

Кейт Хэвиленд, Дайна Грэй, Бен Салама

**Системное программирование в UNIX**  
**Руководство программиста по разработке ПО**



Москва



# **UNIX System Programming**

**A programmer's guide  
to software development**

**Keith Haviland,  
Dina Gray,  
Ben Salama**



**Addison-Wesley**

**An imprint of Addison Wesley Longman, Inc.**

Reading, Massachusetts • Harlow, England • Menlo Park, California

Berkeley, California • Don Mills, Ontario • Sydney

Bonn • Amsterdam • Tokyo • Mexico City



Серия «Для программистов»

# Системное программирование в UNIX

## Руководство программиста по разработке ПО

Кейт Хэвиленд,  
Дайна Грэй,  
Бен Салама



Москва

**ББК 32.973.26-018.2**

**X99**

**Кейт Хэвиленд, Дайна Грэй, Бен Салама**

X99 Системное программирование в UNIX: Пер. с англ. – М., ДМК Пресс. – 368 с., ил. (Серия «Для программистов»).

**ISBN 5-94074-008-1**

Операционная система UNIX всегда занимала важную позицию в научном и техническом сообществах. В настоящее время существует множество крупномасштабных систем управления данными и обработки транзакций на платформе UNIX. Более того, эта ОС является ядром серверов магистральной сети Internet.

Предлагаемое издание адресовано прежде всего программистам, уже знакомым с UNIX, которые собираются разрабатывать программное обеспечение для этой операционной системы на языке C. Помимо обзора основных понятий и терминологии, в книге представлено описание системных примитивов доступа к файлам, процессов UNIX и методов работы с ними. Рассмотрено межпроцессное взаимодействие, освещается работа с основными библиотеками.

Книга также будет полезна разработчикам системного ПО, прикладных и деловых приложений.

**ББК 32.973.26-018.2**

Права на издание книги были получены по соглашению с Addison Wesley Longman, Inc. и Литературным агентством Мэтлок (Санкт-Петербург).

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 0-201-87758-9

ISBN 5-94074-008-1

© Addison Wesley Longman Inc.

© ДМК Пресс



# Содержание

<b>Предисловие .....</b>	<b>13</b>
<b>Соглашения .....</b>	<b>21</b>
<b>Благодарности .....</b>	<b>22</b>
<b>Глава 1. Основные понятия и терминология .....</b>	<b>23</b>
1.1. Файл .....	23
1.1.1. Каталоги и пути .....	24
1.1.2. Владелец файла и права доступа .....	25
1.1.3. Обобщение концепции файла .....	25
1.2. Процесс .....	26
1.2.1. Межпроцессное взаимодействие .....	26
1.3. Системные вызовы и библиотечные подпрограммы .....	26
<b>Глава 2. Файл .....</b>	<b>29</b>
2.1. Примитивы доступа к файлам в системе UNIX .....	29
2.1.1. Введение .....	29
2.1.2. Системный вызов open .....	31
2.1.3. Создание файла при помощи вызова open .....	34
2.1.4. Системный вызов creat .....	35
2.1.5. Системный вызов close .....	36
2.1.6. Системный вызов read .....	36
2.1.7. Системный вызов write .....	39
2.1.8. Пример copyfile .....	41

2.1.9. Эффективность вызовов read и write .....	42
2.1.10. Вызов lseek и произвольный доступ .....	43
2.1.11. Пример: гостиница .....	45
2.1.12. Дописывание данных в конец файла .....	47
2.1.13. Удаление файла .....	47
2.1.14. Системный вызов fcntl .....	48
2.2. Стандартный ввод, стандартный вывод и стандартный вывод диагностики .....	50
2.2.1. Основные понятия .....	50
2.2.2. Программа io .....	51
2.2.3. Использование стандартного вывода диагностики .....	53
2.3. Стандартная библиотека ввода/вывода: взгляд в будущее .....	53
2.4. Системные вызовы и переменная errno .....	56
2.4.1. Подпрограмма perror .....	57

## **Глава 3. Работа с файлами .....**

3.1. Файлы в многопользовательской среде .....	59
3.1.1. Пользователи и права доступа .....	59
3.1.2. Права доступа и режимы файлов .....	60
3.1.3. Дополнительные права доступа для исполняемых файлов .....	62
3.1.4. Маска создания файла и системный вызов umask .....	64
3.1.5. Вызов open и права доступа к файлу .....	65
3.1.6. Определение доступности файла при помощи вызова access .....	66
3.1.7. Изменение прав доступа при помощи вызова chmod .....	67
3.1.8. Изменение владельца при помощи вызова chown .....	68
3.2. Файлы с несколькими именами .....	69
3.2.1. Системный вызов link .....	69
3.2.2. Системный вызов unlink .....	69
3.2.3. Системный вызов rename .....	71
3.2.4. Символьные ссылки .....	71
3.3. Получение информации о файле: вызовы stat и fstat .....	72
3.3.1. Подробнее о вызове chmod .....	77

<b>Глава 4. Каталоги, файловые системы и специальные файлы .....</b>	<b>79</b>
4.1. Введение .....	79
4.2. Каталоги с точки зрения пользователя .....	79
4.3. Реализация каталогов .....	82
4.3.1. Снова о системных вызовах link и unlink .....	83
4.3.2. Точка и двойная точка .....	84
4.3.3. Права доступа к каталогам .....	84
4.4. Использование каталогов при программировании .....	86
4.4.1. Создание и удаление каталогов .....	86
4.4.2. Открытие и закрытие каталогов .....	87
4.4.3. Чтение каталогов: вызовы readdir и rewinddir .....	88
4.4.4. Текущий рабочий каталог .....	91
4.4.5. Смена рабочего каталога при помощи вызова chdir .....	91
4.4.6. Определение имени текущего рабочего каталога .....	92
4.4.7. Обход дерева каталогов .....	93
4.5. Файловые системы UNIX .....	95
4.5.1. Кэширование: вызовы sync и fsync .....	97
4.6. Имена устройств UNIX .....	98
4.6.1. Файлы блочных и символьных устройств .....	99
4.6.2. Структура stat .....	100
4.6.3. Информация о файловой системе .....	101
4.6.4. Ограничения файловой системы: процедуры pathconf и fpathconf .....	103
<b>Глава 5. Процесс .....</b>	<b>105</b>
5.1. Понятие процесса .....	105
5.2. Создание процессов .....	106
5.2.1. Системный вызов fork .....	106
5.3. Запуск новых программ при помощи вызова exec .....	109
5.3.1. Семейство вызовов exec .....	109
5.3.2. Доступ к аргументам, передаваемым при вызове exec .....	113

5.4. Совместное использование вызовов <code>exec</code> и <code>fork</code> .....	115
5.5. Наследование данных и дескрипторы файлов .....	118
5.5.1. Вызов <code>fork</code> , файлы и данные .....	118
5.5.2. Вызов <code>exec</code> и открытые файлы .....	119
5.6. Завершение процессов при помощи системного вызова <code>exit</code> .....	120
5.7. Синхронизация процессов .....	121
5.7.1. Системный вызов <code>wait</code> .....	121
5.7.2. Ожидание завершения определенного потомка: вызов <code>waitpid</code> .....	124
5.8. Зомби-процессы и преждевременное завершение программы .....	125
5.9. Командный интерпретатор <code>smallsh</code> .....	126
5.10. Атрибуты процесса .....	133
5.10.1. Идентификатор процесса .....	133
5.10.2. Группы процессов и идентификаторы группы процессов .....	135
5.10.3. Изменение группы процесса .....	135
5.10.4. Сеансы и идентификатор сеанса .....	136
5.10.5. Переменные программного окружения .....	137
5.10.6. Текущий рабочий каталог .....	139
5.10.7. Текущий корневой каталог .....	139
5.10.8. Идентификаторы пользователя и группы .....	140
5.10.9. Ограничения на размер файла: вызов <code>ulimit</code> .....	141
5.10.10. Приоритеты процессов: вызов <code>nice</code> .....	141

## **Глава 6. Сигналы и их обработка** .....

### 143

6.1. Введение .....	143
6.1.1. Имена сигналов .....	144
6.1.2. Нормальное и аварийное завершение .....	148
6.2. Обработка сигналов .....	150
6.2.1. Наборы сигналов .....	150
6.2.2. Задание обработчика сигналов: вызов <code>sigaction</code> .....	152
6.2.3. Сигналы и системные вызовы .....	157
6.2.4. Процедуры <code>sigsetjmp</code> и <code>siglongjmp</code> .....	158



6.3. Блокирование сигналов ..... 159

6.4. Посылка сигналов ..... 161

6.4.1. Посылка сигналов другим процессам: вызов kill ..... 161

6.4.2. Посылка сигналов самому процессу:  
вызовы raise и alarm ..... 164

6.4.3. Системный вызов pause ..... 166

**Глава 7. Межпроцессное взаимодействие  
при помощи программных каналов ..... 169**

7.1. Каналы ..... 169

7.1.1. Каналы на уровне команд ..... 169

7.1.2. Использование каналов в программе ..... 170

7.1.3. Размер канала ..... 175

7.1.4. Закрытие каналов ..... 177

7.1.5. Запись и чтение без блокирования ..... 178

7.1.6. Использование системного вызова select  
для работы с несколькими каналами ..... 181

7.1.7. Каналы и системный вызов exes ..... 186

7.2. Именованные каналы, или FIFO ..... 189

7.2.1. Программирование при помощи каналов FIFO ..... 191

**Глава 8. Дополнительные методы  
межпроцессного взаимодействия ..... 197**

8.1. Введение ..... 197

8.2. Блокировка записей ..... 198

8.2.1. Мотивация ..... 198

8.2.2. Блокировка записей при помощи вызова fcntl ..... 199

8.3. Дополнительные средства  
межпроцессного взаимодействия ..... 208

8.3.1. Введение и основные понятия ..... 208

8.3.2. Очереди сообщений ..... 210

8.3.3. Семафоры ..... 221

8.3.4. Разделяемая память ..... 229

8.3.5. Команды ipcs и ipcrm ..... 235

<b>Глава 9. Терминал</b>	237
9.1. Введение	237
9.2. Терминал UNIX	239
9.2.1. Управляющий терминал	240
9.2.2. Передача данных	241
9.2.3. Эхо-отображение вводимых символов и опережающий ввод с клавиатуры	241
9.2.4. Канонический режим, редактирование строки и специальные символы	242
9.3. Взгляд с точки зрения программы	244
9.3.1. Системный вызов open	245
9.3.2. Системный вызов read	246
9.3.3. Системный вызов write	248
9.3.4. Утилиты ttyname и isatty	249
9.3.5. Изменение свойств терминала: структура termios	249
9.3.6. Параметры MIN и TIME	257
9.3.7. Другие системные вызовы для работы с терминалом	258
9.3.8. Сигнал разрыва соединения	259
9.4. Псевдотерминалы	260
9.5. Пример управления терминалом: программа tscript	264
 <b>Глава 10. Сокеты</b>	 271
10.1. Введение	271
10.2. Типы соединения	272
10.3. Адресация	272
10.3.1. Адресация Internet	273
10.3.2. Порты	273
10.4. Интерфейс сокетов	274
10.4.1. Создание сокета	274
10.5. Программирование в режиме TCP-соединения	275
10.5.1. Связывание	276
10.5.2. Включение приема TCP-соединений	276
10.5.3. Прием запроса на установку TCP-соединения	277

10.5.4. Подключение клиента .....	278
10.5.5. Пересылка данных .....	279
10.5.6. Закрытие TCP-соединения .....	282
10.6. Программирование в режиме пересылок UDP-дейтаграмм .....	285
10.6.1. Прием и передача UDP-сообщений .....	285
10.7. Различия между двумя моделями .....	288

## **Глава 11. Стандартная библиотека ввода/вывода ..... 289**

11.1. Введение .....	289
11.2. Структура FILE .....	289
11.3. Открытие и закрытие потоков: процедуры fopen и fclose .....	290
11.4. Посимвольный ввод/вывод: процедуры getc и putc .....	292
11.5. Возврат символов в поток: процедура ungetc .....	294
11.6. Стандартный ввод, стандартный вывод и стандартный вывод диагностики .....	296
11.7. Стандартные процедуры опроса состояния .....	297
11.8. Построчный ввод и вывод .....	299
11.9. Ввод и вывод бинарных данных: процедуры fread и fwrite .....	301
11.10. Произвольный доступ к файлу: процедуры fseek, rewind и ftell .....	303
11.11. Форматированный вывод: семейство процедур printf .....	304
11.12. Форматированный ввод: семейство процедур scanf .....	310
11.13. Запуск программ при помощи библиотеки стандартного ввода/вывода .....	313
11.14. Вспомогательные процедуры .....	319
11.14.1. Процедуры freopen и fdopen .....	319
11.14.2. Управление буфером: процедуры setbuf и setvbuf .....	320





# Предисловие

## Назначение этой книги

*Со времени своего появления в Bell Laboratories в 1969 г. операционная система UNIX становилась все более популярной, вначале получив признание в академическом мире, а затем уже в качестве стандартной операционной системы для нового поколения многопользовательских микро- и миникомпьютеров в 80-х годах. И этот рост, по-видимому, продолжается в момент написания данной книги.*

Так начиналось первое издание книги (1987 г.). Более чем десять лет спустя операционная система UNIX оправдала возлагавшиеся на нее надежды и теперь является ключевой деталью технологического пейзажа на рубеже XXI века. Не говоря уже о том, что UNIX всегда занимала сильные позиции в научном и техническом сообществах, в настоящее время существует множество крупномасштабных систем управления данными и обработки транзакций на платформе UNIX. Но, самое главное, ОС UNIX, безусловно, является ядром серверов магистральной сети Internet.

Книга «UNIX System Programming» («Системное программирование в UNIX») прекрасно продавалась, причем первое издание с тех пор переиздавалось один или два раза в год. Эта книга оказалась удивительно жизнеспособной, поскольку раскрывала стандартную и стабильную природу операционной системы UNIX. Однако к настоящему моменту накопилось достаточно изменений, чтобы стала ощутимой потребность в публикации второго издания книги; кроме того, мы хотели охватить темы, касающиеся более распределенных конфигураций, типичных для решений в сфере информационных технологий в конце 90-х.

Вместе с тем основная цель при написании второго издания осталась прежней. Основное внимание, как и прежде, уделяется программному интерфейсу между ядром UNIX (частью UNIX, которая делает ее операционной системой) и прикладным программным обеспечением, которое выполняется в среде UNIX. Этот интерфейс часто называется интерфейсом системных вызовов UNIX (хотя разница между такими вызовами и обычными библиотечными процедурами теперь уже не столь очевидна, как это было раньше). В дальнейшем мы увидим, что системные вызовы – это примитивы, на которых в конечном счете построены все программы UNIX – и поставляемые вместе с операционной системой, и разрабатываемые независимо. Целевая аудитория книги состоит из программистов, уже знакомых с UNIX, которые собираются разрабатывать программное обеспечение для ОС UNIX на языке C. Эта книга в равной мере подойдет разработчикам

системного программного обеспечения и прикладных или деловых приложений – фактически всем, кто серьезно интересуется разработкой программ для ОС UNIX.

Кроме системных вызовов мы также рассмотрим некоторые из наиболее важных библиотек подпрограмм, которые поставляются с системой UNIX. Эти библиотеки, конечно же, написаны с использованием системных вызовов и во многих случаях делают то же самое, но на более высоком уровне, или более удобны для использования программистами. Надеемся, что при исследовании как системных вызовов, так и библиотеки подпрограмм вы получите представление о том, когда можно пользоваться существующими достижениями, не «изобретая велосипед», а также лучше поймете внутреннее устройство этой все еще прекрасной операционной системы.

## Спецификация X/Open

ОС UNIX имеет долгую историю, и существовало множество ее официальных и фактических стандартов, а также коммерческих и учебных вариантов. Тем не менее ядро системы UNIX, находящееся в центре внимания в данной книге, остается необычайно стабильным.

Первое издание книги «Системное программирование в UNIX» было основано на втором выпуске AT&T System V Interface Definition (SVID – описания интерфейса System V) – одной из наиболее важных реализаций UNIX. Во втором описании книги мы положили в основу текста и примеров документы (все датированные 1994 годом) System Interface Definitions (Описания системного интерфейса) и System Interfaces and Headers (Системные интерфейсы и заголовки) из 4-й версии второго выпуска X/Open, а также часть связанного с ними документа Networking Services (Сетевые сервисы). Для удобства мы будем применять сокращение XSI – от X/Open System Interfaces (Системные интерфейсы X/Open). При необходимости особенности различных реализаций системы будут обсуждаться отдельно.

Здесь нужно сделать небольшое отступление. Консорциум X/Open первоначально объединил производителей аппаратного обеспечения, серьезно заинтересованных в открытых операционных системах и платформе UNIX, но со временем число его членов возросло. Одна из главных задач консорциума состояла в выработке практического стандарта UNIX, и руководство по обеспечению мобильности программ, называемое XPG, послужило базовым документом для проекта нескольких основных производителей (включая Sun, IBM, Hewlett Packard, Novell и Open Software Foundation), обычно называемого Spec 1170 Initiative (1170 – это число охватываемых этим документом вызовов, заголовков, команд и утилит). Целью этого проекта было создание единой унифицированной спецификации системных сервисов UNIX, включая системные вызовы, которые являются основой этого документа. В результате получился удобный набор спецификаций, объединивший многие конфликтующие направления в стандартизации UNIX, главную часть которых составляют вышеупомянутые документы.

Другие представленные разработки охватывали основные команды UNIX и обработку вывода на экран.

С точки зрения системного программирования, документы XSI формируют практическую базу, и множество примеров из книги будет выполняться на большинстве существующих платформ UNIX. Стандарт X/Open объединяет ряд соответствующих и дополняющих друг друга стандартов с их практической реализацией. Он объединил в себе ANSI/ISO стандарт языка C, важный базовый стандарт POSIX (IEEE 1003.1-1990) и стандарт SVID, а также позаимствовал элементы спецификаций Open Software Foundation (Организации открытого программного обеспечения) и некоторые известные функции из системы Berkeley UNIX, оказавшей большое влияние на развитие UNIX систем в целом.

Конечно, стандартизация продолжилась и в дальнейшем. В 1996 г. в результате слияния X/Open и OSF (Open Software Foundation) образовалась Группа открытых стандартов (The Open Group). Последние разработки (на момент написания книги) из стандарта POSIX, с учетом опыта практической реализации, Группа открытых стандартов называет второй версией Single UNIX Specification (Единой спецификации UNIX, далее по тексту – SUSV2), которая, в свою очередь, содержит пятый выпуск System Interface Definitions, System Interfaces and Headers и Networking Services. Эти важные, хотя и специализированные, дополнения охватывают такие области, как потоки, расширения реального времени и динамическая компоновка.

В заключение заметим, что:

- все стандарты являются очень обширными, охватывая альтернативные методы реализации и редко используемые, но все же важные функциональные возможности. Основное внимание в этой книге уделяется основам программирования в среде UNIX, а не полному описанию системы в соответствии с базовыми стандартами;
- при необходимости создания программы, строго следующей стандартам, необходимо предусмотреть в ней установку (и проверку) соответствующих флагов, таких как `_XOPEN_SOURCE` или `_POSIX_SOURCE`.

## Структура книги

Книга состоит из двенадцати глав.

Глава 1 представляет собой обзор основных понятий и терминологии. Два наиболее важных из обсуждаемых терминов – это *файл* (file) и *процесс* (process). Мы надеемся, что большинство читателей книги уже хотя бы частично знакомы с приведенным в главе материалом (см. в следующем разделе предпосылки для изучения книги).

В главе 2 описаны системные примитивы доступа к файлам, включая открытие и создание файлов, чтение и запись данных в них, а также произвольный доступ к содержимому файлов. Представлены также способы обработки ошибок, которые могут генерироваться системными вызовами.

В главе 3 изучается контекст работы с файлами. В ней рассмотрены вопросы владения файлами, управления системными привилегиями в системе UNIX и оперирования атрибутами файлов при помощи системных вызовов.

Глава 4 посвящена концепции *дерева каталогов* (directories) с точки зрения программиста. В нее также включено краткое обсуждение *файловых систем* (file systems) и *специальных файлов* (special files), используемых для представления устройств.

Глава 5 посвящена природе процессов UNIX и методам работы с ними. В ней представляются и подробно объясняются системные вызовы `fork` и `exec`. Приводится пример простого *командного интерпретатора* (command processor).

Глава 6 – первая из трех глав, посвященных межпроцессному взаимодействию. Она охватывает *сигналы* (signals) и *обработку сигналов* (signal handling) и весьма полезна для перехвата ошибок и обработки аномальных ситуаций.

В главе 7 рассматривается наиболее полезный метод межпроцессного взаимодействия в системе UNIX – *программные каналы*, или *конвейеры* (pipes), позволяющие передавать выход одной программы на вход другой. Будет исследовано создание каналов, чтение и запись с их помощью, а также выбор из множества каналов.

Глава 8 посвящена методам межпроцессного взаимодействия, которые были впервые введены в ОС System V. В ней описаны блокировка записей (record locking), *передача сообщений* (message passing), *семафоры* (semaphores) и *разделяемая память* (shared memory).

В главе 9 рассматривается работа терминала на уровне системных вызовов. Представлен пример использования *псевдотерминалов* (pseudo terminals).

В главе 10 дается краткое описание сетевой организации UNIX и рассматриваются *сокеты* (sockets), которые могут использоваться для пересылки данных между компьютерами.

В главе 11 мы отходим от системных вызовов и начинаем рассмотрение основных библиотек. В этой главе приведено систематическое изложение стандартной библиотеки ввода/вывода (Standard I/O Library), содержащей намного больше средств для работы с файлами, чем системные примитивы, представленные в главе 2.

И, наконец, глава 12 дает обзор дополнительных системных вызовов и библиотечных процедур, многие из которых очень важны при создании реальных программ. Среди обсуждаемых тем – обработка строк, функции работы со временем и функции управления памятью.

## Что вы должны знать

Эта книга не является учебником по системе UNIX или языку программирования C, а подробно исследует интерфейс системных вызовов UNIX. Чтобы использовать ее наилучшим образом, необходимо хорошо изучить следующие темы:

- вход в систему UNIX;
- создание файлов при помощи одного из стандартных редакторов системы;



- древовидную структуру каталогов UNIX;
- основные команды работы с файлами и каталогами;
- создание и компиляцию простых программ на языке C (включая программы, текст которых находится в нескольких файлах);
- процедуры ввода/вывода `printf` и `getchar`;
- использование аргументов командной строки `argc` и `argv`;
- применение map-системы (интерактивного справочного руководства системы). К сожалению, сейчас уже нельзя давать общие советы для работы со справочным руководством в различных системах, поскольку в формат руководства, прежде считавшийся стандартным, были внесены изменения несколькими производителями. Традиционно руководство было разбито на восемь разделов, каждый из которых был структурирован по алфавитному принципу. Наиболее важными являются три из них: раздел 1, описывающий команды; раздел 2, в котором представлены системные вызовы, и раздел 3, охватывающий функции стандартных библиотек.

Тем из вас, кто не знаком с какими-либо темами из приведенного списка, следует выполнить приведенные упражнения. Если потребуется дополнительная помощь, вы можете найти подходящее руководство, воспользовавшись библиографией в конце книги.

Наконец, стоит отметить, что для изучения информатики недостаточно простого чтения, поэтому на протяжении всей книги делается акцент на упражнения и примеры. Для выполнения упражнений вы должны иметь доступ к компьютеру с системой UNIX.

---

**Упражнение 1.** Объясните назначение следующих команд UNIX:

`ls`   `cat`   `rm`   `cp`   `mv`   `mkdir`   `cc`

---

**Упражнение 2.** Создайте небольшой текстовый файл в вашем любимом текстовом редакторе. Создайте другой файл, содержащий пятькратно повторенный первый файл при помощи команды `cat`.

Подсчитайте число слов и символов в обоих файлах при помощи команды `wc`. Объясните полученный результат. Создайте подкаталог и поместите в него оба файла.

---

**Упражнение 3.** Создайте файл, содержащий список файлов в вашем начальном каталоге и в каталоге `/bin`.

---

**Упражнение 4.** Выведите при помощи одной команды число пользователей, находящихся в данный момент в системе.

---

---

**Упражнение 5.** Напишите, откомпилируйте и запустите на выполнение программу на языке C, которая выводит какое-либо приветствие.

---

**Упражнение 6.** Напишите, откомпилируйте и запустите на выполнение программу на языке C, которая печатает свои аргументы.

---

**Упражнение 7.** Напишите программу, которая подсчитывает и выводит число вводимых слов, строк и символов при помощи функций `getchar()` и `printf()`.

---

**Упражнение 8.** Создайте файл, содержащий процедуру на языке C, которая выводит сообщение `'hello, world'`. Создайте отдельный файл основной программы, который вызывает эту процедуру. Откомпилируйте и выполните полученную программу, назвав ее `hw`.

---

**Упражнение 9.** Найдите в руководстве системы разделы, посвященные команде `cat`, процедуре `printf` и системному вызову `write`.

---

## Изменения по сравнению с первым изданием

В этом разделе описаны введенные изменения для тех, кто знаком с первым изданием книги. Если вы читаете эту книгу впервые, можете пропустить этот раздел. Основные изменения заключаются в следующем:

- для описания параметров и возвращения значений, связанных с системными вызовами и другими функциями, использованы обычные методы создания прототипов функций ANSI C (с некоторыми уступками для удобства). Например, в первом издании первоначальное определение `open` выглядело так:

```
#include <fcntl.h>

int filedес, flags;
char *pathname;
.
.
.
fileдес = open (pathname, flags);
```

Во втором издании используется следующий текст:

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

int open (const char *pathname, int flags, [mode_t mode]);
```

- были включены дополнительные типы, введенные в стандарт UNIX за последнее десятилетие, в том числе:

<code>ssize_t</code>	Информация о размере файлов
<code>mode_t</code>	Информация об атрибутах/правах доступа
<code>off_t</code>	Смещение при произвольном доступе к файлам
<code>uid_t</code>	Номер идентификатора пользователя <code>user-id</code>
<code>gid_t</code>	Номер идентификатора группы <code>group-id</code>
<code>pid_t</code>	Номер идентификатора процесса <code>UNIX process-id</code>

В старых версиях UNIX и первом издании книги вместо указанных типов обычно использовались различные целочисленные типы, которые при реализации обычно сводились просто к целым числам. Вышеприведенные типы обеспечивают лучшую мобильность, хотя и, возможно, за счет усложнения и ухудшения удобочитаемости программ.

Изменения по главам:

- глава 1 претерпела лишь небольшие изменения;
- глава 2 в основном осталась без изменений, хотя в ней подчеркнута использование вызова `open` вместо `creat` и представлен вызов `remove` в качестве альтернативы `unlink`;
- изменения в главе 3 включают описание символьных констант, представляющих значения прав доступа, краткое описание вызова `rename` и обсуждение символьных ссылок;
- глава 4 была обновлена и содержит теперь обсуждение вызова `mkdir` и новые процедуры доступа к каталогам. Также представлены вызовы для управления файлами и ограничения каталогов;
- в главе 5 более внимательно рассмотрены вызовы `wait` и `waitpid`, и эта глава теперь также охватывает группы, сеансы и идентификаторы сеансов;
- глава 6 претерпела значительные изменения. Вместо вызова `signal` вводится более безопасный вызов `sigaction` с дополнительными средствами для работы с наборами сигналов;
- в первом издании глава 7 была частью главы 6, но из соображений удобочитаемости она была разбита на две. Большая часть материала мало изменилась, но был добавлен раздел, посвященный примеру использования вызова `select` для работы с набором конвейеров;
- в главе 8 был модернизирован раздел, посвященный блокировке записей. Небольшие изменения были также внесены в разделы, посвященные разделяемой памяти, семафорам и очередям сообщений;
- глава 9 подверглась значительным изменениям и теперь включает больше данных о внутренней структуре ядра, и в ней вместо `ioctl` применяются более удобные для использования структуры `termios`. Основным пример, демонстрирующий передачу файлов, был заменен (учитывая повсеместное объединение компьютеров в сети в настоящее время) на другой, использующий псевдотерминалы;

- ❑ глава 10 отсутствовала в первом издании, и она посвящена использованию сокетов в сетях UNIX-систем;
- ❑ глава 11 (глава 9 в первом издании) претерпела незначительные изменения;
- ❑ глава 12 (глава 11 в первом издании) теперь включает раздел о вводе/выводе с отображением в память;
- ❑ и, наконец, заметим, что глава 10 из первого издания, которая была посвящена библиотеке работы с алфавитно-цифровым экраном `curses`, была опущена, так как она практически не нужна в эпоху графических интерфейсов.



## Соглашения

В книге приняты следующие выделения:

- ❑ моноширинным шрифтом набраны листинги, параметры командной строки, пути к файлам и значения переменных;
- ❑ также моноширинным шрифтом набран вывод на терминал, при этом *курсивом* выделены символы, вводимые пользователем;
- ❑ **полужирным шрифтом** отмечены названия элементов интерфейса, а также клавиши и их комбинации;
- ❑ *курсивом* выделены слова и утверждения, на которые следует обратить особое внимание, а также точки входа указателя.



## Благодарности

Мы хотели бы поблагодарить Стива Пэйта (Steve Pate), Найджела Барнса (Nigel Barnes), Андриса Несторса (Andris Nestors), Джейсона Рида (Jason Reed), Фила Томкинса (Phil Tomkins), Колин Уайт (Colin White), Джо Коули (Joe Cowley) и Викторию Кейв (Victoria Cave). Мы также хотели бы выразить благодарность Дилану Рейзенбергеру (Dylan Reisenberger), Стиву Темблетту (Steve Temblett) и Кэрен Мосмэн (Karen Mosman) за помощь в создании второго издания книги.

Мы также должны поблагодарить тех, кто оказывал нам помощь с первым изданием книги: Стива Ретклиффа (Steve Ratcliffe), который прочитал множество версий каждой из глав первого издания книги и проверил все примеры в нем; и Джонатану Леффлеру (Jonathan Leffler), Грегу Бругхэму (Greg Brougham), Доминик Данлоп (Dominic Dunlop), Найджелу Мартину (Nigel Martin), Биллу Фрейзеру-Кемпбеллу (Bill Fraser-Campbell), Дэйву Льюксу (David Lukes) и Флойд Вильямсу (Floyd Williams) за их замечания, предложения и помощь при подготовке книги.



# Глава 1. Основные понятия и терминология

В этой главе сделан краткий обзор основных идей и терминологии, которые будут использоваться в книге. Начнем с понятия *файл* (file).

## 1.1. Файл

В системе UNIX информация находится в файлах. Типичные команды UNIX, работающие с файлами, включают в себя следующие:

```
$ vi my_test.c
```

которая вызовет редактор `vi` для создания и редактирования файла `my_test.c`;

```
$ cat my_test.c
```

которая выведет на терминал содержимое файла `my_test.c`;

