):
                                printf ("not exist\n");
                                return 1;

                        case (EISDIR):
                                printf ("is a directory\n");
                                return 1;
```

```
                    case (EACCES):
                            printf ("access denied\n");
                            return 1;

                    default:
                            printf ("unknown error\n");
                            return 1;
                }
        }

        close (fd);
        return 0;
}
```

Вот как работает эта программа:

```
$ gcc -o errno1 errno1.c
$ ./errno1 myfile
not exist
$ mkdir myfile
$ ./errno1 myfile
is a directory
$ rmdir myfile
$ touch myfile
$ ./errno1 myfile
$ chmod 000 myfile
$ ./errno1 myfile
access denied
```

Обратите внимание, что перед обработкой значение `errno` сохраняется в отдельной переменной `errno_local`.

27.2. Сообщение об ошибке: *strerror()*, *perror()*

Для преобразования `errno` в сообщение об ошибке служит функция `strerror()`, которая объявлена в заголовочном файле string.h следующим образом:

```
char * strerror (int ERRN);
```

Функция возвращает строку — сообщение об ошибке, соответствующее коду `ERRN`, полученному из `errno`. Данная функция не предполагает ошибочного завершения.

Следующий пример (листинг 27.2) демонстрирует работу функции `strerror()`.

Листинг 27.2. Программа errno2.c

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

```
#include <errno.h>
#include <string.h>

int main (int argc, char ** argv)
{
        int fd, errno_local;

        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
        }

        fd = open (argv[1], O_RDWR, 0600);
        if (fd == -1) {
                fprintf (stderr, "%s\n", strerror (errno));
                return 1;
        }

        close (fd);
        return 0;
}
```

Проверяем:

```
$ gcc -o errno2 errno2.c
$ ./errno2 myfile
No such file or directory
$ mkdir myfile
$ ./errno2 myfile
Is a directory
$ rmdir myfile
$ touch myfile
$ ./errno2 myfile
$ chmod 000 myfile
$ ./errno2 myfile
Permission denied
```

Для совсем "ленивых" программистов существует функция `perror()`, которая самостоятельно читает значение `errno` и выводит сообщение об ошибке в стандартный поток ошибок. Эта функция объявлена в заголовочном файле stdio.h следующим образом:

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

Здесь `PREFIX` — это произвольная строка, которая выводится перед сообщением об ошибке. Между этой строкой и сообщением `perror()` автоматически вставляет двоеточие и пробел. Обычно в качестве `PREFIX` указывается имя программы или системного вызова.

Примечание

Не стоит пренебрегать префиксами в сообщениях об ошибках. Если две программы разделяют один поток ошибок, то использование префиксов подчас бывает единственным способом отличить сообщения об ошибках, сделанные разными программами.

Следующий пример (листинг 27.3) демонстрирует использование функции `perror()`.

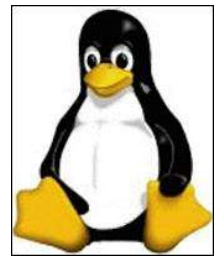
Листинг 27.3. Программа errno3.c

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

int main (int argc, char ** argv)
{
       int fd, errno_local;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       fd = open (argv[1], O_RDWR, 0600);
       if (fd == -1) {
              perror ("open");
              return 1;
       }

       close (fd);
       return 0;
}
```

Вот как работает эта программа:

```
$ gcc -o errno3 errno3.c
$ ./errno3 myfile
open: No such file or directory
$ mkdir myfile
$ ./errno3 myfile
open: Is a directory
$ rmdir myfile
$ touch myfile
$ ./errno3 myfile
$ chmod 000 myfile
$ ./errno3 myfile
open: Permission denied
```

ГЛАВА 28



Использование отладчика gdb

Бывают ситуации, когда от программы ждут одного, а она делает что-то другое. Такие случаи неизбежны; они встречаются как у начинающих, так и у опытных программистов. Обычно причиной неправильного поведения программы является какая-нибудь маленькая ошибка или опечатка, которая играет роль "ложки дегтя в бочке меда".

Отладчик (debugger) — это инструмент, который позволяет изучать программу, выполняя ее в пошаговом режиме. Многие начинающие программисты придумывают собственные методы отладки, включая в программы различные конструкции для вывода различной оперативной информации. Но практика показывает, что трансляция программы под отладчиком — самый быстрый и эффективный способ отыскания ошибок. Время, которое вы затратите на изучение отладчика, многократно окупится в дальнейшем.

В Linux обычно устанавливают свободно распространяемый отладчик gdb (GNU DeBugger). Это очень мощный и гибкий инструмент, позволяющий, кроме прочего, связывать точки выполнения программы с ее исходным кодом. Основное достоинство gdb — его не нужно изучать досконально. Каждый программист выбирает для себя те возможности gdb, которые ему необходимы. Важно понимать, что отладка призвана упрощать создание программ, а не усложнять.

28.1. Добавление отладочной информации

Для использования gdb программа должна быть откомпилирована с опцией `-g`, которая добавляет в исполняемый файл дополнительную отладочную информацию:

```
$ gcc -g -o program program.c
```

Если требуется удалить эту информацию без перекомпиляции, то вызывается программа strip:

```
$ strip program
```

28.2. Запуск отладчика

Для запуска отладчика достаточно набрать команду `gdb`:

```
$ gdb
GNU gdb (GDB) 7.1-1mdv2010.1 (Mandriva Linux release 2010.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying" and
"show warranty" for details.
This GDB was configured as "i586-mandriva-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

Итак, на экране появилось следующее: версия отладчика, лицензионное соглашение и приглашение `(gdb)`, ожидающее от пользователя ввода команд. Если запустить `gdb` с опцией `--quiet` (`-q` — короткий вариант), то информация о версии продукта и лицензионное соглашение выводиться не будут:

```
$ gdb -q
(gdb)
```

Чтобы выйти из отладчика, наберите команду `quit`:

```
(gdb) quit
```

> ***Примечание***
>
> Принцип ввода команд в gdb такой же, как и у командной оболочки: вы набираете команду и нажимаете клавишу <Enter>. Если нажать <Enter> в пустом приглашении, то будет выполнена предыдущая команда.

Теперь напишем простенькую программу (листинг 28.1) и попробуем запустить ее под отладчиком.

Листинг 28.1. Программа program1.c

```
#include <stdio.h>

void change_a (int * mya)
{
        *mya += 10;
}

int main (void)
{
        int a;
        a = 15;

        change_a (&a);
```

```
        printf ("Number: %d\n", a);
        return 0;
}
```

Сначала программу нужно откомпилировать с включением в нее отладочной информации:

```
$ gcc -g -o program1 program1.c
```

Программу можно передать отладчику следующим образом:

```
$ gdb -q program1
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) quit
```

А можно также при помощи команды отладчика `file`:

```
$ gdb -q
(gdb) file program1
Reading symbols from /home/nn/main/work/bhv/src/ch28/program1/program1...done.
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)
```

Для просмотра исходного кода программы используется команда `list`. Если вызвать эту команду без параметров, то будут выведены первые несколько строк исходного кода с нумерацией:

```
(gdb) list
1       #include <stdio.h>
2
3       void change_a (int * mya)
4       {
5               *mya += 10;
6       }
7
8       int main (void)
9       {
10              int a;
(gdb)
```

> ***Примечание***
>
> Некоторые часто используемые команды gdb имеют краткую форму. Например, вместо команды `list` можно вводить просто `l`.

Каждый новый ввод команды `list` без параметров будет выводить следующий набор строк исходного кода программы:

```
(gdb) l
11              a = 15;
12
13              change_a (&a);
14
```

```
15              printf ("Number: %d\n", a);
16              return 0;
17      }
(gdb)
```

Если исходный код закончился, то будет выведено соответствующее сообщение (при очередной попытке вызвать `list` без параметров):

```
(gdb) list
Line number 18 out of range; program1.c has 17 lines.
(gdb)
```

Если в качестве параметра команды `list` указать номер строки, то будет выведен блок исходного кода, "окружающий" эту строку:

```
(gdb) list 10
5               *mya += 10;
6       }
7
8       int main (void)
9       {
10              int a;
11              a = 15;
12
13              change_a (&a);
14
(gdb)
```

Можно также указать через запятую диапазон выводимых строк:

```
(gdb) list 10,15
10              int a;
11              a = 15;
12
13              change_a (&a);
14
15              printf ("Number: %d\n", a);
(gdb)
```

А если в качестве аргумента `list` написать имя функции, то будет выведен блок исходного кода, "окружающий" эту функцию:

```
(gdb) list main
4       {
5               *mya += 10;
6       }
7
8       int main (void)
9       {
10              int a;
```

```
11              a = 15;
12
13              change_a (&a);
(gdb) list change_a
1       #include <stdio.h>
2
3       void change_a (int * mya)
4       {
5               *mya += 10;
6       }
7
8       int main (void)
9       {
10              int a;
(gdb) quit
```

28.3. Трансляция программы под отладчиком

Определимся сначала, что такое стек. В общем понимании *стек* — это обратная очередь. Иными словами, тот, кто попадает в эту очередь последним, обслуживается раньше всех. Например, канал или FIFO — это, наоборот, прямая (справедливая) очередь, которую можно сравнить с очередью покупателей в магазине. Стек похож на стопку чистых листов бумаги: листы обычно кладутся сверху и забираются тоже сверху.

По отношению к программам можно выделить два стека.

- *Стек данных* (data stack) — специальная область памяти процесса, в которой данные читаются и записываются в порядке обратной очереди. Аргументы и локальные переменные функций языка C хранятся в стеке данных.
- *Стек вызовов* (call stack) — это обратная очередь вызовов функций в программе. В самой глубине этой очереди находится функция `main()`: она пришла в программу первой, а выйдет из программы — последней. Если опять представить стек в виде стопки листов чистой бумаги, то вызов функции — это добавление в стопку нового листа, а возврат из функции — это изъятие листа с вершины стопки. Самый нижний "лист" — функция `main()`. Каждый элемент стека вызовов называется *фреймом вызова* (call frame).

> ***Примечание***
>
> Линковщик ld в Linux позволяет помещать в основание стека вызовов любую другую (отличную от `main()`) функцию. Однако в обычных программах эта возможность используется очень редко.

Чтобы просто выполнить программу под отладчиком, введите команду `run`:

```
$ gdb -q program1
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) run
```

```
Starting program:
/home/nn/main/work/bhv/src/ch28/program1/program1
Number: 25

Program exited normally.
(gdb) quit
```

Строка "Program exited normally" сообщает нам, что программа завершилась обычным образом. Изменим теперь нашу программу так, чтобы функция `main()` возвращала числовое значение переданного аргумента (листинг 28.2).

Листинг 28.2. Программа program2.c

```
#include <stdio.h>

void change_a (int * mya)
{
       *mya += 10;
}

int main (int argc, char ** argv)
{
       int a;

       if (argc < 2) {
              fprintf (stderr, "Too few arguments\n");
              return 1;
       }

       a = 15;

       change_a (&a);

       printf ("Number: %d\n", a);
       return atoi (argv[1]);
}
```

Сначала запустим программу без отладчика:

```
$ gcc -g -o program2 program2.c
$ ./program2 10
Number: 25
$ echo $?
10
```

Команда `run` отладчика gdb позволяет задать список аргументов программы так же, как если бы мы делали это в командной оболочке:

```
$ gdb -q
(gdb) file program2
Reading symbols from /home/nn/main/work/bhv/src/ch28/program2/program2...done.
```

```
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) run 5
Starting program:
/home/nn/main/work/bhv/src/ch28/program2/program2 5
Number: 25

Program exited with code 05.
(gdb) quit
```

Сообщение "Program exited with code 05" содержит код возврата программы. Теперь перейдем к самому интересному: научимся выполнять программу под отладчиком в пошаговом режиме. Команда `start` запускает программу и останавливает ее на входе в функцию `main()`. Команде `start` можно передавать аргументы программы:

```
$ gdb -q program2
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) start 7
Breakpoint 1 at 0x8048456: file program2.c, line 9.
Starting program:
/home/nn/main/work/bhv/src/ch28/program2/program2 7
main (argc=Cannot access memory at address 0x0
) at program2.c:9
9       {
(gdb)
```

Итак, программа остановилась на входе в функцию `main()`. Обратите внимание на эту строку:

```
9       {
```

Она означает, что следующим шагом будет выполнение девятой строки исходного кода, т. е. вход в функцию `main()`. Для продвижения по программе в отладчике gdb имеются следующие команды:

- `step` (`s` — краткая форма) — используется для выполнения одной инструкции (строки). Вызов функции, являющейся частью программы, не будет для команды `step` атомарной инструкцией. Иными словами, если в программе встречается вызов внутренней функции, то `step` "заходит" в эту функцию. Этой команде можно передать в качестве аргумента число строк, которые требуется выполнить за один раз;
- `next` (`n` — краткая форма) — работает так же, как и `step`, но без вхождения в функции. Для этой команды вызов функции является просто инструкцией (строкой). Команде `next` можно передать число строк, которые требуется выполнить за один раз;
- `stepi` (`si` — краткая форма) — функционирует как `step`, но атомарной инструкцией считает не строку исходного кода, а машинную команду;
- `nexti` (`ni` — краткая форма) — работает как `next`, но атомарной инструкцией считает не строку исходного кода, а машинную команду;

- `continue` (`c` — краткая форма) — если нет точек останова (см. *разд. 28.4*), то эта команда выполняет оставшуюся часть программы без остановок;
- `kill` — завершает выполнение текущей программы;
- `finish` — действует наподобие `continue`, но не до конца программы, а до возврата из текущей функции.

На каждом этапе выполнения программы мы можем просмотреть ее текущее состояние. Для этого, собственно, и нужна отладка. Далее приведен список команд gdb, позволяющих осуществлять мониторинг текущего состояния программы.

- `backtrace` (`bt` — краткая форма) — выводит на экран текущее состояние стека выполнения программы. Другое название этой команды — `where`.
- `print` (`p` — краткая форма) — выводит на экран значение указанной переменной. В эту команду можно также передавать выражения, составленные по правилам языка C.
- `display` — это очень полезная команда, которая заставляет отладчик выводить значение указанного выражения при каждой остановке программы.
- `show env` — выводит на экран окружение процесса. Если указать в качестве аргумента конкретную переменную, то будет выведено ее значение.
- `info args` — выводит значения аргументов программы.
- `info locals` — выводит значения локальных переменных текущей функции.

Итак, отладчик остановил нашу программу на входе в функцию `main()`. Перейдем на следующую строку командой `step`:

```
(gdb) step
main (argc=2, argv=0xbfd945b4) at program2.c:12
12              if (argc < 2) {
(gdb)
```

> ***Примечание***
>
> Если выполняемая строка не содержит вызовов функций, то команды `step` и `next` выполняют одно и то же.

Посмотрим состояние стека вызовов:

```
(gdb) bt
#0  main (argc=2, argv=0xbfd945b4) at program2.c:12
(gdb)
```

Поскольку мы в данный момент находимся в функции `main()`, которая лежит на дне стека вызовов, то вывод команды `bt` содержит всего один фрейм с обозначением `#0`. Посмотрим теперь текущую информацию об аргументах программы:

```
(gdb) info args
argc = 2
argv = (char **) 0xbfd945b4
(gdb)
```

Адрес массива `argv` нам ни о чем не говорит, поэтому уточним первый аргумент программы при помощи команды `print`:

```
(gdb) print argv[1]
$1 = 0xbfd95244 "7"
(gdb)
```

Узнаем заодно значение переменной окружения `USER`:

```
(gdb) show env USER
USER = nn
(gdb)
```

Выполняем следующую инструкцию:

```
(gdb) step
17              a = 15;
(gdb)
```

Поскольку выражение в скобках оператора ветвления `if` является ложным, то программа сразу "перескочила" на строку 17. Эта строка еще не выполнена, и значение переменной `a` пока не установлено. В этом можно убедиться, если воспользоваться командой `print`:

```
(gdb) print a
$1 = -1208636528
(gdb)
```

Переменная `a` изменяется на протяжении всей программы. Полезно будет включить автоматический вывод ее значения на каждом шаге:

```
(gdb) display a
1: a = -1208636528
(gdb)
```

Сделаем следующий шаг:

```
(gdb) next
19              change_a (&a);
1: a = 15
(gdb)
```

Первая строка вывода говорит, что следующий шаг — вызов функции `change_a()` в строке 19. Вторая строка вывода — это отслеживаемое значение переменной `a`. Чтобы "войти" в функцию `change_a()`, нужно выполнить команду `step`:

```
(gdb) step
change_a (mya=0xbf9f8170) at program2.c:5
5               *mya += 10;
(gdb)
```

Обратите внимание, что функция `change_a()` не видит переменную `a`, поэтому отслеживание ее значения временно прекратилось. Давайте посмотрим состояние стека вызовов:

```
(gdb) backtrace
#0  change_a (mya=0xbf9f8170) at program2.c:5
#1  0x080484b2 in main (argc=2, argv=0xbf9f8214)
at program2.c:19
(gdb)
```

Теперь в стеке находится два фрейма. Посмотрим значение `mya`:

```
(gdb) print mya
$2 = (int *) 0xbf9f8170
(gdb)
```

Это адрес, который нам ни о чем не говорит. Разыменуем указатель:

```
(gdb) print *mya
$3 = 15
(gdb)
```

Другое дело! Команда `print` умеет работать с выражениями. Вот как это делается:

```
(gdb) print *mya * 2 + 17
$4 = 47
(gdb)
```

Сделаем следующий шаг:

```
(gdb) step
6       }
(gdb)
```

Мы перешли к точке возврата из функции `change_a()`. Посмотрим значение `mya`:

```
(gdb) print *mya
$5 = 25
(gdb)
```

Возвращаемся в `main()`:

```
(gdb) step
main (argc=2, argv=0xbf9f8214) at program2.c:21
21              printf ("Number: %d\n", a);
1: a = 25
(gdb)
```

Проверяем стек вызовов:

```
(gdb) where
#0  main (argc=2, argv=0xbf9f8214) at program2.c:21
(gdb)
```

Как и ожидалось, мы опять на дне стека. Выводим значение `a`:

```
(gdb) next
Number: 25
22              return atoi (argv[1]);
1: a = 25
(gdb)
```

И выходим из программы:

```
(gdb) continue
Continuing.

Program exited with code 07.
(gdb) quit
```

28.4. Точки останова

Наша экспериментальная программа настолько мала, что "проиграть" под отладчиком каждую ее строку нетрудно. Но в реальности программистам приходится отлаживать код, вмещающий тысячи строк.

Чтобы отладка программы не превращалась в рутинную процедуру исследования каждой строки исходного кода, программисты используют *точки останова* (breakpoints) и *точки слежения* (watchpoints).

Точка останова — это условная метка (строка исходного кода или функция), по достижении которой отладчик останавливает программу и ждет дальнейших указаний. Точка слежения — это выражение, при каждом изменении которого отладчик останавливает программу. Точек останова и точек слежения можно задать сколько угодно много.

Для работы с точками останова и слежения предусмотрены следующие команды отладчика gdb:

- `break` (`b` — краткая форма) — эта команда задает точку останова. Ее аргумент — имя функции или номер строки;
- `watch` — задает точку слежения, принимая в качестве аргумента выражение или имя переменной;
- `info break` — выводит информацию о точках останова;
- `info watch` — выводит информацию о точках слежения;
- `delete breakpoints` — удаляет все точки останова.

Запустим наш пример под отладчиком еще раз:

```
$ gdb -q
(gdb) file program2
Reading symbols from
/home/nn/main/work/bhv/src/ch28/program2/program2...done.
Using host libthread_db library
"/lib/libthread_db.so.1".
(gdb)
```

Зададим точку останова на инструкции вывода значения переменной `a`. Для этого придется сначала просмотреть исходный код, чтобы узнать номер строки:

```
(gdb) l 15,30
15              }
16
```

```
17              a = 15;
18
19              change_a (&a);
20
21              printf ("Number: %d\n", a);
22              return atoi (argv[1]);
23      }
(gdb) break 21
Breakpoint 1 at 0x80484b2: file program2.c, line 21.
(gdb)
```

Попробуем теперь запустить программу командой `run`:

```
(gdb) run 4
Starting program:
/home/nn/main/work/bhv/src/ch28/program2/program2 4

Breakpoint 1, main (argc=2, argv=0xbff5c784) at program2.c:21
21              printf ("Number: %d\n", a);
(gdb)
```

Программа дошла до строки 21 и остановилась. Чтобы закончить программу, введите команду `continue`:

```
(gdb) continue
Continuing.
Number: 25

Program exited with code 04.
(gdb)
```

Здесь следует отметить, что завершение программы не приводит к удалению точек останова. В этом легко убедиться:

```
(gdb) info break
Num Type       Disp Enb Address    What
1   breakpoint keep y   0x080484b2 in main at program2.c:21
        breakpoint already hit 1 time
(gdb)
```

Зададим еще одну точку останова, но уже для функции `change_a()`:

```
(gdb) break change_a
Breakpoint 2 at 0x8048447: file program2.c, line 5.
(gdb) info break
Num Type       Disp Enb Address    What
1   breakpoint keep y   0x080484b2 in main at program2.c:21
        breakpoint already hit 1 time
2   breakpoint keep y   0x08048447 in change_a at program2.c:5
(gdb)
```

Запустим программу еще раз:

```
(gdb) run 5
Starting program:
/home/nn/main/work/bhv/src/ch28/program2/program2 5

Breakpoint 2, change_a (mya=0xbfa98a10) at program2.c:5
5               *mya += 10;
(gdb)
```

Остановка произошла на первой инструкции функции `change_a()`. Если теперь дать команду `continue`, то программа не завершится, а остановится перед инструкцией вывода значения переменной `a`:

```
(gdb) continue
Continuing.

Breakpoint 1, main (argc=2, argv=0xbfa98ab4) at program2.c:21
21              printf ("Number: %d\n", a);
(gdb)
```

Повторный вызов `continue` завершит программу:

```
(gdb) continue
Continuing.
Number: 25

Program exited with code 05.
(gdb)
```

Добавим теперь в программу точку слежения, которая будет останавливать выполнение программы при каждом изменении переменной `a`. Но здесь следует учитывать, что добавление данной точки слежения возможно только при выполнении программы внутри функции `main()`, т. е. там, где видна отслеживаемая переменная. Чтобы нас не путали установленные ранее точки останова, удалим их:

```
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Теперь запустим программу, шагнем в функцию `main()` и зададим точку слежения:

```
(gdb) start 18
Breakpoint 5 at 0x8048456: file program2.c, line 9.
Starting program:
/home/nn/main/work/bhv/src/ch28/program2/program2 18
main (argc=Cannot access memory at address 0x0
) at program2.c:9
9       {
(gdb) step
```

```
main (argc=2, argv=0xbf9e0a04) at program2.c:12
12              if (argc < 2) {
(gdb) watch a
Hardware watchpoint 6: a
(gdb)
```

Проверяем:

```
(gdb) info watch
Num Type           Disp Enb Address    What
6   hw watchpoint  keep y              a
(gdb)
```

Запускаем выполнение:

```
(gdb) continue
Continuing.
Hardware watchpoint 6: a

Old value = -1208603760
New value = 15
main (argc=2, argv=0xbf9e0a04) at program2.c:19
19              change_a (&a);
(gdb)
```

Отладчик остановил выполнение программы перед строкой 19 и заботливо написал старое и новое значение переменной `a`. Продолжаем:

```
(gdb) c
Continuing.
Hardware watchpoint 6: a

Old value = 15
New value = 25
change_a (mya=0xbf9e0960) at program2.c:6
6       }
(gdb)
```

Следующая инструкция `continue` также останавливает программу, потому что функция `main()`, содержащая переменную `a`, завершается:

```
(gdb) c
Continuing.
Number: 25

Watchpoint 6 deleted because the program has left the block in
which its expression is valid.
0xb7e3c26b in exit () from /lib/libc.so.6
(gdb)
```

Теперь мы можем завершить программу и выйти из отладчика:

```
(gdb) c
Continuing.
```

```
Program exited with code 022.
(gdb) quit
```

Обратите внимание, что отладчик выводит код возврата в восьмеричной форме, о чем свидетельствует нулевой префикс числа.

28.5. Получение дополнительной информации

Дополнительную информацию по использованию отладчика gdb можно получить на соответствующих страницах руководства:

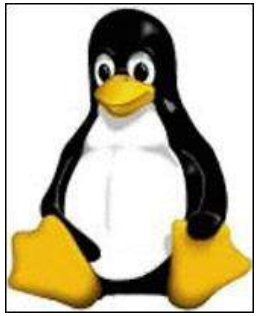
- man 1 gdb;
- info gdb.

