UNIX

Профессиональное программирование

3-е издание

У. Ричард Стивенс,
Стивен А. Раго

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

ПИТЕР

ББК 32.973.2-018.2
УДК 004.451
С80

У. Ричард Стивенс, Стивен А. Раго

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

ISBN 978-5-4461-0649-3

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

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

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

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

ISBN 978-0321637734 англ.
ISBN 978-5-4461-0649-3

Краткое содержание

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

Оглавление

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

От издательства

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

Мы будем рады узнать ваше мнение!

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

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

1 Обзор ОС UNIX

1.1. Введение

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

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

1.2. Архитектура UNIX

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

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

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

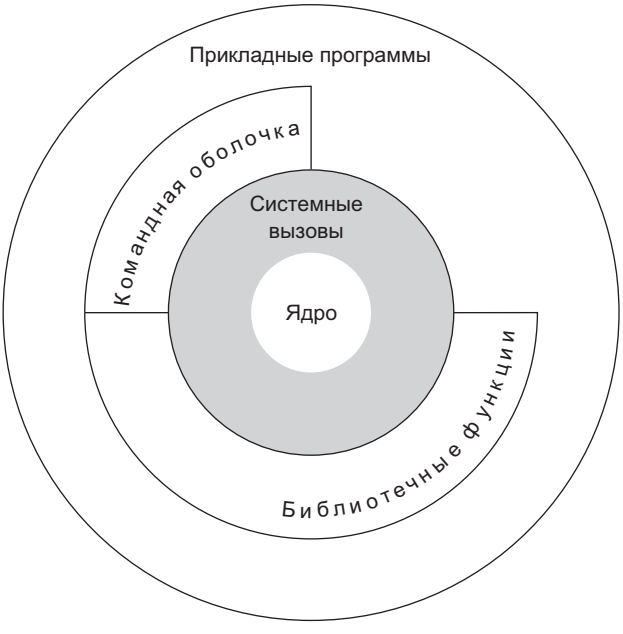

Рис. 1.1. Архитектура системы UNIX

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

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

1.3. Вход в систему

Имя пользователя

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

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

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

Командные оболочки

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

Таблица 1.1. Наиболее распространенные командные оболочки, используемые в UNIX

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

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

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

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

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

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

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

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

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

> *В разных дистрибутивах Linux по умолчанию используются разные командные интерпретаторы. В одних по умолчанию используется командный интерпретатор Bourne-again, в других — BSD-версия Bourne shell — `dash` (Debian Almquist shell, первоначально написан Кеннетом Альмквистом (Kenneth Almquist) и затем перенесен в Linux). В системах FreeBSD по умолчанию используется производная от Almquist shell. В Mac OS X — Bourne-again shell. Операционная система Solaris унаследовала от BSD и System V все командные интерпретаторы, перечисленные в табл. 1.1. Свободно распространяемые версии большинства командных интерпретаторов можно найти в Интернете.*
>
> *На протяжении всей книги в подобных примечаниях мы будем приводить исторические заметки и сравнения различных реализаций UNIX. Зачастую обоснования отдельных методов реализации становятся гораздо понятнее в историческом контексте.*

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

1.4. Файлы и каталоги

Файловая система

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

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

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

Имя файла

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

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

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

Путь к файлу

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

Пример

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

Листинг 1.1. Вывод списка всех файлов в каталоге

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

int
```

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

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

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

    closedir(dp);
    exit(0);
}
```

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

> *Исторически сложилось так, что все восемь разделов были объединены в документ, который называется «UNIX Programmer's Manual» (Руководство программиста UNIX). Но поскольку количество страниц в руководстве постоянно растет, появилась тенденция к распределению разделов по отдельным руководствам: например, одно для пользователей, одно для программистов и одно для системных администраторов.*
>
> *В некоторых версиях UNIX разделы справочного руководства делятся на подразделы, обозначенные заглавными буквами. Так, описания всех стандартных функций ввода/вывода в AT&T [1990e] находятся в разделе 3S, например fopen(3S). В некоторых системах в обозначениях разделов цифры заменены алфавитными символами, например «C» — для раздела с описаниями команд.*

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

```
man 1 ls
```

или

```
man -sl ls
```

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

```
cc myls.c
```

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

Примерный результат работы нашей программы:

```
$ ./a.out /dev
.
..
cdrom
stderr
stdout
stdin
fd
sda4
sda3
sda2
sda1
sda
tty2
tty1
console
tty
zero
null
```

и еще много строк...

```
mem
$ ./a.out /etc/ssl/private
невозможно открыть /etc/ssl/private: Permission denied
$ ./a.out /dev/tty
невозможно открыть /dev/tty: Not a directory
```

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

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

Рассмотрим детальнее эту программу из 20 строк.

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

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

Рабочий каталог

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

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

Домашний каталог

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

1.5. Ввод и вывод

Дескрипторы файлов

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

Стандартный ввод, стандартный вывод, стандартный вывод ошибок

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

```
ls
```

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

```
ls > file.list
```

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

Небуферизованный ввод/вывод

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

Пример

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

Листинг 1.2. Копирование со стандартного ввода на стандартный вывод

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

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

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

    if (n < 0)
```

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

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

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

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

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

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

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

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

Если запустить программу так:

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

она скопирует файл infile в файл outfile.

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

Стандартные функции ввода/вывода

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

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

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

Пример

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

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

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

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

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

    exit(0);
}
```

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

1.6. Программы и процессы

Программа

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

Процессы и идентификаторы процессов

Выполняющая программа называется *процессом*. Этот термин будет встречаться практически на каждой странице этой книги. В некоторых операционных систе-

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

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

Пример

Программа в листинге 1.4 выводит собственный идентификатор процесса.