```
$ cc -o my_test my_test.c
```

которая вызовет компилятор `C` для создания программы `my_test` из исходного файла `my_test.c`, если файл `my_test.c` не содержит ошибок.

Большинство файлов будет принадлежать к некоторой логической структуре, заданной пользователем, который их создает. Например, документ может состоять из слов, строк, абзацев и страниц. Тем не менее, с точки зрения системы, все файлы UNIX представляют собой простые неструктурированные последовательности байтов или символов. Предоставляемые системой примитивы позволяют получить доступ к отдельным байтам последовательно или в произвольном порядке. Не существует встроенных в файлы символов конца записи или конца файла, а также различных типов записей, которые нужно было бы согласовывать.

Эта простота является концептуальной для философии UNIX. Файл в системе UNIX является ясным и общим понятием, на основе которого могут быть сконструированы более сложные и специфические структуры (такие как индексная организация файлов). При этом безжалостно устраняются излишние подробности и особые случаи. Например, в обычном текстовом файле символ перехода на следующую строку (обычно символ перевода строки ASCII), определяющий конец строки текста, в системе UNIX представляет собой всего лишь один из символов, который может читаться и записываться системными утилитами и пользовательскими программами. Только программы, предполагающие, что на их вход подается набор строк, должны заботиться о семантике символа перевода строки.

Кроме этого, система UNIX не различает разные типы файлов. Файл может заключать в себе текст (например, файл, содержащий список покупок, или абзац,

который вы сейчас читаете) или содержать «двоичные» данные (такие как откомпилированный код программы). В любом случае для оперирования файлом могут использоваться одни и те же примитивы или утилиты. Вследствие этого, в UNIX отсутствуют формальные схемы присваивания имен файлам, которые существуют в других операционных системах (тем не менее некоторые программы, например `cc`, следуют определенным простым условиям именования файлов). Имена файлов в системе UNIX совершенно произвольны и в системе SVR4 (System V Release 4) могут включать до 255 символов. Тем не менее, для того чтобы быть переносимыми в соответствии со спецификацией XSI, длина имен не должна превышать 14 символов – предела, заложенного в ранних версиях UNIX.

### 1.1.1. Каталоги и пути

Важное понятие, связанное с файлами, – *каталог* (directory). Каталоги представляют собой набор файлов, позволяя сформировать некоторую логическую структуру содержащейся в системе информации. Например, каждый пользователь обычно имеет начальный каталог, в котором он работает, а команды, системные библиотеки и программы администрирования обычно расположены в своих определенных каталогах. Кроме файлов, каталоги также могут содержать произвольное число подкаталогов, которые, в свою очередь, также могут содержать подкаталоги, и так далее. Фактически каталоги могут иметь любую глубину вложенности. Таким образом, файлы UNIX организованы в иерархической древовидной структуре, в которой каждый узел, кроме конечных, соответствует каталогу. Вершиной этого дерева является один каталог, которая обычно называется *корневым каталогом* (root directory).

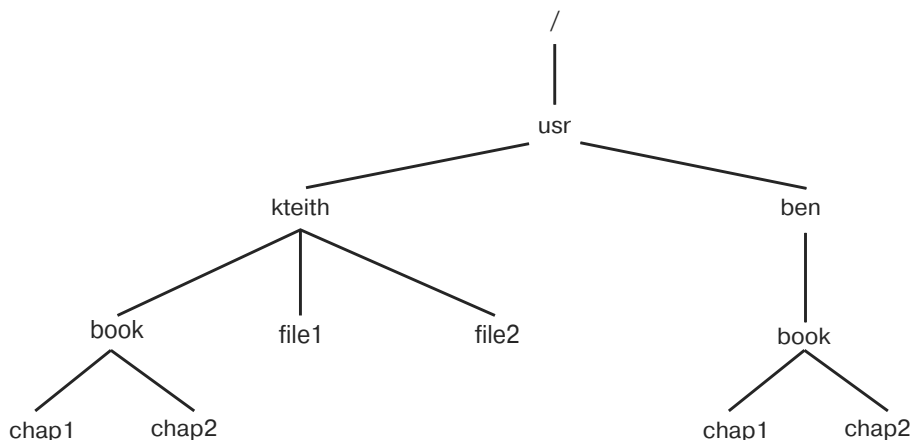


Рис. 1.1. Пример дерева каталогов

Подробное рассмотрение структуры каталогов системы UNIX содержится в главе 4. Тем не менее, поскольку в книге постоянно будет идти речь о файлах UNIX, стоит отметить, что полные их имена, которые называются *путями* (pathnames),



отражают эту древовидную структуру. Каждый путь задает последовательность ведущих к файлу каталогов. Например, полное имя

```
/usr/keith/book/chap1
```

можно разбить на следующие части: первый символ / означает, что путь начинается с корневого каталога, то есть путь дает *абсолютное положение файла* (absolute pathname). Затем идет usr – подкаталог корневого каталога. Каталог keith находится еще на один уровень ниже и поэтому является подкаталогом /usr. Аналогично каталог book является подкаталогом /usr/keith. Последняя часть, chap1, может быть и каталогом, и обычным файлом, поскольку каталоги именуются точно так же, как и обычные файлы. На рис. 1.1 показан пример дерева каталогов, содержащих этот путь.

Путь, который не начинается с символа /, называется *относительным путем* (relative pathname) и задает маршрут к файлу от *текущего рабочего каталога* (current working directory) пользователя. Например, полное имя

```
chap1/intro.txt
```

описывает файл intro.txt, который находится в подкаталоге chap1 текущего каталога. В самом простом случае имя

```
intro.txt
```

просто обозначает файл intro.txt в текущем каталоге. Снова заметим: для того чтобы программа была *действительно* переносимой, каждая из частей полного имени файла должна быть не длиннее 14 символов.

### 1.1.2. Владелец файла и права доступа

Файл характеризуется не только содержащимися в нем данными: существует еще ряд других простых атрибутов, связанных с каждым файлом UNIX. Например, для каждого файла задан определенный пользователь – *владелец файла* (file owner). Владелец файла обладает определенными правами, в том числе возможностью изменять *права доступа* (permissions) к файлу. Как показано в главе 3, права доступа определяют, кто из пользователей может читать или записывать информацию в файл либо запускать его на выполнение, если файл является программой.

### 1.1.3. Обобщение концепции файла

В UNIX концепция файлов расширена и охватывает не только *обычные файлы* (regular files), но и периферийные устройства, а также каналы межпроцессного взаимодействия. Это означает, что одни и те же примитивы могут использоваться для записи и чтения из текстовых и двоичных файлов, терминалов, накопителей на магнитной ленте и даже оперативной памяти. Данная схема позволяет рассматривать программы как обобщенные инструменты, способные использовать любые типы устройств. Например,

```
$ cat file > /dev/rmt0
```

представляет грубый способ записи файла на магнитную ленту (путь /dev/rmt0 обычно обозначает стример).

## 1.2. Процесс

В терминологии UNIX термин *процесс* (process) обычно обозначает экземпляр выполняющейся программы. Простейший способ создания процесса – передать команду для исполнения *оболочке* (shell), которая также называется *командным интерпретатором* (command processor). Например, если пользователь напечатает:

```
$ ls
```

то процесс оболочки создаст другой процесс, в котором будет выполняться программа `ls`, выводящая список файлов в каталоге. UNIX – многозадачная система, поэтому несколько процессов могут выполняться одновременно. Фактически для каждого пользователя, работающего в данный момент с системой UNIX, выполняется хотя бы один процесс.

### 1.2.1. Межпроцессное взаимодействие

Система UNIX позволяет процессам, выполняемым одновременно, взаимодействовать друг с другом, используя ряд методов межпроцессного взаимодействия.

Одним из таких методов является использование *программных каналов* (pipes). Они обычно связывают выход одной программы с входом другой без необходимости сохранения данных в промежуточном файле. Пользователи опять же могут применять эти средства при помощи командного интерпретатора. Командная строка

```
$ ls | wc -l
```

организует конвейер из двух процессов, в одном из которых будет выполняться программа `ls`, а в другом – одновременно программа подсчета числа слов `wc`. При этом выход `ls` будет связан с входом `wc`. В результате будет выведено число файлов в текущем каталоге.

Другими средствами межпроцессного взаимодействия UNIX являются *сигналы* (signals), которые обеспечивают модель взаимодействия по принципу программных прерываний. Дополнительные средства предоставляют *семафоры* (semaphores) и *разделяемая память* (shared memory). Для обмена между процессами одной системы могут также использоваться *сокеты* (sockets), используемые обычно для взаимодействия между процессами в сети.

## 1.3. Системные вызовы и библиотечные подпрограммы

В предисловии уже упомянули, что книга сконцентрирована на *интерфейсе системных вызовов* (system call interface). Тем не менее понятие системного вызова требует дополнительного определения.

Системные вызовы фактически дают разработчику программного обеспечения доступ к *ядру* (kernel). Ядро, с которым познакомились в предисловии, – это блок

кода, который постоянно находится в памяти и занимается планированием системных процессов UNIX и управлением вводом/выводом. По существу ядро является частью UNIX, которое и делает ее операционной системой. Все управление, выделение ресурсов и контроль над пользовательскими процессами, а также весь доступ к файловой системе осуществляется через ядро.

Системные вызовы осуществляются так же, как и вызовы обычных подпрограмм и функций C. Например, можно считать данные из файла, используя библиотечную подпрограмму C `fread`:

```
nread = fread(inputbuf, OBJSIZE, numberobjs, fileptr);
```

или при помощи низкоуровневого системного вызова `read`:

```
nread = read(filedes, inputbuf, BUFSIZE);
```

Основное различие между подпрограммой и системным вызовом состоит в том, что при вызове подпрограммы исполняемый код является частью объектного кода программы, даже если он был скомпонован из библиотеки; при системном вызове основная часть исполняемого кода в действительности является частью ядра, а не вызывающей программы. Другими словами, вызывающая программа напрямую вызывает средства, предоставляемые ядром. Переключение между ядром и пользовательским процессом обычно осуществляется при помощи механизма программных прерываний.

Неудивительно, что большинство системных вызовов выполняет операции либо над файлами, либо над процессами. Фактически системные вызовы составляют основные примитивы, связанные с обоими типами объектов.

При работе с файлами эти операции могут включать передачу данных в файл и из файла, произвольный поиск в файле или изменение связанных с файлом прав доступа.

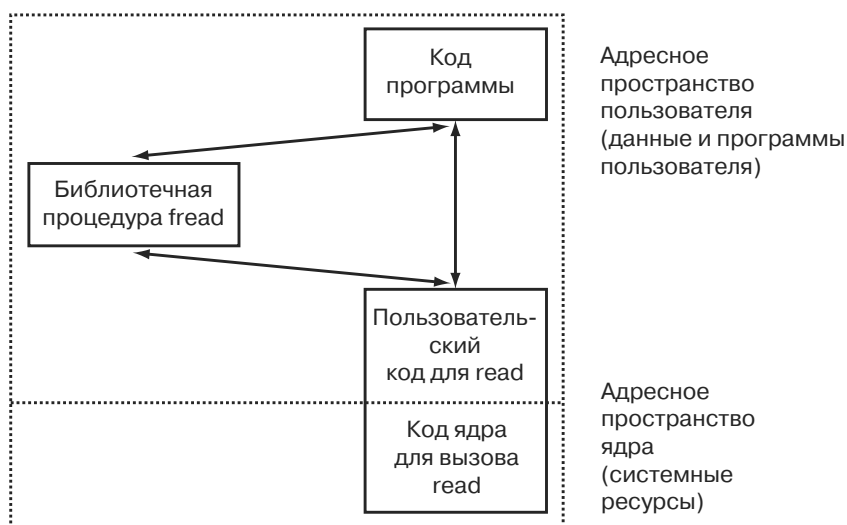


Рис. 1.2. Связь между кодом программы, библиотечной подпрограммой и системным вызовом

При выполнении действий с процессами системные вызовы могут создавать новые процессы, завершать существующие, получать информацию о состоянии процесса и создавать канал взаимодействия между двумя процессами.

Небольшое число системных вызовов не связано ни с файлами, ни с процессами. Обычно системные процессы, входящие в эту категорию, относятся к управлению системой в целом или запросам информации о ней. Например, один из системных вызовов позволяет запросить у ядра текущее время и дату, а другой позволяет переустановить их.

Кроме интерфейса системных вызовов, системы UNIX также предоставляют обширную библиотеку стандартных процедур, одним из важных примеров которых является *Стандартная библиотека ввода/вывода* (Standard I/O Library). Подпрограммы этой библиотеки обеспечивают средства преобразования форматов данных и автоматическую буферизацию, которые отсутствуют в системных вызовах доступа к файлам. Хотя процедуры стандартной библиотеки ввода/вывода гарантируют эффективность, они сами, в конечном счете, используют интерфейс системных вызовов. Их можно представить, как дополнительный уровень средств доступа к файлам, основанный на системных примитивах доступа к файлам, а не отдельную подсистему. Таким образом, любой процесс, взаимодействующий со своим окружением, каким бы незначительным не было это взаимодействие, должен использовать системные вызовы.

На рис. 1.2 показана связь между кодом программы и библиотечной процедурой, а также связь между библиотечной процедурой и системным вызовом. Из рисунка видно, что библиотечная процедура `fread` в конечном итоге является интерфейсом к лежащему в его основе системному вызову `read`.

---

**Упражнение 1.1.** Объясните значение следующих терминов: ядро, системный вызов, подпрограмма *C*, процесс, каталог, путь.

---



# Глава 2. Файл

## 2.1. Примитивы доступа к файлам в системе UNIX

### 2.1.1. Введение

В этой главе будут рассмотрены основные примитивы для работы с файлами, предоставляемые системой UNIX. Эти примитивы состоят из небольшого набора системных вызовов, которые обеспечивают прямой доступ к средствам ввода/вывода, обеспечиваемым ядром UNIX. Они образуют строительные блоки для всего ввода/вывода в системе UNIX, и многие другие механизмы доступа к файлам в конечном счете основаны на них. Названия этих примитивов приведены в табл. 2.1. Дублирование функций, выполняемых различными вызовами, соответствует эволюции UNIX в течение последнего десятилетия.

Типичная программа UNIX вызывает для инициализации файла вызов `open` (или `creat`), а затем использует вызовы `read`, `write` или `lseek` для работы с данными в файле. Если файл больше не нужен программе, она может вызвать `close`, показывая, что работа с файлом завершена. Наконец, если пользователю больше не нужен файл, его можно удалить из системы при помощи вызова `unlink` или `remove`.

Следующая программа, читающая начальный фрагмент некоторого файла, более ясно демонстрирует эту общую схему. Так как это всего лишь вступительный пример, мы опустили некоторые необходимые детали, в частности обработку ошибок. Заметим, что такая практика совершенно недопустима в реальных программах.

Таблица 2.1. Примитивы UNIX

Имя	Функция
<code>open</code>	Открывает файл для чтения или записи либо создает пустой файл
<code>creat</code>	Создает пустой файл
<code>close</code>	Закрывает открытый файл
<code>read</code>	Считывает информацию из файла
<code>write</code>	Записывает информацию в файл
<code>lseek</code>	Перемещается в заданную позицию в файле
<code>unlink</code>	Удаляет файл
<code>remove</code>	Другой метод удаления файла
<code>fcntl</code>	Управляет связанными с файлом атрибутами

```
/* Элементарный пример */

/* Эти заголовки файлов обсуждаются ниже */
#include <fcntl.h>
#include <unistd.h>

main()
{
    int fd;
    ssize_t nread;
    char buf[1024];

    /* Открыть файл "data" для чтения */
    fd = open("data", O_RDONLY);

    /* Прочитать данные */
    nread = read(fd, buf, 1024);

    /* Закрыть файл */
    close(fd);
}
```

Первый системный вызов программа примера делает в строке

```
fd = open("data", O_RDONLY);
```

Это вызов функции `open`, он открывает файл `data` в текущем каталоге. Вторым аргументом функции, `O_RDONLY`, является целочисленной константой, определенной в заголовочном файле `<fcntl.h>`. Это значение указывает на то, что файл должен быть открыт в режиме *только для чтения* (read only). Другими словами, программа сможет только читать содержимое файла и не изменит файл, записав в него какие-либо данные.

Результат вызова `open` крайне важен, в данном примере он помещается в переменную `fd`. Если вызов `open` был успешным, то переменная `fd` будет содержать так называемый *дескриптор файла* (file descriptor) – неотрицательное целое число, значение которого определяется системой. Оно определяет открытый файл при передаче его в качестве параметра другим примитивам доступа к файлам, таким как `read`, `write`, `lseek` и `close`. Если вызов `open` завершается неудачей, то он возвращает значение `-1` (большинство системных вызовов возвращает это значение в случае ошибки). В реальной программе нужно выполнять проверку возвращаемого значения и в случае возникновения ошибки предпринимать соответствующие действия.

После открытия файла программа делает системный вызов `read`:

```
nread = read(fd, buf, 1024);
```

Этот вызов требует считать 1024 символа из файла с идентификатором `fd`, если это возможно, и поместить их в символьный массив `buf`. Возвращаемое значение `nread` дает число считанных символов, которое в нормальной ситуации должно быть равно 1024, но может быть и меньше, если длина файла оказалась меньше 1024 байт. Так же, как и `open`, вызов `read` возвращает в случае ошибки значение `-1`.

Переменная `nread` имеет тип `ssize_t`, определенный в файле `<sys/types.h>`. Этот файл не включен в пример, потому что некоторые основные типы, такие как

`ssize_t`, определены также в файле `<unistd.h>`. Тип `ssize_t` является первым примером различных особых типов, определенных для безопасного использования системных вызовов. Они обычно сводятся к основному целочисленному типу (в первом издании книги переменная `nread` имела тип `int` – в прежних версиях UNIX все было несколько проще).

Этот оператор демонстрирует еще один важный момент: примитивы доступа к файлам имеют дело с простыми линейными последовательностями символов или байтов. Вызов `read`, например, не будет выполнять никаких полезных преобразований типа перевода символьного представления целого числа в форму, используемую для внутреннего представления целых чисел. Не нужно путать системные вызовы `read` и `write` с одноименными операторами более высокого уровня в таких языках, как Fortran или Pascal. Системный вызов `read` типичен для философии, лежащей в основе интерфейса системных вызовов: он выполняет одну простую функцию и представляет собой строительный блок, с помощью которого могут быть реализованы другие возможности.

В конце примера файл закрывается:

```
close(fd);
```

Этот вызов сообщает системе, что программа закончила работу с файлом, связанным с идентификатором `fd`. Легко увидеть, что вызов `close` противоположен вызову `open`. В действительности, так как программа на этом завершает работу, вызов `close` не является необходимым, поскольку все открытые файлы автоматически закрываются при завершении процесса. Тем не менее обязательное использование вызова `close` считается хорошей практикой.

Этот короткий пример должен дать представление о примитивах UNIX для доступа к файлам. Теперь каждый из этих примитивов будет рассмотрен более подробно.

### 2.1.2. Системный вызов `open`

Для выполнения операций записи или чтения данных в существующем файле его следует открыть при помощи системного вызова `open`. Ниже приведено описание этого вызова. Для ясности и согласованности с документацией системы все описания системных вызовов будут использовать структуру прототипов функций ANSI. В них также будут приводиться заголовочные файлы, в которых декларируются прототипы и определяются все важные постоянные:

#### Описание

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

```
int open(const char *pathname, int flags, [mode_t mode]);
```

Первый аргумент, `pathname`, является указателем на строку маршрутного имени открываемого файла. Значение `pathname` может быть абсолютным путем, например:

```
/usr/keith/junk
```

Данный путь задает положение файла по отношению к корневому каталогу. Аргумент `pathname` может также быть относительным путем, задающим маршрут от текущего каталога к файлу, например:

```
keith/junk
```

или просто:

```
junk
```

В последнем случае программа откроет файл `junk` в текущем каталоге. В общем случае, если один из аргументов системного вызова или библиотечной процедуры – имя файла, то в качестве него можно задать любое допустимое маршрутное имя файла UNIX.

Второй аргумент системного вызова `open`, который в нашем описании называется `flags`, имеет целочисленный тип и определяет метод доступа. Параметр `flags` принимает одно из значений, заданных постоянными в заголовочном файле `<fcntl.h>` при помощи директивы препроцессора `#define` (`fcntl` является сокращением от *file control*, «управление файлом»). Так же, как и большинство стандартных заголовочных файлов, файл `<fcntl.h>` обычно находится в каталоге `/usr/include` и может быть включен в программу при помощи директивы: `#include <fcntl.h>`

В файле `<fcntl.h>` определены три постоянных, которые сейчас представляют для нас интерес:

<code>O_RDONLY</code>	Открыть файл только для чтения
<code>O_WRONLY</code>	Открыть файл только для записи
<code>O_RDWR</code>	Открыть файл для чтения и записи

В случае успешного завершения вызова `open` и открытия файла возвращаемое вызовом `open` значение будет содержать неотрицательное целое число – дескриптор файла. Значение дескриптора файла будет наименьшим целым числом, которое еще не было использовано в качестве дескриптора файла выполнившим вызов процессом – знание этого факта иногда может понадобиться. Как отмечено во введении, в случае ошибки вызов `open` возвращает вместо дескриптора файла значение `-1`. Это может произойти, например, если файл не существует. Для создания нового файла можно использовать вызов `open` с параметром `flags`, равным `O_CREAT`, – эта операция описана в следующем разделе.

**Необязательный** (optional) третий параметр `mode`, используемый только вместе с флагом `O_CREAT`, также будет обсуждаться в следующем разделе – он связан с правами доступа к файлу. Следует обратить внимание на квадратные скобки в описании, которые обозначают, что параметр `mode` является необязательным.

Следующий фрагмент программы открывает файл `junk` для чтения и записи и проверяет, не возникает ли при этом ошибка. Этот последний момент особенно важен: имеет смысл устанавливать проверку ошибок во все программы, которые используют системные вызовы, поскольку каким бы простым не было приложение, иногда может произойти сбой. В этом примере используются библиотечные процедуры `printf` для вывода сообщения и `exit` – для завершения процесса. Обе эти процедуры являются стандартными в любой системе UNIX.



```
#include <stdlib.h>      /* Для вызова exit */
#include <fcntl.h>

char *workfile="junk"; /* Задать имя рабочего файла */
main()
{
    int filedес;

    /* Открыть файл, используя постоянную O_RDWR из <fcntl.h> */
    /* Файл открывается для чтения/записи */

    if((fileдес = open(workfile, O_RDWR)) == -1)
    {
        printf("Невозможно открыть %s\n", workfile);
        exit(1);      /* Выход по ошибке */
    }

    /* Остальная программа */

    exit(0);          /* Нормальный выход */
}
```

Обратите внимание, что используется `exit` с параметром 1 в случае ошибки, и 0 – в случае удачного завершения. Это соответствует соглашениям UNIX и является правильной практикой программирования. Как будет показано в следующих главах, после завершения программы можно получить передаваемый вызову `exit` аргумент (*program's exit status* – код завершения программы). Следует также обратить внимание на использование заголовочного файла `<stdlib.h>`, который содержит прототип системного вызова `exit`.

### Предостережение

Здесь необходимо сделать несколько предостережений. Во-первых, существует предельное число файлов, которые могут быть одновременно открыты программой, – в стандарте POSIX (а следовательно, и в спецификации XSI) не менее двадцати.<sup>1</sup> Чтобы обойти эту проблему, требуется использовать системный вызов `close`, тем самым сообщая системе, что работа с файлом закончена. Вызов `close` будет рассмотрен в разделе 2.1.5. Может также существовать предел суммарного числа файлов, открытых всеми процессами, определяемый размером таблицы в ядре. Во-вторых, в ранних версиях UNIX не существовал заголовочный файл `<fcntl.h>`, и в параметре `flags` непосредственно задавались численные значения. Это все еще достаточно распространенный, хотя и не совсем удовлетворительный прием – задание численных значений вместо имен постоянных, определенных в файле `<fcntl.h>`. Поэтому может встретиться оператор типа

```
fileдес = open(filename, 0);
```

который в обычных условиях открывает файл только для чтения и эквивалентен следующему оператору:

```
fileдес = open (filename, O_RDONLY);
```

---

<sup>1</sup> Согласно спецификации SUSV2 заголовочный файл `limits.h` должен определять константу `OPEN_MAX`, задающую это значение. – *Прим. науч. ред.*

**Упражнение 2.1.** *Создайте небольшую программу, описанную выше. Проверьте ее работу при условии, что файл junk не существует. Затем создайте файл junk с помощью текстового редактора и снова запустите программу. Содержимое файла junk не имеет значения.*

### 2.1.3. Создание файла при помощи вызова open

Вызов open может использоваться для создания файла, например:

```
filedes = open("/tmp/newfile", O_WRONLY | O_CREAT, 0644);
```

Здесь объединены флаги O\_CREAT и O\_WRONLY, задающие создание файла /tmp/newfile при помощи вызова open. Если /tmp/newfile не существует, то будет создан файл нулевой длины с таким именем и открыт только для записи.

В этом примере вводится третий параметр mode вызова open, который нужен только при создании файла. Не углубляясь в детали, заметим, что параметр mode содержит число, определяющее *права доступа* (access permissions) к файлу, указывающие, кто из пользователей системы может осуществлять чтение, запись или выполнение файла. В вышеприведенном примере используется значение 0644. При этом пользователь, создавший файл, может выполнять чтение из файла и запись в него. Остальные пользователи будут иметь доступ только для чтения файла. В следующей главе показано, как вычисляется это значение. Для простоты оно будет использовано во всех примерах этой главы.

Следующая программа создает файл newfile в текущем каталоге:

```
#include <stdlib.h>
#include <fcntl.h>
#define PERMS 0644 /* Права доступа при открытии с O_CREAT */

char *filename="newfile";

main()
{
    int filedes;

    if((filedes = open (filename, O_RDWR|O_CREAT, PERMS)) == -1)
    {
        printf("Невозможно создать %s\n",filename);
        exit(1);      /* Выход по ошибке */
    }

    /* Остальная программа */

    exit(0);
}
```

Что произойдет, если файл newfile уже существует? Если позволяют права доступа к нему, то он будет открыт на запись, как если бы флаг O\_CREAT не был задан. В этом случае параметр mode не будет иметь силы. С другой стороны, объединение флагов O\_CREAT и O\_EXCL (exclusive – исключительный) приведет к ошибке во время вызова create, если файл уже существует. Например, следующий вызов

```
fd = open("lock", O_WRONLY|O_CREAT|O_EXCL, 0644);
```

означает, что если файл `lock` не существует, его следует создать с правами доступа `0644`. Если же он существует, то в переменную `fd` будет записано значение `-1`, свидетельствующее об ошибке. Имя файла `lock` (защелка) показывает, что он создается для обозначения исключительного доступа к некоторому ресурсу.

Еще один полезный флаг – флаг `O_TRUNC`. При его использовании вместе с флагом `O_CREAT` файл будет усечен до нулевого размера, если он существует, и права доступа к файлу позволяют это. Например:

```
fd = open ("file", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Это может понадобиться, если вы хотите, чтобы программа писала данные поверх данных, записанных во время предыдущих запусков программы.

---

**Упражнение 2.2.** *Интересно, что флаг `O_TRUNC` может использоваться и без флага `O_CREAT`. Попробуйте предугадать, что при этом получится, а затем проверьте это при помощи программы в случаях, когда файл существует и не существует.*

---

### 2.1.4. Системный вызов `creat`

Другой способ создания файла заключается в использовании системного вызова `creat`. В действительности это *исходный* способ создания файла, но сейчас он в какой-то мере является излишним и предоставляет меньше возможностей, чем вызов `open`. Мы включили его для полноты описания. Так же, как и вызов `open`, он возвращает либо ненулевой дескриптор файла, либо `-1` в случае ошибки. Если файл успешно создан, то возвращаемое значение является дескриптором этого файла, открытого для записи. Вызов `creat` осуществляется так:

#### Описание

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

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

Первый параметр `pathname` указывает на маршрутное имя файла UNIX, определяющее имя создаваемого файла и путь к нему. Так же, как в случае вызова `open`, параметр `mode` задает права доступа. При этом, если файл существует, то второй параметр также игнорируется. Тем не менее, в отличие от вызова `open`, в результате вызова `creat` файл всегда будет усечен до нулевой длины. Пример использования вызова `creat`:

```
filedes = creat("/tmp/newfile", 0644);
```

что эквивалентно вызову

```
filedes = open("/tmp/newfile", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Следует отметить, что вызов `creat` всегда открывает файл только для записи. Например, программа не может создать файл при помощи `creat`, записать в него данные, затем вернуться назад и попытаться прочитать данные из файла, если предварительно не закроет его и не откроет снова при помощи вызова `open`.

**Упражнение 2.3.** *Напишите небольшую программу, которая вначале создает файл при помощи вызова `creat`, затем, не вызывая `close`, сразу же открывает его при помощи системного вызова `open` для чтения в одном случае и записи в другом. В обоих случаях выведите сообщение об успешном или неуспешном завершении, используя `printf`.*

---

### 2.1.5. Системный вызов `close`

Системный вызов `close` противоположен вызову `open`. Он сообщает системе, что вызывающий его процесс завершил работу с файлом. Необходимость этого вызова определяется тем, что число файлов, которые могут быть одновременно открыты программой, ограничено.

#### Описание

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

```
int close(int filedес);
```

Системный вызов `close` имеет всего один аргумент – дескриптор закрываемого файла, обычно получаемый в результате предыдущего вызова `open` или `creat`. Следующий фрагмент программы поясняет простую связь между вызовами `open` и `close`:

```
fileдес = open("file", O_RDONLY);
```

```
.  
.   
.
```

```
close(fileдес);
```

Системный вызов `close` возвращает 0 в случае успешного завершения и `-1` – в случае ошибки (которая может возникнуть, если целочисленный аргумент не является допустимым дескриптором файла).

При завершении работы программы все открытые файлы закрываются автоматически.

### 2.1.6. Системный вызов `read`

Системный вызов `read` используется для копирования произвольного числа символов или байтов из файла в буфер. Буфер формально устанавливается как указатель на переменную типа `void`; это означает, что он может содержать элементы любого типа. Хотя обычно буфер является массивом данных типа `char`, он также может быть массивом структур, определенных пользователем.

Заметим, что программисты на языке C часто любят использовать термины «символ» и «байт» как взаимозаменяемые. Байт является единицей памяти, необходимой для хранения символа, и на большинстве машин имеет длину восемь бит. Термин «символ» обычно описывает элемент из набора символов ASCII, который является комбинацией всего из семи бит. Поэтому обычно байт может

содержать больше значений, чем число символов ASCII; такая ситуация возникает при работе с двоичными данными. Тип `char` языка C представляет более общее понятие байта, поэтому название данного типа является не совсем правильным.

### Описание

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

```
ssize_t read(int filedes, void *buffer, size_t n);
```

Первый параметр, `filedes`, является дескриптором файла, полученным во время предыдущего вызова `open` или `creat`. Второй параметр, `buffer`, – это указатель на массив или структуру, в которую должны копироваться данные. Во многих случаях в качестве этого параметра будет выступать просто имя массива, например:

```
int fd;
ssize_t nread;
char buffer[SOMEVALUE];

/* Дескриптор файла fd получен в результате вызова open */
.
.
.
nread = read(fd, buffer, SOMEVALUE);
```

Как видно из примера, третьим параметром вызова `read` является положительное число (имеющее специальный тип `size_t`), задающее число байтов, которое требуется считать из файла.

Возвращаемое вызовом `read` число (присваиваемое в примере переменной `nread`) содержит число байтов, которое было считано в действительности. Обычно это число запрошенных программой байтов, но, как будет показано в дальнейшем, – не всегда, и значение переменной `nread` может быть меньше. Кроме того, в случае ошибки вызов `read` возвращает значение `-1`. Это происходит, например, если передать `read` недопустимый дескриптор файла.

### Указатель чтения-записи

Достаточно естественно, что программа может последовательно вызывать `read` для просмотра файла. Например, если предположить, что файл `foo` содержит не менее 1024 символов, то следующий фрагмент кода поместит первые 512 символов из файла `foo` в массив `buf1`, а вторые 512 символов – в массив `buf2`.