Кроме того, сам отладчик содержит встроенную интерактивную справочную систему. Просто запустите отладчик и наберите команду `help`:

```
$ gdb -q
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution
without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name
for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Система помощи предложила целый ряд справочных разделов. Например, если вы хотите получить информацию о командах исследования стека, просто введите следующее "заклинание":

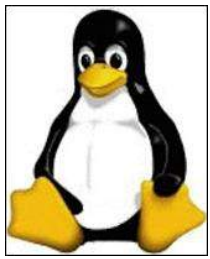
```
(gdb) help stack
```



ЧАСТЬ VII

Программирование в Linux на языке Python

ГЛАВА 29



Язык Python

Python — это высокоуровневый язык программирования, которому отдают предпочтение многие Linux-программисты. Язык Python совмещает в себе простоту, лаконичность и широту применения с обилием возможностей и высокой надежностью.

29.1. Несколько слов о языке Python

В конце 1980-х годов сотрудник Национального исследовательского института математики и информатики Нидерландов Гвидо ван Россум (Guido van Rossum) начал работу над усовершенствованием языка программирования ABC. Так появился язык Python, первая версия которого была опубликована в 1991 г.

Название "Python" к змеям никакого отношения не имеет. Создатель языка был поклонником многосерийного комедийного шоу "Monty Python's Flying Circus" на канале BBC, откуда и было взято название.

Python — это язык, включающий в себя три основные парадигмы программирования: *императивную*, *объектно-ориентированную* и *функциональную*. В этой книге мы не будем подробно рассматривать объектно-ориентированное и функциональное программирование, однако это нетрудно сделать самостоятельно, воспользовавшись свободно распространяемым руководством The Python Tutorial, которое доступно для чтения и скачивания по адресу **http://docs.python.org/py3k/tutorial/index.html**.

С точки зрения синтаксиса язык Python очень прост. При этом простота достигается не за счет примитивности, а за счет вычленения формализма. Иными словами, в языке Python отброшено почти все то, что не имеет непосредственного отношения к алгоритму программы.

Краткая философия языка Python изложена в небольшом документе под названием "The Zen of Python", который объявлен народным достоянием и доступен в Интернете по адресу **http://www.python.org/dev/peps/pep-0020/**.

Язык Python, несмотря на свою простоту, содержит множество причудливых конструкций, которые значительно сокращают размер программы, но в то же время выглядят весьма "опрятно". Например, одной строкой можно объявить сразу несколько переменных и присвоить им значения (листинг 29.1).

Листинг 29.1. Пример множественного присваивания посредством одной инструкции

```
a, b, c = 10, "Hello", -20
```

В синтаксисе Python, помимо прочего, есть одна особенность, которая поначалу обескураживает программистов, привыкших к языкам C, C++, Pascal или PHP. Речь идет о выделении блоков "отступами" (пробелами или табуляциями). *Блок* — это логически обособленная часть программы, относящаяся к циклу, условию, функции, классу и т. п. В листинге 29.2 показан пример использования вложенных блоков в языках C и C++.

Листинг 29.2. Демонстрация вложенных блоков в языках семейства C/C++

```
if (a == b)
{
        a = b + 1;
        if (c < d)
        {
                c = a + d;
        }
}
```

То же самое ветвление на языке Python будет выглядеть иначе (листинг 29.3).

Листинг 29.3. Использование отступов для выделения блоков в языке Python

```
if a == b:
        a = b + 1
        if c < d:
                c = a + d
```

Поначалу отказ от скобок или специальных операторов (например, `begin` и `end`) кажется странным. Но на практике исходный код становится более читаемым и компактным. Да и стиль языков, использующих специальные конструкции для выделения блоков, диктует те же самые отступы.

Синтаксис языка Python чувствителен к регистру символов (строчные и прописные буквы). Например, ключевое слово `for` не может писаться как `For` или `FOR`. Это также означает, что, например, из букв `A` и `a` можно сконструировать четыре независимых имени `aa`, `Aa`, `aA` и `AA`.

Сам язык Python отвечает за алгоритмизацию, а все расширения для работы в конкретных предметных областях доступны в виде библиотек. Подобно языкам C и C++ язык Python поставляется с собственной стандартной библиотекой The Python Standard Library, в которой содержатся механизмы общего назначения. Остальные библиотеки обычно доступны в дистрибутивах Linux в виде отдельных пакетов.

Язык Python является интерпретируемым. Программа, написанная на таком языке, исполняется другой программой — интерпретатором, а не компилируется в исполняемый код. Этот подход имеет как положительные, так и отрицательные стороны. Главный недостаток интерпретируемых программ — более медленное выполнение по сравнению с исполняемым кодом, однако различные методы оптимизации позволяют сократить разрыв в скорости до минимума.

С другой стороны, благодаря тому, что Python является интерпретируемым языком, процесс создания и отладки программ становится более быстрым и удобным. Помимо этого программа, написанная на Python, без какой-либо модификации будет работать в любой операционной системе, где есть интерпретатор языка. Единственное, что может этому помешать — отсутствие нужных библиотек. И наконец, интерпретируемые языки являются более динамичными, т. е. позволяют во время выполнения программы совершать такие действия, которые компилируемым программам не под силу. Но об этом речь пойдет позднее.

29.2. Инструментарий

В *главе 1* уже говорилось, что инструментарий — это совокупный набор средств, необходимый для создания программ с учетом поставленных задач. До сих пор при изучении материала этой книги мы использовали инструментарий языка C в Linux. Но для программирования на языке Python нужен другой "арсенал".

Сначала нам потребуется совсем немного: компьютер с операционной системой GNU/Linux, текстовый редактор, командная оболочка и интерпретатор языка Python. При решении более сложных задач в *главе 30* наш инструментарий немного расширится.

Для написания программ на языке Python подойдет любой текстовый редактор. Желательно, чтобы это был редактор с подсветкой синтаксиса Python, который распознает и выделяет графически различные элементы языка. К таким редакторам можно отнести Vim, Emacs, kate, gedit, jedit и многие другие, в изобилии присутствующие в современных дистрибутивах GNU/Linux.

Нам также понадобится интерпретатор языка Python версии не ниже 3.1. И здесь нужно сделать небольшое отступление, касающееся версий языка Python. Дело в том, что за многие годы существования языка у разработчиков накопилась "критическая масса" мелких претензий и пожеланий к определенным аспектам Python. В итоге, в 2008 г. была выпущена третья версия языка Python, не совместимая на уровне синтаксиса с предыдущей второй версией. Нельзя сказать, что язык был изменен до неузнаваемости — скорее, это был "косметический ремонт". А для пере-

хода со второй версии на третью достаточно "пропустить" программу через специальную утилиту, так что в ручном переписывании кода нет никакой необходимости.

Примечание

Третье поколение языка Python носит кодовое название Py3k или Py3000.

Тем не менее третья версия языка несовместима со второй. На момент сдачи в печать этой книги разработчиками Python официально поддерживаются две стабильные версии языка — 2.7.1 и 3.2. Однако сами разработчики Python рекомендуют использовать для новых программ именно третью версию языка, которая уже достаточно стабильна для полноценного использования. Единственным существенным поводом к тому, чтобы на время отложить переход на Py3k, может явиться отсутствие обновленных версий каких-нибудь важных библиотек.

Современные дистрибутивы GNU/Linux, следуя за разработчиками Python, также поддерживают обе версии языка. Таким образом, в вашей Linux-системе может быть интерпретатор второй версии Python, который запускается командой `python`, а также интерпретатор третьей версии, запускаемый командой `python3`. Для получения версии интерпретатора используется ключ `-V`:

```
$ python -V
Python 2.6.5
$ python3 -V
Python 3.1.2
```

Даже если по каким-либо причинам вы не можете получить доступ к третьей версии Python из вашего дистрибутива, вы можете скачать исходный код интерпретатора по адресу **http://python.org/download/**, а затем собрать и установить его самостоятельно, даже не имея прав доступа суперпользователя root.

29.3. Первая программа

Нашей первой программой будет традиционное приветствие "Hello World!", переделанное для разнообразия в наставление "Be yourself!" (Будь собой!). Для начала откройте текстовый редактор и наберите текст программы (листинг 29.4).

Листинг 29.4. Программа hello1

```
print("Be yourself!")
```

Теперь сохраните файл под именем hello1.py. Это исходный файл программы, а его содержимое — исходный код. Для исходных файлов на языке Python используется расширение py.

Поскольку Python является интерпретируемым языком, компиляция программы не требуется. Иными словами, исходный код является готовой программой. Нам остается только передать этот код интерпретатору языка. Для этого в командной обо-

лочке сделайте текущим тот каталог, в котором находится файл hello1.py, и введите следующую команду:

```
$ python3 hello1.py
Be yourself!
```

Вообще говоря, нам не обязательно было заносить исходный код программы в файл. Интерпретатор Python поддерживает так называемый интерактивный режим. Иными словами, мы можем запустить интерпретатор Python, вводя в него команды вручную. Далее показан пример ввода команд в интерактивном режиме.

```
$ python3
Python 3.1.2 (r312:79147, Apr 22 2010, 16:15:31)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Be yourself!")
Be yourself!
>>>
```

Для выхода из интерактивного режима интерпретатора Python обратно в командную оболочку нажмите комбинацию клавиш <Ctrl>+<D>.

Работа с интерпретатором Python в интерактивном режиме может оказаться полезной в определенных случаях, но широкого применения этот подход не имеет. Поэтому далее мы будем записывать наши программы в файлы и передавать их интерпретатору в качестве аргумента.

> ***Примечание***
>
> Интерактивный режим языка Python можно использовать в качестве калькулятора. Вы можете вводить сложные выражения, например `(48+52)*16.5 – (155+17.3)`, при этом нет нужды использовать функцию `print()` для вывода результата. Просто введите выражение и нажмите клавишу <Enter>.

Итак, вернемся к исходной программе. Функция `print()` выводит на экран свой аргумент — строку "Be yourself!". В языке Python строки помещаются в одинарные или двойные кавычки; об этом речь пойдет позже. Мы можем также передать функции `print()` несколько аргументов, отделенных друг от друга запятыми. В этом случае каждый аргумент будет выводиться на экран через пробел. В листинге 29.5 показана модифицированная версия нашей исходной программы.

Листинг 29.5. Программа hello2

```
print("Be", "yourself!")
```

Результат работы программы будет тем же, что и в первом случае:

```
$ python3 hello2.py
Be yourself!
```

Функция `print()` автоматически добавляет в конец вывода символ переноса строки. Мы можем убедиться в этом, если разобьем нашу программу на две инструкции, как показано в листинге 29.6.

Листинг 29.6. Программа hello3

```
print("Be")
print("yourself!")
```

В результате слово "yourself" и восклицательный знак окажутся на новой строке:

```
$ python3 hello3.py
Be
yourself!
```

Если вам не нравится то, что функция без вашего ведома дополняет вывод переносом строки, вы можете при помощи именованного аргумента `end` функции `print()` указать то, чем будет дополняться вывод (листинг 29.7).

Листинг 29.7. Программа hello4

```
print("Be", end=' ')
print("yourself!")
```

Как видим, все слова опять оказались на одной строке:

```
$ python3 hello4.py
Be yourself!
```

29.4. Структура программы

Исходный код программы состоит из инструкций и комментариев. В языке Python инструкция — это не являющаяся комментарием строка исходного кода или несколько соединенных строк. Естественно, пустая строка, как и комментарий, инструкцией не является. Чтобы объединить две строки в одну инструкцию, в конце незавершенной первой строки ставят символ обратной косой черты (бэкслэш). Такой подход часто называют *конкатенацией строк исходного кода*. Мы уже делали это в *главе 4*, когда работали с make-файлами. Следует также отметить, что при конкатенации строк исходного кода между ними автоматически вставляется пробел.

Чтобы не путаться в дальнейшем, давайте определимся с формулировками. В английском языке, откуда берется большинство терминов, связанных с программированием, строка исходного кода обозначается как line, а текстовая строка (например, "Be yourself!") определяется словом string. В русском языке за оба значения отвечает слово "строка", поэтому, чтобы не путаться, далее слово line будем переводить как "строка исходного кода", а string будем называть просто "строка" или "обычная строка".

Рассмотрим теперь пример, демонстрирующий конкатенацию строк исходного кода (листинг 29.8).

Листинг 29.8. Программа hello5

```
print(\
"Be youself!")
```

Вот что получилось:

```
$ python3 hello5.py
Be youself!
```

Аналогичным образом можно разрывать обычные строки, как показало в листинге 29.9.

Листинг 29.9. Программа hello6

```
print("Be \
yourself!")
```

Вывод будет тем же, что и в предыдущем случае. Следует отметить, что когда разрываются обычные строки, пробел между расчлененными частями не вставляется. Следующий пример демонстрирует это (листинг 29.10).

Листинг 29.10. Программа hello7

```
print("Be\
yourself!")
```

Вот результат:

```
$ python3 hello7.py
Beyourself!
```

С технической точки зрения нет необходимости разделять инструкции на несколько строк исходного кода. Однако в среде программистов на языке Python использование в исходном коде слишком длинных строк (свыше 79 символов) считается плохим стилем. Во-первых, слишком длинные строки делают программу трудночитаемой. Во-вторых, при миграции исходного кода из одной программы в другую (например, из одного текстового редактора в другой или из Web-браузера в текстовый редактор) может получиться, что слишком длинная строка будет автоматически разбита на две коротких, но без связующего символа обратной косой черты. Это неизбежно приведет к ошибкам при запуске программы.

В языке Python используются однострочные комментарии, которые начинаются с символа # (решетка) и действуют до конца текущей строки исходного кода. Комментарии используются в двух случаях: для разъяснения смысла отдельных участков исходного кода, а также для комментирования самого исходного кода при отладке. В первом случае программист объясняет себе (в качестве напоминания), а также всем тем, кто будет читать исходный код, что означают те или иные инструкции. Это вносит ясность в исходный код и помогает лучше его структурировать.

Комментарии игнорируются интерпретатором Python, как если бы их вообще не было в исходном коде. Этим удобно пользоваться при отладке программ, когда перед строкой исходного кода ставится символ комментария, чтобы на время сделать инструкцию "невидимой" для интерпретатора. Следует также отметить, что комментарий нельзя расчленить на две строки исходного кода при помощи бэкслэша, как мы это делали ранее. Чтобы написать многострочный комментарий, на каждой строке исходного кода ставится новый символ решетки. И, наконец, комментарий нельзя разместить внутри обычной строки. Следующий пример (листинг 29.11) демонстрирует использование комментариев.

Листинг 29.11. Программа comment1