Листинг 1.4. Вывод идентификатора процесса

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

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

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

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

Управление процессами

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

Пример

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

Листинг 1.5. Чтение команд со стандартного ввода и их выполнение

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

int
```

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

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

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

Перечислим наиболее важные аспекты в этой 30-строчной программе.

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

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

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

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

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

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

Потоки выполнения и идентификаторы потоков

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

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

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

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

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

1.7. Обработка ошибок

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

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

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

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

```
extern int errno;
```

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

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

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

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

Для вывода сообщений об ошибках стандарт C предусматривает две функции.

```
#include <string.h>

char *strerror(int errnum);
                                    Возвращает указатель на строку сообщения
```

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

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

```
#include <stdio.h>

void perror(const char *msg);
```

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

Пример

В листинге 1.6 приводится пример использования этих функций.

Листинг 1.6. Демонстрация функций strerror и perror

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

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

Скомпилировав и запустив эту программу, мы получим

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

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

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

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

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

Восстановление после ошибок

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

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

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

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

1.8. Идентификация пользователя

Идентификатор пользователя

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

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

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

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

Идентификатор группы

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

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

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

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

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

Пример

Программа в листинге 1.7 выводит идентификаторы пользователя и группы.

Листинг 1.7. Вывод идентификаторов пользователя и группы

```
#include "apue.h"

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

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

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

Идентификаторы дополнительных групп

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

1.9. Сигналы

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

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

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

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

Пример

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

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

Листинг 1.8. Чтение команд со стандартного ввода и их выполнение

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

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

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

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

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

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

1.10. Представление времени

Исторически в системе UNIX поддерживается два способа представления времени.

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

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

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

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

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

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

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

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

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

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

```
$ cd /usr/include
$ time -p grep _POSIX_SOURCE */*.h > /dev/null

real    0m0.81s
user    0m0.11s
sys     0m0.07s
```

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

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

1.11. Системные вызовы и библиотечные функции

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

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

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

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

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

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

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

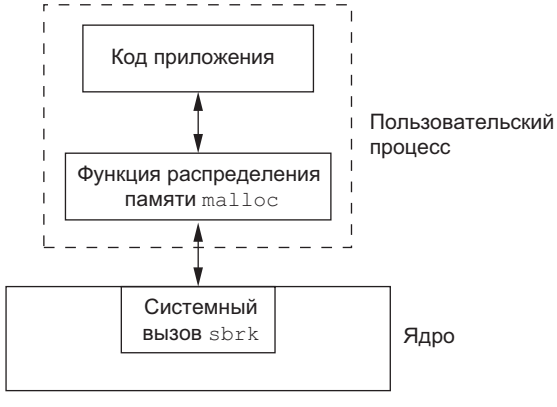

Рис. 1.2. Разделение обязанностей функции malloc и системного вызова sbrk

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

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

Прикладная программа может обращаться к системному вызову и к библиотечной функции. Кроме того, следует помнить, что библиотечные функции, в свою очередь, также могут обращаться к системным вызовам. Это наглядно показано на рис. 1.3.

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

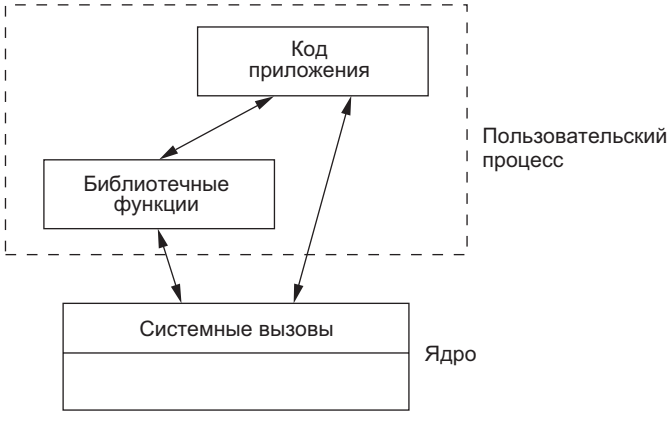

Рис. 1.3. Разделение обязанностей функции malloc и системного вызова sbrk

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

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

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

1.12. Подведение итогов

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

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

Упражнения

1.1 В своей системе проверьте и убедитесь, что каталоги «.» и «..» являются различными каталогами, за исключением корневого каталога.

1.2 Просмотрите еще раз результат работы примера в листинге 1.4 и скажите, куда пропали процессы с идентификаторами 852 и 853.

1.3 Аргумент функции `perror` в разделе 1.7 определен с атрибутом `const` (в соответствии со стандартом ISO C), в то время как целочисленный аргумент функции `strerror` определен без этого атрибута. Почему?

1.4 Если предположить, что календарное время хранится в виде 32-разрядного целого числа со знаком, в каком году наступит переполнение? Какими способами можно отдалить дату переполнения? Будут ли найденные решения совместимы с существующими приложениями?

1.5 Если предположить, что время работы процесса хранится в виде 32-разрядного целого числа со знаком и система отсчитывает 100 тактов в секунду, через сколько дней наступит переполнение счетчика?