```
int fd;
ssize_t n1, n2;
char buf1[512], buf2[512];
.
.
.
if(( fd = open("foo", O_RDONLY)) == -1)
    return (-1);

n1 = read(fd, buf1, 512);
n2 = read(fd, buf2, 512);
```

Система отслеживает текущее положение в файле при помощи объекта, который называется *указателем ввода/вывода* (read-write pointer), или *указателем файла* (file pointer). По существу, в этом указателе записано положение очередного байта в файле, который должен быть считан (или записан) следующим для определенного дескриптора файла; следовательно, указатель файла можно себе представить в виде закладки. Его значение отслеживает система, и программисту нет необходимости выделять под него переменную. Произвольный доступ, при котором положение указателя ввода/вывода изменяется явно, может осуществляться при помощи системного вызова `lseek`, который описан в разделе 2.1.10. В случае вызова `read` система просто перемещает указатель ввода/вывода вперед на число байтов, считанных в результате данного вызова.

Поскольку вызов `read` может использоваться для просмотра файла с начала и до конца, программа должна иметь возможность определять конец файла. При этом становится важным возвращаемое вызовом `read` значение. Если число запрошенных во время вызова `read` символов больше, чем оставшееся число символов в файле, то система передаст только оставшиеся символы, установив соответствующее возвращаемое значение. Любые последующие вызовы `read` будут возвращать значение 0. При этом больше не останется данных, которые осталось бы прочитать. Обычным способом проверки достижения конца файла в программе является проверка равенства нулю значения, возвращаемого вызовом `read`, по крайней мере, для программы, использующей вызов `read`.

Следующая программа `count` иллюстрирует некоторые из этих моментов:

```
/* Программа count - подсчитывает число символов в файле */  
  
#include <stdlib.h>  
#include <fcntl.h>  
#include <unistd.h>  
  
#define BUFSIZE 512  
  
main()  
{  
    char buffer[BUFSIZE];  
    int filedес;  
    ssize_t nread;  
    long total = 0;  
  
    /* Открыть файл "anotherfile" только для чтения */  
    if(( filedес = open("anotherfile", O_RDONLY)) == -1)  
    {  
        printf ("Ошибка при открытии файла anotherfile\n");  
        exit(1);  
    }  
  
    /* Повторять до конца файла, пока nread не будет равно 0 */  
    while( (nread = read(fileдес, buffer, BUFSIZE)) >0)  
        total += nread; /* Увеличить total на единицу */
```

```
printf("число символов в файле anotherfile: %ld\n", total);  
exit(0);  
}
```

Эта программа будет выполнять чтение из файла `anotherfile` блоками по 512 байт. После каждого вызова `read` значение переменной `total` будет увеличиваться на число символов, действительно скопированных в массив `buffer`. Почему `total` объявлена как переменная типа `long`?

Здесь использовано для числа считываемых за один раз символов значение 512, поскольку система UNIX сконфигурирована таким образом, что наибольшая производительность достигается при перемещении данных блоками, размер которых кратен размеру блока на диске, в этом случае 512. (В действительности размер блока зависит от конкретной системы и может составлять до и более 8 Кбайт.) Тем не менее мы могли бы задавать в вызове `read` произвольное число, в том числе единицу. Введение определенного значения, соответствующего вашей системе, не дает выигрыша в функциональности, а лишь повышает производительность программы, но, как мы увидим в разделе 2.1.9, это улучшение может быть значительным.

Чтобы учесть действительный размер блока системы, можно использовать определение `BUFSIZ` из файла `/usr/include/stdio.h` (который связан со стандартной библиотекой ввода/вывода). Например:

```
#include <stdio.h>  
  
.  
.  
.  
nread = read(filedes, buffer, BUFSIZ);
```

---

**Упражнение 2.4.** Если вы знаете, как это сделать, перепишите программу `count` так, чтобы вместо использования постоянного имени файла она принимала его в качестве аргумента командной строки. Проверьте работу программы на небольшом файле, состоящем из нескольких строк.

---

**Упражнение 2.5.** Измените программу `count` так, чтобы она также выводила число слов и строк в файле. Определите слово как знак препинания или алфавитно-цифровую строку, не содержащую пробельных символов, таких как пробел, табуляция или перевод строки. Строкой, конечно же, будет любая последовательность символов, завершающаяся символом перевода строки.

---

## 2.1.7. Системный вызов `write`

Системный вызов `write` противоположен вызову `read`. Он копирует данные из буфера программы, рассматриваемого как массив, во внешний файл.

### Описание

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

```
ssize_t write(int filedес, const void *buffer, size_t n);
```

Так же, как и вызов `read`, вызов `write` имеет три аргумента: дескриптор файла `fileдес`, указатель на записываемые данные `buffer` и `n` – положительное число записываемых байтов. Возвращаемое вызовом значение является либо числом записанных символов, либо кодом ошибки `-1`. Фактически, если возвращаемое значение не равно `-1`, то оно почти всегда будет равно `n`. Если оно меньше `n`, значит, возникли какие-то серьезные проблемы. Например, это может произойти, если в процессе вызова `write` было исчерпано свободное пространство на выходном носителе. (Если носитель уже был заполнен до вызова `write`, то вызов вернет значение `-1`.)

Вызов `write` часто использует дескриптор файла, полученный при создании нового файла. Легко увидеть, что происходит в этом случае. Изначально файл имеет нулевую длину (он только что создан или получен усечением существующего файла до нулевой длины), и каждый вызов `write` просто дописывает данные в конец файла, перемещая указатель чтения-записи на позицию, следующую за последним записанным байтом. Например, в случае удачного завершения фрагмент кода

```
int fd;
ssize_t w1, w2;
char header1[512], header2[1024];
.
.
.
if ( fd = open("newfile", O_WRONLY|O_CREAT|O_EXCL, 0644)) == -1)
    return (-1);

w1 = write(fd, header1, 512);
w2 = write(fd, header2, 1024);
```

дает в результате файл длиной 1536 байт, с содержимым массивов `header1` и `header2`.

Что произойдет, если программа откроет существующий файл на запись и сразу же запишет в него что-нибудь? Ответ очень прост: старые данные в файле будут заменены новыми, символ за символом. Например, предположим, что файл `oldhat` имеет длину 500 символов. Если программа откроет файл `oldhat` для записи и выведет в него 10 символов, то первые 10 символов в файле будут заменены содержимым буфера записи программы. Следующий вызов `write` заменит очередные 10 символов и так далее. После достижения конца исходного файла в процессе дальнейших вызовов `write` его длина будет увеличиваться. Если нужно избежать переписывания файла, можно открыть файл с флагом `O_APPEND`. Например:

```
fileдес = open(filename, O_WRONLY|O_APPEND);
```

Теперь в случае успешного вызова `open` указатель чтения-записи будет помещен сразу же за последним байтом в файле, и вызов `write` будет добавлять данные в конец файла. Этот прием более подробно будет объяснен в разделе 2.1.12.



### 2.1.8. Пример *copyfile*

Теперь можем закрепить материал на практике. Задача состоит в написании функции `copyfile`, которая будет копировать содержимое одного файла в другой. Возвращаемое значение должно быть равно нулю в случае успеха или отрицательному числу – в случае ошибки.

Основная логика действий понятна: открыть первый файл, затем создать второй и выполнять чтение из первого файла и запись во второй до тех пор, пока не будет достигнут конец первого файла. И, наконец, закрыть оба файла.

Окончательное решение может выглядеть таким образом:

```
/* Программа copyfile - скопировать файл name1 в файл name2 */
#include <unistd.h>
#include <fcntl.h>
#define BUFSIZE 512    /* Размер считываемого блока */
#define PERM 0644      /* Права доступа для нового файла */

/* Скопировать файл name1 в файл name2 */
int copyfile( const char *name1, const char *name2)
{
    int infile, outfile;
    ssize_t nread;
    char buffer[BUFSIZE];

    if( ( infile = open(name1, O_RDONLY ) ) == -1)
        return (-1);
    if((outfile=open(name2,O_WRONLY|O_CREAT|O_TRUNC,PERM))== -1)
    {
        close(infile);
        return (-2);
    }

    /* Чтение из файла name1 по BUFSIZE символов */
    while( (nread = read (infile, buffer, BUFSIZE) ) > 0)
    {
        /* Записать buffer в выходной файл */
        if( write(outfile, buffer, nread) < nread )
        {
            close(infile);
            close(outfile);
            return (-3); /* Ошибка записи */
        }
    }

    close(infile);
    close(outfile);

    if( nread == -1)
        return (-4); /* Ошибка при последнем чтении */
    else
        /* Все в порядке */
        return (0);
}
```

Теперь функцию `copyfile` можно вызывать так:

```
retcode = copyfile("squarepeg", "roundhole");
```

**Упражнение 2.6.** Измените функцию `copyfile` так, чтобы в качестве ее параметров могли выступать дескрипторы, а не имена файлов. Проверьте работу новой версии программы.

**Упражнение 2.7.** Если вы умеете работать с аргументами командной строки, используйте одну из процедур `copyfile` для создания программы `тупср`, копирующей первый заданный в командной строке файл во второй.

### 2.1.9. Эффективность вызовов `read` и `write`

Процедура `copyfile` дает возможность оценить эффективность примитивов доступа к файлам в зависимости от размера буфера. Один из методов заключается просто в компиляции `copyfile` с различными значениями `BUFSIZE`, а затем – в измерении времени ее выполнения при помощи команды UNIX `time`. Мы сделали это, используя функцию `main`

```
/* Функция main для тестирования функции copyfile */
main()
{
    copyfile("test.in", "test.out");
}
```

и получили при копировании одного и того же большого файла (68307 байт) на компьютере с системой SVR4 UNIX для диска, разбитого на блоки по 512 байт, результаты, приведенные в табл. 2.2.

Таблица 2.2. Результаты тестирования функции `copyfile`

BUFSIZE	Real time	User time	System time
1	0:24.49	0:3.13	0:21.16
64	0:0.46	0:0.12	0:0.33
512	0:0.12	0:0.02	0:0.08
4096	0:0.07	0:0.00	0:0.05
8192	0:0.07	0:0.01	0:0.05

Формат данных в таблице отражает вывод команды `time`. В первом столбце приведены значения `BUFSIZE`, во втором – действительное время выполнения процесса в минутах, секундах и десятых долях секунды. В третьем столбце приведено «пользовательское» время, то есть время, занятое частями программы, не являющимися системными вызовами. Из-за дискретности используемого таймера одно из значений в таблице ошибочно записано как нулевое. В последнем, четвертом, столбце приведено время, затраченное ядром на обслуживание системных вызовов. Как видно из таблицы, третий и четвертый столбцы в сумме не дают

действительное время выполнения. Это связано с тем, что в системе UNIX одновременно выполняется несколько процессов. Не все время тратится на выполнение ваших программ!

Полученные результаты достаточно убедительны – чтение и запись по одному байту дает очень низкую производительность, тогда как увеличение размера буфера значительно повышает производительность. Наибольшая производительность достигается, если BUFSIZE кратно размеру блока диска на диске, как видно из результатов, для значений BUFSIZE 512, 4096 и 8192 байта.

Следует также отметить, что большая часть прироста (но не всего) эффективности получается просто от уменьшения числа системных вызовов. Переключение между программой и ядром может обойтись достаточно дорого. В общем случае, если нужна максимальная производительность, следует минимизировать число генерируемых программой системных вызовов.

### 2.1.10. Вызов *lseek* и произвольный доступ

Системный вызов `lseek` позволяет пользователю изменять положение указателя чтения-записи, то есть изменять номер байта, который будет первым считан или записан в следующей операции ввода/вывода. Таким образом, использование вызова `lseek` позволяет получить произвольный доступ к файлу.

#### Описание

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

```
off_t lseek(int filedес, off_t offset, int start_flag);
```

Первый параметр, `fileдес`, – это дескриптор открытого файла. Второй параметр, `offset`, обычно определяет новое положение указателя чтения-записи и задает число байтов, которое нужно добавить к начальному положению указателя. Третий целочисленный параметр, `start_flag`, определяет, что принимается в качестве начального положения, то есть откуда вычисляется смещение `offset`. Флаг `start_flag` может принимать одно из символьных значений (определенных в файле `<unistd.h>`), как показано ниже:

SEEK_SET	Смещение <code>offset</code> вычисляется от начала файла, обычно имеет значение = (int)0
SEEK_CUR	Смещение <code>offset</code> вычисляется от текущего положения в файле, обычное значение = (int)1
SEEK_END	Смещение <code>offset</code> вычисляется от конца файла, обычное значение = (int)2

Эти значения показаны в графическом виде на рис. 2.1, на котором представлен файл из 7 байт.

Пример использования вызова `lseek`:

```
off_t newpos;
.
.
.
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

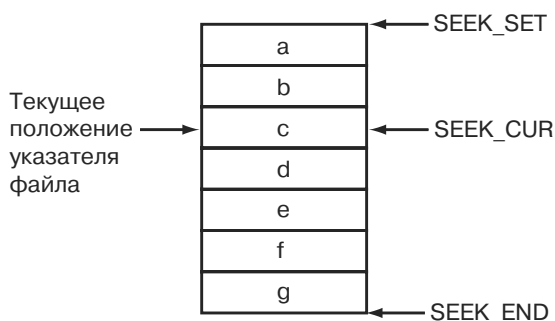


Рис. 2.1  
Символьные значения флага `start_flag`

который задает положение указателя в 16 байтах от конца файла. Обратите внимание на приведение значения `-16` байт к типу `off_t`.

Во всех случаях возвращаемое значение (содержащееся в переменной `newpos` в примере) дает новое положение в файле. В случае ошибки оно будет содержать стандартный код ошибки `-1`.

Существует ряд моментов, которые следует отметить. Во-первых, обе переменные `newpos` и `offset` имеют тип `off_t`, определенный в файле `<sys/types.h>`, и должны вмещать смещение для любого файла в системе. Во-вторых, как показано в примере, смещение `offset` может быть отрицательным. Другими словами, возможно перемещение в обратную сторону от начального положения, заданного флагом `offset_flag`. Ошибка возникнет только при попытке переместиться при этом на позицию, находящуюся до начала файла. В-третьих, можно задать позицию за концом файла. В этом случае, очевидно, не существует данных, которые можно было бы прочесть – невозможно предугадать будущие записи в этот участок (UNIX не имеет машины времени) – но последующий вызов `write` имеет смысл и приведет к увеличению размера файла. Пустое пространство между старым концом файла и начальным положением новых данных не обязательно выделяется физически, но для последующих вызовов `read` оно будет выглядеть как заполненное символами `null ASCII`.

В качестве простого примера мы можем создать фрагмент программы, который будет дописывать данные в конец существующего файла, открывая файл, перемещаясь на его конец при помощи вызова `lseek` и начиная запись:

```
filesdes = open(filename, O_RDWR);
lseek(filesdes, (off_t)0, SEEK_END);
write(filesdes, outbuf, OBSIZE);
```

Здесь параметр направления поиска для вызова `lseek` установлен равным `SEEK_END` для перемещения в конец файла. Так как перемещаться дальше нам не нужно, то смещение задано равным нулю.

Вызов `lseek` также может использоваться для получения размера файла, так как он возвращает новое положение в файле.

```
off_t filesize;
int filesdes;
.
.
.
filesize = lseek(filesdes, (off_t)0, SEEK_END);
```

**Упражнение 2.8.** Напишите функцию, которая использует вызов `lseek` для получения размера открытого файла, не изменяя при этом значения указателя чтения-записи.

### 2.1.11. Пример: гостиница

В качестве несколько надуманного, но возможно наглядного примера, предположим, что имеется файл `residents`, в котором записаны фамилии постояльцев гостиницы. Первая строка содержит фамилию жильца комнаты 1, вторая – жильца комнаты 2 и т.д. (очевидно, это гостиница с прекрасно организованной системой нумерации комнат). Длина каждой строки составляет ровно 41 символ, в первые 40 из которых записана фамилия жильца, а 41-й символ является символом перевода строки для того, чтобы файл можно было вывести на дисплей при помощи команды UNIX `cat`.

Следующая функция `getoccupier` вычисляет по заданному целому номеру комнаты положение первого байта фамилии жильца, затем перемещается в эту позицию и считывает данные. Она возвращает либо указатель на строку с фамилией жильца, либо нулевой указатель в случае ошибки (мы будем использовать для этого значение `NULL`, определенное в файле `<stdio.h>`). Обратите внимание, что мы присвоили переменной дескриптора файла `infile` исходное значение `-1`. Благодаря этому мы можем гарантировать, что файл будет открыт всего один раз.

```
/* Функция getoccupier - получить фамилию из файла residents */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define NAMELENGTH 41

char namebuf[NAMELENGTH]; /* Буфер для фамилии */
int infile = -1;           /* Для хранения дескриптора файла */
char *getoccupier(int roomno)
{
    off_t offset;
    ssize_t nread;

    /* Убедиться, что файл открывается впервые */
    if( infile == -1 &&
        (infile = open("residents", O_RDONLY)) == -1)
    {
        return (NULL);    /* Невозможно открыть файл */
    }

    offset = (roomno - 1) * NAMELENGTH;

    /* Найти поле комнаты и считать фамилию жильца */
    if(lseek(infile, offset, SEEK_SET) == -1)
        return (NULL);

    if( (nread = read(infile, namebuf, NAMELENGTH)) <= 0)
        return (NULL);
```

```

/* Создать строку, заменив символ перевода строки на "\0" */
namebuf[nread - 1] = "\0";
return (namebuf);
}

```

Если предположить, что в гостинице 10 комнат, то следующая программа будет последовательно вызывать функцию `getoccupier` для просмотра файла и выводить каждую найденную фамилию при помощи процедуры `printf` из стандартной библиотеки ввода/вывода:

```

/* Программа listoc - выводит все фамилии жильцов */
#define NROOMS 10

main ()
{
    int j;
    char *getoccupier(int), *p;
    for( j = 1; j <= NROOMS; j++)
    {
        if(p = getoccupier(j))
            printf("Комната %2d, %s\n", j, p);
        else
            printf("Ошибка для комнаты %d\n", j);
    }
}

```

---

**Упражнение 2.9.** Придумайте алгоритм для определения пустых комнат. Измените функцию `getoccupier` и файл данных, если это необходимо, так, чтобы он отражал эти изменения. Затем напишите процедуру с названием `findfree` для поиска свободной комнаты с наименьшим номером.

---



---

**Упражнение 2.10.** Напишите процедуру `freeroom` для удаления записи о жильце. Затем напишите процедуру `addguest` для внесения новой записи о жильце, с предварительной проверкой того, что выделяемая комната свободна.

---



---

**Упражнение 2.11.** Объедините процедуры `getoccupier`, `freeroom`, `addguest` и `findfree` в простой программе с названием `frontdesk`, которая управляет файлом данных. Используйте аргументы командной строки или напишите интерактивную программу, которая вызывает функции `printf` и `getchar`. В обоих случаях для вычисления номера комнаты вам потребуется преобразовывать строки в целые числа. Вы можете использовать для этого библиотечную процедуру `atoi`:

```
i = atoi(string);
```

где `string` – указатель на строку символов, а `i` – целое число.

---

**Упражнение 2.12.** В качестве обобщенного примера напишите программу на основе системного вызова `lseek`, которая копирует в обратном порядке байты из одного файла в другой. Насколько эффективным получилось ваше решение?

**Упражнение 2.13.** Используя вызов `lseek`, напишите процедуры для копирования последних 10 символов, последних 10 слов и последних 10 строк из одного файла в другой.

### 2.1.12. Дописывание данных в конец файла

Как должно быть ясно из раздела 2.1.10, для дописывания данных в конец файла может использоваться следующий код:

```
/* Поиск конца файла */
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, appbuf, BUFSIZE);
```

Тем не менее более изящный способ состоит в использовании одного из дополнительных флагов вызова `open`, `O_APPEND`. Если установлен этот флаг, то перед каждой записью указатель будет устанавливаться в конец файла. Это может быть полезно, если нужно лишь дополнить файл, застраховавшись от случайной перезаписи данных в начале файла.

Можно использовать флаг `O_APPEND` следующим образом:

```
filedes = open("yetanother", O_WRONLY | O_APPEND);
```

Каждый последующий вызов `write` будет дописывать данные в конец файла. Например:

```
write(filedes, appbuf, BUFSIZE);
```

**Упражнение 2.14.** Напишите процедуру `fileopen`, имеющую два аргумента: первый – строку, содержащую имя файла, и второй – строку, которая может иметь одно из следующих значений:

- `r` – открыть файл только для чтения;
- `w` – открыть файл только для записи;
- `rw` – открыть файл для чтения и записи;
- `a` – открыть файл для дописывания.

процедура `fileopen` должна возвращать дескриптор файла или код ошибки –1.

### 2.1.13. Удаление файла

Существует два метода удаления файла из системы – при помощи вызовов `unlink` и `remove`.

#### Описание

```
#include <unistd.h>
int unlink(const char *pathname);
```

```
#include <stdio.h>
int remove(const char *pathname);
```

Оба вызова имеют единственный аргумент – строку с именем удаляемого файла, например:

```
unlink ("/tmp/usedfile");
remove ("/tmp/tmpfile");
```

Оба вызова возвращают 0 в случае успешного завершения и -1 – в случае ошибки.

Зачем нужны два различных вызова? Вначале существовал только вызов `unlink`. Вызов `remove`, определенный в стандарте ANSI C, является недавним дополнением к спецификации XSI. При удалении обычных файлов вызов `remove` идентичен вызову `unlink`. При удалении пустых каталогов вызов `remove(path)` эквивалентен вызову `rmdir(path)` – другому системному вызову, который всегда должен использоваться для удаления каталогов вместо `unlink`. Мы снова встретимся с ним в главе 4.

### 2.1.14. Системный вызов `fcntl`

Системный вызов `fcntl` был введен для управления уже открытыми файлами. Это довольно странная конструкция, которая может выполнять разные функции.

#### Описание

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

/\* Примечание: тип последнего параметра может меняться,  
что показано многоточием "...". \*/

```
int fcntl(int filedes, int cmd, ...);
```

Системный вызов `fcntl` работает с открытым файлом, заданным дескриптором файла `filedes`. Конкретная выполняемая функция задается выбором одного из значений параметра `cmd` из файла `<fcntl.h>`. Тип третьего параметра зависит от значения параметра `cmd`. Например, если вызов `fcntl` используется для установки флагов статуса файла, тогда третий параметр будет целым числом. Если же, как можно будет увидеть позже, вызов `fcntl` будет использоваться для блокировки файла, то третий параметр будет указателем на структуру `lock`. Иногда третий параметр вообще не используется.

Некоторые из этих функций относятся к взаимодействию файлов и процессов, и мы не будем рассматривать их здесь; тем не менее две из этих функций, заданные значениями `F_GETFL` и `F_SETFL` параметра `cmd`, представляют для нас сейчас интерес.

При задании параметра `F_GETFL` вызов `fcntl` возвращает текущие флаги статуса файла, установленные вызовом `open`. Следующая функция `filestatus` использует `fcntl` для вывода текущего статуса открытого файла.



```
/* Функция filestatus - узнает текущий статус файла */
#include <fcntl.h>

int filestatus(int filedes)
{
    int arg1;

    if(( arg1 = fcntl (filedes, F_GETFL)) == -1)
    {
        printf("Ошибка чтения статуса файла\n");
        return (-1);
    }
    printf("Дескриптор файла %d: ",filedes);

    /* Сравнить аргумент с флагами открытия файла */
    switch( arg1 & O_ACCMODE){
    case O_WRONLY:
        printf("Только для записи");
        break;
    case O_RDWR:
        printf("Для чтения-записи");
        break;
    case O_RDONLY:
        printf("Только для чтения");
        break;
    default:
        printf("Режим не существует");
    }

    if(arg1 & O_APPEND)
        printf(" - установлен флаг append");

    printf("\n");
    return (0);
}
```

Следует обратить внимание на проверку установки определенного бита во флаги статуса файла в переменной `arg1` при помощи побитового оператора И, обозначаемого одиночным символом `&`. Поле интересующих нас битов вырезается с помощью специальной маски `O_ACCMODE`, определенной в файле `<fcntl.h>`. Дальнейшие действия осуществляются с учетом того, что в данном поле не может быть выставлено более одного бита, поскольку эти три режима доступа к файлу не совместимы.

Значение `F_SETFL` используется для переустановки связанных с файлом флагов статуса. Новые флаги задаются в третьем аргументе вызова `fcntl`. При этом могут быть установлены только некоторые флаги, например, нельзя вдруг превратить файл, открытый только для чтения, в файл, открытый для чтения и записи. Тем не менее с помощью `F_SETFL` можно задать режим, при котором все следующие операции записи будут только дописывать информацию в конец файла:

```
if( fcntl(filedes, F_SETFL, O_APPEND) == -1)
    printf("Ошибка вызова fcntl \n");
```

## 2.2. Стандартный ввод, стандартный вывод и стандартный вывод диагностики

### 2.2.1. Основные понятия

Система UNIX автоматически открывает три дескриптора файла для любой выполняющейся программы. Эти дескрипторы называются *стандартным вводом* (standard input), *стандартным выводом* (standard output) и *стандартным выводом диагностики* (standard error). Они всегда имеют значения 0, 1 и 2 соответственно. Недопустимо путать эти дескрипторы с похожими по названию стандартными потоками `stdin`, `stdout` и `stderr` из стандартной библиотеки ввода/вывода.

По умолчанию вызов `read` для стандартного ввода приведет к чтению данных с клавиатуры. Аналогично запись в стандартный вывод или стандартный вывод диагностики приведет по умолчанию к выводу сообщения на экран терминала. Это первый пример использования примитивов доступа к файлам для ввода/вывода на устройства, отличные от обычных файлов.

Программа, применяющая эти стандартные дескрипторы файлов, тем не менее, не ограничена использованием терминала. Каждый из этих дескрипторов может быть независимо переназначен, если программа вызывается с использованием средств перенаправления, обеспечиваемых стандартным командным интерпретатором UNIX. Например, команда

```
$ prog_name < infile
```

приведет к тому, что при чтении из дескриптора со значением 0 программа будет получать данные из файла `infile`, а не с терминала, обычного источника для стандартного ввода.

Аналогично все данные, записываемые в стандартный вывод, могут быть перенаправлены в выходной файл, например:

```
$ prog_name > outfile
```

Возможно, наиболее полезно то, что можно связать стандартный вывод одной программы со стандартным вводом другой при помощи каналов UNIX. Следующая команда оболочки означает, что все данные, записываемые программой `prog_1` в ее стандартный вывод, попадут на стандартный ввод программы `prog_2` (такие команды называются конвейерами):

```
$ prog_1 | prog_2
```

Дескрипторы файлов стандартного ввода и вывода позволяют писать гибкие и совместимые программы. Программа может представлять собой инструмент, способный при необходимости принимать ввод от пользователя, из файла, или даже с выхода другой программы. Программа настроена на чтение из стандартного ввода, использует файловый дескриптор 0, а выбор входного источника данных откладывается до момента запуска программы.

## 2.2.2. Программа *io*

В качестве очень простого примера использования стандартных дескрипторов файлов приведем программу *io*, применяющую системные вызовы *read* и *write* и дескрипторы файлов со значениями 0 и 1 для копирования стандартного ввода в стандартный вывод. В сущности, это усеченная версия программы UNIX *cat*. Обратите внимание на отсутствие вызовов *open* и *creat*.

```
/* Программа io - копирует стандартный ввод */
/* в стандартный вывод */
#include <stdlib.h>
#include <unistd.h>
#define SIZE 512

main()
{
    ssize_t nread;
    char buf[SIZE];

    while ( (nread = read(0, buf, SIZE)) > 0)
        write(1, buf, nread);
    exit(0);
}
```

Предположим, что исходный код этой программы находится в файле *io.c*, который компилируется для получения исполняемого файла *io*:

```
$ cc -o io io.c
```

Если теперь запустить программу *io* на выполнение, просто набрав имя файла программы, то она будет ожидать ввода с терминала. Если пользователь напечатает строку и затем нажмет клавишу **Return** или **Enter** на клавиатуре, то программа *io* просто выведет на дисплей напечатанную строку, то есть запишет строку в стандартный вывод. При этом диалог с системой в действительности будет выглядеть примерно так:

\$ io	Пользователь печатает <i>io</i> и нажимает Return
Это строка 1	Пользователь печатает строку и нажимает Return
Это строка 1	Программа <i>io</i> выводит строку на дисплей
.	
.	
.	

После вывода строки на экран программа *io* будет ожидать дальнейшего ввода. Пользователь может продолжать печатать, и программа *io* будет послушно выводить каждую строку на экран при нажатии на клавишу **Return** или **Enter**.

Для завершения программы пользователь может напечатать строку из единственного символа конца файла. Обычно это символ **^D**, то есть **Ctrl+D**, который набирается одновременным нажатием клавиш **Ctrl** и **D**. При этом вызов *read* вернет 0, указывая на то, что достигнут конец файла. Весь диалог с системой мог бы выглядеть примерно так:

```
$ io
Это строка 1
Это строка 1
Это строка 2
Это строка 2
<Ctrl-D>
$
```

Пользователь печатает **Ctrl+D**

Обратите внимание, что программа `io` ведет себя не совсем так, как можно было бы ожидать. Вместо того чтобы считать все 512 символов до начала вывода на экран, как, казалось бы, следует делать, она выводит строку на экран при каждом нажатии клавиши **Return**. Это происходит из-за того, что вызов `read`, который использовался для ввода данных с терминала, обычно возвращает значение после каждого символа перевода строки для облегчения взаимодействия с пользователем. Если быть еще более точным, это будет иметь место только для обычных настроек терминала. Терминалы могут быть настроены в другом режиме, позволяя осуществлять, например, посимвольный ввод. Дополнительные соображения по этому поводу изложены в главе 9.

Поскольку программа `io` использует стандартные дескрипторы файлов, к ней можно применить стандартные средства оболочки для перенаправления и организации конвейеров. Например, выполнение команды

```
$ io < /etc/motd > message
```

приведет к копированию при помощи программы `io` сообщения с цитатой дня команды `/etc/motd` в файл `message`, а выполнение команды

```
$ io < /etc/motd | wc
```

направит стандартный вывод программы `io` в утилиту UNIX для подсчета числа слов `wc`. Так как стандартный вывод программы `io` будет фактически идентичен содержимому `/etc/motd`, это просто еще один (более громоздкий) способ подсчета слов, строк и символов в файле.

---

**Упражнение 2.15.** Напишите версию программы `io`, которая проверяет наличие аргументов командной строки. Если существует хотя бы один из них, то программа должна рассматривать каждый из аргументов как имя файла и копировать содержимое каждого файла в стандартный вывод. Если аргументы командной строки отсутствуют, то ввод должен осуществляться из стандартного ввода. Как должна действовать программа `io`, если она не может открыть файл?

---

---

**Упражнение 2.16.** Иногда данные в файле могут медленно накапливаться в течение продолжительного промежутка времени. Напишите версию программы `io` с именем `watch`, которая будет выполнять чтение из стандартного ввода до тех пор, пока не встретится символ конца файла, выводя данные на стандартный вывод. После

достижения конца файла программа `watch` должна сделать паузу на пять секунд, а затем снова начать чтение стандартного ввода, чтобы проверить, не поступили ли новые данные, не открывая при этом файл заново и не изменяя положение указателя чтения-записи. Для прекращения работы процесса на заданное время вы можете использовать стандартную библиотечную процедуру `sleep`, которая имеет единственный аргумент — целое число, задающее продолжительность ожидания в секундах. Например, вызов

```
sleep(5);
```

заставляет процесс прекратить работу на 5 секунд. Программа `watch` аналогична программе `readslow`, существующей в некоторых версиях UNIX. Посмотрите также в руководстве системы описание ключа `-f` команды `tail`.

---

### 2.2.3. Использование стандартного вывода диагностики

Стандартный вывод диагностики является особым файловым дескриптором, который по принятому соглашению зарезервирован для сообщений об ошибках и для предупреждений, что позволяет программе отделить обычный вывод от сообщений об ошибках. Например, использование стандартного вывода диагностики позволяет программе выводить сообщения об ошибках на терминал, в то время как стандартный вывод записывается в файл. Тем не менее при необходимости стандартный вывод диагностики может быть перенаправлен аналогично перенаправлению стандартного вывода. Например, часто используется такая форма команды запуска системы `make`:

```
$ make > log.out 2>log.err
```

В результате все сообщения об ошибках работы `make` направляются в файл `log.err`, а стандартный вывод направляется в файл `log.out`.

Можно выводить сообщения в стандартный вывод диагностики при помощи системного вызова `write` со значением дескриптора файла равным 2:

```
char msg[6]="boob\n";  
.  
.  
write(2, msg, 5);
```

Тем не менее это достаточно грубый и громоздкий способ. Мы приведем лучшее решение в конце этой главы.

## 2.3. Стандартная библиотека ввода/вывода: взгляд в будущее

Системные вызовы доступа к файлам лежат в основе всего ввода и вывода программ UNIX. Тем не менее эти вызовы действительно примитивны и работают

с данными только как с простыми последовательностями байтов, оставляя все остальное на усмотрение программиста. Соображения эффективности также ложатся на плечи разработчика.

Чтобы несколько упростить ситуацию, система UNIX предоставляет стандартную библиотеку ввода/вывода, которая содержит намного больше средств, чем уже описанные системные вызовы. Поскольку книга в основном посвящена интерфейсу системных вызовов с ядром, подробное рассмотрение стандартной библиотеки ввода/вывода отложено до главы 11. Тем не менее для сравнения стоит и в этой главе кратко описать возможности стандартного ввода/вывода.