```
# This program prints "Be yourself!"
print("Be ", end='')
print("yourself!") # This statement prints string "World"
# print("Goodbye!")
```

В первой строке исходного кода содержится комментарий, разъясняющий, что, собственно, делает эта программа. В третьей строке кода комментарий располагается вслед за инструкцией, объясняя ее назначение. И, наконец, последняя строка исходного кода — это закомментированная инструкция, которая будет игнорироваться компилятором до тех пор, пока мы не уберем символ комментария. Вот что выводит программа:

```
$ python3 comment1.py
Be yourself!
```

Существует несколько правил в отношении комментариев, которые не обязательны к соблюдению, но, тем не менее, позволяют придерживаться хорошего стиля программирования на языке Python. Хороший стиль программирования делает исходный код легко читаемым, легко модифицируемым, а также позволяет избегать многих потенциальных проблем. Эти правила, по большей части, применимы и для других языков программирования.

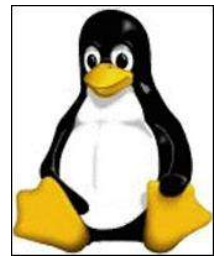
Итак, первое правило гласит, что комментарии должны писаться на английском языке, особенно если речь идет о программе с открытым исходным кодом. Вообще говоря, даже если программа изначально нацелена на русскоязычного пользователя, в исходном коде все равно не должно быть русского языка. Это касается не только комментариев, но и таких вещей, как интерфейс программы, сообщения об ошибках и т. п. Чтобы "заставить" программу говорить на каком-нибудь языке, например по-русски, существуют различные инструменты для интернационализации и локализации программ, которые и следует использовать. Это правило часто нарушается, что обычно имеет негативные последствия.

Второе правило касается избыточного комментирования. Комментарии должны появляться в исходном коде лишь там, где они действительно нужны. Если в исходном коде слишком много комментариев, то это только усложняет восприятие.

У этой медали есть еще одна сторона. Придерживаясь правила комментировать только то, что может вызвать вопросы, вы все равно можете оказаться в море "объяснительных записок". Обычно такой результат свидетельствует о том, что в программе используются слишком сложные подходы к решению задач, нуждающиеся в упрощении.

И третье правило говорит о том, что короткие комментарии, относящиеся к одной инструкции, лучше размещать на одном уровне с этой инструкцией. Если же комментарий касается группы инструкций или слишком длинный, чтобы поместиться на одной строке вместе с инструкцией, то его следует помещать над объектом комментирования, а не под ним.

ГЛАВА 30



Типы данных

Язык Python является *типизированным*. Это означает, что все данные, которыми оперирует программа, относятся к какому-либо типу. Вдобавок ко всему, Python является объектно-ориентированным языком программирования, что позволяет создавать собственные типы данных. Однако в основе всего лежат предопределенные, стандартные типы данных.

Python содержит богатый набор стандартных типов данных. Сравнивая различные версии языка, нетрудно заметить, что этот набор мутирует во времени. Одни типы данных появляются, другие исчезают, а некоторые объединяются в один общий тип. Чтобы не запутаться во всем этом разнообразии, мы рассмотрим лишь наиболее полезные и часто используемые типы данных. Вообще говоря, здесь важно понять именно основополагающие концепции, а не загромождать ум горой названий и справочных данных, которые и так всегда доступны. Например, умея оперировать целыми числами, вы без труда освоите тип данных, отвечающий за комплексные числа, если они вам понадобятся.

В основе языка Python лежит парадигма *объектно-ориентированного программирования*, а это значит, что типы данных выступают здесь не только в виде контейнеров, но и как набор операций над данными, которые находятся в этих контейнерах. То есть тип данных описывает не только сами данные, но и то, что с ними можно сделать. В основе парадигмы объектно-ориентированного программирования лежит также концепция *полиморфизма*, сутью которой является различное поведение одних и тех же операций в зависимости от типа данных, над которыми производятся операции.

В этой главе будут рассматриваться следующие стандартные типы данных:

- Целые числа.
- Числа с плавающей точкой.
- Строки.
- Списки.

30.1. Переменные

Переменные — это имена, через которые мы обращаемся к данным. Python является языком программирования с динамической типизацией, а это означает, что к переменным изначально не привязывается тип данных. Иными словами, тип переменной определяется типом данных, на которые эта переменная ссылается в данный момент: одна и та же переменная может быть, к примеру, сначала числом, затем строкой.

Имя переменной может содержать буквы латиницы, цифры и знаки подчеркивания. Имя переменной не может начинаться с цифры и не должно совпадать с ключевыми словами языка Python: `False`, `None`, `True`, `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with` и `yield`.

Вот несколько примеров недопустимых имен переменных:

- `38parrots` (имя начинается с цифры);
- `rock-n-roll` (имя содержит дефис);
- `pass` (имя является ключевым словом языка).

В языке Python переменная создается в момент присваивания ей первого значения операцией равенства `=`. В листинге 30.1 показан пример создания и использования переменной.

Листинг 30.1. Программа var1

```
s = "Be yourself!"
print(s)
```

Здесь все просто. Мы создали переменную `s` и присвоили ей строку `"Be yourself!"`, а затем передали эту переменную функции `print()` в качестве аргумента. Вот что получилось:

```
$ python3 var1.py
Be yourself!
```

Переменная, оправдывая свое название, может менять значение во время выполнения программы, т. е. меняться. Кроме того, как уже сообщалось ранее, тип присваемого значения может быть произвольным. Рассмотрим следующий пример (листинг 30.2).

Листинг 30.2. Программа var2

```
a = "Hello"
print(a)
a = 125
print(a)
```

В этом примере переменной a сначала было присвоено строковое значение, а потом — целое число. Вот что получилось:

```
$ python3 var2.py
Hello
125
```

Можно также создать несколько переменных одновременно, присвоив им одно и то же значение, как показано в листинге 30.3.

Листинг 30.3. Программа var3

```
a = b = c = 5
print(a, b, c)
```

Результат очевиден:

```
$ python3 var3.py
5 5 5
```

Язык Python также способен на различные "фокусы", недоступные во многих других языках программирования. Например, мы можем одной инструкцией объявить несколько переменных и присвоить им значения. Аналогичным образом, одной лишь инструкцией мы можем поменять значения двух переменных. В следующем примере демонстрируются эти две возможности (листинг 30.4).

Листинг 30.4. Программа var4

```
a, b = 7, 8
print(a, b)
a, b = b, a
print(a, b)
```

Результат работы программы такой:

```
$ python3 var4.py
7 8
8 7
```

30.2. Целые числа

Целочисленный тип описывает положительные и отрицательные недробные числа в широком диапазоне. Конкретные пределы значений могут зависеть от архитектуры компьютера, от версии интерпретатора и даже от количества свободной оперативной памяти.

Для работы с целыми числами чаще всего используются следующие операции:

- сложение: a+b;
- вычитание: a-b;

- умножение: `a*b`;
- целочисленное деление: `a//b`;
- остаток от деления: `a%b`;
- модуль: `abs(a)`;
- возведение в степень: `a**b` или `pow(a,b)`.

Следующий простой пример (листинг 30.5) демонстрирует операции над целыми числами.

Листинг 30.5. Программа num1

```
print(10+20, 10-15, 5*2, 13//7, 13%7)
print(abs(5), abs(-5), pow(2,3), 2**3)
```

Вот что выводит эта программа:

```
$ python3 num1.py
30 -5 10 1 6
5 5 8 8
```

Обратите внимание на то, как работает целочисленное деление. Это не деление с последующим округлением до целого. Результатом деления 13 на 7 является дробное число 1,857 (округлено до тысячных). Если округлить это число до целых, получится 2. Но результатом целочисленного деления будет именно 1. Таким образом, целочисленное деление — это изъятие целой части от деления. При решении конкретных задач важно помнить об этом. Иногда программисты используют целочисленное деление там, где нужно применять дробное деление с округлением результата до целых. Это чревато получением очень большой погрешности, которая еще и непостоянна.

> ***Примечание***
>
> Приоритет операций над целыми числами в языке Python такой же, как и в обычной арифметике. Например, умножение и деление выполняется раньше, чем сложение и вычитание. Для переопределения приоритета, а также для формирования математических выражений любой сложности используются скобки. Следует отметить, что скобки можно использовать просто для внесения ясности в сложное выражение, даже тогда, когда переопределение приоритета не требуется.

Вместо явных чисел мы можем использовать переменные. Следующий пример (листинг 30.6) демонстрирует вычисление 99-го члена арифметической прогрессии с первым членом 1 и шагом 8.

Листинг 30.6. Программа num2

```
a1, d, n = 1, 8, 99
an = a1+(n-1)*d
print("n:", n, ", an:", an)
```

Вот что выводит эта программа:

```
$ python3 num2.py
n: 99 , an: 785
```

Обратите внимание на то, как в программе отформатирован вывод: функции `print()` передается четыре аргумента (строка, число, строка, число), разделенные запятыми. При выводе аргументы автоматически отделяются друг от друга пробелами.

30.3. Числа с плавающей точкой

О *числах с плавающей точкой* (floating point numbers) можно говорить очень долго. Их эволюция в истории вычислительной техники проходила много этапов. Существуют также различные представления чисел с плавающей точкой, а вопросы, связанные с диапазоном значений и точностью, нередко становятся объектами научных диссертаций. По этой теме доступно много информации, поэтому не будем глубоко погружаться в теорию, а обозначим лишь базовые понятия, необходимые нам для практического использования чисел с плавающей точкой.

Итак, числа с плавающей точкой — это наиболее популярная на сегодняшний день абстракция действительных (вещественных) чисел в вычислительной технике. Современные компьютеры позволяют оперировать огромными количествами информации, но, тем не менее, мы все равно находимся в ограничениях. Например, доказанная Иоганном Ламбертом иррациональность числа π означает невозможность точного представления последнего в вычислительной технике; мы можем говорить лишь о некоторой точности таких представлений.

Идея чисел с плавающей точкой заключается в повышении точности представления и экономии памяти, необходимой для их хранения. Вместо того чтобы выделять отдельные фиксированные ячейки под целую и дробную части (числа с фиксированной точкой), числа с плавающей точкой задаются в виде *мантиссы* и *порядка*. Мантисса — это что-то вроде большого целого числа, которое умножается на 10 в определенной степени. Эта степень и есть порядок. Таким образом, изменяя порядок, мы заставляем точку "плавать" по мантиссе. Отсюда числа с плавающей точкой и берут свое название.

Приведенное объяснение не является абсолютно точным и содержит много "но", однако, несмотря на это, позволяет понять основную идею чисел с плавающей точкой. В языке Python, как и в большинстве других языков программирования, целая часть числа отделяется от дробной не запятой, как принято в России, а точкой.

В языке Python числа с плавающей точкой представляются в форме `x.y`, где `x` — целая часть, `y` — дробная часть. Если целая или дробная часть равна нулю, то ее можно опустить. Например, .72 аналогично 0.72, а 2. аналогично 2.0. Казалось бы, зачем писать точку после двойки, если дробная часть нулевая? Но тут следует вспомнить про полиморфизм, о котором вкратце говорилось в *разд. 30.1*: одни и те

же операции могут выполнять над разными типами данных различные действия. Примером тому может послужить операция деления: мы уже говорили о том, что целочисленное деление обладает рядом особенностей.

Числа с плавающей точкой поддерживают все те же операции, что и целые числа. Только операция деления в данном случае возвращает обычное частное, а не только его целую часть. Хотя, для целочисленного деления чисел с плавающей точкой предусмотрена отдельная операция `a//b`. Следующий пример (листинг 30.7) демонстрирует операции над числами с плавающей точкой.

Листинг 30.7. Программа float1

```
print(10.5+20.3, -7.5+0.19, 2.0*3.4)
print(13./7., 13.0%7.0, 13.0//7.0)
print(abs(-0.5), pow(9.0,0.5), 3.2**2.)
```

Вот что получилось:

```
$ python3 float1.py
30.8 -7.31 6.8
1.85714285714 6.0 1.0
0.5 3.0 10.24
```

В арифметических операциях числа с плавающей точкой можно использовать совместно с целыми числами. В этом случае используются именно операции, относящиеся к числовому типу с плавающей точкой, и результат выражения будет тоже с плавающей точкой. В листинге 30.8 показан предыдущий пример, переделанный таким образом, чтобы числа с плавающей точкой использовались вместе с целыми числами.

Листинг 30.8. Программа float2

```
print(10+20.3, -7+0.19, 2*3.4)
print(13./7, 13%7.0, 13.//7)
print(pow(9, 0.5), 3.2**2)
```

Как видим, результат каждой операции — число с плавающей точкой:

```
$ python3 float2.py
30.3 -6.81 6.8
1.85714285714 6.0 1.0
3.0 10.24
```

При совместном использовании чисел с плавающей точкой и целых чисел в сложных выражениях важно не допустить одну часто встречающуюся, но трудноуловимую ошибку. Если в выражении присутствуют числа с плавающей точкой, это вовсе не значит, что целочисленные операции исключены. Следующий пример (листинг 30.9) демонстрирует эту ошибку.

Листинг 30.9. Программа float3

```
pi, r = 3.14159, 20.0
v1 = (4//3)*pi*pow(r, 3)
v2 = (4./3.)*pi*pow (r, 3)
print("volume1 =", v1)
print("volume2 =", v2)
```

В этом примере мы вычисляем объем шара радиусом 20 единиц, используя два похожих выражения. В итоге получаем два совсем разных результата:

```
$ python3 float3.py
volume1 = 25132.72
volume2 = 33510.2933333
```

В первом выражении не было учтено, что делимое и делитель — целые числа. Следовательно, результатом деления будет просто единица, которая уже в дальнейшем умножается на π и радиус в кубе. В итоге мы получили просто недопустимую погрешность, о чем и говорилось в *разд. 30.2*.

30.4. Строки

Вы уже знаете, что строки можно формировать, заключая их содержимое в двойные кавычки. Аналогичным образом можно использовать строки, заключенные в одинарные кавычки. Разница между двумя представлениями лишь в том, что строки из двойных кавычек могут содержать внутри себя одинарные, и наоборот. В следующем примере (листинг 30.10) показано, как это происходит.

Листинг 30.10. Программа str1

```
print("Be 'yourself'!")
print('Be "yourself"!')
```

Вот результат работы программы:

```
$ python3 str1.py
Be 'yourself'!
Be "yourself"!
```

Возникает вопрос: а что делать, если в одной строке требуется наличие как одинарных, так и двойных кавычек? Для этих целей используются специальные последовательности, которые встраиваются в строку и преобразовываются интерпретатором в нечто, что нам нужно, точно так же, как это делается в языках C и C++. Каждая последовательность начинается с символа обратной косой черты (бэкслэш). Приведем список наиболее часто используемых последовательностей:

- одинарная кавычка `\'`;
- двойная кавычка `\"`;
- перенос строки `\n`;

- табуляция `\t`;
- бэкслэш `\\`.

Следующий пример (листинг 30.11) демонстрирует использование специальных последовательностей в строках.

Листинг 30.11. Программа str2

```
s = "\tTutorial\nYou may insert \', \", and \\ into string."
print(s)
```

Вот результат работы программы:

```
$ python3 str2.py
        Tutorial
You may insert ', ", and \ into string.
```

В языке Python предусмотрен даже тот случай, когда надо оставить специальные последовательности как есть, не преобразовывая их в соответствующие символы. Для этого используются *сырые строки* (raw strings). Для формирования сырой строки перед открывающейся кавычкой добавляют символ `r`. В следующем примере (листинг 30.12) показано, как это работает.

Листинг 30.12. Программа str3

```
print(r'Leave us\n untouched')
```

В результате последовательность `\n` осталась нетронутой:

```
$ python3 str3.py
Leave us\n untouched
```

Для полного удобства в языке Python предусмотрены также длинные строки, которые заключаются в последовательности из трех одинарных или из трех двойных кавычек. Эти строки имеют ряд особенностей:

- Их можно разносить на несколько строк исходного кода, причем символ переноса строки будет добавляться автоматически.
- Внутри них можно использовать одинарные и двойные кавычки, не прибегая к помощи специальных последовательностей.
- В отличие от сырых строк в них обрабатываются последовательности переноса строки, табуляции и бэкслэша.

В листинге 30.13 показан пример использования длинной строки.

Листинг 30.13. Программа str4

```
print("""Menu:
\t- "Black" tea;
\t- 'Arabica' coffee;
\t- Cake\\Ice-cream.""")
```

Вот что выводит эта программа:

```
$ python3 str4.py
Menu:
        — "Black" tea;
        — 'Arabica' coffee;
        — Cake\Ice-cream.
```

Стандартный строковый тип в языке Python содержит большое количество операций. Вот список часто используемых операций над строками:

- ❒ слияние строк (конкатенация): `s1+s2`;
- ❒ умножение (повторение строки n раз): `s*n`;
- ❒ получение длины строки: `len(s)`;
- ❒ преобразование строки в целое число: `int(s)`;
- ❒ преобразование строки в число с плавающей точкой: `float(s)`;
- ❒ извлечение символа с индексом n: `s[n]`;
- ❒ извлечение подстроки между позициями `n` и `m-1` (включительно): `s[n:m]`;
- ❒ извлечение первых `n` символов строки: `s[:n]`;
- ❒ извлечение всех символов, кроме `n` первых: `s[n:]`;
- ❒ извлечение n-го символа от конца строки: `s[-n]`;
- ❒ извлечение n последних символов строки: `s[-n:]`;
- ❒ извлечение всех символов строки, кроме n последних: `s[:-n]`.

Перед рассмотрением конкретных примеров с операциями над строками важно уяснить некоторые концептуальные моменты.

Строка — это набор символов (characters), каждый из которых имеет свой номер, называемый *индексом*. Нумерация символов начинается с нуля. Таким образом, если строка состоит из 5 символов, то первый символ будет иметь индекс 0, второй символ — индекс 1 и т. д. до последнего символа, который будет располагаться под номером 4. В итоге индекс последнего символа строки всегда равен длине строки минус 1. Как видите, в этом отношении строки Python ничем не отличаются от строк языков C и C++.

Второй концептуальный момент заключается в том, что в языке Python (в отличие от языков C и C++) нет специального типа для работы с отдельными символами. Любой отдельно извлеченный символ — это строка единичной длины.

Также следует знать, что операция присваивания (знак равенства) не предназначена для изменения конкретных частей строки. Единственное назначение этой операции при работе со строками — присвоение переменной нового значения. Для модификации строки используются специальные операции, о которых речь пойдет далее.

Конкретные значения строкового типа (то, что мы помещаем в кавычки) называются строковыми константами (string literals). Например, `"Be yourself!"` является строковой константой, а некоторая переменная `s` или результат какой-нибудь из

перечисленных ранее операций строковой константой являться не будет. Если несколько строковых констант следуют друг за другом, то автоматически происходит их конкатенация. Наличие удобной операции сложения строк делает указанную возможность практически ненужной, однако лучше о ней знать, чтобы не удивляться, встретив ее в чужом коде. Следующий пример (листинг 30.14) демонстрирует конкатенацию строковых констант.

Листинг 30.14. Программа str5

```
print("Bob's " 'dog')
```

В результате получим:

```
$ python3 str5.py
Bob's dog
```

В следующем примере (листинг 30.15) демонстрируются операции над строками.

Листинг 30.15. Программа str6

```
s = "Be"
print(s+" yourself!")
print(s*3)
print(len(s))
print(int("1024"))
print(float("3.14"))
print(s[1])
print(s[1:4])
print(s[:4])
print(s[-4:])
```

А вот результат работы этой программы:

```
$ python3 str6.py
Be yourself!
BeBeBe
2
1024
3.14
e
e
Be
Be
```

Возвращаемыми значениями перечисленных строковых операций, кроме операций `len()`, `int()` и `float()`, являются строки. Над этими строками также можно выполнять какие-нибудь операции, формируя тем самым сложные выражения. Следующий пример (листинг 30.16) демонстрирует причудливое преобразование одной строки в другую.

Листинг 30.16. Программа str7

```
s = "Humpty Dumpty sat on a wall"
sresult = s[0]+"e" + s[-2:]+s[-9]
sresult = sresult+' W' + sresult[len(sresult)-1]
print(sresult+"r"+s[-1]+"d!")
```

В результате получилось старое знакомое приветствие:

```
$ python3 str7.py
Hello World!
```

Обратите внимание на предпоследнюю строку исходного кода. Это один из видов рекурсии: переменная использует сама себя для переопределения значения, так же, как и в языках C и C++. Итак, если вы поняли, как работает последний пример, то можете считать тему строк пройденной.

30.5. Списки

Списки — это тип, предназначенный для хранения обработки наборов (массивов) каких-либо данных. Каждая единица данных в списке называется элементом списка и имеет свой номер, называемый индексом. Нумерация элементов списка начинается с нуля.

Приятной особенностью языка Python является то, что элементы одного списка могут иметь различный тип. Как и в случае со строками, списки располагают солидным набором операций.

Значением списка является набор элементов любого типа, заключенный в квадратные скобки. Элементы отделяются друг от друга запятыми. Запись `[]` обозначает пустой список. Списки в языке Python можно целиком и в удобной форме выводить на экран функцией `print()`. В листинге 30.17 показан пример создания списка.

Листинг 30.17. Программа list1

```
lst = [10, 20.5, "Hello"]
print(lst)
```

Вот что получилось:

```
$ python3 list1.py
[10, 20.5, 'Hello']
```

При работе со строками операция присваивания (знак равенства) используется только для занесения в переменную нового значения, но не для модификации строки. Однако при работе со списками оператор присваивания позволяет работать с конкретными элементами. Вот список часто используемых операций над списками:

- получение n-го элемента списка: `lst[n]`;
- сложение списков (продолжение вторым первого): `lst1+lst2`;

- получение длины списка (количество элементов): `len(lst)`;
- удаление элементов между позициями `n` и `m-1` (включительно): `lst[n:m]=[]`;
- замена элементов между позициями `n` и `m-1` (включительно): `lst[n:m]=[a1, a2, ...]`;
- вставка элементов между позициями `n-1` и `n`: `lst[n:n]=[a1, a2,...]`;
- извлечение части списка между позициями `n` и `m-1` (включительно): `lst[n:m]`;
- извлечение первых n элементов списка: `lst[:n]`;
- извлечение всех элементов списка, кроме n первых: `lst[n:]`;
- извлечение n-го элемента от конца списка: `lst[-n]`;
- извлечение n последних элементов списка: `lst[-n:]`;
- извлечение всех элементов списка, кроме n последних: `lst[:-n]`.

Следующий пример (листинг 30.18) демонстрирует перечисленные операции над списками.

Листинг 30.18. Программа list2

```
l = [1, 2, 3, "hello"]
l = l + [4, 5, "world"]
print(l)
print (len(l), l[0], l[3])
l[1:3] = []
print(l)
l[1:3] = ["begin", "end"]
print(l)
l[2:2] = [3, 4]
print(l)
print(l[1:5])
print(l[:2])
print(l[2:])
print(l[-2])
print(l[-2:])
print(l[:-2])
```

А вот результат работы этой программы:

```
$ python3 list2.py
[1, 2, 3, 'hello', 4, 5, 'world']
7 1 hello
[1, 'hello', 4, 5, 'world']
[1, 'begin', 'end', 5, 'world']
[1, 'begin', 3, 4, 'end', 5, 'world']
['begin', 3, 4, 'end']
[1, 'begin']
```

```
[3, 4, 'end', 5, 'world']
5
[5, 'world']
[1, 'begin', 3, 4, 'end']
```

Элементами списков могут быть не только числа или строки, но и сами списки. Такой подход позволяет произвольным образом формировать многомерные наборы данных. Следующий пример (листинг 30.19) показывает, как это работает на практике.

Листинг 30.19. Программа list3

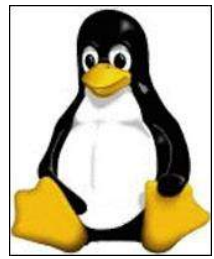
```
lst1 = ["a", "b", "c"]
lst2 = [1, 2, 3]
lst = ["hello", lst1, lst2, "goodbye"]
print(lst)
print(len(lst), len(lst[1]), len(lst[2]))
print(lst[1][0], lst[2][0])
lst[1][2] = "C"
print(lst)
lst[2][0], lst[2][1] = "one", "two"
print(lst)
```

И вот что получилось:

```
$ python3 list3.py
['hello', ['a', 'b', 'c'], [1, 2, 3], 'goodbye']
4 3 3
a 1
['hello', ['a', 'b', 'C'], [1, 2, 3], 'goodbye']
['hello', ['a', 'b', 'C'], ['one', 'two', 3], 'goodbye']
```

В приведенном примере в список `lst` были "внедрены" списки `lst1` и `lst2` в качестве членов.

ГЛАВА 31



Программирование на языке Python

В этой главе будут рассмотрены практические аспекты программирования на языке Python:

- Обработка ошибок.
- Использование логических операций в ветвлениях.
- Циклы и функции.

31.1. Логические операции

В языке Python предусмотрены три логические (булевы) операции:

- логическое И (конъюнкция) `a and b`;
- логическое ИЛИ (дизъюнкция) `a or b`;
- логическое НЕ (отрицание) `not a`.

Результатом логической операции является одно из двух значений: истина `True` или ложь `False`.

Операция `and` возвращает истину лишь в том случае, если оба аргумента истинные. В противном случае возвращается ложь. Таким образом:

```
True and True = True
True and False = False
False and True = False
False and False = False
```

Операция `or` возвращает `False` лишь в том случае, если оба аргумента ложные. Во всех остальных случаях возвращается `True`:

```
True or True = True
True or False = True
False or True = True
False or False = False
```

И наконец, операция `not` возвращает `False` при истинном аргументе, и наоборот:

```
not True = False
not False = True
```

Логические операции можно использовать совместно друг с другом и со скобками, формируя тем самым логические выражения любой сложности. В языке Python существует также отдельный логический тип, позволяющий хранить и обрабатывать логические данные. В следующем примере (листинг 31.1) демонстрируется работа с логическими данными.

Листинг 31.1. Программа bool1

```
a, b = True, False
print(not((a and b) or not(a or b)))
```

Вот результат работы программы:

```
$ python3 bool1.py
True
```

Основное назначение логических операций в языке Python состоит не в решении логических задач, а в совместном использовании с операциями сравнения. Операции сравнения возвращают истину или ложь, сравнивая значения различных типов данных. Полученные логические значения используются в специальных конструкциях языка (ветвления, циклы и т. п.), которые позволяют изменять поведение программы в зависимости от текущих условий. В языке Python используются следующие операции сравнения:

- `a == b` — a равно b;
- `a < b` — a меньше b;
- `a > b` — a больше b;
- `a <= b` — a меньше или равно b;
- `a >= b` — a больше или равно b;
- `a != b` — a не равно b;
- `a is b` — a имеет тот же тип, что и b;
- `a is not b` — a не того же типа, что b.

Операции сравнения применимы не только к числам, но и к строкам и даже к спискам, хотя последние почти никогда не сравниваются целиком. Что касается строк, то тут все просто: одна строка больше другой в том случае, если в отсортированном по алфавиту возрастающем списке она будет стоять дальше. Например:

```
"Иванов" < "Петров"
"Иванов" > "Иваненко"
"Иван" < "Иванов"
"иванов" < "Иван"
```

Следующий пример (листинг 31.2) демонстрирует совместное использование операций сравнения и логических операций.

Листинг 31.2. Программа bool2

```
a, b = 10, 20.5
print(a<b, not a<b)
print((a<=15) and (b>=17))
b = 10.0
print(a == b, a is b)
b = 10
print(a is b)
```

Вот результат работы программы:

```
$ python3 bool2.py
True False
True
True False
True
```

Как видно из примера, сравнивать можно значения разных типов. Обратите внимание на операцию `is`: она возвращает истину лишь в том случае, если типы значений или переменных совпадают.

Приоритет операций сравнения выше, чем у логических операций. Иными словами, сначала выполняются сравнения, а потом уже логические операции. Однако в сложных выражениях бывает полезным использование скобок даже там, где это технически не обязательно. Скобки позволяют не только переопределять приоритет, но и делают выражение более ясным для понимания.

31.2. Сообщения об ошибках

Если вы проверяли на практике изложенные в предыдущих разделах или свои собственные примеры, то наверняка допускали опечатки или неточности, приводившие к сообщениям об ошибках. Если же ошибок не было, то остается только позавидовать вашей внимательности.

В языке Python сообщения об ошибках делятся на два вида: *синтаксические ошибки* (syntax errorrs) и *исключения* (exceptions). Синтаксические ошибки выявляются еще до запуска программы и означают, что программа нарушает правила языка Python и не может быть выполнена. Рассмотрим пример (листинг 31.3), в котором умышленно содержится синтаксическая ошибка.

Листинг 31.3. Программа err1

```
print("Be ")
prind "yourself!")
```

Суть ошибки видна невооруженным глазом: упущена скобка перед списком аргументов функции `print()`. При попытке интерпретации этой программы мы получим сообщение об ошибке:

```
$ python3 err1.py
  File "err1.py", line 2
    print "yourself!")
                     ^
SyntaxError: invalid syntax
```

Итак, сообщение говорит о том, что в файле err1.py во второй строке исходного кода (`line 2`) допущена синтаксическая ошибка. В сообщении об ошибке выводится строка исходного кода, вызвавшая ошибку, а под этой строкой — стрелка, показывающая место, в котором ошибка была обнаружена. Ошибку обычно следует искать слева от стрелки.

Синтаксические ошибки обнаруживаются в процессе анализа исходного кода до запуска программы. Но есть другой вид ошибок — исключения, которые выявляются во время выполнения программы. Рассмотрим следующий пример (листинг 31.4).

Листинг 31.4. Программа err2

```
a, b = 10, 0
print("Hello")
print(a/b)
print("Goodbye")
```

В этой программе нет синтаксических ошибок, поэтому она без проблем запустится и начнет работать, но на предпоследней инструкции "споткнется".

> ***Примечание***
>
> Программисты иногда шутят, что успешное деление на ноль способно вызвать нарушение пространственно-временного континуума, что неизбежно приводит к разрушительным последствиям, вплоть до краха Вселенной.

К счастью, успешного деления в нашем случае не произошло:

```
$ python3 err2.py
Hello
Traceback (most recent call last):
  File "err2.py", line 3, in <module>
    print(a/b)
ZeroDivisionError: int division or modulo by zero
```

Обратите внимание на то, что программа работала до тех пор, пока не произошло исключение, о чем свидетельствует выведенная на экран строка `Hello`. А вот слово `Goodbye` уже не вывелось, потому что программа была прервана из-за возникшего исключения.

Прерывание выполнения программы является типичным поведением интерпретатора Python при возникновении исключений. Однако язык располагает арсеналом средств, позволяющим обрабатывать возникающие исключения произвольным способом, а не завершать программу, но эта тема не рассматривается в нашей книге.

31.3. Ветвления

Управляющая логика практически любого языка программирования позволяет устанавливать обратную связь между данными и алгоритмом, когда выполнение программы ставится в зависимость от содержащихся в ней данных. В языке Python для этих целей используются ветвления, циклы и исключения. В рамках данной книги мы ограничимся только ветвлениями и циклами, и начнем с ветвлений.

В языке Python для ветвлений используются операторы `if`, `elif` и `else`. С операциями `if` и `else` вы уже знакомы из языков C и C++, и, возможно, уже догадались, что `elif` — это сокращенная версия комбинации `else if`. Работают эти операции так же, как и в языках C, C++, Pascal и т. п.

Синтаксис языков C и C++ требует заключать логическое выражение для оператора `if` в скобки. В языке Python этого не требуется. Однако, как уже было сказано в *разд. 30.2*, скобки могут быть использованы не только для переопределения приоритета выполнения операций (в том числе логических), но и для внесения ясности в сложные выражения.

Ветвления в языке Python реализуются следующим образом. Сначала указывается оператор `if`, затем логическое выражение (в скобках или без них), затем двоеточие и, наконец, блок инструкций, который будет выполняться в случае, если условие верное. Если нужно выполнить всего одну инструкцию, то можно написать ее сразу после двоеточия. Это минимальная форма ветвления. И вот как это выглядит в условной форме:

```
if УСЛОВИЕ:
        ИНСТРУКЦИЯ1
        ИНСТРУКЦИЯ2
```

А вот сокращенная форма ветвления:

```
if УСЛОВИЕ: ИНСТРУКЦИЯ
```

Приведенная минимальная форма ветвления может быть дополнена операторами `elif` и/или `else`. Вот как условно выглядит полная форма ветвления в языке Python:

```
if УСЛОВИЕ1:
        ИНСТРУКЦИЯ1
        ИНСТРУКЦИЯ2
elif УСЛОВИЕ2:
        ИНСТРУКЦИЯ3
        ИНСТРУКЦИЯ4
else:
        ИНСТРУКЦИЯ5
```

И, наконец, как и в языках семейства C/C++, Python позволяет реализовывать вложенные ветвления. Вот условный пример такого ветвления:

```
if УСЛОВИЕ1:
        if УСЛОВИЕ2:
                ИНСТРУКЦИЯ1
        else:
                ИНСТРУКЦИЯ2
                ИНСТРУКЦИЯ3
elif: УСЛОВИЕ3:
        ИНСТРУКЦИЯ4
        ИНСТРУКЦИЯ5
elif УСЛОВИЕ4:
        ИНСТРУКЦИЯ6
else:
        ИНСТРУКЦИЯ7
        ИНСТРУКЦИЯ8
```

Как видим, `elif` может использоваться в ветвлении сколько угодно раз. Обратите внимание на вложенное ветвление. Оно демонстрирует то, что для использования `else` вовсе не обязательно предварительно включать в ветвление `elif`.

И здесь мы впервые сталкиваемся с тем, о чем шла речь в *разд. 29.1*: для выделения логических блоков программы в языке Python используются не скобки, как в языках C и C++, и не операторы `begin` и `end`, как в языке Pascal, а обычные отступы, выполненные пробелами и/или табуляциями. Пришла пора применить этот подход на практике.

Здесь есть всего два основных правила.

1. Последовательно идущие инструкции, имеющие одинаковый отступ, находятся в одном логическом блоке.
2. Для формирования вложенных блоков используется более глубокий отступ.

Следующий пример (листинг 31.5) демонстрирует изложенные правила.

Листинг 31.5. Программа branching1

```
a, b, c = 5, 6, 7

if a > b:
        print("Yeah, A > B")
elif b > c:
        print("B > C")
        print("Blah-blah-blah...")
else:
        print("A <= B")
        if b < c:
                print("B < C")
                print("Something else...")
```

И вот как выполнилась эта программа:

```
$ python3 branching1.py
A <= B
B < C
Something else...
```

Обратите внимание, что неважно, используете ли вы для отступа один пробел или десять табуляций. Главное, чтобы в одном блоке отступы были одинаковые и вложенные блоки были более "глубокими", чем родительский. Также полезным может оказаться мысленное представление вершины стека выполнения программы как блока с нулевым отступом.

> ***Примечание***
>
> Разработчики языка Python рекомендуют использовать шаг отступа, равный четырем пробелам, и не рекомендуют пользоваться табуляциями. Тем не менее, все отступы в примерах этой книги имеют шаг глубины в одну табуляцию. Иногда приятно нарушать правила, если это не вредит делу!

Пытливый ум читателя может задаться вопросом о том, есть ли в языке Python конструкции множественного выбора, наподобие тех, что реализованы в языках C и C++ операторами `switch`, `case`, `default` и `break`. Ответ прост: нет. Подобные конструкции разработчики Python считают избыточными и рекомендуют пользоваться вместо этого ветвлениями на основе операторов `if`, `elif` и `else`.

31.4. Циклы

Не вдаваясь детально в тему сравнений с циклами языков C и C++, рассмотрим некоторые особенности циклов языка Python.

Во-первых, в языке Python используется два основных типа циклов: *циклы по условию* (while-циклы) и *циклы по счетчику* (for-циклы). При этом конструкции указанных циклов не такие сложные, как в языках C и C++. Но в то же время упрощение не привело к потере функциональности. Как и в языках семейства C/C++, в циклах языка Python могут использоваться операторы `break` и `continue`.

В while-циклах языка Python проверка условия производится всегда перед входом в цикл, т. е. конструкция `do-while` здесь не предусмотрена. Зато в языке Python широко употребима конструкция `while-else`. Блок, относящийся к `else`, выполняется в случае нарушения условия цикла, но только в том случае, если выход из цикла произошел "естественным образом", а не посредством использования оператора `break`. Подобное привнесение в циклы элемента ветвлений открывает совершенно новый подход к реализации итеративных (основанных на повторениях) алгоритмов.

Итак, простейший цикл по условию на языке Python выглядит так:

```
while УСЛОВИЕ:
        ИНСТРУКЦИЯ1
        ИНСТРУКЦИЯ2
```

Как и в случае с простейшим ветвлением, если в теле цикла используется лишь одна инструкция, допускается оставить ее прямо за двоеточием:

```
while УСЛОВИЕ: ИНСТРУКЦИЯ
```

Более полная форма while-цикла выглядит так:

```
while УСЛОВИЕ:
       ИНСТРУКЦИЯ1
       ИНСТРУКЦИЯ2
else:
       ИНСТРУКЦИЯ3
```

И, наконец, вложенные циклы формируются аналогично вложенным ветвлениям.

> ***Примечание***
>
> Современные языки программирования технически позволяют создавать очень глубокие вложенные ветвления и циклы. Однако следует помнить, что какой бы сложной ни была программа, исходный код должен быть прост и понятен. Использование более трех-четырех уровней глубины в ветвлениях и циклах является дурным тоном: такие программы иногда называют "ветвистыми" или "лохматыми".

Следующий пример (листинг 31.6) демонстрирует работу цикла по условию.

Листинг 31.6. Программа while1

```
a = 0
while a < 10:
       print("a=", a)
       a+=1
else:
       print("The loop condition has been broken, a=", a)
```

Вот что получилось:

```
$ python3 while1.py
a= 0
a= 1
a= 2
a= 3
a= 4
a= 5
a= 6
a= 7
a= 8
a= 9
The loop condition has been broken, a= 10
```

Теперь рассмотрим циклы по счетчику. Как и в большинстве языков программирования, в языке Python они задаются при помощи оператора `for`. Однако синтаксис этого оператора разительно отличается от того, что используется в языках C и C++.

Как мы хорошо помним, в языках семейства C/C++ оператору `for` передаются три аргумента, которые можно условно назвать инициализатором, условием повторения и инкрементом. В языке Python все намного проще: for-цикл "прогоняет" заданную переменную по указанной последовательности (список, строка и т. п.), после чего завершается. И, естественно, имеется также возможность "досрочного" завершения цикла оператором `break`.

Такой подход поначалу кажется несколько неуклюжим. Но на практике выходит как раз наоборот: циклы языка Python выглядят весьма лаконично и опрятно. Дело в том, что язык Python предоставляет множество стандартных механизмов для генерирования и корректировки наборных типов данных (строки, списки, словари и т. п.). Так, например, часто используемая функция `range()` генерирует арифметическую прогрессию с заданными первым членом, последним членом и шагом. По умолчанию шаг прогрессии равен 1, а первый член равен 0.

Вот примеры использования функции `range()`:

- `range(10)`: арифметическая прогрессия от 0 до 10 с шагом 1;
- `range(1, 5)`: арифметическая прогрессия от 1 до 5 с шагом 1;
- `range(3, 20, 2)`: арифметическая прогрессия от 3 до 20 с шагом 2.

Условный пример цикла по счетчику в языке Python может выглядеть так:

```
for ПЕРЕМЕННАЯ in ПОСЛЕДОВАТЕЛЬНОСТЬ:
       ИНСТРУКЦИЯ1;
       ИНСТРУКЦИЯ2;
else:
       ИНСТРУКЦИЯ3;
       ИНСТРУКЦИЯ4;
```

И, как и в случае с циклом по условию, оператор `else` можно опустить. А если тело цикла состоит из одной инструкции, то можно реализовать цикл одной строкой:

```
for ПЕРЕМЕННАЯ in ПОСЛЕДОВАТЕЛЬНОСТЬ: ИНСТРУКЦИЯ
```

Таким образом, в for-циклах языка Python задается не способ прохождения счетчика через последовательность, а сама последовательность. Следующий пример (листинг 31.7) является реализацией предыдущей программы while1 (листинг 31.6) при помощи цикла по счетчику.

Листинг 31.7. Программа for1

```
for a in range(10): print("a=", a)
else: print("The loop condition has been broken, a=", a)
```

Программа for1 оказалась компактнее, чем while1, а различаются эти программы только значением переменной `a` при выходе из цикла.

31.5. Функции

Функции в языке Python работают обычным способом: принимают аргументы, выполняют заданные действия и, если нужно, возвращают какой-нибудь результат.

В процессе программирования на языках C и C++ мы привыкли, что исполнение программы начинается с входа в функцию `main()` и заканчивается возвратом из функции `main()`. В этом случае говорят, что функция `main()` находится на вершине стека вызовов. Программы на языке Python тоже имеют стек вызовов, но с тем лишь отличием, что для вершины стека не используется специальная функция.

Итак, в языке Python используется следующий формат определения функций:

```
def ИМЯ(АРГУМЕНТЫ):
        """ ОПИСАНИЕ ФУНКЦИИ """
        ИНСТРУКЦИЯ1
        ИНСТРУКЦИЯ2
        ...
        return ВОЗВРАЩАЕМОЕ_ЗНАЧЕНИЕ
```

Аргументы функции задаются так же, как и в языках C и C++ — через запятую. После двоеточия идет строка описания функции (docstring). Строка описания не только комментирует функцию, но также используется при автоматическом построении документации. Эту строку можно опустить, однако разработчики Python рекомендуют документировать каждую функцию. Если функция ничего не возвращает, то операцию `return` можно опустить. Можно также использовать `return` без аргумента, когда требуется прервать выполнение функции без возврата значения.

Следующая программа (листинг 31.8) демонстрирует работу функций в языке Python на примере простой функции `sum3()`, которая возвращает сумму трех чисел, переданных в качестве аргументов.

Листинг 31.8. Программа func1

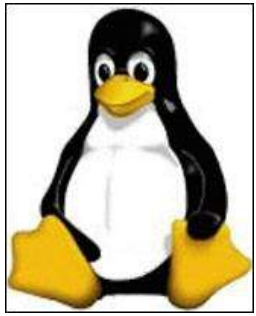
```
def sum3(a, b, c):
        """ Returns a sum of agruments """
        return a+b+c

print("Here is the result:", sum3(10, 20, 30))
```

Вот что получилось:

```
$ python3 func1.py
Here is the result: 60
```

Обратите внимание, что в отличие от функций языков семейства C/C++, аргументы функций языка Python задаются без указания типа.

Удачного вам программирования!



ПРИЛОЖЕНИЯ

Приложение 1. Именованные константы

Приложение 2. Коды ошибок системных вызовов

Приложение 3. Сигналы Linux

Приложение 4. Примеры программ

ПРИЛОЖЕНИЕ 1

Именованные константы

Таблица П1.1. *Константы флагов открытия файла*

Константа	Описание
O_RDONLY	Открыть только для чтения
O_WRONLY	Открыть только для записи
O_RDWR	Открыть для чтения и записи
O_APPEND	Дописать в конец файла
O_ASYNC	Генерировать сигнал (по умолчанию SIGIO), когда ввод-вывод для данного файлового дескриптора становится возможным
O_CREAT	Если не существует, создать файл
O_DIRECT	Минимизировать эффекты кэширования ввода-вывода для данного файла
O_DIRECTORY	Завершиться ошибкой, если файл не является каталогом
O_EXCL	При использовании с O_CREAT генерирует ошибку, если файл существует
O_LARGEFILE	Допустить использование больших файлов, размер которых превышает максимальное значение для off_t
O_NOATIME	Не обновлять время последнего доступа к файлу (начиная с версии ядра 2.6.8), если производится чтение
O_NOCTTY	Если имя файла ссылается на терминальное устройство, то оно не становится управляющим терминалом процесса, даже если процесс не имеет такового
O_NOFOLLOW	Завершиться ошибкой, если открываемый файл является символической ссылкой
O_NONBLOCK	Открыть файл в неблокируемом режиме, если это возможно

Таблица П1.1 *(окончание)*

Константа	Описание
O_NDELAY	Синоним для O_NONBLOCK
O_SYNC	Открыть файл для синхронного ввода-вывода: каждая операция write() блокирует вызывающий процесс до тех пор, пока данные не будут реально записаны на физический носитель
O_TRUNC	Установить длину файла в 0, если он уже существует, является обычным файлом и открывается с возможностью записи

Таблица П1.2. *Флаги режима файла*

Константа	Восьмеричное значение	Описание
S_IRUSR	0400	Право владельца на чтение
S_IWUSR	0200	Право владельца на запись
S_IXUSR	0100	Право владельца на выполнение
S_IRGRP	0040	Право группы на чтение
S_IWGRP	0020	Право группы на запись
S_IXGRP	0010	Право группы на выполнение
S_IROTH	0004	Право других пользователей на чтение
S_IWOTH	0002	Право других пользователей на запись
S_IXOTH	0001	Право других пользователей на выполнение
S_ISUID	04000	Бит SUID
S_ISGID	02000	Бит SGID
S_ISVTX	01000	Липкий бит
S_IFDIR	0040000	Каталог
S_IFCHR	0020000	Символьное устройство
S_IFBLK	0060000	Блочное устройство
S_IFREG	0100000	Обычный файл
S_IFIFO	0010000	Именованный канал FIFO
S_IFLNK	0120000	Символическая ссылка
S_IFSOCK	0140000	Локальный сокет

ПРИЛОЖЕНИЕ 2

Коды ошибок системных вызовов

Таблица П2.1. *Значения* `errno` *для* `open()`

Код ошибки	Описание
EACCESS	Доступ запрещен
EEXIST	Файл уже существует
EFAULT	Имя файла представлено недоступным указателем
EISDIR	Файл является каталогом
ELOOP	Слишком много символических ссылок, ссылающихся на файл
EMFILE	Превышен лимит открытых файлов для процесса
ENFILE	Превышен лимит открытых файлов для системы
ENODEV	Имя файла ссылается на устройство
ENOENT	Файл не существует
ENOMEM	Не хватает памяти
ENOSPC	На устройстве нет места для нового файла
ENOTDIR	Файл не является каталогом
ENXIO	Файл ссылается на несуществующее устройство
EOVERFLOW	Имя файла ссылается на слишком большой файл
EPERM	У процесса нет соответствующих привилегий для открытия файла
EROFS	Файловая система, на которой находится файл, смонтирована только для чтения

Таблица П2.2. *Значения* `errno` *для* `read()`

Код ошибки	Описание
EAGAIN	Данные не могут быть прочитаны в настоящее время, повторите попытку позже
EBADF	Неподходящий для чтения файловый дескриптор
EFAULT	Буфер для чтения находится вне адресного пространства процесса
EINTR	Системный вызов был прерван по сигналу
EINVAL	Дескриптор ссылается на файл или иной объект, который недоступен для чтения
EIO	Ошибка ввода-вывода
EISDIR	Дескриптор ссылается на каталог

Таблица П2.3. *Значения* `errno` *для* `write()`

Код ошибки	Описание
EAGAIN	Данные не могут быть прочитаны в настоящее время, повторите попытку позже
EBADF	Неподходящий для записи файловый дескриптор
EFAULT	Буфер для записи находится вне адресного пространства процесса
EFBIG	Слишком большой файл
EINTR	Системный вызов был прерван по сигналу
EINVAL	Дескриптор ссылается на файл или иной объект, который недоступен для записи
EIO	Ошибка ввода-вывода
ENOSPC	На устройстве нет доступного места для записи данных в файл
EPIPE	Дескриптор ссылается на канал или сокет, читающий конец которого закрыт

Таблица П2.4. *Значения* `errno` *для* `lseek()`

Код ошибки	Описание
EBADF	Дескриптор не соответствует открытому файлу
EINVAL	Неверный третий аргумент
EOVERFLOW	Новая позиция ввода-вывода не умещается в `off_t`
ESPIPE	Дескриптор соответствует каналу, FIFO или сокету

Таблица П2.5. *Значения* `errno` *для* `close()`

Код ошибки	Описание
EBADF	Дескриптор не соответствует открытому файлу
EINTR	Системный вызов прерван по сигналу
EIO	Ошибка ввода-вывода

Таблица П2.6. *Значения* `errno` *для* `fork()`

Код ошибки	Описание
EAGAIN	Невозможно выделить достаточное количество памяти для создания нового процесса или превышен лимит другого ресурса
ENOMEM	Ошибка выделения памяти для создания нового процесса

Таблица П2.7. *Значения* `errno` *для семейства* `exec()`

Код ошибки	Описание
E2BIG	Слишком длинный список аргументов
EACCESS	Доступ запрещен
EINVAL	Формат исполняемого файла не поддерживается системой
ELOOP	Слишком много символических ссылок
ENAMETOOLONG	Слишком длинное имя файла
ENOENT	Файл не существует
ENOTDIR	Компонент пути не является каталогом
ENOMEM	Не хватает памяти

Таблица П2.8. *Значения* `errno` *для семейства* `wait()`

Код ошибки	Описание
ECHILD	Нет дочернего процесса для ожидания
EINTR	Прерывание по сигналу `SIGCHLD` при отсутствии флага `WNOHANG`
EINVAL	Неверный аргумент, задающий опции

ПРИЛОЖЕНИЕ 3

Сигналы Linux

Таблица П3.1. *Сигналы Linux*

Сигнал	Действие по умолчанию	Описание
SIGHUP	Завершение	Отключение от терминала или завершение процесса, контролирующего терминал
SIGINT	Завершение	Прерывание от клавиатуры (обычно <Ctrl>+<C>)
SIGQUIT	Дамп	Завершение от клавиатуры (обычно <Ctrl>+<\>)
SIGILL	Дамп	Недопустимая инструкция
SIGABRT	Дамп	Аварийное завершение; этот сигнал процесс посылает сам себе при вызове функции `abort()`
SIGFPE	Дамп	Ошибка операции с плавающей точкой
SIGKILL	Завершение	Завершение (убийство) процесса
SIGSEGV	Дамп	Ошибка сегментации памяти; ядро Linux посылает этот сигнал процессу, если тот пытается обратиться к памяти вне своего адресного пространства
SIGPIPE	Завершение	Поврежденный канал; чтение в канал, читающий конец которого закрыт
SIGALRM	Завершение	Сигнал таймера
SIGTERM	Завершение	Запрос на завершение процесса
SIGUSR1	Завершение	Пользовательский сигнал 1
SIGUSR2	Завершение	Пользовательский сигнал 2
SIGCHLD	Игнорирование	Остановка или завершение дочернего процесса
SIGCONT	Продолжение	Продолжение выполнения, если процесс остановлен
SIGSTOP	Остановка	Остановка выполнения процесса
SIGTSTP	Остановка	Остановка от терминала

Таблица П3.1 *(окончание)*

Сигнал	Действие по умолчанию	Описание
SIGTTIN	Остановка	Ввод терминала для фонового процесса
SIGTTOU	Остановка	Вывод терминала для фонового процесса
SIGBUS	Дамп	Ошибка шины
SIGPOLL	Завершение	Опрос события
SIGPROF	Завершение	Профилирующий таймер
SIGSYS	Дамп	Неверный аргумент или системный вызов
SIGTRAP	Дамп	Сигнал ловушки
SIGURG	Игнорирование	Срочное сообщение сокета
SIGVTALRM	Завершение	Сигнал виртуального таймера
SIGXCPU	Дамп	Превышение временного лимита процессора
SIGXFSZ	Дамп	Превышение лимита размера файла
SIGIOT	Дамп	Синоним для SIGABRT
SIGEMT	Завершение	Сигнал ловушки эмулятора (EMulator Trap)
SIGSTKFLT	Завершение	Нарушение стека сопроцессора (не используется)
SIGIO	Завершение	Ввод-вывод доступен
SIGCLD	Игнорирование	Синоним для SIGCHLD
SIGPWR	Завершение	Отключение или сбой питания
SIGINFO	Завершение	Синоним для SIGPWR
SIGLOST	Завершение	Потеря блокировки файла
SIGWINCH	Игнорирование	Изменение размеров окна
SIGUNUSED	Завершение	Неиспользуемый сигнал

ПРИЛОЖЕНИЕ 4

Примеры программ

Исходные тексты рассмотренных в книге программ на языках C, C++ и Python можно скачать по ссылке **ftp://85.249.45.166/ 9785977507448.zip**, а также на странице книги на сайте **www.bhv.ru**.

Таблица П4.1. *Файлы примеров*

Листинг	**Файл**
Листинг 1.1	ch01/myclock/myclock.c
Листинг 1.2	ch01/printup/print_up.h
Листинг 1.3	ch01/printup/print_up.c
Листинг 1.4	ch01/printup/main.c
Листинг 2.1	ch02/printup1/make_printup
Листинг 2.2	ch02/printup2/Makefile
Листинг 2.3	ch02/printup3/Makefile
Листинг 2.4	ch02/printup4/Makefile
Листинг 2.5	ch02/printup5/Makefile
Листинг 2.6	ch02/printup6/Makefile
Листинг 2.7	ch02/printup7/Makefile
Листинг 2.8	ch02/upver/readver/readver.h
Листинг 2.9	ch02/upver/readver/readver.c
Листинг 2.10	ch02/upver/readver/Makefile
Листинг 2.11	ch02/upver/toup/toup.h
Листинг 2.12	ch02/upver/toup/toup.c
Листинг 2.13	ch02/upver/toup/Makefile
Листинг 2.14	ch02/upver/upver.c
Листинг 2.15	ch02/upver/Makefile

Таблица П4.1 *(продолжение)*

Листинг	**Файл**
Листинг 3.1	ch03/myenv1/myenv1.c
Листинг 3.2	ch03/myenv2/myenv2.c
Листинг 3.3	ch03/setenvdemo/setenvdemo.c
Листинг 3.4	ch03/putenvdemo/putenvdemo.c
Листинг 3.5	ch03/unsetenvdemo/unsetenvdemo.c
Листинг 4.1	ch04/power/power.c
Листинг 4.2	ch04/myenv/mysetenv.c
Листинг 4.3	ch04/myenv/myprintenv.c
Листинг 4.4	ch04/myenv/myenv.h
Листинг 4.5	ch04/myenv/Makefile
Листинг 4.6	ch04/myenv/envmain.c
Листинг 4.7	ch04/myenv1/Makefile
Листинг 4.8	ch04/myenv2/Makefile
Листинг 4.9	ch04/program/first.c
Листинг 4.10	ch04/program/second.c
Листинг 4.11	ch04/program/third.c
Листинг 4.12	ch04/program/program.c
Листинг 4.13	ch04/program/common.h
Листинг 4.14	ch04/Makefile
Листинг 5.1	ch05/mainargs/mainargs.c
Листинг 5.2	ch05/getoptdemo/getoptdemo.c
Листинг 5.3	ch05/longoptdemo/longoptdemo.c
Листинг 6.1	ch06/mytail1/mytail.c
Листинг 6.2	ch06/stdoutreopen/stdoutreopen.c
Листинг 6.3	ch06/mytail2/mytail.cpp
Листинг 7.1	ch07/creat1/creat1.c
Листинг 7.2	ch07/mymask/mymask.c
Листинг 7.3	ch07/creatopen/creatopen.c
Листинг 7.4	ch07/closetwice/closetwice.c
Листинг 7.5	ch07/read1/read1.c
Листинг 7.6	ch07/read2/read2.c
Листинг 7.7	ch07/mycp/mycp.c
Листинг 7.8	ch07/mycp1/mycp1.c
Листинг 7.9	ch07/reverse/reverse.c

Таблица П4.1 *(продолжение)*

Листинг	Файл
Листинг 7.10	ch07/mylseektail/mylseektail.c
Листинг 7.11	ch07/readblock/readblock.c
Листинг 8.1	ch08/hw/hw.c
Листинг 8.2	ch08/filenodemo/filenodemo.c
Листинг 8.3	ch08/brokenfile/brokenfile.c
Листинг 8.4	ch08/fdopendemo/fdopendemo.c
Листинг 8.5	ch08/readtar/readtar.c
Листинг 8.6	ch08/abook/abook.c
Листинг 8.7	ch08/blackhole/blackhole.c
Листинг 8.8	ch08/showhole/showhole.c
Листинг 9.1	ch09/system1/system1.c
Листинг 9.2	ch09/system2/system2.c
Листинг 9.3	ch09/system3/system3.c
Листинг 9.4	ch09/getpinfo/getpinfo.c
Листинг 10.1	ch10/fork1/fork1.c
Листинг 10.2	ch10/fork2/fork2.c
Листинг 10.3	ch10/race1/race1.c
Листинг 10.4	ch10/execve1/execve1.c
Листинг 10.5	ch10/execve2/execve2.c
Листинг 10.6	ch10/execve2/newprog.c
Листинг 10.7	ch10/execve2/Makefile
Листинг 10.8	ch10/execve3/execve3.c
Листинг 10.9	ch10/forkexec1/forkexec1.c
Листинг 10.10	ch10/execvels/execvels.c
Листинг 10.11	ch10/execlls/execlls.c
Листинг 10.12	ch10/execlels/execlels.c
Листинг 10.13	ch10/execlpls/execlpls.c
Листинг 10.14	ch10/execvls/execvls.c
Листинг 10.15	ch10/execvpls/execvpls.c
Листинг 10.16	ch10/lsroot/lsroot.c
Листинг 10.17	ch10/lsstatus/lsstatus.c
Листинг 10.18	ch10/killedchild/killedchild.c
Листинг 11.1	ch11/pdata/pdata.c
Листинг 11.2	ch11/nothread/nothread.c

Таблица П4.1 *(продолжение)*

Листинг	Файл
Листинг 11.3	ch11/thread1/thread1.c
Листинг 11.4	ch11/thread1/Makefile
Листинг 11.5	ch11/threadarg/threadarg.c
Листинг 11.6	ch11/threadstruct/threadstruct.c
Листинг 11.7	ch11/multithread/multithread.c
Листинг 11.8	ch11/threadexit/threadexit.c
Листинг 11.9	ch11/join1/join1.c
Листинг 11.10	ch11/join2/join2.c
Листинг 11.11	ch11/join3/join3.c
Листинг 11.12	ch11/join4/join4.c
Листинг 11.13	ch11/pcancel1/pcancel1.c
Листинг 11.14	ch11/pcancel2/pcancel2.c
Листинг 12.1	ch12/nice1/nice1.c
Листинг 12.2	ch12/nice2/nice2.c
Листинг 12.3	ch12/waitpid1/waitpid1.c
Листинг 12.4	ch12/waitpid2/waitpid2.c
Листинг 12.5	ch12/zombie1/zombie1.c
Листинг 14.1	ch14/statvfsdemo/statvfsdemo.c
Листинг 14.2	ch14/fstatvfsdemo/fstatvfsdemo.c
Листинг 14.3	ch14/cwd/cwd.c
Листинг 14.4	ch14/cwdauto/cwdauto.c
Листинг 15.1	ch15/chdirdemo/chdirdemo.c
Листинг 15.2	ch15/fchdirdemo/fchdirdemo.c
Листинг 15.3	ch15/readdir1/readdir1.c
Листинг 15.4	ch15/readdir2/readdir2.c
Листинг 15.5	ch15/stat1/stat1.c
Листинг 15.6	ch15/stat2/stat2.c
Листинг 15.7	ch15/stat3/stat3.c
Листинг 15.8	ch15/readdir3/readdir3.c
Листинг 15.9	ch15/readdir4/readdir4.c
Листинг 16.1	ch16/unlink1/unlink1.c
Листинг 16.2	ch16/statvfsinode/statvfsinode.c
Листинг 16.3	ch16/statinode/statinode.c
Листинг 16.4	ch16/unlink2/unlink2.c

Таблица П4.1 *(продолжение)*

Листинг	Файл
Листинг 16.5	ch16/rename1/rename1.c
Листинг 16.6	ch16/rename2/rename2.c
Листинг 16.7	ch16/linkdemo/linkdemo.c
Листинг 16.8	ch16/symlinkdemo/symlinkdemo.c
Листинг 16.9	ch16/mkdir1/mkdir1.c
Листинг 16.10	ch16/mkdir2/mkdir2.c
Листинг 16.11	ch16/mkdir3/mkdir3.c
Листинг 16.12	ch16/mkdir4/mkdir4.c
Листинг 16.13	ch16/rmdirdemo/rmdirdemo.c
Листинг 17.1	ch17/chmoddemo/chmoddemo.c
Листинг 17.2	ch17/fchmoddemo/fchmoddemo.c
Листинг 18.1	ch18/mkstempdemo/mkstempdemo.c
Листинг 18.2	ch18/abook1/abook1.c
Листинг 19.1	ch19/kinsfolk1/kinsfolk-child1.c
Листинг 19.2	ch19/kinsfolk1/kinsfolk-parent1.c
Листинг 20.1	ch20/kill1/kill1.c
Листинг 20.2	ch20/kill2/kill2.c
Листинг 20.3	ch20/sigaction1/sigaction1.c
Листинг 20.4	ch20/sigaction2/sigaction2.c
Листинг 20.5	ch20/kinsfolk2/kinsfolk-child2.c
Листинг 20.6	ch20/kinsfolk2/kinsfolk-parent2.c
Листинг 21.1	ch21/shm1/shm1-owner.c
Листинг 21.2	ch21/shm1/shm1-user.c
Листинг 21.3	ch21/shm2/shm2-owner.c
Листинг 21.4	ch21/shm2/shm2-user.c
Листинг 22.1	ch22/mmap1/mmap1.c
Листинг 22.2	ch22/mmap2/mmap2-owner.c
Листинг 22.3	ch22/mmap2/mmap2-user.c
Листинг 23.1	ch23/pipe1/pipe1-parent.c
Листинг 23.2	ch23/pipe1/pipe1-src.c
Листинг 23.3	ch23/pipe1/pipe1-dst.c
Листинг 23.4	ch23/dup01/dup01.c
Листинг 23.5	ch23/dup02/dup02.c
Листинг 23.6	ch23/pipe2/pipe2.c

Таблица П4.1 *(продолжение)*

Листинг	Файл
Листинг 24.1	ch24/fifo1/mkfifo1.c
Листинг 24.2	ch24/fifo2/fifo2-server.c
Листинг 24.3	ch24/fifo2/fifo2-client.c
Листинг 25.1	ch25/socket1/socket1.c
Листинг 25.2	ch25/getwwwpage/getwwwpage.c
Листинг 25.3	ch25/socket2/socket2-server.c
Листинг 25.4	ch25/socket2/socket2-client.c
Листинг 25.5	ch25/socket3/socket3-server.c
Листинг 25.6	ch25/socket3/socket3-client.c
Листинг 26.1	ch26/joke1/joke1.c
Листинг 26.2	ch26/assert1/assert1.c
Листинг 26.3	ch26/assert1/Makefile
Листинг 26.4	ch26/assert2/assert2.c
Листинг 27.1	ch27/errno1/errno1.c
Листинг 27.2	ch27/errno2/errno2.c
Листинг 27.3	ch27/errno3/errno3.c
Листинг 28.1	ch28/program1/program1.c
Листинг 28.2	ch28/program2/program2.c
Листинг 29.4	ch29/hello1/hello1.py
Листинг 29.5	ch29/hello2/hello2.py
Листинг 29.6	ch29/hello3/hello3.py
Листинг 29.7	ch29/hello4/hello4.py
Листинг 29.8	ch29/hello5/hello5.py
Листинг 29.9	ch29/hello6/hello6.py
Листинг 29.10	ch29/hello7/hello7.py
Листинг 29.11	ch29/comment1/comment1.py
Листинг 30.1	ch30/var1/var1.py
Листинг 30.2	ch30/var2/var2.py
Листинг 30.3	ch30/var3/var3.py
Листинг 30.4	ch30/var4/var4.py
Листинг 30.5	ch30/num1/num1.py
Листинг 30.6	ch30/num2/num2.py
Листинг 30.7	ch30/float1/float1.py
Листинг 30.8	ch30/float2/float2.py

Таблица П4.1 *(окончание)*

Листинг	**Файл**
Листинг 30.9	ch30/float4/float3.py
Листинг 30.10	ch30/str1/str1.py
Листинг 30.11	ch30/str2/str2.py
Листинг 30.12	ch30/str3/str3.py
Листинг 30.13	ch30/str4/str4.py
Листинг 30.14	ch30/str5/str5.py
Листинг 30.15	ch30/str6/str6.py
Листинг 30.16	ch30/str7/str7.py
Листинг 30.17	ch30/list1/list1.py
Листинг 30.18	ch30/list2/list2.py
Листинг 30.19	ch30/list3/list3.py
Листинг 31.1	ch31/bool1/bool1.py
Листинг 31.2	ch31/bool2/bool2.py
Листинг 31.3	ch31/err1/err1.py
Листинг 31.4	ch31/err2/err2.py
Листинг 31.5	ch31/branching1/branching1.py
Листинг 31.6	ch31/while1/while1.py
Листинг 31.7	ch31/for1/for1.py
Листинг 31.8	ch31/func1/func1.py

Предметный указатель

Б

В

Д

З

И

К

Л

М

О

П

P

C

Т

У

Ф

Э