Возможно, наиболее очевидное отличие между стандартным вводом/выводом и примитивами системных вызовов состоит в способе описания файлов. Вместо целочисленных дескрипторов файлов, процедуры стандартного ввода/вывода явно или неявно работают со структурой `FILE`. Следующий пример показывает, как открывается файл при помощи процедуры `fopen`:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *stream;

    if( ( stream = fopen("junk", "r") ) == NULL)
    {
        printf("Невозможно открыть файл junk\n");
        exit(1);
    }
}
```

Первая строка примера

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

подключает заголовочный файл стандартной библиотеки ввода/вывода `<stdio.h>`. Этот файл, кроме всего прочего, содержит определение `FILE`, `NULL` и объявления `extern` для таких функций, как `fopen`. На сегодняшний день `NULL` также определяется в файле `<unistd.h>`.

Настоящее содержание этого примера заключается в операторе:

```
if( ( stream = fopen("junk", "r") ) == NULL)
{
    .
    .
}
```

Здесь `junk` – это имя файла, а строка `"r"` означает, что файл открывается только для чтения. Строка `"w"` может использоваться для усечения файла до нулевой длины или создания файла и открытия его на запись. В случае успеха функция `fopen` проинициализирует структуру `FILE` и вернет ее адрес в переменной `stream`. Указатель `stream` может быть передан другим процедурам из библиотеки. Важно понимать, что где-то внутри тела функции `fopen` осуществляется вызов нашего

старого знакомого `open`. И, естественно, где-то внутри структуры `FILE` находится дескриптор файла, привязывающий структуру к файлу. Существенно то, что процедуры стандартного ввода/вывода написаны на основе примитивов системных вызовов. Основная функция библиотеки состоит в создании более удобного интерфейса и автоматической буферизации.

После открытия файла существует множество стандартных процедур ввода/вывода для доступа к нему. Одна из них – процедура `getc`, считывающая одиночный символ, другая, `putc`, выводит один символ. Они используются следующим образом:

### Описание

```
#include <stdio.h>

int getc(FILE *istream); /* Считать символ из istream */

int putc(int c, FILE *ostream); /* Поместить символ в ostream */
```

Можно поместить обе процедуры в цикл для копирования одного файла в другой:

```
int c;
FILE *istream, *ostream;

/* Открыть файл istream для чтения и файл ostream для записи */
.
.
.
while( ( c = getc(istream)) != EOF)
    putc (c, ostream);
```

Значение `EOF` определено в файле `<stdio.h>`, и оно возвращается функцией `getc` при достижении конца файла. В действительности значение `EOF` равно `-1`, поэтому тип возвращаемого функцией `getc` значения определен как `int`.

На первый взгляд, функции `getc` и `putc` могут вызывать определенное беспокойство, поскольку они работают с одиночными символами, а это, как уже было продемонстрировано на примере системных вызовов, чрезвычайно неэффективно. Процедуры стандартного ввода/вывода избегают этой неэффективности при помощи изящного механизма буферизации, который работает следующим образом: первый вызов функции `getc` приводит к чтению из файла `BUFSIZ` символов при помощи системного вызова `read` (как показано в разделе 2.1.6, `BUFSIZ` – это постоянная, определенная в файле `<stdio.h>`). Данные находятся в буфере, созданном библиотекой и находящемся в пользовательском адресном пространстве. Функция `getc` возвращает только первый символ. Все остальные внутренние действия скрыты от вызывающей программы. Последующие вызовы функции `getc` поочередно возвращают символы из буфера. После того как при помощи функции `getc` программе будут переданы `BUFSIZ` символов и будет выполнен очередной вызов `getc`, из файла снова будет считан буфер целиком. Аналогичный механизм реализован в функции `putc`.

Такой подход весьма удобен, так как он освобождает программиста от беспокойства по поводу эффективности работы программы. Это также означает, что

данные записываются большими блоками, и запись в файл будет производиться с задержкой (для терминалов сделаны специальные оговорки). Поэтому весьма неблагоприятно использовать для одного и того же файла и стандартные процедуры ввода/вывода, и системные вызовы, такие как `read`, `write` или `lseek`. Это может привести к хаосу, если не представлять четко, что при этом происходит. С другой стороны, вполне допустимо смешивать системные вызовы и процедуры стандартного ввода/вывода для разных файлов.

Кроме механизма буферизации стандартный ввод/вывод предоставляет утилиты для форматирования и преобразования, например, функция `printf` обеспечивает форматированный вывод:

```
printf("Целое число %d\n", ival);
```

Эта функция должна быть уже известна большинству читателей этой книги. (Между прочим, функция `printf` неявно осуществляет запись в стандартный вывод.)

### **Вывод сообщений об ошибках при помощи функции `fprintf`**

Функция `printf` может использоваться для вывода диагностических ошибок. К сожалению, она осуществляет запись в стандартный вывод, а не в стандартный вывод диагностики. Тем не менее можно использовать для этого функцию `fprintf`, которая является обобщением функции `printf`. Следующий фрагмент программы показывает, как можно это сделать:

```
#include <stdio.h>      /* Для определения stderr */
.
.
fprintf(stderr, "Ошибка номер %d\n", errno);
```

Единственное отличие между использованием `fprintf` и вызовом `printf` заключается в параметре `stderr`, являющемся указателем на стандартную структуру `FILE`, автоматически связанную с потоком вывода стандартной диагностики.

Следующая процедура расширяет возможности использования функции `fprintf` в более общей процедуре вывода сообщения об ошибке:

```
/* Функция notfound - вывести сообщение об ошибке и выйти */
#include <stdio.h>
#include <stdlib.h>
int notfound(const char *programe, const char *filename)
{
    fprintf(stderr, "%s: файл %s не найден\n", programe, filename);
    exit(1);
}
```

В последующих примерах для вывода сообщений об ошибках будет использована функция `fprintf`, а не `printf`. Это обеспечит совместимость с большинством команд и программ, применяющих для диагностики стандартный вывод диагностики.

## **2.4. Системные вызовы и переменная `errno`**

Из вышеизложенного материала видно, что все описанные до сих пор системные вызовы файлового ввода/вывода могут завершиться неудачей. В этом случае



возвращаемое значение всегда равно `-1`. Чтобы помочь программисту получить информацию о причине ошибки, система UNIX предоставляет глобальную целочисленную переменную, содержащую код ошибки. Значение кода ошибки связано с сообщением об ошибке, таким как *no permission* (нет доступа) или *invalid argument* (недопустимый аргумент). Полный список кодов и описаний ошибок приведен в Приложении 1. Текущее значение кода ошибки соответствует типу последней ошибки, произошедшей во время системного вызова.

Переменная, содержащая код ошибки, имеет имя `errno` (сокращение от *error number* – номер ошибки). Программист может использовать переменную `errno` в программе на языке C, подключив заголовочный файл `<errno.h>`. Многие старые программы требуют, чтобы переменная `errno` была объявлена при компоновке как внешняя, но теперь это требование является устаревшим в соответствии со спецификацией XSI.<sup>1</sup>

Следующая программа использует вызов `open`, и в случае его неудачного завершения использует функцию `fprintf` для вывода значения переменной `errno`:

```
/* Программа err1.c - открывает файл с обработкой ошибок */

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
    int fd;
    if( (fd = open ("nonesuch", O_RDONLY)) == -1)
        fprintf(stderr, "Ошибка %d\n", errno);
}
```

Если, например, файл `nonesuch` не существует, то код соответствующей ошибки в стандартной реализации UNIX будет равен 2. Так же, как и остальные возможные значения переменной `errno`, этот код является значением определенной в заголовочном файле `<errno.h>` константы, в данном случае – константы `ENOENT`, имя которой является сокращением от *no such entry* (нет такого файла или каталога). Эти константы можно непосредственно использовать в программе.

При использовании переменной `errno` следует проявлять осторожность, так как при следующем системном вызове ее значение не сбрасывается. Поэтому наиболее безопасно использовать `errno` сразу же после неудачного системного вызова.

### 2.4.1. Подпрограмма `perror`

Кроме `errno` UNIX обеспечивает библиотечную процедуру (не системный вызов) `perror`. Для большинства традиционных команд UNIX использование `perror` является стандартным способом вывода сообщений об ошибках. Она имеет единственный аргумент строчного типа и при вызове отправляет на стандартный вывод диагностики сообщение, состоящее из переданного ей строчного аргумента,

<sup>1</sup> Кроме того, современные стандарты допускают реализацию `errno` в виде макроса, возвращающего выражение вида `*errno_object`, которое может использоваться по обе стороны оператора присваивания. – *Прим. науч. ред.*

двоеточия и дополнительного описания ошибки, соответствующей текущему значению переменной `errno`. Важно, что сообщение об ошибке отправляется на стандартный вывод диагностики, а не в стандартный вывод.

В вышеприведенном примере можно заменить строку, содержащую вызов `printf`, на:

```
perror("Ошибка при открытии файла nonesuch");
```

Если файл `nonesuch` не существует, то функция `perror` выведет сообщение:

```
Ошибка при открытии файла nonesuch: No such file or directory
```

---

**Упражнение 2.17.** *Напишите процедуры, выполняющие те же действия, что и примитивы доступа к файлам, описанные в этой главе, но вызывающие функцию `perror` при возникновении ошибок или исключительных ситуаций.*

---



## Глава 3. Работа с файлами

Файлы не определяются полностью содержащимися в них данными. Каждый файл UNIX содержит ряд простых дополнительных свойств, необходимых для администрирования этой сложной многопользовательской системы. В данной главе будут изучены дополнительные свойства и оперирующие ими системные вызовы.

### 3.1. Файлы в многопользовательской среде

#### 3.1.1. Пользователи и права доступа

Для каждого файла в системе UNIX задан его *владелец* (owner – один из пользователей системы; обычно пользователь, создавший файл). Истинный идентификатор пользователя представлен неотрицательным числом *user-id*, сокращенно *uid*, которое связывается с файлом при его создании.

В типичной системе UNIX связанный с определенным именем пользователя идентификатор *uid* находится в третьем поле записи о пользователе в файле паролей, то есть в строке файла `/etc/passwd/`, которая идентифицирует пользователя в системе. Типичная запись

```
keith:x:35:10::/usr/keith:/bin/ksh
```

указывает, что пользователь `keith` имеет *uid* 35.

Поля в записи о пользователе в файле паролей разделяются двоеточием. Первое поле задает имя пользователя. Второе, в данном случае `x`, – это маркер пароля пользователя. В отличие от ранних версий UNIX сам зашифрованный пароль обычно находится в другом файле, отличающемся для разных систем. Как уже было показано, третье поле содержит идентификатор пользователя *uid*. В четвертом поле находится идентификатор группы пользователя по умолчанию – *group-id*, сокращенно *gid*; подробнее он будет рассмотрен ниже. Пятое поле – это необязательное поле комментария. Шестое задает домашний каталог пользователя. Последнее поле – полное имя программы, которая запускается после входа пользователя в систему. Например, `/bin/ksh` – одна из стандартных оболочек UNIX.

Фактически для идентификации пользователя в системе UNIX нужен только идентификатор *user-id*. Каждый процесс UNIX обычно связывается с идентификатором пользователя, который запустил его на выполнение. При этом процесс является просто экземпляром выполняемой программы. При создании файла система устанавливает его владельца на основе идентификатора *uid*, создающего файл процесса.

Владелец файла позже может быть изменен, но только суперпользователем или владельцем файла. Следует отметить, что суперпользователь имеет имя root и его идентификатор uid всегда равен 0.

Помимо владельца файлы могут быть связаны с *группами пользователей* (groups), которые представляют произвольный набор пользователей, благодаря чему становится возможным простой способ управления проектами, включающими несколько человек. Каждый пользователь принадлежит как минимум к одной группе.

Группы пользователей определяются в файле /etc/group. Каждая из них определена своим идентификатором *gid*, который, как и uid, является неотрицательным числом. Группа пользователя по умолчанию задается четвертым полем записи о нем в файле паролей.

Так же, как идентификатор пользователя uid, идентификатор группы gid пользователя наследуется процессом, который запускает пользователь. Поэтому при создании файла связанный с создающим его процессом идентификатор группы gid записывается наряду с идентификатором пользователя uid.

### **Действующие идентификаторы пользователей и групп**

Необходимо сделать одно уточнение: создание файла определяется связанным с процессом *действующим идентификатором пользователя euid* (effective user-id). Хотя процесс может быть запущен одним пользователем (скажем, keith), при определенных обстоятельствах он может получить права доступа другого пользователя (например, dina). Вскоре будет показано, как это можно осуществить. Идентификатор пользователя, запустившего процесс, называется *истинным идентификатором пользователя* (real user-id, сокращенно ruuid) этого процесса. Разумеется, в большинстве случаев действующий и истинный идентификаторы пользователя совпадают.

Аналогично с процессом связывается *действующий идентификатор группы* (effective group-id, сокращенно egid), который может отличаться от *истинного идентификатора группы* (real group-id, сокращенно rgid).

### **3.1.2. Права доступа и режимы файлов**

Владелец обладает исключительными правами обращения с файлами.

В частности, владелец может изменять связанные с файлом *права доступа* (permissions). Права доступа определяют возможности доступа к файлу других пользователей. Они затрагивают три группы пользователей:

- владелец файла;
- все пользователи, кроме владельца файла, принадлежащие к связанной с файлом группе;
- все пользователи, не входящие в категории 1 или 2.

Для каждой категории пользователей существуют три основных типа прав доступа к файлам. Они определяют, может ли пользователь определенной категории выполнять:

- чтение из файла;
- запись в файл;

□ запуск файла на выполнение. В этом случае файл обычно является программой или последовательностью команд оболочки.

Как обычно, суперпользователь выделен в отдельную категорию и может оперировать любыми файлами, независимо от связанных с ними прав чтения, записи или выполнения.

Система хранит связанные с файлом права доступа в битовой маске, называемой *кодом доступа к файлу* (file mode). Хотя заголовочный файл `<sys/stat.h>` и определяет символьные имена для битов прав доступа, большинство программистов все еще предпочитает использовать восьмеричные постоянные, приведенные в табл. 3.1 – при этом символьные имена являются относительно недавним и весьма неудобным нововведением. Следует обратить внимание, что в языке C восьмеричные постоянные всегда начинаются с нуля, иначе компилятор будет расценивать их как десятичные.

Таблица 3.1. Восьмеричные значения для прав доступа к файлам

Восьмеричное значение	Символьное обозначение	Значение
0400	S_IRUSR	Владелец имеет доступ для чтения
0200	S_IWUSR	Владелец имеет доступ для записи
0100	S_IXUSR	Владелец может выполнять файл
0040	S_IRGRP	Группа имеет доступ для чтения
0020	S_IWGRP	Группа имеет доступ для записи
0010	S_IXGRP	Группа может выполнять файл
0004	S_IROTH	Другие пользователи имеют доступ для чтения
0002	S_IWOTH	Другие пользователи имеют доступ для записи
0001	S_IXOTH	Другие пользователи могут выполнять файл

Из таблицы легко увидеть, что можно сделать файл доступным для чтения всем типам пользователей, сложив 0400 (доступ на чтение для владельца), 040 (доступ на чтение для членов группы файла) и 04 (доступ на чтение для всех остальных пользователей). В итоге это дает код доступа к файлу 0444. Такой код может быть получен и при помощи побитовой операции ИЛИ (|) для соответствующих символьных представлений; например, 0444 эквивалентен выражению:

```
S_IRUSR | S_IRGRP | S_IROTH
```

Поскольку все остальные значения из таблицы не включены, код доступа 0444 также означает, что никто из пользователей, включая владельца файла, не может получить доступ к файлу на запись или выполнение.

Чтобы устранить это неудобство, можно использовать более одного восьмеричного значения, относящегося к одной категории пользователей. Например, сложив 0400, 0200 и 0100, получим в сумме значение 0700, которое показывает, что владелец файла может читать его, производить в него запись и запускать файл на выполнение.

Поэтому чаще встречается значение кода доступа:

0700 + 050 + 05 = 0755

Это означает, что владелец файла может читать и писать в файл или запускать файл на выполнение, в то время как права членов группы, связанной с файлом, и всех остальных пользователей ограничены только чтением или выполнением файла.

Легко понять, почему программисты UNIX предпочитают использовать восьмеричные постоянные, а не имена констант из файла `<sys/stat.h>`, когда простое значение 0755 представляется выражением:

```
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

Рассказ о правах доступа еще не закончен. В следующем подразделе будет продемонстрировано, как три других типа прав доступа влияют на файлы, содержащие исполняемые программы. В отношении доступа к файлам важно то, что каждый каталог UNIX, почти как обычный файл, имеет набор прав доступа, которые влияют на доступность файлов в каталоге. Этот вопрос будет подробно рассмотрен в главе 4.

**Упражнение 3.1.** Что означают следующие значения прав доступа: 0761, 0777, 0555, 0007 и 0707?

**Упражнение 3.2.** Замените восьмеричные значения из упражнения 3.1 эквивалентными символьными выражениями.

**Упражнение 3.3.** Напишите процедуру `lsoct`, которая переводит набор прав доступа из формы, получаемой на выходе команды `ls` (например, `rwxr-xr-x`) в эквивалентные восьмеричные значения. Затем напишите обратную процедуру `octls`.

### 3.1.3. Дополнительные права доступа для исполняемых файлов

Существуют еще три типа прав доступа к файлам, задающие особые атрибуты и обычно имеющие смысл только в том случае, если файл содержит исполняемую программу. Соответствующие восьмеричные значения и символьные имена также соответствуют определенным битам в коде доступа к файлу и обозначают следующее:

04000	<code>S_ISUID</code>	Задать user-id при выполнении
02000	<code>S_ISGID</code>	Задать group-id при выполнении
01000	<code>S_ISVTX</code>	Сохранить сегмент кода (бит фиксации)

Если установлен флаг доступа `S_ISUID`, то при запуске на выполнение находящейся в файле программы система задает в качестве действующего идентификатора пользователя полученного процесса не идентификатор пользователя, запустившего процесс (как обычно), а идентификатор владельца файла. Процессу при этом присваиваются права доступа владельца файла, а не пользователя, запустившего процесс.

Подобный механизм может использоваться для управления доступом к критическим данным. Конфиденциальная информация может быть защищена от публичного доступа или изменения при помощи стандартных прав доступа на

чение/запись/выполнение. Владелец файла создает программу, которая будет предоставлять ограниченный доступ к файлу. Затем для файла программы устанавливается флаг доступа `S_ISUID`, что позволяет другим пользователям получать ограниченный доступ к файлу только при помощи данной программы. Очевидно, программа должна быть написана аккуратно во избежание случайного и умышленного нарушения защиты.<sup>1</sup>

Классический пример этого подхода представляет программа `passwd`. Администратор системы ожидает неприятности, если он позволит всем пользователям выполнять запись в файл паролей. Тем не менее все пользователи должны иногда изменять этот файл при смене своего пароля. Решить проблему позволяет программа `passwd`, так как ее владельцем является суперпользователь и для нее установлен флаг `S_ISUID`.

Не столь полезна установка флага `S_ISGID`, которая выполняет те же функции для идентификатора группы файла `group-id`. Если указанный флаг установлен, то при запуске файла на выполнение получившийся процесс получает действующий идентификатор группы `egid` владельца файла, а не пользователя, который запустил программу на выполнение.

Исторически бит `S_ISVTX` обычно использовался для исполняемых файлов и назывался флагом *сохранения сегмента кода* (save-text-image), или *битом фиксации* (sticky bit). В ранних версиях системы, если для файла был установлен этот бит, при его выполнении код программы оставался в файле подкачки до выключения системы. Поэтому при следующем запуске программы системе не приходилось искать файл в структуре каталогов системы, а можно было просто и быстро переместить программу в память из файла подкачки. В современных системах UNIX указанный бит является избыточным, и в спецификации XSI бит `S_ISVTX` определен только для каталогов. Использование `S_ISVTX` будет подробнее рассмотрено в главе 4.

---

**Упражнение 3.4.** Следующие примеры показывают, как команда `ls` выводит на экран права доступа `set-user-id` и `group-id`, соответственно:

```
r-sr-xr-x  
r-xr-sr-x
```

При помощи команды `ls -l` найдите в каталогах `/bin`, `/etc` и `/usr/bin` файлы с необычными правами доступа (если это командные файлы оболочки и у вас есть право на чтение этих файлов, посмотрите, что они делают и надежно ли они защищены). Более опытные читатели могут ускорить поиск, воспользовавшись программой `grep`. Если вам не удастся найти файлы с необычными правами доступа, объясните, почему это произошло.

---

<sup>1</sup> Использование `suid`-программ для нарушения защиты – известнейший способ взлома систем. Существует набор строгих правил составления защищенных `suid`-программ. Самое простое из этих правил (но явно недостаточное) – не давать никому права читать содержимое таких программ. Благодаря этому иногда можно скрыть слабое место программы от посторонних глаз. – Прим. науч. ред.

### 3.1.4. Маска создания файла и системный вызов `umask`

Как уже было отмечено в главе 2, первоначально права доступа к файлу задаются в момент его создания при помощи вызова `creat` или `open` в расширенной форме, например:

```
filedes = open("datafile", O_CREAT, 0644);
```

С каждым процессом связано значение, называемое *маской создания файла* (file creation mask). Эта маска используется для автоматического выключения битов прав доступа при создании файлов, независимо от режима, заданного соответствующим вызовом `creat` или `open`. Это полезно для защиты всех создаваемых за время существования процесса файлов, так как предотвращается случайное включение лишних прав доступа.

Основная идея просматривается четко: если в маске создания файлов задан какой-либо бит доступа, то при создании файлов он всегда остается выключенным. Биты в маске могут быть установлены при помощи тех же восьмеричных постоянных, которые были описаны ранее для кода доступа к файлам, хотя при этом могут использоваться только основные права доступа на чтение, запись и выполнение. Экзотические права доступа, такие как `S_ISUID`, не имеют смысла для маски создания файла.

Таким образом, с точки зрения программиста, оператор

```
filedes = open(pathname, O_CREAT, mode);
```

эквивалентен оператору

```
filedes = open(pathname, O_CREAT, (~mask) & mode);
```

где переменная `mask` содержит текущее значение маски создания файла, `~` (тильда) — это оператор побитового отрицания, а `&` — (амперсанд) оператор побитового И.

Например, если значение маски равно `04+02+01=07`, то права доступа, обычно задаваемые этими значениями, при создании файла выключаются. Поэтому файл, создаваемый при помощи оператора

```
filedes = open(pathname, O_CREAT, 0644);
```

в действительности будет иметь код доступа `0640`. Это означает, что владелец файла и пользователи из связанной с файлом группы смогут использовать файл, а пользователи всех остальных категорий не будут иметь доступа к нему.

Маску создания файла можно изменить при помощи системного вызова `umask`.

#### Описание

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

```
mode_t umask(mode_t newmask);
```

Например:

```
mode_t oldmask;
```

```
.
.
.
```

```
oldmask = umask(022);
```



Значение 022 запрещает присваивание файлу прав доступа на запись всем, кроме владельца файла. Тип `mode_t` специально предназначен для хранения кодов доступа к файлам, то есть информации о правах доступа. Он определен в заголовочном файле `<sys/types.h>`, который, в свою очередь, включен в файл `<sys/stat.h>`. После вызова в переменную `oldmask` будет помещено предыдущее значение маски.

Поэтому, если вы хотите быть абсолютно уверены, что файлы создаются именно с кодами доступа, заданными в вызовах `creat` или `open`, вам следует вначале вызвать `umask` с нулевым аргументом. Так как все биты в маске создания файла будут равны нулю, ни один из битов в коде доступа, передаваемом вызовам `open` или `creat`, не будет сброшен. В следующем примере этот подход используется для создания файла с заданным кодом доступа, а затем восстанавливается старая маска создания файла. Программа возвращает дескриптор файла, полученный в результате вызова `open`.

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

int specialcreat(const char *pathname, mode_t mode)
{
    mode_t oldu;
    int filedес;
    /* Установить маску создания файла равной нулю */
    if( (oldu = umask(0)) == -1)
    {
        perror("Ошибка сохранения старой маски");
        return (-1);
    }

    /* Создать файл */
    if((fileдес=open(pathname,O_WRONLY|O_CREAT|O_EXCL,mode)) == -1)
        perror("Ошибка открытия файла");

    /* Восстановить прежний режим доступа к файлу */
    if(umask(oldu) == -1)
        perror ("Ошибка восстановления старой маски");

    /* Вернуть дескриптор файла */
    return filedес;
}
```

### 3.1.5. Вызов `open` и права доступа к файлу

Если вызов `open` используется для открытия существующего файла на чтение или запись, то система проверяет, разрешен ли запрошенный процессом режим доступа (только для чтения, только для записи или для чтения-записи), проверяя права доступа к файлу. Если режим не разрешен, вызов `open` вернет значение `-1`, указывающее на ошибку, а переменная `errno` будет содержать код ошибки `EACCESS`, означающий: *нет доступа* (permission denied).

Если для создания файла используется расширенная форма вызова `open`, то использование флагов `O_CREAT`, `O_TRUNC` и `O_EXCL` позволяет по-разному работать

с существующими файлами. Примеры использования вызова `open` с заданием прав доступа к файлу:

```
filedes = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

и:

```
filedes = open(pathname, O_WRONLY | O_CREAT | O_EXCL, 0600);
```

В первом примере, если файл существует, он будет усечен до нулевой длины в случае, когда права доступа к файлу разрешают вызывающему процессу доступ на запись. Во втором – вызов `open` завершится ошибкой, если файл существует, независимо от заданных прав доступа к нему, а переменная `errno` будет содержать код ошибки `EEXIST`.

### Упражнение 3.5.

*A. Предположим, что действующий идентификатор пользователя `eu` равен 100, а его действующий идентификатор группы `eg` равен 200. Владелец файла `testfile` является пользователем с идентификатором 101, а идентификатор группы файла `gid` равен 200. Для каждого возможного режима доступа (только для чтения, только для записи, для записи-чтения) определите, будет ли успешным вызов `open`, если файл `testfile` имеет следующие права доступа:*

```
rwrx-rwx  r-xrwxr-x  rwx--x--  rwsrw-r--
--s--s--x  ---rwx---  ---r-x--x
```

*B. Что произойдет, если `real user-id` (действующий идентификатор пользователя) процесса равен 101, а `real group-id` (действующий идентификатор группы) равен 201?*

### 3.1.6. Определение доступности файла при помощи вызова `access`

Системный вызов `access` определяет, может ли процесс получить доступ к файлу в соответствии с *истинным* (`real`), а не *действующим* (`effective`) идентификатором пользователя (и группы) процесса. Такой вызов позволяет процессу, получившему права при помощи бита `S_ISUID`, определить настоящие права пользователя, запустившего это процесс, что облегчает написание безопасных программ.

#### Описание

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

```
int access(const char *pathname, int amode);
```

Как мы уже видели, существует несколько режимов доступа к файлу, поэтому параметр `amode` содержит значение, указывающее на интересующий нас метод доступа. Параметр `amode` может принимать следующие значения, определенные в файле `<unistd.h>`:

- `R_OK` – имеет ли вызывающий процесс доступ на чтение;
- `W_OK` – имеет ли вызывающий процесс доступ на запись;
- `X_OK` – может ли вызывающий процесс выполнить файл.

Аргумент `amode` не конкретизирует, к какой категории пользователей относится вопрос, так как вызов `access` сообщает права доступа к файлу конкретного пользователя, имеющего `guid` и `rgid` текущего процесса. Переменная `amode` также может принимать значение `F_OK`, в этом случае проверяется лишь существование файла. Как обычно, параметр `pathname` задает имя файла.

Значение, возвращаемое вызовом `access`, либо равно нулю (доступ разрешен) или `-1` (доступ не разрешен). В последнем случае переменная `errno` будет содержать значение кода ошибки. Значение `EACCESS`, например, означает, что запрошенный режим доступа к файлу не разрешен, а значение `ENOENT` показывает, что указанного файла просто не существует.

Следующий пример программы использует вызов `access` для проверки, разрешено ли пользователю чтение файла при любом значении бита `S_ISUID` исполняемого файла этой программы:

```
/* Пример использования вызова access */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main ()
{
    char *filename = "afile";

    if(access(filename, R_OK) == -1)
    {
        fprintf(stderr, "Пользователь не имеет доступа \
            на чтение к файлу %s\n", filename);
        exit(1);
    }

    printf("%s доступен для чтения, продолжаем\n", filename);

    /* Остальная программа */
}
```

---

**Упражнение 3.6.** Напишите программу `whatable`, которая будет сообщать, можете ли вы выполнять чтение, запись или выполнение заданного файла. Если доступ невозможен, программа `whatable` должна сообщать почему (используйте коды ошибок, возвращаемых в переменной `perror`).

---

### 3.1.7. Изменение прав доступа при помощи вызова `chmod`

#### Описание

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

int chmod(const char *pathname, mode_t newmode);
```

Для изменения прав доступа к существующему файлу применяется системный вызов `chmod`. Вызов разрешен владельцу файла или суперпользователю.

Параметр `pathname` указывает имя файла. Параметр `newmode` содержит новый код доступа файла, образованный описанным в первой части главы способом.

Пример использования вызова `chmod`:

```
if(chmod(pathname, 0644) == -1)
    perror("Ошибка вызова chmod");
```

---

**Упражнение 3.7.** *Напишите программу `setperm`, которая имеет два аргумента командной строки. Первый – имя файла, второй – набор прав доступа в восьмеричной форме или в форме, выводимой командой `ls`. Если файл существует, то программа `setperm` должна попытаться поменять права доступа к файлу на заданные. Используйте процедуру `lsoc`, которую вы разработали в упражнении 3.3.*

---

### 3.1.8. Изменение владельца при помощи вызова `chown`

Вызов `chown` используется для изменения владельца и группы файла.

#### Описание

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

int chown(const char *pathname, uid_t owner_id, gid_t group_id);
```

Например:

```
int retval;
.
.
.
retval = chown("/usr/dina", 56, 3);
```

Вызов имеет три аргумента: `pathname`, указывающий имя файла, `owner_id`, задающий нового владельца, и `group_id`, задающий новую группу. Возвращаемое значение `retval` равно нулю в случае успеха и `-1` – в случае ошибки. Оба типа `uid_t` и `gid_t` определены в заголовочном файле `<sys/types.h>`.

В системе, удовлетворяющей спецификации XSI, вызывающий процесс должен быть процессом суперпользователя или владельца файла (точнее, действующий идентификатор вызывающего процесса должен либо совпадать с идентификатором владельца файла, либо быть равным 0). При несанкционированной попытке изменить владельца файла выставляется код ошибки `EPERM`.

Поскольку вызов `chown` разрешен текущему владельцу файла, обычный пользователь может передать свой файл другому пользователю. При этом пользователь не сможет впоследствии отменить это действие, так как идентификатор пользователя уже не будет совпадать с идентификатором пользователя файла. Следует также обратить внимание на то, что при смене владельца файла в целях предотвращения неправомерного использования вызова `chown` для получения системных полномочий, сбрасываются права доступа `set-user-id` и `set-group-id`. (Что могло бы произойти, если бы это было не так?)

## 3.2. Файлы с несколькими именами

Любой файл UNIX может иметь несколько имен. Другими словами, один и тот же набор данных может быть связан с несколькими именами UNIX без необходимости создания копий файла. Поначалу это может показаться странным, но для экономии свободного пространства на диске и увеличения числа пользователей, использующих один и тот же файл, — весьма полезно.

Каждое такое имя называется *жесткой ссылкой* (hard link). Число связанных с файлом ссылок называется *счетчиком ссылок* (link count).

Новая жесткая ссылка создается при помощи системного вызова `link`, а существующая жесткая ссылка может быть удалена при помощи системного вызова `unlink`.

Следует отметить полную равноправность жестких ссылок на файл и настоящего имени файла. Нет способа отличить настоящее имя файла от созданной позднее жесткой ссылки. Это становится очевидным, если рассмотреть организацию файловой системы, — см. главу 4.

### 3.2.1. Системный вызов `link`

#### Описание

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

```
int link(const char *original_path, const char *new_path);
```

Первый параметр, `original_path`, является указателем на массив символов, содержащий полное имя файла в системе UNIX. Он должен задавать существующую ссылку на файл, то есть фактическое имя файла. Второй параметр, `new_path`, задает новое имя файла или ссылку на файл, но файл, заданный параметром `new_path`, еще не должен существовать.

Системный вызов `link` возвращает значение 0 в случае успешного завершения и `-1` — в случае ошибки. В последнем случае новая ссылка на файл не будет создана.

Например, оператор

```
link("/usr/keith/chap.2", "/usr/ben/2.chap");
```

создаст новую ссылку `/usr/ben/2.chap` на существующий файл `/usr/keith/chap.2`. Теперь к файлу можно будет обратиться, используя любое из имен. Пример показывает, что ссылка не обязательно должна находиться в одном каталоге с файлом, на который она указывает.

### 3.2.2. Системный вызов `unlink`

В разделе 2.1.13 мы представили системный вызов `unlink` в качестве простого способа удаления файла из системы. Например:

```
unlink("/tmp/scratch");
```

удалит файл `/tmp/scratch`.

Фактически системный вызов `unlink` просто удаляет указанную ссылку и уменьшает *счетчик ссылок* (link count) файла на единицу. Данные в файле будут

безвозвратно потеряны только после того, как счетчик ссылок на него станет равным нулю, и он не будет открыт ни в одной программе. В этом случае занятые файлом блоки на диске добавляются к поддерживаемому системой списку свободных блоков. Хотя данные могут еще существовать физически в течение какого-то времени, восстановить их будет невозможно. Так как многие файлы имеют лишь одну ссылку – принятое имя файла, удаление файла является обычным результатом вызова `unlink`. И наоборот, если счетчик ссылок не уменьшится до нуля, то данные в файле останутся нетронутыми, и к ним можно будет обратиться при помощи других ссылок на файл.

Следующая короткая программа переименовывает файл, вначале создавая на него ссылку с новым именем и удаляя в случае успеха старую ссылку на файл. Это упрощенная версия стандартной команды UNIX `mv`:

```
/* Программа move - переименование файла */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
char *usage = "Применение: move файл1 файл2\n";

/*  Функция main использует аргументы командной строки,
 *  передаваемые обычным способом.
 */

main(int argc, char **argv)
{
    if(argc != 3)
    {
        fprintf(stderr, usage);
        exit(1);
    }

    if( link(argv[1], argv[2]) == -1)
    {
        perror("Ошибка вызова link");
        exit(1);
    }

    if(unlink(argv[1]) == -1)
    {
        perror("Ошибка вызова unlink");
        unlink(argv[2]);
        exit(1);
    }

    printf("Успешное завершение\n");
    exit(0);
}
```

До сих пор не было упомянуто взаимодействие вызова `unlink` и прав доступа к файлу, связанных с аргументом, задающим имя файла. Это объясняется тем, что

права просто не влияют на вызов `unlink`. Вместо этого успешное или неуспешное завершение вызова `unlink` определяется правами доступа к содержащему файл каталогу. Эту тема будет рассмотрена в главе 4.

### 3.2.3. Системный вызов `rename`

Фактически задачу предыдущего примера можно выполнить гораздо легче, используя системный вызов `rename`, который был добавлен в систему UNIX сравнительно недавно. Системный вызов `rename` может использоваться для переименования как обычных файлов, так и каталогов.

#### Описание

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

```
int rename(const char *oldpathname, const char *newpathname);
```

Файл, заданный аргументом `oldpathname`, получает новое имя, заданное вторым параметром `newpathname`. Если файл с именем `newpathname` уже существует, то перед переименованием файла `oldpathname` он удаляется.

---

**Упражнение 3.8.** *Напишите свою версию команды `rm`, используя вызов `unlink`. Ваша программа должна проверять, имеет ли пользователь право записи в файл при помощи вызова `access` и в случае его отсутствия запрашивать подтверждение перед попыткой удаления ссылки на файл. (Почему?) Будьте осторожны при тестировании программы!*

---

### 3.2.4. Символьные ссылки

Существует два важных ограничения на использование вызова `link`. Обычный пользователь не может создать ссылку на каталог (в некоторых версиях UNIX и суперпользователь не имеет права этого делать), и невозможно создать ссылку между различными *файловыми системами* (file systems). Файловые системы являются основными составляющими всей файловой структуры UNIX и будут изучаться более подробно в главе 4.

Для преодоления этих ограничений спецификация XSI поддерживает понятие *символьных ссылок* (symbolic links). Символьная ссылка в действительности представляет собой файл, содержащий вместо данных путь к файлу, на который указывает ссылка. Можно сказать, что символьная ссылка является указателем на другой файл.

Для создания символьной ссылки используется системный вызов `symlink`:

#### Описание

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

```
int symlink(const char *realname, const char *symname);
```

После завершения вызова `symlink` создается файл `symname`, указывающий на файл `realname`. Если возникает ошибка, например, если файл с именем `symname`

уже существует, то вызов `symlink` возвращает значение `-1`. В случае успеха вызов возвращает нулевое значение.

Если затем файл символической ссылки открывается при помощи `open`, то системный вызов `open` корректно прослеживает путь к файлу `realname`. Если необходимо считать данные из самого файла `symlinkname`, то нужно использовать системный вызов `readlink`.

### Описание

```
#include <unistd.h>

int readlink(const char *sympath, char *buffer, size_t bufsize);
```

Системный вызов `readlink` вначале открывает файл `sympath`, затем читает его содержимое в переменную `buffer`, и, наконец, закрывает файл `sympath`. К сожалению, спецификация XSI не гарантирует, что строка в переменной `buffer` будет заканчиваться нулевым символом. Возвращаемое вызовом `readlink` значение равно числу символов в буфере или `-1` – в случае ошибки.

Следует сделать предупреждение, касающееся использования и прослеживания символических ссылок. Если файл, на который указывает символическая ссылка, удаляется, то при попытке доступа к файлу при помощи символической ссылки выдается ошибка, которая может ввести вас в заблуждение. Программа все еще сможет «видеть» символическую ссылку, но, к сожалению, вызов `open` не сможет проследовать по указанному в ссылке пути и вернет ошибку, установив значение переменной `errno` равным `EEXIST`.

## 3.3. Получение информации о файле: вызовы `stat` и `fstat`

До сих пор были лишь рассмотрены вопросы, как можно установить или изменить основные связанные с файлами свойства. Два системных вызова `stat` и `fstat` позволяют процессу определить значения этих свойств в существующем файле.

### Описание

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Системный вызов `stat` имеет два аргумента: первый из них – `pathname`, как обычно, указывает на полное имя файла. Второй аргумент `buf` является указателем на структуру `stat`. Эта структура после успешного вызова будет содержать связанную с файлом информацию.

Системный вызов `fstat` функционально идентичен системному вызову `stat`. Отличие состоит в интерфейсе: вместо полного имени файла вызов `fstat` ожидает дескриптор файла, поэтому он может использоваться только для открытых файлов.



```
.  
.   
.   
struct stat s;  
int filedес, retval;  
  
fileдес = open("/tmp/dina", O_RDWR);  
  
/* Структура s может быть заполнена при помощи вызова .. */  
retval = stat("/tmp/dina", &s);  
  
/* ... или */  
retval = fstat(fileдес, &s);
```

Определение структуры stat находится в системном заголовочном файле <sys/stat.h> и включает следующие элементы:

```
dev_t      st_dev;  
ino_t      st_ino;  
mode_t     st_mode;  
nlink_t    st_nlink;  
uid_t      st_uid;  
gid_t      st_gid;  
dev_t      st_rdev;  
off_t      st_size;  
time_t     st_atime;  
time_t     st_mtime;  
time_t     st_ctime;  
long       st_blksize;  
long       st_blocks;
```

Используемые структурой stat типы определены в системном заголовочном файле <sys/types.h>.

Элементы структуры stat имеют следующие значения:

❑ st\_dev, st\_ino

Первый из элементов структуры описывает логическое устройство, на котором находится файл, а второй задает *номер индексного дескриптора* (inode number), который вместе с st\_dev однозначно определяет файл. Фактически и st\_dev, и st\_ino относятся к низкоуровневому управлению структурой файлов UNIX. Эти понятия будут рассмотрены в следующей главе.

❑ st\_mode

Этот элемент задает *режим* доступа к файлу и позволяет программисту вычислить связанные с файлом права доступа. Здесь следует сделать предостережение. Значение, содержащееся в переменной st\_mode, также дает информацию о типе файла, и только младшие 12 бит относятся к правам доступа. Это станет очевидно в главе 4.

❑ st\_nlink

Число ссылок, указывающих на этот файл (другими словами, число различных имен файла, так как жесткие ссылки неотличимы от «настоящего» имени). Это значение обновляется при каждом системном вызове link и unlink.

□ `st_uid, st_gid`

Идентификаторы пользователя `uid` и группы `gid` файла. Первоначально устанавливаются вызовом `creat` и изменяются системным вызовом `chown`.

□ `st_rdev`

Этот элемент имеет смысл только в случае использования файла для описания устройства. На него пока можно не обращать внимания.

□ `st_size`

Текущий *логический* размер файла в байтах. Нужно понимать, что способ хранения файла определяется реальными параметрами устройства, и поэтому физический размер занимаемого пространства может быть больше, чем логический размер файла. Элемент `st_size` изменяется при каждом вызове `write` в конце файла.

□ `st_atime`

Содержит время последнего чтения из файла (хотя первоначальные вызовы `creat` и `open` устанавливают это значение).

□ `st_mtime`

Указывает время последней модификации файла – изменяется при каждом вызове `write` для файла.

□ `st_ctime`

Содержит время последнего изменения информации, возвращаемой в структуре `stat`. Это время изменяется системными вызовами `link` (меняется элемент `st_nlink`), `chmod` (меняется `st_mode`) и `write` (меняется `st_mtime` и, возможно, `st_size`).

□ `st_blksize`

Содержит размер блока ввода/вывода, зависящий от настроек системы. Для некоторых систем этот параметр может различаться для разных файлов.

□ `st_blocks`

Содержит число физических блоков, занимаемых определенным файлом.

Следующий пример – процедура `filedata` выводит данные, связанные с файлом, определяемым переменной `pathname`. Пример сообщает размер файла, идентификатор пользователя, группу файла, а также права доступа к файлу.

Чтобы преобразовать права доступа к файлу в удобочитаемую форму, похожую на результат, выводимый командой `ls`, был использован массив `octarray` чисел типа `short integer`, содержащий значения для основных прав доступа, и массив символов `perms`, содержащий символьные эквиваленты прав доступа.

```
/* Процедура filedata - выводит данные о файле */
```

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

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

```
/*
```

```
 * Массив octarray используется для определения
```

```
 * установки битов прав доступа.
```

```
*/
```

```
static short octarray[9] = { 0400,0200,0100,  
                             0040,0020,0010,  
                             0004,0002,0001};
```

```
/* Мнемонические коды для прав доступа к файлу,
 * длиной 10 символов, включая нулевой символ в конце строки.
 */
static char perms[10] = "rwxrwxrwx";

int filedata(const char *pathname)
{
    struct stat statbuf;
    char descrip[10];
    int j;

    if(stat(pathname, &statbuf) == -1)
    {
        fprintf(stderr, "Ошибка вызова stat %s\n", pathname);
        return (-1);
    }
    /* Преобразовать права доступа в удобочитаемую форму */

    for(j=0; j<9; j++)
    {
        /*
         * Проверить, установлены ли права доступа
         * при помощи побитового И.
         */
        if(statbuf.st_mode & octarray[j])
            descrip[j] = perms[j];
        else
            descrip[j] = '-';
    }
    descrip[9] = '\0'; /* Задать строку */

    /* Вывести информацию о файле */
    printf("\nФайл %s : \n", pathname);
    printf("Размер %ld байт\n", statbuf.st_size);
    printf("User-id %d, Group-id %d\n\n", statbuf.st_uid,
           statbuf.st_gid);
    printf("Права доступа: %s\n", descrip);
    return (0);
}
```

Более полезным инструментом является следующая программа `lookout`. Она раз в минуту проверяет, изменился ли какой-либо из файлов из заданного списка, опрашивая время модификации каждого из файлов (`st_mtime`). Это утилита, которая предназначена для запуска в качестве фонового процесса.<sup>1</sup>

```
/* Программа lookout - сообщает об изменении файла */

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

<sup>1</sup> Фоновый процесс при попытке вывода на консоль остановится, получив сигнал SIGTTOU (см. главу 6). Следует придумать иной способ оповещения об изменениях файлов. – *Прим. науч. ред.*

```
#define MFILE 10

void cmp(const char *, time_t);
struct stat sb;

main(int argc, char **argv)
{
    int j;
    time_t last_time[MFILE+1];

    if (argc < 2)
    {
        fprintf(stderr, "Синтаксис: lookout имя_файла ...\n");
        exit(1);
    }

    if(-argc > MFILE)
    {
        fprintf(stderr, "lookout: слишком много имен файлов\n") ;
        exit(1);
    }

    /* Инициализация */
    for(j=1; j<=argc; j++)
    {
        if(stat(argv[j], &sb) == -1)
        {
            fprintf(stderr, "lookout: ошибка вызова stat %s\n",
                argv[j]);
            exit(1);
        }
        last_time[j] = sb.st_mtime;
    }

    /* Повторять до тех пор, пока файл не изменится */
    for(;;)
    {
        for(j=1; j<=argc; j++)
            cmp(argv[j], last_time[j]);

        /*
         * Остановиться на 60 секунд.
         * Функция "sleep" - стандартная
         * библиотечная процедура UNIX.
         */
        sleep (60);
    }
}

void cmp(const char *name, time_t last)
{
    /* Проверять время изменения файла,
     * если можно считать данные о файле */
}
```

```
if(stat(name, &sb) == -1 || sb.st_mtime != last)
{
    fprintf(stderr, "lookout: файл %s изменился\n", name);
    exit(0);
}
```

---

**Упражнение 3.9.** Напишите программу, которая проверяет и записывает изменения размера файла в течение часа. В конце работы она должна строить простую гистограмму, демонстрирующую изменения размера во времени.

---

**Упражнение 3.10.** Напишите программу `slowwatch`, которая периодически проверяет время изменения заданного файла (она не должна завершаться ошибкой, если файл изначально не существует). При изменении файла программа `slowwatch` должна копировать его на свой стандартный вывод. Как можно убедиться (или предположить), что обновление файла закончено до того, как он будет скопирован?

---

### 3.3.1. Подробнее о вызове `chmod`

Системные вызовы `stat` и `fstat` расширяют использование вызова `chmod`, поскольку позволяют предварительно узнать значение кода доступа к файлу, что дает возможность изменять отдельные биты, а не менять весь код доступа целиком.

Следующая программа `addx` демонстрирует сказанное. Она вначале вызывает `stat` для получения режима доступа к файлу из списка аргументов вызова программы. В случае успешного завершения вызова программа изменяет существующие права доступа так, чтобы файл был доступен для выполнения его владельцем. Эта программа может быть полезна для придания командным файлам, составленным пользователем, статуса исполняемых файлов.

```
/* Программа addx - разрешает доступ на выполнение файла */

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

#define XPERM 0100 /* Право на выполнение для владельца */

main(int argc, char **argv)
{
    int k;
    struct stat statbuf;
    /* Выполнить для всех файлов в списке аргументов */
    for(k=1; k<argc; k++)
    {
        /* Получить текущий код доступа к файлу */
        if(stat(argv[k], &statbuf) == -1)
```

```
{
    fprintf(stderr, "addx: ошибка вызова stat %s\n",
               argv[k]);
    continue;
}
/* Попытайтесь разрешить доступ на выполнение
   при помощи оператора побитового ИЛИ */

statbuf.st_mode |= XPERM;
if(chmod(argv[k], statbuf.st_mode) == -1)
    fprintf(stderr, "addx: ошибка изменения прав доступа
                   для файла %s\n", argv[k]);
} /* Конец цикла */

exit(0);
}
```

Наиболее интересный момент заключается здесь в способе изменения кода доступа файла при помощи побитового оператора ИЛИ. Это гарантирует, что устанавливается бит, заданный определением XPERM. Фактически мы могли бы расписать этот оператор в виде:

```
statbuf.st_mode = (statbuf.st_mode) | XPERM;
```

Для ясности использована более короткая форма. Можно было бы также использовать вместо XPERM предусмотренную в системе постоянную S\_IXUSR.

---

**Упражнение 3.11.** Приведенную задачу можно решить проще. Если вы знаете, как это сделать, напишите эквивалент этой программы при помощи командного интерпретатора.

---

---

**Упражнение 3.12.** Напишите свою версию команды chmod, используя ее описание в справочном руководстве вашей системы UNIX.

---



# Глава 4. Каталоги, файловые системы и специальные файлы

## 4.1. Введение

В двух предыдущих главах внимание было сконцентрировано на основном компоненте файловой структуры UNIX – обычных файлах. В этой главе будут рассмотрены другие компоненты файловой структуры, а именно:

- *каталоги*. Каталоги выступают в качестве хранилища имен файлов и, следовательно, позволяют пользователям группировать произвольные наборы файлов. Понятие каталогов должно быть знакомо большинству пользователей UNIX и многим «эмигрантам» из других операционных систем. Далее будет показано, что каталоги UNIX могут быть вложенными, это придает структуре файлов древовидную иерархическую форму;
- *файловые системы*. Файловые системы представляют собой набор каталогов и файлов и являются подразделами иерархического дерева каталогов и файлов, образующих общую файловую структуру UNIX. Файловые системы обычно соответствуют *физическим разделам* (partitions) дискового устройства или всему дисковому устройству. При решении большинства задач файловые системы остаются невидимыми для пользователя;
- *специальные файлы*. Концепция файла получила в системе UNIX дальнейшее развитие и включает в себя присоединенные к системе периферийные устройства. Эти периферийные устройства, такие как принтеры, дисковые накопители и даже системная память, представляются в файловой структуре именами файлов. Файл, представляющий устройство, называется *специальным файлом* (special file). К устройствам можно получить доступ при помощи обычных системных вызовов доступа к файлам, описанных в главах 2 и 3 (например, вызовов `open`, `read` и `write`). Каждый такой вызов задействует код драйвера устройства в ядре системы, отвечающий за управление заданным устройством. Тем не менее программе не нужно ничего знать об этом, так как система позволяет обращаться со специальными файлами почти как с обычными.

## 4.2. Каталоги с точки зрения пользователя

Даже случайный пользователь системы UNIX будет иметь некоторое представление о том, как выглядит структура каталогов системы. Тем не менее для полноты

изложения кратко опишем обычное расположение файлов с точки зрения пользователя.

В сущности каталоги являются просто списками имен файлов, которые обеспечивают способ разбиения файлов на логически связанные группы. Например, каждый пользователь имеет *домашний каталог* (home directory), в который он попадает при входе в систему и где может создавать файлы и работать с ними. Смысл этого, очевидно, состоит в том, чтобы разделить файлы отдельных пользователей. Аналогично программы, доступные всем пользователям, такие как `cat` или `ls`, помещаются в общеизвестные каталоги, обычно `/bin` или `/usr/bin`. Используя общепринятую метафору, каталоги можно сравнить с ящиками в шкафу, в которых хранятся папки файлов с документами.

Вместе с тем у каталогов есть некоторые преимущества по сравнению с ящиками в шкафу, так как кроме файлов они также могут включать другие каталоги, которые называются *подкаталогами* (subdirectories) и позволяют организовать следующие уровни классификации. Подкаталоги, в свою очередь, могут содержать другие подкаталоги и так далее. Допустим любой уровень вложенности, хотя может быть ограничение на длину абсолютного пути файла (см. пункт 4.6.4).

Фактически файловую структуру UNIX можно представить в виде иерархической структуры, напоминающей перевернутое дерево. Упрощенное дерево каталогов показано на рис. 4.1 (это тот же пример, что и рисунок в главе 1). Конечно же, любая реальная система будет иметь более сложную структуру.

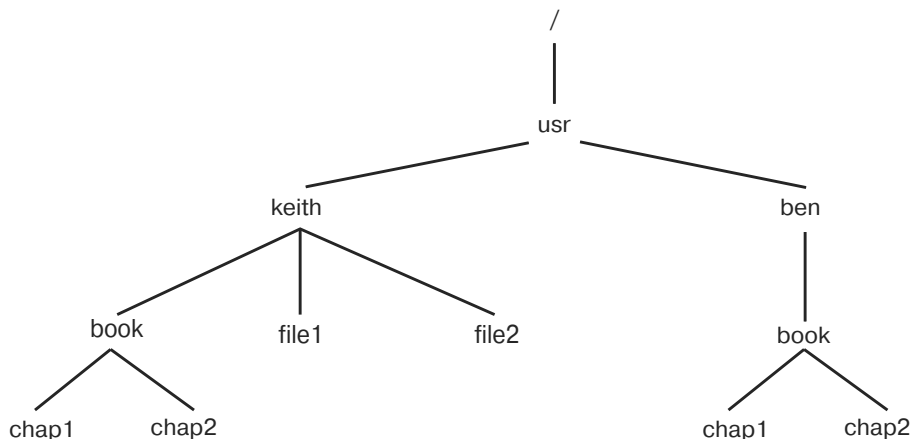


Рис. 4.1. Пример дерева каталогов

На вершине этого дерева, так же как и на вершине любого дерева каталогов UNIX, находится единственный каталог, который называется *корневым каталогом* (root directory) и имеет очень короткое имя `/`. Все узлы дерева, кроме конечных, например узлы `keith` или `ben`, всегда являются каталогами. Конечные узлы, например узлы `file1` или `file2`, являются файлами или пустыми каталогами. В настоящее время в большинстве систем UNIX имена каталогов могут содержать до 255 символов, но, так же как и в случае с файлами, для обеспечения



совместимости со старыми версиями системы их длина не должна превышать 14 символов.

В нашем примере узлы `keith` и `ben` являются подкаталогами родительского каталога `usr`. В каталоге `keith` находятся три элемента: два обычных файла `file1` и `file2` и подкаталог `book`. Каталог `keith` является родительским для каталога `book`. В свою очередь каталог `book` содержит два файла `chap1` и `chap2`. Как было показано в главе 1, положение файла в иерархии может быть задано заданием пути к нему. Например, полное имя файла `chap2` в каталоге `keith`, включающее путь к нему, будет `/usr/keith/book/chap2`. Аналогично можно указать и полное имя каталога. Полное имя каталога `ben` будет `/usr/ben`.

Следует обратить внимание, что каталог `/usr/ben/book` также содержит два файла с именами `chap1` и `chap2`. Они не обязательно связаны со своими тезками в каталоге `/usr/keith/book`, так как только полное имя файла однозначно идентифицирует его. Тот факт, что в разных каталогах могут находиться файлы с одинаковыми именами, означает, что пользователям нет необходимости изобретать странные и уникальные имена для файлов.

### Текущий рабочий каталог

После входа в систему пользователь находится в определенном месте файловой структуры, называемом *текущим рабочим каталогом* (current working directory) или иногда просто *текущим каталогом* (current directory). Это будет, например, каталог, содержимое которого выведет команда `ls` при ее запуске без параметров. Первоначально в качестве текущего рабочего каталога для пользователя выступает его домашний каталог, заданный в файле паролей системы. Можно перейти в другой каталог при помощи команды `cd`, например, команда

```
$ cd /usr/keith
```

сделает `/usr/keith` текущим каталогом. Имя текущего каталога можно при необходимости узнать при помощи команды *вывести рабочий каталог* (print working directory, сокращенно `pwd`):

```
$ pwd  
/usr/keith
```

В отношении текущего каталога основной особенностью является то, что с него система начинает поиск при задании относительного пути – то есть такого, который не начинается с корня `/`. Например, если текущий рабочий каталог `/usr/keith`, то команда

```
$ cat book/chap1
```

эквивалентна команде

```
$ cat /usr/keith/book/chap1
```

а команда

```
$ cat file1
```

эквивалентна команде

```
$ cat usr/keith/file1
```





### 4.3.2. Точка и двойная точка

В каждом каталоге всегда присутствуют два странных имени файлов: точка (.) и двойная точка (..). Точка является стандартным для системы UNIX способом обозначения текущего рабочего каталога, как в команде

```
$ cat ./fred
```

которая выведет на экран файл `fred` в текущем каталоге, или

```
$ ls .
```

которая выведет список файлов в текущем каталоге. Двойная точка является стандартным способом ссылки на родительский каталог текущего рабочего каталога, то есть каталог, содержащий текущий каталог. Поэтому команда

```
$ cd ..
```

позволяет пользователю переместиться на один уровень вверх по дереву каталогов.

Фактически имена «точка» (.) и «двойная точка» (..) просто являются ссылками на текущий рабочий каталог и родительский каталог соответственно, и любой каталог UNIX содержит в первых двух позициях эти два имени. Другими словами, во время создания каталога в него автоматически добавляются эти два имени.

Можно более ясно это представить себе, рассмотрев участок дерева каталогов, приведенный на рис. 4.4.

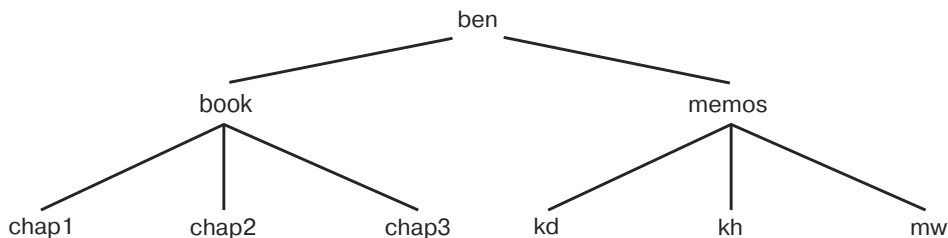


Рис. 4.4. Часть дерева каталогов

Если рассмотреть каждый из каталогов `ben`, `book` и `memos`, то откроется картина, похожая на рис. 4.5. Нужно обратить внимание на то, что в каталоге `book` номер записи с именем `.` равен 260, а номер записи с именем `..` равен 123, и эти номера соответствуют элементам `book` и `.` в родительском каталоге `ben`. Аналогично имена `.` и `..` в каталоге `memos` (с номерами узлов 401 и 123) соответствуют каталогу `memos` и имени `.` в каталоге `ben`.

### 4.3.3. Права доступа к каталогам

Так же как и с обычными файлами, с каталогами связаны права доступа, определяющие возможность доступа к ним различных пользователей.

Права доступа к каталогам организованы точно так же, как и права доступа к обычным файлам, разбиты на три группы битов `rwX`, определяющих права владельца файла, пользователей из его группы и всех остальных пользователей системы.



Здесь строка, описывающая подкаталог `expenses`, помечена буквой `d` в начале строки. Видно, что владелец этого каталога (пользователь `ben`) имеет права на чтение, запись и выполнение (поиск), пользователи группы файла (называющейся `other`) имеют права на чтение и выполнение (переход в каталог), а для всех остальных пользователей доступ полностью запрещен.

Если требуется получить информацию о текущем каталоге, можно задать в команде `ls` кроме параметра `-l` еще и параметр `-d`, например:

```
$ ls -ld
drwxr-x--- 3 ben other 128 Oct 12 22:02 .
```

Помните, что имя `.` (точка) в конце листинга обозначает текущий каталог.

## 4.4. Использование каталогов при программировании

Как уже упоминалось, для работы с каталогами существует особое семейство системных вызовов. Главным образом эти вызовы работают со структурой `dirent`, которая определена в заголовочном файле `<dirent.h>` и содержит следующие элементы:

```
ino_t  d_ino;           /* Номер индексного дескриптора */
char   d_iname[];       /* Имя файла, заканчивается null */
```

Тип данных `ino_t` определяется в заголовочном файле `<sys/types.h>`, который включен в заголовочный файл `<dirent.h>`. Спецификация XSI не определяет размер `d_name`, но гарантирует, что число байтов, предшествующих нулевому символу, будет меньше, чем число, хранящееся в переменной `_PC_NAME_MAX`, определенной в заголовочном файле `<unistd.h>`. Обратите внимание, что нулевое значение переменной `d_ino` обозначает пустую запись в каталоге.

### 4.4.1. Создание и удаление каталогов

Как уже упоминалось ранее, каталоги нельзя создать при помощи системных вызовов `creat` или `open`. Для выполнения этой задачи существует специальный системный вызов `mkdir`.

#### Описание

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

int mkdir(const char *pathname, mode_t mode);
```

Первый параметр, `pathname`, указывает на строку символов, содержащую имя создаваемого каталога. Второй параметр, `mode`, является набором прав доступа к каталогу. Права доступа будут изменяться с учетом значения `umask` процесса, например:

```
int retval;
retval = mkdir("/tmp/dir1", 0777);
```

Как обычно, системный вызов `mkdir` возвращает нулевое значение в случае успеха, и `-1` — в случае неудачи. Обратите внимание, что `mkdir` также помещает

две ссылки ( . и .. ) в создаваемый новый каталог. Если бы этих элементов не было, работать с полученным каталогом было бы невозможно.

Если каталог больше не нужен, то его можно удалить при помощи системного вызова `rmdir`.

### **Описание**

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Параметр `pathname` определяет путь к удаляемому каталогу. Этот вызов завершается успехом, только если удаляемый каталог пуст, то есть содержит только записи «точка» ( . ) и «двойная точка» ( .. ).

## **4.4.2. Открытие и закрытие каталогов**

Для открытия каталога UNIX спецификация XSI определяет особую функцию `opendir`.

### **Описание**

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

DIR *opendir(const char *dirname);
```

Передаваемый вызову `opendir` параметр является именем открываемого каталога. При успешном открытии каталога `dirname` вызов `opendir` возвращает указатель на переменную типа `DIR`. Определение типа `DIR`, представляющего дескриптор открытого каталога, находится в заголовочном файле `<dirent.h>`. Это определение аналогично определению типа `FILE`, используемого в стандартной библиотеке ввода/вывода, описанной в главах 2 и 11. Указатель позиции ввода/вывода в полученном от функции `opendir` дескрипторе установлен на первую запись каталога. Если вызов завершился неудачно, то функция возвращает `NULL`. Всегда следует проверять возвращаемое значение, прежде чем это значение может быть использовано.

После того, как программа закончит работу с каталогом, она должна закрыть его. Это можно сделать при помощи функции `closedir`.

### **Описание**

```
#include <dirent.h>

int closedir(DIR *dirptr);
```

Функция `closedir` закрывает дескриптор открытого каталога, на который указывает аргумент `dirptr`. Обычно его значение является результатом предшествующего вызова `opendir`, что демонстрирует следующий пример:

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

```
main()
{
    DIR *dp;
    if((dp = opendir("/tmp/dir1")) == NULL)
    {
        fprintf(stderr, "Ошибка открытия каталога /tmp/dir1\n");
        exit(1);
    }
    /* Код, работающий с каталогом */

    .
    .
    closedir(dp);
}
```

#### 4.4.3. Чтение каталогов: вызовы *readdir* и *rewinddir*

После открытия каталога из него можно начать считывать записи.

##### Описание

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

struct dirent *readdir(DIR *dirptr);
```

Функции *readdir* должен передаваться допустимый указатель на дескриптор открытого каталога, обычно возвращаемый предшествующим вызовом *opendir*. При первом вызове *readdir* в структуру *dirent* будет считана первая запись в каталоге. В результате успешного вызова указатель каталога переместится на следующую запись.

Когда в результате последующих вызовов *readdir* достигнет конца каталога, то вызов вернет нулевой указатель. Если в какой-то момент потребуется начать чтение каталога с начала, то можно использовать системный вызов *rewinddir*, определенный следующим образом:

##### Описание

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

void rewinddir(DIR *dirptr);
```

Следующий после вызова *rewinddir* вызов *readdir* вернет первую запись в каталоге, на который указывает переменная *dirptr*.

В приведенном ниже примере функция *my\_double\_ls* дважды выведет на экран имена всех файлов в заданном каталоге. Она принимает в качестве параметра имя каталога и в случае ошибки возвращает значение *-1*.

```
#include <dirent.h>

int my_double_ls(const char *name)
{
    struct dirent *d;
```



```
DIR *dp;

/* Открытие каталога с проверкой ошибок */
if((dp=opendir(name)) == NULL)
    return (-1);

/* Выполнить обход каталога,
 * выводя записи из него, пока возвращается правильный
 * индексный дескриптор.
 */
while(d = readdir(dp))
{
    if(d->d_ino != 0)
        printf("%s\n", d->d_name);
}

/* Вернуться к началу каталога .. */
rewinddir(dp) ;

/* ... и снова вывести его содержимое */
while(d = readdir(dp))
{
    if(d->d_ino != 0)
        printf("%s\n", d->d_name);
}

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

Порядок выводимых функцией `my_double_ls` имен файлов будет совпадать с порядком расположения файлов в каталоге. Если вызвать функцию `my_double_ls` в каталоге, содержащем три файла `abc`, `bookmark` и `fred`, то ее вывод может выглядеть так:

```
.
..
fred
bookmark
abc
.
..
fred
bookmark
abc
```

### ***Второй пример: процедура `find_entry`***

Процедура `find_entry` будет искать в каталоге следующий файл (или подкаталог), заканчивающийся определенным суффиксом. Она имеет три параметра: имя каталога, в котором будет выполняться поиск, строка суффикса и флаг, определяющий, нужно ли продолжать дальнейший поиск после того, как искомый элемент будет найден.

Процедура `find_entry` использует процедуру проверки совпадения строк `match` с целью определения, заканчивается ли файл заданным суффиксом. Процедура `match`, в свою очередь, вызывает две процедуры из стандартной библиотеки С системы UNIX: функцию `strlen`, возвращающую длину строки в символах, и функцию `strcmp`, которая сравнивает две строки, возвращая нулевое значение в случае их совпадения.

```
#include <stdio.h> /* Для NULL */
#include <dirent.h>
#include <string.h> /* Для функций работы со строками */

int match(const char *, const char *);

char *find_entry(char *dirname, char *suffix, int cont)
{
    static DIR *dp=NULL;
    struct dirent *d;

    if(dp == NULL || cont == 0){
        if(dp != NULL)
            closedir(dp);
        if((dp = opendir(dirname)) == NULL)
            return (NULL);
    }
    while(d = readdir(dp))
    {
        if(d->d_ino == 0)
            continue;
        if(match(d->d_name, suffix))
            return (d->d_name);
    }
    closedir(dp);
    dp = NULL;
    return(NULL) ;
}

int match(const char *s1, const char *s2)
{
    int diff = strlen(s1) - strlen(s2);

    if(strlen(s1) > strlen(s2))
        return(strcmp(&s1[diff], s2) == 0);
    else
        return (0);
}
```

---

**Упражнение 4.1.** Измените функцию `my_double_ls` из предыдущего примера так, чтобы она имела второй параметр – целочисленную переменную `skip`. Если значение `skip` равно нулю, то функция `my_double_ls` должна выполняться так же, как и раньше.

Если значение переменной `skip` равно 1, функция `my_double_ls` должна пропускать все имена файлов, которые начинаются с точки (.).

**Упражнение 4.2.** В предыдущей главе мы познакомились с использованием системных вызовов `stat` и `fstat` для получения информации о файле. Структура `stat`, возвращаемая вызовами `stat` и `fstat`, содержит поле `st_mode`, режим доступа к файлу. Режим доступа к файлу образуется при помощи выполнения побитовой операции ИЛИ значения кода доступа с константами, определяющими, является ли этот файл обычным файлом, каталогом, специальным файлом, или механизмом межпроцессного взаимодействия, таким как именованный канал. Наилучший способ проверить, является ли файл каталогом – использовать макрос `S_ISDIR`:

```
/* Переменная buf получена в результате вызова stat */
if (S_ISDIR(buf.st_mode))
    printf("Это каталог\n");
else
    printf("Это не каталог\n");
```

Измените процедуру `my_double_ls` так, чтобы она вызывала `stat` для каждого найденного файла и выводила звездочку после каждого имени каталога.

#### 4.4.4. Текущий рабочий каталог

Как уже было рассмотрено в разделе 4.2, после входа в систему пользователь работает в текущем рабочем каталоге. Фактически каждый процесс UNIX, то есть каждый экземпляр выполняемой программы, имеет свой текущий рабочий каталог, который используется в качестве начальной точки при поиске относительных путей в вызовах `open` и им подобных. Текущий рабочий каталог пользователя на самом деле является текущим рабочим каталогом процесса оболочки, интерпретирующего команды пользователя.

Первоначально в качестве текущего рабочего каталога процесса задается текущий рабочий каталог запустившего его процесса, обычно оболочки. Процесс может поменять свой текущий рабочий каталог при помощи системного вызова `chdir`.

#### 4.4.5. Смена рабочего каталога при помощи вызова `chdir`

##### Описание

```
#include <unistd.h>

int chdir(const char *path);
```

После выполнения системного вызова `chdir` каталог `path` становится текущим рабочим каталогом вызывающего процесса. Важно отметить, что эти изменения

относятся только к процессу, который выполняет вызов `chdir`. Смена текущего каталога в программе не затрагивает запустивший программу командный интерпретатор, поэтому после выхода из программы пользователь окажется в том же рабочем каталоге, в котором он находился перед запуском программы, независимо от перемещений программы.

Системный вызов `chdir` завершится неудачей и вернет значение `-1`, если путь `path` не является корректным именем каталога или если вызывающий процесс не имеет доступ на выполнение (прохождение) для всех каталогов в пути.

Системный вызов может успешно использоваться, если нужно получить доступ к нескольким файлам в заданном каталоге. Смена каталога и задание имен файлов относительно нового каталога будет более эффективной, чем использование абсолютных имен файлов. Это связано с тем, что системе приходится поочередно проверять все каталоги в пути, пока не будет найдено искомое имя файла, поэтому уменьшение числа составляющих в пути файла экономит время. Например, вместо использования следующего фрагмента программы

```
fd1 = open("/usr/ben/abc", O_RDONLY);
fd2 = open("/usr/ben/xyz", O_RDWR);
```

можно использовать:

```
chdir("/usr/ben");
fd1 = open("abc", O_RDONLY);
fd2 = open("xyz", O_RDWR);
```

#### 4.4.6. Определение имени текущего рабочего каталога

Спецификация XSI определяет функцию (а не системный вызов) `getcwd`, которая возвращает имя текущего рабочего каталога.

##### Описание

```
#include <unistd.h>

char *getcwd(char *name, size_t size);
```

Функция `getcwd` возвращает указатель на имя текущего каталога. Следует помнить, что значение второго аргумента `size` должно быть больше длины имени возвращаемого пути не менее чем на единицу. В случае успеха имя текущего каталога копируется в массив, на который указывает переменная `name`. Если значение `size` равно нулю или меньше значения, необходимого для возвращения строки имени текущего каталога, то вызов завершится неудачей и вернет нулевой указатель. В некоторых реализациях, если переменная `name` содержит нулевой указатель, то функция `getcwd` сама запросит `size` байтов оперативной памяти; тем не менее, так как эта семантика зависит от системы, не рекомендуется вызывать функцию `getcwd` с нулевым указателем.

Эта короткая программа имитирует команду `pwd`:

```
/* Программа my_pwd - вывод полного имени рабочего каталога */
#include <stdio.h>
#include <unistd.h>
```

```
#define VERYBIG 200

void my_pwd(void);

main()
{
    my_pwd();
}

void my_pwd(void)
{
    char dirname[VERYBIG];

    if( getcwd(dirname, VERYBIG) == NULL)
        perror("Ошибка вызова getcwd ");
    else
        printf("%s\n", dirname);
}
```

#### 4.4.7. Обход дерева каталогов

Иногда необходимо выполнить операцию над иерархией каталогов, начав от стартового каталога, и обойти все лежащие ниже файлы и подкаталоги. Для этого в системе UNIX существует процедура `ftw`, выполняющая обход дерева каталогов, начиная с заданного, и вызывающая процедуру, определенную пользователем для каждой встретившейся записи в каталоге.

##### Описание

```
#include <ftw.h>

int ftw(const char *path, int(*func)(), int depth);
```

Первый параметр `path` определяет имя каталога, с которого должен начаться рекурсивный обход дерева. Параметр `depth` управляет числом используемых функцией `ftw` различных дескрипторов файлов. Чем больше значение `depth`, тем меньше будет случаев повторного открытия каталогов, что сократит общее время отработки вызова. Хотя на каждом уровне дерева будет использоваться только один дескриптор, следует быть уверенным, что значение переменной `depth` не больше числа свободных дескрипторов файлов. Для определения максимально возможного числа дескрипторов, которые может задействовать процесс, рекомендуется использовать системный вызов `getrlimit`, обсуждаемый в главе 12.

Второй параметр `func` – это определенная пользователем функция, вызываемая для каждого файла или каталога, найденного в поддереве каталога `path`. Как можно увидеть из описания, параметр `func` передается процедуре `ftw` как указатель на функцию, поэтому функция должна быть объявлена до вызова процедуры `ftw`. При каждом вызове функции `func` будут передаваться три аргумента: заканчивающаяся нулевым символом строка с именем объекта, указатель на структуру `stat` с данными об объекте и целочисленный код. Функция `func`, следовательно, должна быть построена следующим образом:

```
int func(const char *name, const struct stat *sptr, int type)
{
    /* Тело функции */
}
```

Целочисленный аргумент `type` может принимать одно из нескольких возможных значений (определенных в заголовочном файле `<ftw.h>`) и описывающих тип встретившегося объекта. Вот эти значения:

<code>FTW_F</code>	Объект является файлом
<code>FTW_D</code>	Объект является каталогом
<code>FTW_DNR</code>	Объект является каталогом, который нельзя прочесть
<code>FTW_SL</code>	Объект является символьной ссылкой
<code>FTW_NS</code>	Объект не является символьной ссылкой, и для него нельзя успешно выполнить вызов <code>stat</code>

Если объект является каталогом, который нельзя прочесть (`type = FTW_DNR`), то его потомки не будут обрабатываться. Если нельзя успешно выполнить функцию `stat` (`type = FTW_NS`), то передаваемая для объекта структура `stat` будет иметь неопределенные значения.

Работа вызова будет продолжаться до тех пор, пока не будет завершен обход дерева или не возникнет ошибка внутри функции `ftw`. Обход также закончится, если определенная пользователем функция возвратит ненулевое значение. Тогда функция `ftw` прекратит работу и вернет значение, возвращенное функцией пользователя. Ошибки внутри функции `ftw` приведут к возврату значения `-1`, тогда в переменной `errno` будет выставлен соответствующий код ошибки.

Следующий пример использует функцию `ftw` для обхода поддерева каталогов, выводящего имена всех встретившихся файлов (каталогов) и права доступа к ним. Каталоги и символьные ссылки при выводе будут обозначаться дополнительной звездочкой.

Сначала рассмотрим функцию `list`, которая будет передаваться в качестве аргумента функции `ftw`.

```
#include <sys/stat.h>
#include <ftw.h>

int list(const char *name, const struct stat *status,
        int type)
{
    /* Если вызов stat завершился неудачей, просто вернуться */
    if (type == FTW_NS)
        return 0;

    /*
     * Иначе вывести имя объекта,
     * права доступа к нему и постфикс "*",
     * если объект является каталогом или символьной ссылкой.
     */
    if (type == FTW_F)
```

```
printf("%-30s\t0%3o\n", name, status->st_mode&0777);
else
printf("%-30s*\t0%3o\n", name, status->st_mode&0777);

return 0;
}
```

Теперь запишем основную программу, которая принимает в качестве параметра путь и использует его в качестве начальной точки для обхода дерева. Если аргументы не заданы, то обход начинается с текущего рабочего каталога:

```
main(int argc, char **argv)
{
    int list(const char *, const struct stat *, int);

    if (argc == 1)
        ftw(".", list, 1);
    else
        ftw(argv[1], list, 1);
    exit(0) ;
}
```

Вывод программы `list` для простой иерархии каталогов будет выглядеть так:

```
$ list
. * 0755
./list * 0755
./file1 0644
./subdir * 0777
./subdir/another 0644
./subdir/subdir2 * 0755
./subdir/yetanother 0644
```

Обратите внимание на порядок обхода каталогов.

## 4.5. Файловые системы UNIX

Как уже было рассмотрено, файлы могут быть организованы в различные каталоги, которые образуют иерархическую древовидную структуру. Каталоги могут быть сгруппированы вместе, образуя *файловую систему* (file system). Обычно с файловыми системами имеет дело только системный администратор UNIX. Они позволяют распределять структуру каталогов по нескольким различным физическим дискам или разделам диска, сохраняя однородность структуры с точки зрения пользователя.

Каждая файловая система начинается с каталога в иерархическом дереве. Это свойство позволяет системным администраторам разбивать иерархию файлов UNIX и отводить под ее части отдельные области на диске или даже распределять файловую структуру между несколькими физическими дисковыми устройствами. В большинстве случаев физическое разбиение файловой системы остается невидимым для пользователей.

Файловые системы также называются *монтируемыми томами* (mountable volumes), поскольку их можно динамически *монтировать* и *демонтировать* в виде целых поддеревьев в определенные точки общей древовидной структуры каталогов системы. Демонтирование файловой системы делает все ее содержимое временно недоступным для пользователей. Операционной системе могут быть доступны несколько файловых систем, но не все из них обязательно будут видны как части древовидной структуры.

Информация, содержащаяся в файловой системе, находится на разделе диска, доступном через *файл устройства* (device file), также называемый *специальным файлом* (special file). Этот тип файлов будет описан ниже, а пока просто упомянем, что в системе UNIX каждая файловая система однозначно определяется некоторым именем файла.

Реальное расположение данных файловой системы на носителе никак не связано с высокоуровневым иерархическим представлением каталогов с точки зрения пользователя. Кроме того, расположение данных файловой системы не определяется спецификацией XSI – существуют разные реализации. Ядро может поддерживать одновременно несколько типов файловых систем с различной организацией хранения данных. Здесь будет описано только традиционное расположение.

Традиционная файловая система разбита на ряд логических частей. Каждая такая файловая система содержит четыре определенных секции: *загрузочная область* (bootstrap area), *суперблок* (superblock), ряд блоков, зарезервированных для структур *индексных дескрипторов* (inode) файловой системы, и области, отведен-



Рис. 4.6. Расположение традиционной файловой системы

ной для блоков данных, образующих файлы этой файловой системы. Это расположение схематично представлено на рис. 4.6. Первый из этих блоков (блок с нулевым логическим номером, физически он может быть расположен где угодно внутри раздела диска) зарезервирован для использования в качестве загрузочного блока. Это означает, что он может содержать зависящую от оборудования загрузочную программу, которая используется для загрузки ОС UNIX при старте системы.

Логический блок 1 в файловой системе называется *суперблоком*. Он содержит всю жизненно важную информацию о системе, например, полный размер файловой системы (г блоков на приведенном рисунке), число блоков, отведенных для индексных дескрип-

торов (n-2), дату и время последнего обновления файловой системы. Суперблок содержит также два списка. В первом из них находится часть цепочки номеров свободных блоков секции данных, а во втором – часть цепочки номеров свободных



индексных дескрипторов. Эти два списка обеспечивают ускорение доступа к файловой системе при выделении новых блоков на диске для хранения дополнительных данных или при создании нового файла или каталога. Суперблок смонтированной файловой системы находится в памяти для обеспечения быстрого доступа к списку свободных блоков и свободных узлов. Эти списки в памяти пополняются с диска по мере их исчерпания.

Размер структуры индексных дескрипторов зависит от файловой системы; например, в определенных файловых системах она имеет размер 64 байта, а в других – 128 байт. Индексные дескрипторы последовательно нумеруются, начиная с единицы, поэтому для определения положения структуры индексного дескриптора с заданным номером, прочтенным из записи каталога (как это происходит при переходе в подкаталог или при открытии определенного файла каталога), используется совсем простой алгоритм.

Файловые системы создаются при помощи программы `mkfs`, и при ее запуске задаются размеры области индексных дескрипторов и области данных. В традиционных файловых системах размеры этих областей нельзя изменять динамически, поэтому можно было исчерпать пространство файловой системы одним из двух способов. Во-первых, это может произойти, если были использованы все блоки данных (даже если еще есть доступные номера индексных дескрипторов). Во-вторых, могут быть использованы все номера индексных дескрипторов (при создании большого числа мелких файлов), и, следовательно, дальнейшее создание новых файлов в файловой системе станет невозможным, даже если есть еще свободные блоки данных. В настоящее время современные файловые системы могут иметь переменный размер, и пространство под индексные дескрипторы часто выделяется динамически.

Теперь понятно, что номера индексных дескрипторов являются уникальными только в пределах файловой системы, вот почему невозможно использовать жесткие ссылки между файловыми системами.

#### **4.5.1. Кэширование: вызовы `sync` и `fsync`**

Из соображений эффективности в традиционной файловой системе копии суперблоков смонтированных систем находятся в оперативной памяти. Их обновление может выполняться очень быстро, без необходимости обращаться к диску. Аналогично все операции между памятью и диском обычно кэшируются в области данных оперативной системы вместо немедленной записи на диск. Операции чтения также буферизуются в кэше. Следовательно, в любой заданный момент времени данные на диске могут оказаться устаревшими по сравнению с данными кэша в оперативной памяти. В UNIX существуют две функции, которые позволяют процессу убедиться, что содержимое кэша совпадает с данными на диске. Системный вызов `sync` используется для сброса на диск всего буфера памяти, содержащего информацию о файловой системе, а вызов `fsync` используется для сброса на диск всех данных и атрибутов, связанных с определенным файлом.

### Описание

```
#include <unistd.h>

void sync(void);

int fsync(int filedес);
```

Важное отличие между этими двумя вызовами состоит в том, что вызов `fsync` не завершается до тех пор, пока все данные не будут записаны на диск. Вызов `sync` может завершиться, но запись данных при этом может быть не завершена, а только занесена в планировщик (более того, в некоторых реализациях вызов `sync` может быть ненужным и не иметь эффекта).

Функция `sync` не возвращает значения. Функция `fsync` будет возвращать нулевое значение в случае успеха и `-1` – в случае ошибки. Вызов `fsync` может завершиться неудачей, если, например, переменная `filedes` содержит некорректный дескриптор файла.

Чтобы убедиться, что содержимое файловых систем на диске не слишком долго отстает от времени, в системе UNIX регулярно производится вызов `sync`. Обычно период запуска `sync` равен 30 секундам, хотя этот параметр может изменяться системным администратором.

## 4.6. Имена устройств UNIX

Подключенные к системе UNIX периферийные устройства (диски, терминалы, принтеры, дисковые массивы и так далее) доступны при помощи их имен в файловой системе. Эти файлы называются *файлами устройств* (device files). Соответствующие файловым системам разделы дисков также относятся к классу объектов, представленных этими специальными файлами.

В отличие от обычных дисковых файлов, чтение и запись в файлы устройств приводит к пересылке данных напрямую между системой и соответствующим периферийным устройством.

Обычно эти специальные файлы находятся в каталоге `/dev`. Поэтому, например, имена

```
/dev/tty00
/dev/console
/dev/pts/as    (псевдотерминал для сетевого доступа)
```

могут соответствовать трем портам терминалов системы, а имена

```
/dev/lp
/dev/rmt0
/dev/rmt/0cbn
```

могут обозначать матричный принтер и два накопителя на магнитной ленте.

Имена разделов диска могут иметь разнообразный формат, например:

```
/dev/dsk/c0b0t0d0s3
/dev/dsk/hd0d
```

В командах оболочки и в программах файлы устройств могут использоваться так же, как и обычные файлы, например, команды

```
$ cat fred > /dev/lp
$ cat fred > /dev/rmt0
```

выведут файл `fred` на принтер и накопитель на магнитной ленте соответственно (если это позволяют права доступа). Очевидно, что пытаться таким образом оперировать разделами диска с файловыми системами – огромный риск. Одна неосторожная команда может привести к случайной потере большого объема ценных данных. Кроме того, если бы права доступа к таким файлам устройств были бы не очень строгими, то продвинутые пользователи могли бы обойти ограничения прав доступа, наложенные на файлы в файловой системе. Поэтому системные администраторы должны задавать для файлов дисковых разделов соответствующие права доступа, чтобы иметь уверенность в том, что такие действия невозможны.

Для доступа к файлам устройств в программе могут использоваться вызовы `open`, `close`, `read` и `write`, например, программа

```
#include <fcntl.h>

main()
{
    int i, fd;

    fd = open("/dev/tty00", O_WRONLY);

    for(i=0; i<100; i++)
        write(fd, "x", 1);

    close(fd);
}
```

приведет к выводу 100 символов `x` на порт терминала `tty00`. Конечно, работа с терминалом является отдельной важной темой, поэтому она подробнее будет рассмотрена в главе 9.

#### 4.6.1. Файлы блочных и символьных устройств

Файлы устройств UNIX разбиваются на две категории: *блочные устройства* (block devices) и *символьные устройства* (character devices):

- семейство файлов блочных устройств соответствует устройствам класса дисковых накопителей (съемных и встроенных) и накопителей на магнитной ленте. Передача данных между ядром и этими устройствами осуществляется блоками стандартного размера. Все блочные устройства обеспечивают произвольный доступ. Внутри ядра доступ к этим устройствам управляется хорошо структурированным набором процедур и структур ядра. Этот общий интерфейс к блочным устройствам означает, что обычно драйверы блочных устройств очень похожи, различаясь только в низкоуровневом управлении заданным устройством;

□ семейство файлов символьных устройств соответствует устройствам терминалов, модемных линий, устройствам печати, то есть тем устройствам, которые не используют блочный механизм структурированной пересылки данных. Произвольный доступ для символьных устройств может как поддерживаться, так и не поддерживаться. Данные передаются не блоками фиксированного размера, а в виде потоков байтов произвольной длины.

Важно заметить, что файловые системы могут находиться только на блочных устройствах, и блочные устройства имеют связанные с ними символьные устройства для быстрого и простого доступа, которые называются *устройствами прямого доступа* (raw device). Утилиты `mkfs` и `fsck` используют интерфейс прямого доступа.

ОС UNIX использует две конфигурационные таблицы для связи периферийного устройства с кодом его управления, эти таблицы называются *таблицей блочных устройств* (block device switch) и *таблицей символьных устройств* (character device switch). Обе таблицы проиндексированы при помощи значения *старшего номера устройства* (major device number), который записан в номере индексного дескриптора файла устройства. Последовательность передачи данных к периферийному устройству и от него выглядит так:

1. Системные вызовы `read` или `write` обращаются к индексному дескриптору файла устройства обычным способом.
2. Система проверяет флаг в структуре индексного дескриптора и определяет, является ли устройство блочным или символьным. Также извлекается старший номер устройства.
3. Старший номер используется для индексирования соответствующей таблицы устройств и нахождения процедуры драйвера устройства, нужной для непосредственного выполнения передачи данных.

Таким образом, порядок доступа к периферийным устройствам полностью согласуется с порядком доступа к обычным дисковым файлам.

Кроме старшего номера устройства, в индексном дескрипторе также записано второе значение, называемое *младшим номером устройства* (minor device number) и передаваемое процедурам драйвера устройства для точного задания номера порта на устройствах, которые поддерживают более одного порта, или для обозначения одного из разделов жесткого диска, обслуживаемых одним драйвером. Например, на 8-портовой плате терминала все линии будут иметь один и тот же старший номер устройства и, соответственно, тот же набор процедур драйвера устройства, но каждая конкретная линия будет иметь свой уникальный младший номер устройства в диапазоне от 0 до 7.

#### 4.6.2. Структура `stat`

Структура `stat`, которую уже была обсуждена в главе 3, позволяет хранить информацию о файле устройства в двух полях:

`st_mode`      В случае файла устройства это поле содержит права доступа к файлу, к которым прибавлено восьмеричное значение `060000` для

блочных устройств или 020000 для символьных устройств. В заголовочном файле `<stat.h>` определены константы `S_IFBLK` и `S_IFCHR`, которые могут использоваться вместо этих чисел

`st_rdev` Это поле содержит старший и младший номера устройства

Можно вывести эту информацию при помощи команды `ls` с параметром `-l`, например:

```
$ ls -l /dev/tty3
crw--w--w- 1 ben other 8,3 Sep 13 10:19 /dev/tty3
```

Обратите внимание на символ `c` в первой строке вывода, что говорит о том, что `/dev/tty3` является символьным устройством. Значения 8 и 3 представляют старший и младший номера устройства соответственно.

Можно получить в программе значение поля `st_mode` при помощи методики, введенной в упражнении 4.2:

```
if(S_ISCHR(buf.st_mode))
    printf("Символьное устройство\n");
else
    printf("Не символьное устройство\n");

S_ISCHR – это макрос, определенный в файле <stat.h>.
```

### 4.6.3. Информация о файловой системе

Для устройств, которые представляют файловые системы, применимы две функции, сообщающие основную информацию о файловой системе, – полное число блоков, число свободных блоков, число свободных индексных дескрипторов и т.д. Это функции `statvfs` и `fstatvfs`.

#### Описание

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

int statvfs(const char *path, struct statvfs *buf);

int fstatvfs(int fd, struct statvfs *buf);
```

Обе функции возвращают информацию о файловой системе, заданной либо именем файла устройства `path` для функции `statvfs`, либо дескриптором открытого файла `fd` для функции `fstatvfs`. Параметр `buf` является указателем на структуру `statvfs`, определенную в заголовочном файле `<sys/statvfs.h>`. Структура `statvfs` включает, по меньшей мере, следующие элементы:

<code>unsigned long f_bsize</code>	Размер блока данных, при котором система имеет наибольшую производительность. Например, значение <code>f_frsize</code> может быть равно 1 Кбайт, но значение <code>f_bsize</code> может составлять при этом 8 Кбайт, что означает, что система обеспечивает более эффективный ввод/вывод при операциях с такими порциями данных
------------------------------------	---

<code>unsigned long f_frsize</code>	Основной размер блоков файловой системы (заданный при ее создании)
<code>unsigned long f_blocks</code>	Полное число блоков в файловой системе в единицах, заданных в переменной <code>f_frsize</code>
<code>unsigned long f_bfree</code>	Полное число свободных блоков
<code>unsigned long f_avail</code>	Число свободных блоков, доступных непривилегированным процессам
<code>unsigned long f_files</code>	Полное число номеров индексных дескрипторов
<code>unsigned long f_ffree</code>	Полное число свободных номеров индексных дескрипторов
<code>unsigned long f_favail</code>	Число номеров узлов, доступных непривилегированным процессам
<code>unsigned long f_fsid</code>	Идентификатор файловой системы
<code>unsigned long f_flag</code>	Битовая маска значений флагов
<code>unsigned long f_namemax</code>	Максимальная длина файла

Следующий пример делает примерно то же самое, что и стандартная команда `df`. Эта программа использует функцию `statvfs` для вывода числа свободных блоков и свободных индексных дескрипторов в файловой системе.

```
/* Программа fsys - вывод информации о файловой системе */
/* Имя файловой системы передается в качестве аргумента */

#include <sys/types.h>
#include <sys/statvfs.h>
#include <stdlib.h>
#include <stdio.h>

main(int argc, char **argv)
{
    struct statvfs buf;

    if(argc != 2)
    {
        fprintf(stderr, "Применение: fsys имя_файла\n");
        exit(1);
    }

    if(statvfs(argv[1], &buf) != 0)
    {
        fprintf(stderr, "Ошибка вызова statvfs\n");
        exit (2);
    }
    printf("%s:\tсвободных блоков %d\tсвободных индексных\n",
        argv[1], buf.f_bfree, buf.f_ffree);
    exit(0);
}
```

#### 4.6.4. Ограничения файловой системы: процедуры *pathconf* и *fpathconf*

Комитет разработки стандарта POSIX и другие группы разработки стандартов несколько формализовали способ определения системных ограничений. Так как система может поддерживать различные типы файловых систем, определенные ограничения могут различаться для разных файлов и каталогов. Для запроса этих ограничений, связанных с определенным каталогом, могут использоваться две процедуры, *pathconf* и *fpathconf*.

##### Описание

```
#include <unistd.h>

long int pathconf(const char *pathname, int name);

long int fpathconf(int filedes, int name);
```

Обе эти процедуры работают одинаково и возвращают значение для запрошенного ограничения или переменной. Различие между ними заключается в первом параметре: для процедуры *pathconf* это имя файла или каталога, а для процедуры *fpathconf* – дескриптор открытого файла. Второй параметр является значением одной из констант, определенных в файле *<unistd.h>* и обозначающих запрашиваемое ограничение.

Следующая программа *lookup* может использоваться для вывода системных ограничений для заданного файла/каталога. В этом примере программа *lookup* выводит наиболее интересные из этих значений для стандартного каталога */tmp*:

```
/* Программа lookup – выводит установки ограничений файлов */
```

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

typedef struct{
    int val;
    char *name;
} Table;

main()
{
    Table *tb;
    static Table options[] = {
        { _PC_LINK_MAX, "Максимальное число ссылок"},
        { _PC_NAME_MAX, "Максимальная длина имени файла"},
        { _PC_PATH_MAX, "Максимальная длина пути"},
        {-1, NULL}
    };

    for(tb=options; tb->name != NULL; tb++)
        printf("%-28.28s\t%ld\n", tb->name,
            pathconf("/tmp", tb->val));
}
```

На одной из систем эта программа вывела следующий результат:

Максимальное число ссылок	32767
Максимальная длина имени файла	256
Максимальная длина пути	1024

Эти значения относятся к каталогу `/tmp`. Максимально возможное число ссылок является характеристикой самого каталога, а максимальная длина имени файла относится к файлам в каталоге. Существуют также *общесистемные ограничения* (system-wide limits), они декларируются в файле `<limits.h>`, и их значения могут быть определены при помощи похожей процедуры `sysconf`.





# Глава 5. Процесс

## 5.1. Понятие процесса

Как было уже рассмотрено в главе 1, процессом в терминологии UNIX является просто экземпляр выполняемой программы, соответствующий определению задачи в других средах. Каждый процесс объединяет код программы, значения данных в переменных программы и более экзотические элементы, такие как значения регистров процессора, стек программы и т.д.<sup>1</sup>

Командный интерпретатор для выполнения команд обычно запускает один или несколько процессов. Например, командная строка

```
$ cat file1 file2
```

приведет к созданию процесса для выполнения команды `cat`. Немного более сложная команда

```
$ ls | wc -l
```

приведет к созданию двух процессов для одновременного выполнения команд `ls` и `wc`. (Кроме этого, результат программы `ls`, вывод списка файлов в каталоге, *перенаправляется* с помощью программного канала (`pipe`) на вход программы подсчета числа слов `wc`.)

Так как процессы соответствуют выполняемым программам, не следует путать их с программами, которые они выполняют. Несколько процессов могут выполнять одну и ту же программу. Например, если несколько пользователей выполняют одну и ту же программу редактора, то каждый из экземпляров программы будет отдельным процессом.

Любой процесс UNIX может, в свою очередь, запускать другие процессы. Это придает среде процессов UNIX иерархическую структуру, подобную дереву каталогов файловой системы. На вершине дерева процессов находится единственный управляющий процесс, экземпляр очень важной программы `init`, которая является предком всех системных и пользовательских процессов.

Система UNIX предоставляет программисту набор системных вызовов для создания процессов и управления ими. Если исключить различные средства

---

<sup>1</sup> Не следует путать это понятие с понятием потока выполнения, когда несколько копий кода могут работать с одним набором данных. Потоки выполнения сейчас доступны в некоторых реализациях UNIX, и они охватываются последними расширениями стандарта POSIX и спецификации XSI. Тем не менее мы не будем более подробно описывать сложности многопоточной модели. За дальнейшей информацией обратитесь к справочному руководству системы.

для межпроцессного взаимодействия, то наиболее важными из оставшихся будут:

fork	Используется для создания нового процесса, дублирующего вызывающий. Вызов <code>fork</code> является основным примитивом создания процессов
exec	Семейство библиотечных процедур и один системный вызов, выполняющих одну и ту же функцию – смену задачи процесса за счет перезаписи его пространства памяти новой программой. Различие между вызовами <code>exec</code> в основном лежит в способе задания списка их аргументов
wait	Этот вызов обеспечивает элементарную синхронизацию процессов. Он позволяет процессу ожидать завершения другого процесса, обычно логически связанного с ним
exit	Используется для завершения процесса

Далее рассмотрим, что представляют собой процессы UNIX в целом и вышеприведенные четыре важных системных вызова в частности.

## 5.2. Создание процессов

### 5.2.1. Системный вызов `fork`

Основным примитивом для создания процессов является системный вызов `fork`. Он является механизмом, который превращает UNIX в многозадачную систему.

#### Описание

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

pid_t fork(void);
```

В результате успешного вызова `fork` ядро создает новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе, за одним важным исключением, которое вскоре обсудим.

Созданный процесс называется *дочерним процессом* (child process), а процесс, осуществивший вызов `fork`, называется *родителем* (parent).

После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом `fork`.

Идею, заключенную в вызове `fork`, быть может, достаточно сложно понять тем, кто привык к схеме последовательного программирования. Рис. 5.1 иллюстрирует это понятие. На рисунке рассматриваются три строки кода, состоящие из вызова `printf`, за которым следует вызов `fork` и еще один вызов `printf`.

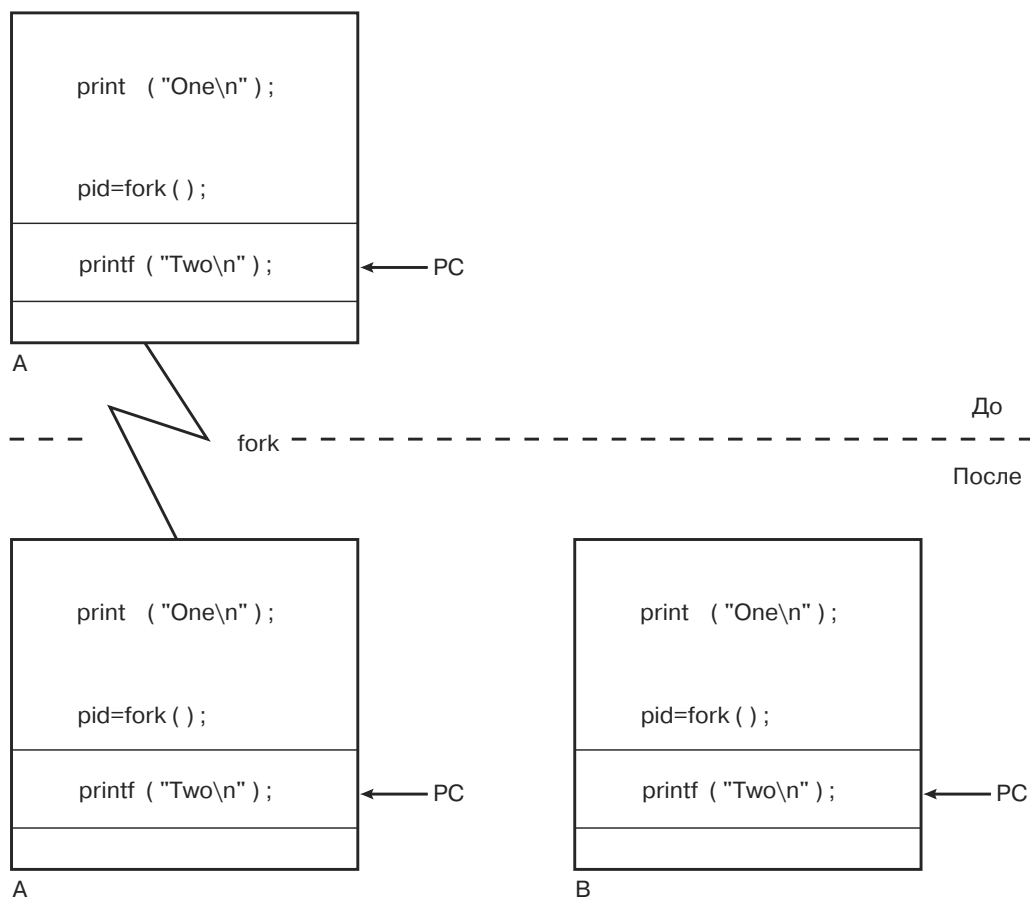
Рис. 5.1. Вызов `fork`

Рисунок разбит на две части: *До* и *После*. Часть рисунка *До* показывает состояние до вызова `fork`. Существует единственный процесс *A* (его обозначили буквой *A* только для удобства, для системы это ничего не значит). Стрелка, обозначенная *PC* (*program counter* – программный счетчик), указывает на выполняемый в настоящий момент оператор. Так как стрелка указывает на первый оператор `printf`, на стандартный вывод выдается тривиальное сообщение `One`.

Часть рисунка *После* показывает ситуацию сразу же после вызова `fork`. Теперь существуют два выполняющихся одновременно процесса: *A* и *B*. Процесс *A* – это тот же самый процесс, что и в части рисунка *До*. Процесс *B* – это новый процесс, порожденный вызовом `fork`. Этот процесс является копией процесса *A* за одним важным исключением – он имеет другое значение идентификатора процесса `pid`, но выполняет ту же самую программу, что и процесс *A*, то есть те же три строки исходного кода, приведенные на рисунке. В соответствии с введенной выше терминологией процесс *A* является родительским процессом, а процесс *B* – дочерним.

Две стрелки с надписью *PC* в этой части рисунка показывают, что следующим оператором, который выполняется родителем и потомком после вызова `fork`, является вызов `printf`. Другими словами, оба процесса *A* и *B* продолжают выполнение с той же точки кода программы, хотя процесс *B* и является новым процессом для системы. Поэтому сообщение `Two` выводится дважды.

### Идентификатор процесса

Как было уже отмечено в начале этого раздела, вызов `fork` не имеет аргументов и возвращает идентификатор процесса `pid_t`. В файле `<sys/types.h>` определен специальный тип `pid_t`, который обычно является целым. Пример вызова:

```
#include <unistd.h> /* Включает определение pid_t */
pid_t pid;

pid = fork();
```

Родитель и потомок отличаются значением переменной `pid`. В родительском процессе значение переменной `pid` будет ненулевым положительным числом, для потомка же оно равно нулю. Так как возвращаемые в родительском и дочернем процессе значения различаются, то программист может задавать различные действия для двух процессов.

Значение, возвращаемое родительскому процессу в переменной `pid`, называется *идентификатором процесса* (*process-id*) дочернего процесса. Это число идентифицирует процесс в системе аналогично идентификатору пользователя. Поскольку все процессы порождаются при помощи вызова `fork`, то каждый процесс UNIX имеет уникальный идентификатор процесса.

Следующая короткая программа более наглядно показывает работу вызова `fork` и использование идентификатора процесса:

```
/* Программа spawn — демонстрация вызова fork */
#include <unistd.h>

main()
{
    pid_t pid; /* process-id в родительском процессе */

    printf("Пока всего один процесс\n");
    printf("Вызов fork...\n");

    pid = fork(); /* Создание нового процесса */

    if (pid == 0)
        printf("Дочерний процесс\n");
    else if (pid > 0)
        printf("Родительский процесс, pid потомка %d\n", pid);
    else
        printf("Ошибка вызова fork, потомок не создан\n");
}
```

Оператор `if`, следующий за вызовом `fork`, имеет три ветви. Первая определяет дочерний процесс, соответствующий нулевому значению переменной `pid`. Вто-

рая задает действия для родительского процесса, соответствуя положительному значению переменной `pid`. Третья ветвь неявно соответствует отрицательному, а на самом деле равному `-1`, значению переменной `pid`, которое возвращается, если вызову `fork` не удастся создать дочерний процесс. Это может означать, что вызывающий процесс попытался нарушить одно из двух ограничений; первое из них – системное ограничение на число процессов; второе ограничивает число процессов, одновременно выполняющихся и запущенных одним пользователем. В обоих случаях переменная `errno` содержит код ошибки `EAGAIN`. Обратите также внимание на то, что поскольку оба процесса, созданных программой, будут выполняться одновременно без синхронизации, то нет гарантии, что вывод родительского и дочернего процессов не будет смешиваться.

Перед тем как продолжить, стоит обсудить, зачем нужен вызов `fork`, поскольку сам по себе он может показаться бессмысленным. Существенный момент заключается в том, что вызов `fork` обретает ценность в сочетании с другими средствами UNIX. Например, возможно, что родительский и дочерний процессы будут выполнять различные, но связанные задачи, организуя совместную работу при помощи одного из механизмов межпроцессного взаимодействия, такого как сигналы или каналы (описываемые в следующих главах). Другим средством, часто используемым совместно с вызовом `fork`, является системный вызов `exec`, позволяющий выполнять другие программы, и который будет рассмотрен в следующем разделе.

---

**Упражнение 5.1.** Программа может осуществлять вызов `fork` несколько раз. Аналогично каждый дочерний процесс может вызывать `fork`, порождая своих потомков. Чтобы доказать это, напишите программу, которая создает два подпроцесса, а они, в свою очередь, – свой подпроцесс. После каждого вызова `fork` каждый родительский процесс должен использовать функцию `printf` для вывода идентификаторов своих дочерних процессов.

---

## 5.3. Запуск новых программ при помощи вызова `exec`

### 5.3.1. Семейство вызовов `exec`

Если бы вызов `fork` был единственным доступным для программиста примитивом создания процессов, то система UNIX была бы довольно скучной, так как в ней можно было бы создавать копии только одной программы. К счастью, для смены исполняемой программы можно использовать функции семейства `exec`. На рис. 5.2 показано дерево семейства функций `exec`. Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рисунка, в конечном итоге все эти функции выполняют один системный вызов `execve`.

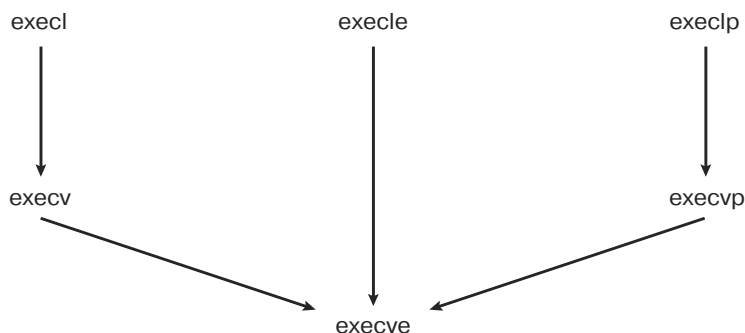


Рис. 5.2  
Дерево семейства  
вызовов `exec`

### Описание

```

#include <unistd.h>

/* Для семейства вызовов exec1 аргументы должны быть списком,
   заканчивающимся NULL */

/* Вызову exec1 нужно передать полный путь к файлу программы */
int exec1(const char *path,
          const char *arg0, ..., const char *argn,
          (char *)0);

/* Вызову execlp нужно только имя файла программы */
int execlp(const char *file,
           const char *arg0, ..., const char *argn,
           (char *)0);

/* Семейству вызовов execv нужно передать массив аргументов */
/* Вызову execv нужно передать полный путь к файлу программы */
int execv(const char *path, char *const argv[]);

/* Вызову execvp нужно только имя файла программы */
int execvp(const char *file, char *const argv[]);

```

Все множество системных вызовов `exec` выполняет одну и ту же функцию: они преобразуют вызывающий процесс, загружая новую программу в его пространство памяти. Если вызов `exec` завершился успешно, то вызывающая программа полностью замещается новой программой, которая запускается с начала. Результат вызова можно рассматривать как запуск нового процесса, который при этом сохраняет идентификатор вызывающего процесса и по умолчанию наследует файловые дескрипторы (см. пункт 5.5.2).

Важно отметить, что вызов `exec` не создает новый подпроцесс, который выполняется одновременно с вызывающим, а вместо этого новая программа загружается на место старой. Поэтому, в отличие от вызова `fork`, успешный вызов `exec` не возвращает значения.

Для простоты осветим только один из вызовов `exec`, а именно `exec1`.

Все аргументы функции `exec1` являются указателями строк. Первый из них, аргумент `path`, задает имя файла, содержащего программу, которая будет запущена на выполнение; для вызова `exec1` это должен быть полный путь к программе,

абсолютный или относительный. Сам файл должен содержать программу или последовательность команд оболочки и быть доступным для выполнения. Система определяет, содержит ли файл программу, просматривая его первые байты (обычно первые два байта). Если они содержат специальное значение, называемое *магическим числом* (magic number), то система рассматривает файл как программу. Второй аргумент, `arg0`, является, по соглашению, именем программы или команды, из которого исключен путь к ней. Этот аргумент и оставшееся переменное число аргументов (от `arg1` до `argn`) доступны в вызываемой программе, аналогично аргументам командной строки при запуске программы из оболочки. В действительности командный интерпретатор сам вызывает команды, используя один из вызовов `exec` совместно с вызовом `fork`. Так как список аргументов имеет произвольную длину, он должен заканчиваться нулевым указателем для обозначения конца списка.

Короткий пример ценнее тысячи слов – следующая программа использует вызов `execl` для запуска программы вывода содержимого каталога `ls`:

```
/* Программа runls - использование "execl" для запуска ls */
#include <unistd.h>
main()
{
    printf("Запуск программы ls\n");
    execl("/bin/ls", "ls", "-l", (char *)0);

    /* Если execl возвращает значение, то вызов был неудачным */
    perror("Вызов execl не смог запустить программу ls");
    exit(1);
}
```

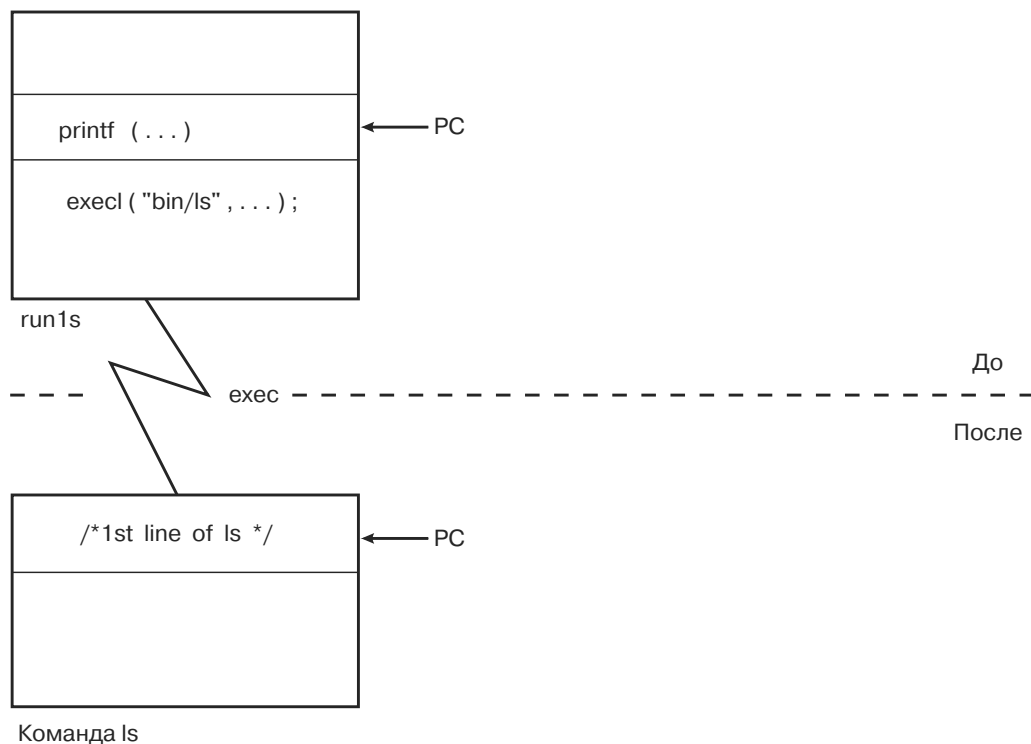
Работа этой демонстрационной программы показана на рис. 5.3. Часть *До* показывает процесс непосредственно перед вызовом `execl`. Часть *После* показывает измененный процесс после вызова `execl`, который при этом выполняет программу `ls`. Программный счетчик *РС* указывает на первую строку программы `ls`, показывая, что вызов `execl` запускает программу с начала.

Обратите внимание, что в примере за вызовом `execl` следует безусловный вызов библиотечной процедуры `perror`. Это отражает то, что успешный вызов функции `execl` (и других родственных функций) стирает вызывающую программу. Если вызывающая программа сохраняет работоспособность и происходит возврат из вызова `execl`, значит, произошла ошибка. Поэтому возвращаемое значение `execl` и родственных функций всегда равно `-1`.

### **Вызовы `execv`, `execvp` и `execsv`**

Другие формы вызова `exec` упрощают задание списков параметров запуска загружаемой программы. Вызов `execv` принимает два аргумента: первый (`path` в описании применения вызова) является строкой, которая содержит полное имя и путь к запускаемой программе. Второй аргумент (`argv`) является массивом строк, определенным как:

```
char * const argv[];
```

Рис. 5.3. Вызов `exec`

Первый элемент этого массива указывает, по принятому соглашению, на имя запускаемой программы (исключая префикс пути). Оставшиеся элементы указывают на все остальные аргументы программы. Так как этот список имеет неопределенную длину, он всегда должен заканчиваться нулевым указателем.

Следующий пример использует вызов `execv` для запуска той же программы `ls`, что и в предыдущем примере:

```
/* Программа runls2 - использует вызов execv для запуска ls */
#include <unistd.h>

main()
{
    char * const av[]={ "ls", "-l", (char *)0 };
    execv("/bin/ls", av);

    /* Если мы оказались здесь, то произошла ошибка */
    perror("execv failed");
    exit(1);
}
```

Функции `execlp` и `execvp` почти эквивалентны функциям `execl` и `execv`. Основное отличие между ними состоит в том, что первый аргумент обеих функций `execlp` и `execvp` – просто имя программы, не включающее путь к ней. Путь к файлу



находится при помощи поиска в каталогах, заданных в переменной среды `PATH`. Переменная `PATH` может быть легко задана на уровне командного интерпретатора с помощью следующих команд:

```
$ PATH = /bin:/usr/bin:/usr/keith/mybin
$ export PATH
```

Теперь командный интерпретатор и вызов `exesvr` будут вначале искать команды в каталоге `/bin`, затем в `/usr/bin`, и, наконец, в `/usr/keith/mybin`.

---

**Упражнение 5.2.** В каком случае нужно использовать вызов `exesv` вместо `exesl`?

---

**Упражнение 5.3.** Предположим, что вызовы `exesvr` и `exesclp` не существуют. Напишите эквиваленты этих процедур, используя вызовы `exesl` и `exesv`. Параметры этих процедур должны состоять из списка каталогов и набора аргументов командной строки.

---

### 5.3.2. Доступ к аргументам, передаваемым при вызове `exes`

Любая программа может получить доступ к аргументам активизировавшего ее вызова `exes` через параметры, передаваемые функции `main`. Эти параметры могут быть описаны при определении функции `main` следующим образом:

```
main(int argc, char **argv)
{
    /* Тело программы */
}
```

Такое описание должно быть знакомо большинству программистов, так как похожий метод используется для доступа к аргументам командной строки при обычном старте программы — еще один признак того, что командный интерпретатор также использует для запуска процессов вызовы `exes`. (Несколько предшествующих примеров и упражнений были составлены с учетом того, что читателям книги известен метод получения программой параметров ее командной строки. Ниже эта тема будет рассмотрена подробнее.)

В вышеприведенном определении функции `main` значение переменной `argc` равно числу аргументов, переменная `argv` указывает на массив самих аргументов. Поэтому, если программа запускается на выполнение при помощи вызова `exesvr` следующим образом:

```
char * const argin[] = ("команда", "с", "аргументами", (char *)0);
exesvr("prog", argin);
```

то в программе `prog` будут истинны следующие выражения (выражения вида `argv[x] == "xxx"` следует считать фигуральным равенством, а не выражением языка C):

```
argc == 3
argv[0] == "команда"
```

```
argv[1] == "c"
argv[2] == "аргументами"
argv[3] == (char *)0
```

В качестве простой иллюстрации этого метода рассмотрим следующую программу, которая печатает свои аргументы, за исключением нулевого, на стандартный вывод:<sup>1</sup>

```
/* Программа myecho – вывод аргументов командной строки */
main(int argc, char **argv)
{
    while(--argc > 0)
        printf("%s ", **++argv);

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

Если вызвать эту программу в следующем фрагменте кода

```
char * const argin[]={ "myecho", "hello", "world", (char *)0 };
execvp(argin[0], argin);
```

то переменная `argc` в программе `myecho` будет иметь значение 3, и в результате на выходе программы получим:

```
hello world
```

Тот же самый результат можно получить при помощи команды оболочки:

```
$ myecho hello world
```

---

**Упражнение 5.4.** *Напишите программу `waitcmd`, которая выполняет произвольную команду при изменении файла. Она должна принимать в качестве аргументов командной строки имя контролируемого файла и команду, которая должна выполняться в случае его изменения. Для слежения за файлом можно использовать вызовы `stat` и `fstat`. Программа не должна расходовать напрасно системные ресурсы, поэтому следует использовать процедуру `sleep` (представленную в упражнении 2.16), для приостановки выполнения программы `waitcmd` в течение заданного интервала времени, после того как она проверит файл. Как должна действовать программа, если файл изначально не существует?*

---

---

<sup>1</sup> Здесь следует отметить, что значение `argv[0]` – не пустая трата памяти, а весьма важный параметр. Во-первых, он напоминает программе ее имя: признаком хорошего стиля программирования считается вывод диагностики от имени программы `argv[0]`, ведь заранее не известно, как впоследствии переименует программу пользователь. Во-вторых, у исполняемого файла может быть несколько имен (вспомните про ссылки), и это можно выгодно использовать. Часто множество утилит на самом деле

## 5.4. Совместное использование вызовов `exec` и `fork`

Системные вызовы `fork` и `exec`, объединенные вместе, представляют мощный инструмент для программиста. Благодаря ветвлению при использовании вызова `exec` во вновь созданном дочернем процессе программа может выполнять другую программу в дочернем процессе, не стирая себя из памяти. Следующий пример показывает, как это можно сделать. В нем мы также представим простую процедуру обработки ошибок `fatal` и системный вызов `wait`. Системный вызов `wait`, описанный ниже, заставляет процесс ожидать завершения работы дочернего процесса.

```
/* Программа runls3 – выполнить ls как subprocess */

#include <unistd.h>
main()
{
    pid_t pid;

    switch (pid = fork()){
    case -1:
        fatal("Ошибка вызова fork");
        break;
    case 0:
        /* Потомок вызывает exec */
        execl("/bin/ls", "ls", "-l", (char *)0);
        fatal("Ошибка вызова exec ");
        break;
    default:
        /* Родительский процесс вызывает wait для приостановки
         * работы до завершения дочернего процесса.
         */
        wait( (int *)0);
        printf("Программа ls завершилась\n") ;
        exit(0);
    }
}
```

Процедура `fatal` использует функцию `perror` для вывода сообщения, а затем завершает работу процесса. (Оператор `break`, следующий за вызовом `fatal`, в данном примере не нужен, но он страхует код от любых будущих изменений в процедуре.) Процедура `fatal` реализована следующим образом:

```
int fatal(char *s)
{
    perror(s);
    exit(1) ;
}
```

---

является одной программой, которая ведет себя по-разному в зависимости от использованного имени. Это кажется странным, но прекрасно работает. Так, программа удаленного выполнения команд `rsh`, будучи названной именем `peibog`, ведет себя так, как будто ей сообщили дополнительный первый аргумент `peibog`. Можно заготовить набор псевдонимов этой программы для запуска команд на соседних системах сети. – *Прим. науч. ред.*

Снова графическое представление, в данном случае рис. 5.4, используется для наглядного объяснения работы программы. Рисунок разбит на три части: *До вызова fork*, *После вызова fork* и *После вызова exec*.

В начальном состоянии, *До вызова fork*, существует единственный процесс *A*, и программный счетчик *PC* направлен на оператор `fork`, показывая, что это следующий оператор, который должен быть выполнен.

После вызова `fork` существует два процесса, *A* и *B*. Родительский процесс *A* выполняет системный вызов `wait`. Это приведет к приостановке выполнения процесса *A* до тех пор, пока процесс *B* не завершится. В это время процесс *B* использует вызов `exec` для запуска на выполнение команды `ls`.

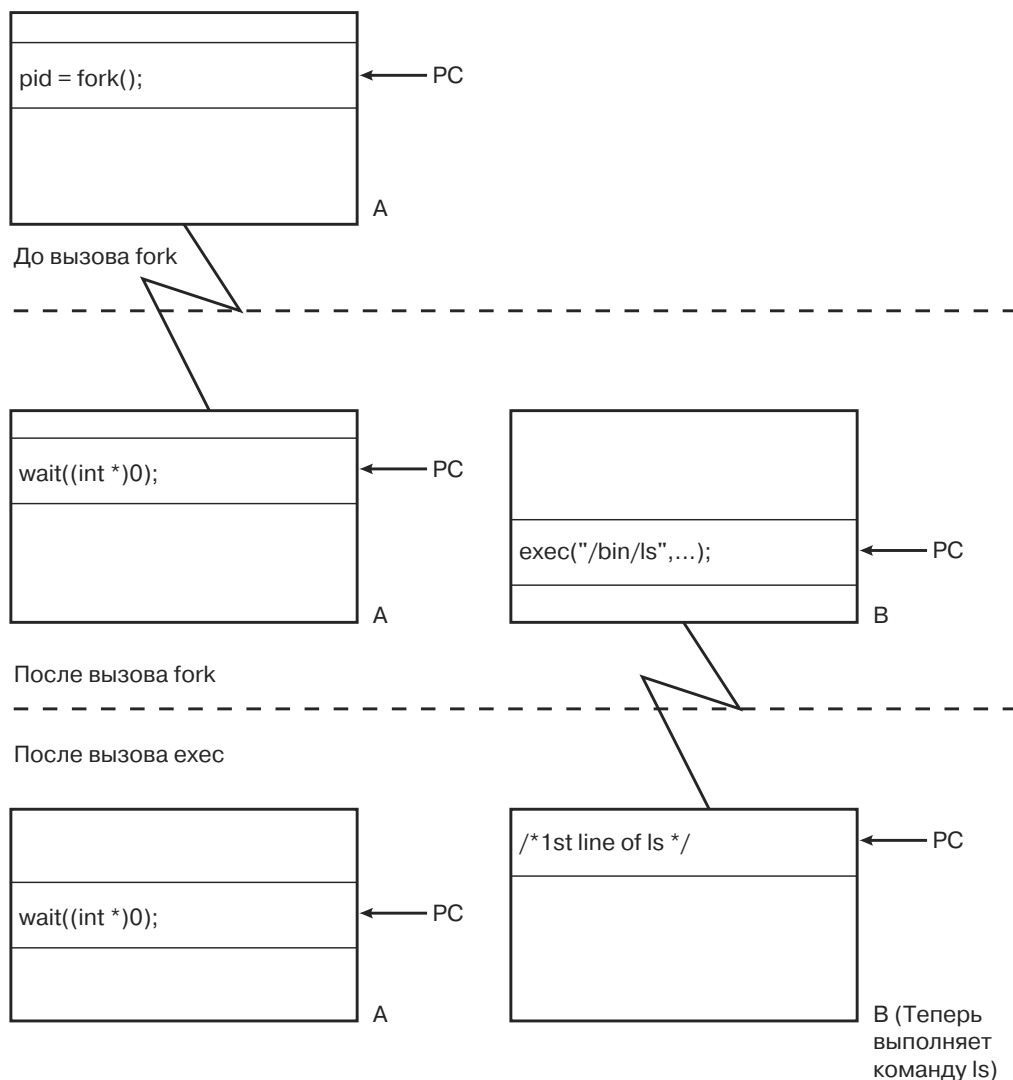


Рис. 5.4. Совместное использование вызовов `fork` и `exec`

Что происходит дальше, показано в части *После вызова* `exec` на рис. 5.4. Процесс *B* изменился и теперь выполняет программу `ls`. Программный счетчик процесса *B* установлен на первый оператор команды `ls`. Так как процесс *A* ожидает завершения процесса *B*, то положение его программного счетчика *PC* не изменилось.

Теперь можно увидеть в общих чертах механизмы, используемые командным интерпретатором. Например, при обычном выполнении команды оболочка использует вызовы `fork`, `exec` и `wait` приведенным выше образом. При фоновом исполнении команды вызов `wait` пропускается, и оба процесса – команда и оболочка – продолжают выполняться одновременно.

### **Пример `docommand`**

ОС UNIX предоставляет библиотечную процедуру `system`, которая позволяет выполнить в программе команду оболочки. Для примера создадим упрощенную версию этой процедуры `docommand`, используя вызовы `fork` и `exec`. В качестве посредника вызовом стандартную оболочку (заданную именем `/bin/sh`), а не будем пытаться выполнять программу напрямую. Это позволит программе `docommand` воспользоваться преимуществами, предоставляемыми оболочкой, например, раскрытием шаблонов имен файлов. Задание параметра `-c` вызова оболочки определяет, что команды передаются не со стандартного ввода, а берутся из следующего строчного аргумента.

```
/* Программа docommand — запуск команды оболочки, 1 версия */

#include <unistd.h>

int docommand(char *command)
{
    pid_t pid;
    if ((pid = fork()) < 0 )
        return (-1);

    if (pid == 0) /* Дочерний процесс */
    {
        execl("/bin/sh", "sh", "-c", command, (char *)0);
        perror("execl"),
        exit(1);
    }

    /* Код родительского процесса */
    /* Ожидание возврата из дочернего процесса */
    wait( (int *)0);
    return (0);
}
```

Следует сказать, что это только первое приближение к настоящей библиотечной процедуре `system`. Например, если конечный пользователь программы нажмет клавишу прерывания во время выполнения команды оболочки, то и вызывающая программа, и команда остановятся. Существуют способы обойти это ограничение, которые будут рассмотрены в следующей главе.

## 5.5. Наследование данных и дескрипторы файлов

### 5.5.1. Вызов *fork*, файлы и данные

Созданный при помощи вызова *fork* дочерний процесс является почти точной копией родительского. Все переменные в дочернем процессе будут иметь те же самые значения, что и в родительском (единственным исключением является значение, возвращаемое самим вызовом *fork*). Так как данные в дочернем процессе являются *копией* данных в родительском процессе и занимают другое абсолютное положение в памяти, важно понимать, что последующие изменения в одном процессе не будут затрагивать переменные в другом.

Аналогично все файлы, открытые в родительском процессе, также будут открытыми и в потомке; при этом дочерний процесс будет иметь свою копию связанных с каждым файлом дескрипторов. Тем не менее файлы, открытые до вызова *fork*, остаются тесно связанными в родительском и дочернем процессах. Это обусловлено тем, что указатель чтения-записи для каждого из таких файлов используется совместно родительским и дочерним процессами благодаря тому, что он поддерживается системой и существует не только в самом процессе. Следовательно, если дочерний процесс изменяет положение указателя в файле, то в родительском процессе он также окажется в новом положении. Это поведение демонстрирует следующая короткая программа, в которой использована процедура *fatal*, приведенная ранее в этой главе, а также новая процедура *printpos*. Дополнительно введено допущение, что существует файл с именем *data* длиной не меньше 20 символов.

```
/* Программа proc_file — поведение файлов при ветвлении */
/* Предположим, что длина файла "data" не менее 20 символов */

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

main()
{
    int fd;
    pid_t pid;      /* Идентификатор процесса */
    char buf[10];   /* Буфер данных для файла */

    if(( fd = open("data", O_RDONLY)) == -1)
        fatal("Ошибка вызова open ");

    read(fd, buf, 10); /* Переместить вперед указатель файла */

    printpos ("До вызова fork", fd);

    /* Создать два процесса */
    switch(pid = fork()){
    case -1:          /* Ошибка */
        fatal("Ошибка вызова fork ");
        break;
    case 0:           /* Потомок */
```

```
    printpos("Дочерний процесс до чтения", fd),
    read(fd, buf, 10);
    printpos ("Дочерний процесс после чтения", fd);
    break;
default:      /* Родитель */
    wait((int *)0);
    printpos ("Родительский процесс после ожидания", fd);
}
}
```

Процедура `printpos` просто выводит текущее положение в файле, а также короткое сообщение. Ее можно реализовать следующим образом:

```
/* Вывести положение в файле */
int printpos (const char *string, int filedes)
{
    off_t pos;

    if(( pos = lseek(filedes, 0, SEEK_CUR)) == -1)
        fatal("Ошибка вызова lseek");
    printf("%s:%ld\n", string, pos);
}
```

После запуска этого примера получены результаты, которые убедительно подтверждают то, что указатель чтения-записи совместно используется обоими процессами:

```
До вызова fork:10
Дочерний процесс до чтения:10
Дочерний процесс после чтения:20
Родительский процесс после ожидания:20
```

---

**Упражнение 5.5.** *Напишите программу, показывающую, что значения переменных программы в родительском и дочернем процессах первоначально совпадают, но не зависят друг от друга.*

---

---

**Упражнение 5.6.** *Определите, что происходит в родительском процессе, если дочерний процесс закрывает файл, дескриптор которого он унаследовал после ветвления. Другими словами, останется ли файл открытым в родительском процессе или же будет закрыт?*

---

### 5.5.2. Вызов `exes` и открытые файлы

Дескрипторы открытых файлов обычно сохраняют свое состояние также во время вызова `exes`, то есть файлы, открытые в исходной программе, остаются открытыми, когда совершенно новая программа запускается при помощи вызова `exes`. Указатели чтения-записи на такие файлы остаются неизменными после вызова `exes`. (Очевидно, не имеет смысла говорить о сохранении значений переменных

после вызова `exec`, так как в общем случае новая программа совершенно отличается от старой.)

Тем не менее есть связанный с файловым дескриптором флаг *close-on-exec* (закрывать при вызове `exec`), который может быть установлен с помощью универсальной процедуры `fcntl`. Если этот флаг установлен (по умолчанию он сброшен), то файл закрывается при вызове любой функции семейства `exec`. Следующий фрагмент показывает, как устанавливается флаг `close-on-exec`:

```
#include <fcntl.h>

.
.
.
int fd;

fd = open("file", O_RDONLY);

.
.
.

/* Установить флаг close-on-exec flag */
fcntl(fd, F_SETFD, 1);
```

Флаг `close-on-exec` можно сбросить так:

```
fcntl(fd, F_SETFD, 0);
```

Значение флага можно получить следующим образом:

```
res = fcntl(fd, F_GETFD, 0);
```

Целое `res` будет иметь значение 1, если флаг `close-on-exit` установлен для дескриптора файла `fd`, и 0 – в противном случае.

## 5.6. Завершение процессов при помощи системного вызова `exit`

### Описание

```
#include <stdlib.h>

void exit(int status);
```

Системный вызов `exit` уже известен, но теперь следует дать его правильное описание. Этот вызов используется для завершения процесса, хотя это также происходит, когда управление доходит до конца тела функции `main` или до оператора `return` в функции `main`.

Единственный целочисленный аргумент вызова `exit` называется *статусом завершения* (`exit status`) процесса, младшие восемь бит которого доступны родительскому процессу при условии, если он выполнил системный вызов `wait` (подробнее об этом см. в следующем разделе). При этом возвращаемое вызовом `exit` значение обычно используется для определения успешного или неудачного завершения выполнявшейся процессом задачи. По принятому соглашению, нулевое



возвращаемое значение соответствует нормальному завершению, а ненулевое значение говорит о том, что что-то случилось.

Кроме завершения вызывающего его процесса, вызов `exit` имеет еще несколько последствий: наиболее важным из них является закрытие всех открытых дескрипторов файлов. Если, как это было в последнем примере, родительский процесс выполнял вызов `wait`, то его выполнение продолжится. Процедура `exit` также вызовет определенные программистом процедуры обработки выхода и выполнит так называемые процедуры очистки. Они могут относиться, например, к буферизации в стандартной библиотеке ввода/вывода. Программист также может выбрать не менее 32 процедур обработки вывода при помощи функции `atexit`.

### Описание

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Процедура `atexit` регистрирует функцию, на которую указывает ссылка `func`, которая будет вызываться без параметров. Каждая из заданных в процедуре `atexit` функций будет вызываться при выходе в порядке, обратном порядку их расположения.

Для полноты изложения следует также упомянуть системный вызов `_exit`, который отличается от вызова `exit` наличием символа подчеркивания в начале. Он используется в точности так же, как и вызов `exit`. Тем не менее он не включает описанные ранее действия по очистке. В большинстве случаев следует избегать использования вызова `_exit`.

---

**Упражнение 5.7.** Статус завершения программы можно получить в командном интерпретаторе при помощи переменной `$?`, например:

```
$ ls nonesuch
nonesuch: No such file or directory
$ echo $?
2
```

Напишите программу `fake`, которая использует целочисленное значение первого аргумента в качестве статуса завершения. Используя намеченный выше метод, выполните программу `fake`, задавая различные значения аргументов, включая большие и отрицательные. Есть ли польза от программы `fake`?

---

## 5.7. Синхронизация процессов

### 5.7.1. Системный вызов `wait`

#### Описание

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

pid_t wait(int *status);
```

Как было уже обсуждено, вызов `wait` временно приостанавливает выполнение процесса, в то время как дочерний процесс продолжает выполняться. После завершения дочернего процесса выполнение родительского процесса продолжится. Если запущено более одного дочернего процесса, то возврат из вызова `wait` произойдет после выхода из любого из потомков.

Вызов `wait` часто осуществляется родительским процессом после вызова `fork`, например:

```
.
.
.
int status;
pid_t cpid;
cpid = fork(); /*Создать новый процесс */
if(cpid == 0){

    /* Дочерний процесс */
    /* Выполнить какие-либо действия ... */

}else{

    /* Родительский процесс, ожидание завершения дочернего */
    cpid = wait(&status);
    printf("Дочерний процесс %d завершился\n",cpid);

}
.
.
.
```

Сочетание вызовов `fork` и `wait` наиболее полезно, если дочерний процесс предназначен для выполнения совершенно другой программы при помощи вызова `exec`.

Возвращаемое значение `wait` обычно является идентификатором дочернего процесса, который завершил свою работу. Если вызов `wait` возвращает значение (`pid_t`)-1, это может означать, что дочерние процессы не существуют, и в этом случае переменная `errno` будет содержать код ошибки `ECHILD`. Возможность определить завершение каждого из дочерних процессов по отдельности означает, что родительский процесс может выполнять цикл, ожидая завершения каждого из потомков, а после того, как все они завершатся, продолжать свою работу.

Вызов `wait` принимает один аргумент, `status`, – указатель на целое число. Если указатель равен `NULL`, то аргумент просто игнорируется. Если же вызову `wait` передается допустимый указатель, то после возврата из вызова `wait` переменная `status` будет содержать полезную информацию о статусе завершения процесса. Обычно эта информация будет представлять собой код завершения дочернего процесса, переданный при помощи вызова `exit`.

Следующая программа `status` показывает, как может быть использован вызов `wait`:

```
/* Программа status – получение статуса завершения потомка */
#include <sys/wait.h>
```

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

main()
{
    pid_t pid;
    int status, exit_status;

    if((pid = fork()) < 0)
        fatal("Ошибка вызова fork ");

    if(pid == 0)    /* Потомок */
    {
        /* Вызвать библиотечную процедуру sleep
         * для временного прекращения работы на 4 секунды.
         */
        sleep(4);
        exit(5);    /* Выход с ненулевым значением */
    }

    /* Если мы оказались здесь, то это родительский процесс, */
    /* поэтому ожидать завершения дочернего процесса */

    if((pid = wait(&status)) == -1)
    {
        perror("Ошибка вызова wait ");
        exit(2);
    }

    /* Проверка статуса завершения дочернего процесса */
    if(WIFEXITED(status))
    {
        exit_status = WEXITSTATUS(status);
        printf("Статус завершения %d равен %d\n", pid, exit_status);
    }
    exit(0);
}
```

Значение, возвращаемое родительскому процессу при помощи вызова `exit`, записывается в старшие восемь бит целочисленной переменной `status`. Чтобы оно имело смысл, младшие восемь бит должны быть равны нулю. Макрос `WIFEXITED` (определенный в файле `<sys/wait.h>`) проверяет, так ли это на самом деле. Если макрос `WIFEXITED` возвращает 0, то это означает, что выполнение дочернего процесса было остановлено (или прекращено) другим процессом при помощи межпроцессного взаимодействия, называемого *сигналом* (`signal`) и рассматриваемого в главе 6.

---

**Упражнение 5.8.** Переделайте процедуру `docommand` так, чтобы она возвращала статус вызова `exit` выполняемой команды. Что должно происходить, если вызов `wait` возвращает значение `-1`?

---

### 5.7.2. Ожидание завершения определенного потомка: вызов `waitpid`

Системный вызов `wait` позволяет родительскому процессу ожидать завершения любого дочернего процесса. Тем не менее, если нужна большая определенность, то можно использовать системный вызов `waitpid` для ожидания завершения определенного дочернего процесса.

#### Описание

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

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Первый аргумент `pid` определяет идентификатор дочернего процесса, завершения которого будет ожидать родительский процесс. Если этот аргумент установлен равным `-1`, а аргумент `options` установлен равным `0`, то вызов `waitpid` ведет себя в точности так же, как и вызов `wait`, поскольку значение `-1` соответствует любому дочернему процессу. Если значение `pid` больше нуля, то родительский процесс будет ждать завершения дочернего процесса с идентификатором процесса равным `pid`. Во втором аргументе `status` будет находиться статус дочернего процесса после возврата из вызова `waitpid`.

Последний аргумент, `options`, может принимать константные значения, определенные в файле `<sys/wait.h>`. Наиболее полезное из них – константа `WNOHANG`. Задание этого значения позволяет вызывать `waitpid` в цикле без блокирования процесса, контролируя ситуацию, пока дочерний процесс продолжает выполняться. Если установлен флаг `WNOHANG`, то вызов `waitpid` будет возвращать `0` в случае, если дочерний процесс еще не завершился.

Функциональные возможности вызова `waitpid` с параметром `WNOHANG` можно продемонстрировать, переписав предыдущий пример. На этот раз родительский процесс проверяет, завершился ли уже дочерний процесс. Если нет, он выводит сообщение, говорящее о том, что он продолжает ждать, затем делает секундную паузу и снова вызывает `waitpid`, проверяя, завершился ли дочерний процесс. Обратите внимание, что потомок получает свой идентификатор процесса при помощи вызова `getpid`. Об этом вызове расскажем в разделе 5.10.1.

```
/* Программа status2 – получение статуса завершения
 * дочернего процесса при помощи вызова waitpid */
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
```

```
main()
{
    pid_t pid;
    int status, exit_status;

    if((pid = fork()) < 0)
        fatal("Ошибка вызова fork ");
```

```
if(pid == 0)    /* потомок */
{
    /* Вызов библиотечной процедуры sleep
     * для приостановки выполнения на 4 секунды.
     */
    printf("Потомок %d пауза...\n",getpid());
    sleep(4);
    exit(5);    /* Выход с ненулевым значением */
}

/* Если мы оказались здесь, то это родительский процесс */
/* Проверить, закончился ли дочерний процесс, и если нет, */
/* то сделать секундную паузу, и потом проверить снова */
while(waitpid(pid, &status, WNOHANG) == 0)
{
    printf("Ожидание продолжается...\n");
    sleep(1);
}

/* Проверка статуса завершения дочернего процесса */
if(WIFEXITED(status))
{
    exit_status = WEXITSTATUS(status);
    printf("Статус завершения %d равен %d\n",pid, exit_status);
}

exit(0) ;
}
```

При запуске программы получим следующий вывод:

```
Ожидание продолжается...
Потомок 12857 пауза...
Ожидание продолжается...
Ожидание продолжается...
Ожидание продолжается...
Статус завершения 12857 равен 5
```

## 5.8. Зомби-процессы и преждевременное завершение программы

До сих пор предполагалось, что вызовы `exit` и `wait` используются правильно, и родительский процесс ожидает завершения каждого подпроцесса. Вместе с тем иногда могут возникать две другие ситуации, которые стоит обсудить:

- ❑ в момент завершения дочернего процесса родительский процесс не выполняет вызов `wait`;
- ❑ родительский процесс завершается, в то время как один или несколько дочерних процессов продолжают выполняться.

В первом случае завершающийся процесс как бы «теряется» и становится *зомби-процессом* (zombie). Зомби-процесс занимает ячейку в таблице, поддерживаемой ядром для управления процессами, но не использует других ресурсов ядра. В конце концов, он будет освобожден, если его родительский процесс вспомнит о нем и вызовет `wait`. Тогда родительский процесс сможет прочесть статус завершения процесса, и ячейка освободится для повторного использования. Во втором случае родительский процесс завершается нормально. Дочерние процессы (включая зомби-процессы) принимаются процессом `init` (процесс, идентификатор которого `pid = 1`, становится их новым родителем).

## 5.9. Командный интерпретатор `smallsh`

В этом разделе создается простой командный интерпретатор `smallsh`. Этот пример имеет два достоинства. Первое состоит в том, что он развивает понятия, введенные в этой главе. Второе – в том, что подтверждается отсутствие в стандартных командах и утилитах UNIX чего-то особенного. В частности, пример показывает, что оболочка является обычной программой, которая запускается при входе в систему.

Наши требования к программе `smallsh` просты: она должна транслировать и выполнять команды – на переднем плане и в фоновом режиме – а также обрабатывать строки, состоящие из нескольких команд, разделенных точкой с запятой. Другие средства, такие как перенаправление ввода/вывода и раскрытие имен файлов, могут быть добавлены позднее.

Основная логика понятна:

```
while(не встретится EOF)
{
    получить строку команд от пользователя
    оттранслировать аргументы и выполнить
    ожидать возврата из дочернего процесса
}
```

Дадим имя `userin` функции, выполняющей «получение командной строки от пользователя». Эта функция должна выводить приглашение, а затем ожидать ввода строки с клавиатуры и помещать введенные символы в буфер программы. Функция `userin` реализована следующим образом:

```
/* Заголовочный файл для примера */
#include "smallsh.h"

/* Буферы программы и рабочие указатели */
static char inbuf[MAXBUF], tokbuf[2*MAXBUF],
    *ptr = inbuf, *tok = tokbuf;

/* Вывести приглашение и считать строку */
int userin(char *p)
{
    int c, count;

    /* Инициализация для следующих процедур */
```

```
ptr = inbuf;
tok = tokbuf;

/* Вывести приглашение */
printf("%s", p);

count = 0;

while(1)
{
    if((c = getchar()) == EOF)
        return(EOF) ;

    if (count < MAXBUF)
        inbuf[count++] = c;

    if( c == '\n' && count < MAXBUF)
    {
        inbuf[count] = '\0';
        return count;
    }

    /* Если строка слишком длинная, начать снова */
    if(c == '\n')
    {
        printf("smallsh: слишком длинная входная строка\n");
        count = 0;
        printf ("%s ",p);
    }
}
}
```

Некоторые детали инициализации можно пока не рассматривать. Главное, что функция `userin` вначале выводит приглашение ввода команды (передаваемое в качестве параметра), а затем считывает ввод пользователя по одному символу до тех пор, пока не встретится символ перевода строки или конец файла (последний случай обозначается символом EOF).

Процедура `getchar` содержится в стандартной библиотеке ввода/вывода. Она считывает один символ из стандартного ввода программы, который обычно соответствует клавиатуре. Функция `userin` помещает каждый новый символ (если это возможно) в массив символов `inbuf`. После своего завершения функция `userin` возвращает либо число считанных символов, либо EOF, обозначающий конец файла. Обратите внимание, что символы перевода строки не отбрасываются, а добавляются в массив `inbuf`.

Заголовочный файл `smallsh.h`, упоминаемый в функции `userin`, содержит определения для некоторых полезных постоянных (например, `MAXBUF`). В действительности файл содержит следующее:

```
/* smallsh.h – определения для интерпретатора smallsh */

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

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

#define EOL      1      /* Конец строки */
#define ARG      2      /* Обычные аргументы */
#define AMPERSAND 3      /* Символ '&' */
#define SEMICOLON 4      /* Точка с запятой */

#define MAXARG    512    /* Макс. число аргументов */
#define MAXBUF    512    /* Макс. длина строки ввода */
#define FOREGROUND 0      /* Выполнение на переднем плане */
#define BACKGROUND 1      /* Фоновое выполнение */
```

Другие постоянные, не упомянутые в функции `userin`, встретятся в следующих процедурах.

Файл `smallsh.h` включает стандартный заголовочный файл `<stdio.h>`, в котором определены процедура `getchar` и постоянная `EOF`.

Рассмотрим следующую процедуру, `gettok`. Она выделяет *лексемы* (tokens) из командной строки, созданной функцией `userin`. (Лексема является минимальной единицей языка, например, имя или аргумент команды.) Процедура `gettok` вызывается следующим образом:

```
toktype = gettok(&tptr);
```

Целочисленная переменная `toktype` будет содержать значение, обозначающее тип лексемы. Диапазон возможных значений берется из файла `smallsh.h` и включает символы `EOL` (конец строки), `SEMICOLON` и так далее. Переменная `tptr` является символьным указателем, который будет указывать на саму лексему после вызова `gettok`. Так как процедура `gettok` сама выделяет пространство под строки лексем, нужно передать адрес переменной `tptr`, а не ее значение.

Исходный код процедуры `gettok` приведен ниже. Обратите внимание, что поскольку она ссылается на символьные указатели `tok` и `ptr`, то должна быть включена в тот же исходный файл, что и `userin`. (Теперь должно быть понятно, зачем была нужна инициализация переменных `tok` и `ptr` в начале функции `userin`.)

```
/* Получить лексему и поместить ее в буфер tokbuf */
int gettok(char **outptr)
{
    int type;

    /* Присвоить указателю на строку outptr значение tok */
    *outptr = tok;

    /* Удалить пробелы из буфера, содержащего лексемы */
    while( *ptr == ' ' || *ptr == '\t')
        ptr++;

    /* Установить указатель на первую лексему в буфере */
    *tok++ = *ptr;

    /* Установить значение переменной type в соответствии
     * с типом лексемы в буфере */
    switch(*ptr++){
```



```
case '\n':
    type = EOL;
    break;
case '&':
    type = AMPERSAND;
    break;
case ';':
    type = SEMICOLON;
    break;
default:
    type = ARG;
    /* Продолжить чтение обычных символов */
    while(inarg(*ptr))
        *tok++ = *ptr++;
}

*tok++ = '\0';
return type;
}
```

Функция `inarg` используется для определения того, может ли символ быть частью «обычного» аргумента. Пока можно просто проверять, является ли символ особым для командного интерпретатора команд `smallsh` или нет:

```
static char special [] = {' ', '\t', '&', ';', '\n', '\0'};

int inarg(char c)
{
    char *wrk;

    for(wrk = special; *wrk; wrk++)
    {
        if(c == *wrk)
            return (0);
    }

    return (1);
}
```

Теперь можно составить функцию, которая будет выполнять главную работу нашего интерпретатора. Функция `procline` будет разбирать командную строку, используя процедуру `gettok`, создавая тем самым список аргументов процесса. Если встретится символ перевода строки или точка с запятой, то она вызывает для выполнения команды процедуру `runcommand`. При этом она предполагает, что командная строка уже была считана при помощи функции `userin`.

```
#include "smallsh.h"
```

```
int procline(void)          /* Обработка строки ввода */
{
    char *arg[MAXARG + 1]; /* Массив указателей для runcommand */
    int toktype;           /* Тип лексемы в команде */
    int nargs;             /* Число аргументов */
}
```

```

int type;          /* На переднем плане или в фоне */
narg=0;
for(;;)           /* Бесконечный цикл */
{
    /* Выполнить действия в зависимости от типа лексемы */
    switch (toktype = gettok (&arg[narg])){
    case ARG:
        if(narg < MAXARG)
            narg++;
        break;
    case EOL:
    case SEMICOLON:
    case AMPERSAND:
        if ( toktype == AMPERSAND)
            type = BACKGROUND;
        else
            type = FOREGROUND;
        if(narg != 0)
        {
            arg[narg] = NULL;
            runcommand(arg, type);
        }

        if(toktype == EOL)
            return;

        narg = 0;
        break;
    }
}
}
}

```

Следующий этап состоит в определении процедуры `runcommand`, которая в действительности запускает командные процессы. Процедура `runcommand`, в сущности, является переделанной процедурой `docommand`, с которой встречались раньше. Она имеет еще один целочисленный параметр `where`. Если параметр `where` принимает значение `BACKGROUND`, определенное в файле `smallsh.h`, то вызов `waitpid` пропускается, и процедура `runcommand` просто выводит идентификатор процесса и завершает работу.

```
#include "smallsh.h"
```

```

/* Выполнить команду, возможно ожидая ее завершения */
int runcommand(char **cline, int where)
{
    pid_t pid;
    int status;

    switch(pid = fork()){
    case -1:

```

```
perror("smallsh");
return (-1);
case 0:
    execvp(*cline, cline);
    perror(*cline) ;
    exit(1);
}

/* Код родительского процесса */
/* Если это фоновый процесс, вывести pid и выйти */
if(where == BACKGROUND)
{
    printf("[Идентификатор процесса %d]\n", pid);
    return (0);
}

/* Ожидание завершения процесса с идентификатором pid */
if(waitpid(pid, &status, 0) == -1)
    return (-1);
else
    return (status);
}
```

Обратите внимание, что простой вызов `wait` из функции `docommand` был заменен вызовом `waitpid`. Это гарантирует, что выход из процедуры `docommand` произойдет только после завершения процесса, запущенного в этом вызове `docommand`, и помогает избавиться от проблем с фоновыми процессами, которые завершаются в это время. (Если это кажется не совсем ясным, следует вспомнить, что вызов `wait` возвращает идентификатор первого завершающегося дочернего процесса, а не идентификатор последнего запущенного.)

Процедура `runcommand` также использует системный вызов `execvp`. Это гарантирует, что при запуске программы, заданной командой, выполняется ее поиск во всех каталогах, указанных в переменной окружения `PATH`, хотя, в отличие от настоящего командного интерпретатора, в программе `smallsh` нет никаких средств для работы с переменной `PATH`.

Последний шаг состоит в написании функции `main`, которая связывает вместе остальные функции. Это простое упражнение:

```
/* Программа smallsh – простой командный интерпретатор */
#include "smallsh.h"

char *prompt = "Command> "; /* Приглашение ввода командной строки */
main()
{
    while(userin(prompt) != EOF)
        procline();
}
```

Эта процедура завершает первую версию программы `smallsh`. И снова следует отметить, что это только набросок законченного решения. Так же, как в случае

процедуры `docommand`, поведение программы `smallsh` далеко от идеала, когда пользователь вводит символ прерывания, поскольку это приводит к завершению работы программы `smallsh`. В следующей главе будет показано, как можно сделать программу `smallsh` более устойчивой.

---

**Упражнение 5.9.** Включите в программу `smallsh` механизм для выключения с помощью символа `\` (escaping) специального значения символов, таких как точка с запятой и символ `&`, так чтобы они могли входить в список аргументов программы. Программа должна также корректно интерпретировать комментарии, обозначаемые символом `#` в начале. Что должно произойти с приглашением командной строки, если пользователь выключил таким способом специальное значение символа возврата строки?

---

**Упражнение 5.10.** Процедуру `fcntl` можно использовать для получения копии дескриптора открытого файла. В этом случае он вызывается следующим образом:

```
newfdes = fcntl(filedes, F_DUPFD, reqvalue);
```

где `filedes` – это исходный дескриптор открытого файла. Постоянная `F_DUPFD` определена в системном заголовочном файле `<fcntl.h>`. Значение переменной `reqvalue` должно быть небольшим целым числом. После успешного вызова переменная `newfdes` будет содержать дескриптор файла, который ссылается на тот же самый файл, что и дескриптор `filedes`, и имеет то же численное значение, что и переменная `reqvalue` (если `reqvalue` еще не является дескриптором файла). Следующий фрагмент программы показывает, как перенаправить стандартный ввод, то есть дескриптор файла со значением 0:

```
fd1 = open("somefile", O_RDONLY);  
close (0);  
fd2 = fcntl(fd1, F_DUPFD, 0);
```

После этого вызова значение дескриптора `fd2` будет равно 0. Используя этот вызов вместе с системными вызовами `open` и `close`, переделайте программу `smallsh` так, чтобы она поддерживала перенаправление стандартного ввода и стандартного вывода, используя ту же систему обозначений, что и стандартный командный интерпретатор UNIX. Помните, что стандартный ввод и вывод соответствует дескрипторам 0 и 1 соответственно. Обратите внимание, что копирование дескрипторов файлов можно также осуществить при помощи системного вызова `dup2`. (Существует также близкий по смыслу вызов `dup`.)

---

## 5.10. Атрибуты процесса

С каждым процессом UNIX связан набор атрибутов, которые помогают системе управлять выполнением и планированием процессов, обеспечивать защиту файловой системы и так далее. Один из атрибутов, с которым мы уже встречались, – это идентификатор процесса, то есть число, которое однозначно идентифицирует процесс. Другие атрибуты простираются от окружения, которое является набором строк, определяемых программистом и находящихся вне области данных, до действующего идентификатора пользователя, определяющего права доступа процесса к файловой системе. В оставшейся части этой главы рассмотрим наиболее важные атрибуты процесса.

### 5.10.1. Идентификатор процесса

Как уже отмечено в начале этой главы, система присваивает каждому процессу неотрицательное число, которое называется идентификатором процесса. В любой момент времени идентификатор процесса является уникальным, хотя после завершения процесса он может использоваться снова для другого процесса. Некоторые идентификаторы процесса зарезервированы системой для особых процессов. Процесс с идентификатором 0, хотя он и называется *планировщиком* (scheduler), на самом деле является процессом *подкачки памяти* (swapper). Процесс с идентификатором 1 – это процесс инициализации, выполняющий программу `/etc/init`. Этот процесс, явно или неявно, является предком всех других процессов в системе UNIX.

Программа может получить свой идентификатор процесса при помощи следующего системного вызова:

```
pid = getpid();
```

Аналогично вызов `getppid` возвращает идентификатор родителя вызывающего процесса:

```
ppid = getppid();
```

Следующая процедура `gentemp` использует вызов `getpid` для формирования уникального имени временного файла. Это имя имеет форму:

```
/tmp/tmp<pid>.<no>
```

Суффикс номера `<no>` увеличивается на единицу при каждом вызове процедуры `gentemp`. Процедура также вызывает функцию `access`, чтобы убедиться, что файл еще не существует:

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

static int num = 0;

static char namebuf[20] ;
static char prefix [] = "/tmp/tmp";

char *gentemp(void)
{
```

```
int length;
pid_t pid;

pid = getpid();    /* Получить идентификатор процесса */

/* Стандартные процедуры работы со строками */
strcpy(namebuf, prefix);
length = strlen(namebuf);

/* Добавить к имени файла идентификатор процесса */
itoa(pid, &namebuf[length]);

strcat(namebuf, ".");
length = strlen(namebuf);

do{
    /* Добавить суффикс с номером */
    itoa(num++, &namebuf[length]);
} while (access(namebuf, F_OK) != -1);

return (namebuf);
}
```

Процедура `itoa` просто преобразует целое число в эквивалентную строку:

```
/* Процедура itoa – преобразует целое число в строку */

int itoa(int i, char *string)
{
    int power, j;

    j = i;

    for(power = 1; j >= 10; j /= 10)
        power *= 10;

    for( ; power > 0; power /= 10)
    {
        *string++ = '0' + i/power;
        i %= power;
    }

    *string = '\0';
}
```

Обратите внимание на способ преобразования цифры в ее символьный эквивалент в первом операторе во втором цикле `for` – он опирается на знание таблицы символов ASCII. Следует также отметить, что большую часть работы можно было бы выполнить гораздо проще при помощи процедуры `sprintf`. Описание процедуры `sprintf` смотрите в главе 11.

---

**Упражнение 5.11.** Переделайте процедуру `gentemp` так, чтобы она принимала в качестве аргумента префикс имени временного файла.

---

### 5.10.2. Группы процессов и идентификаторы группы процессов

Система UNIX позволяет легко помещать процессы в группы. Например, если в командной строке задано, что процессы связаны при помощи программного канала, они обычно помещаются в одну группу процессов. На рис. 5.5 показана такая типичная группа процессов, установленная из командной строки.

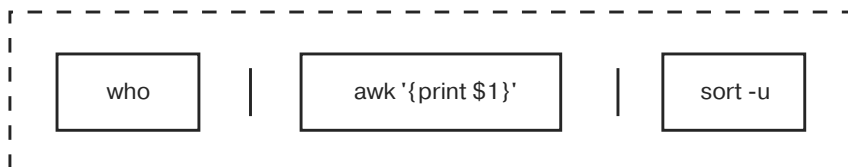


Рис. 5.5  
Группа  
процессов

Группы процессов удобны для работы с набором процессов в целом, с помощью механизма межпроцессного взаимодействия, который называется *сигналами*, о чем будет сказано подробнее в главе 6. Обычно сигнал «посылается» отдельному процессу и может вызывать завершение этого процесса, но можно послать сигнал и целой группе процессов.

Каждая *группа процессов* (process group) обозначается *идентификатором группы процессов* (process group-id), имеющим тип `pid_t`. Процесс, идентификатор которого совпадает с идентификатором группы процессов, считается *лидером* (leader) группы процессов, и при его завершении выполняются особые действия. Первоначально процесс наследует идентификатор группы во время вызова `fork` или `exec`.

Процесс может получить свой идентификатор группы при помощи системного вызова `getpgrp`:

#### Описание

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

pid_t getpgrp(void);
```

### 5.10.3. Изменение группы процесса

В оболочке UNIX, поддерживающей управление заданиями, может потребоваться переместить процесс в другую группу процессов. Управление заданиями позволяет оболочке запускать несколько групп процессов (заданий) и контролировать, какие группы процессов должны выполняться на переднем плане и, следовательно, иметь доступ к терминалу, а какие должны выполняться в фоне. Управление заданиями организуется при помощи сигналов.

Процесс может создать новую группу процессов или присоединиться к существующей при помощи системного вызова `setpgid`:

#### Описание

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

int setpgid(pid_t pid, pid_t pgid);
```

Вызов `setpgid` устанавливает идентификатор группы процесса с идентификатором `pid` равным `pgid`. Если `pid` равен 0, то используется идентификатор *вызывающего* процесса. Если значения идентификаторов `pid` и `pgid` одинаковы, то процесс становится лидером группы процессов. В случае ошибки возвращается значение `-1`. Если идентификатор `pgid` равен нулю, то в качестве идентификатора группы процесса используется идентификатор процесса `pid`.

#### 5.10.4. Сеансы и идентификатор сеанса

В свою очередь, каждая группа процессов принадлежит к сеансу. В действительности сеанс относится к связи процесса с *управляющим терминалом* (`controlling terminal`). Когда пользователи входят в систему, все процессы и группы процессов, которые они явно или неявно создают, будут принадлежать к сеансу, связанному с их текущим терминалом. Сеанс обычно представляет собой набор из одной группы процессов переднего плана, использующей терминал, и одной или более групп фоновых процессов. Сеанс обозначается при помощи *идентификатора сеанса* (`session-id`), который имеет тип `pid_t`.

Процесс может получить идентификатор сеанса при помощи вызова `getsid`:

##### Описание

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

pid_t getsid(pid_t pid);
```

Если передать вызову `getsid` значение 0, то он вернет идентификатор сеанса вызывающего процесса, в противном случае возвращается идентификатор сеанса процесса, заданного идентификатором `pid`.

Понятие сеанса полезно при работе с фоновыми процессами или *процессами-демонами* (`daemon processes`). Процессом-демоном называется просто процесс, не имеющий управляющего терминала. Примером такого процесса является процесс `cron`, запускающий команды в заданное время. Демон может задать для себя сеанс без управляющего терминала, переместившись в другой сеанс при помощи системного вызова `setsid`.

##### Описание

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

pid_t setsid(void);
```

Если вызывающий процесс не является лидером группы процессов, то создается новая группа процессов и новый сеанс, и идентификатор вызывающего процесса станет идентификатором созданного сеанса. Он также не будет иметь управляющего терминала. Процесс-демон теперь будет находиться в странном состоянии, так как он будет единственным процессом в группе процессов, содержащейся в новом сеансе, а его идентификатор процесса `pid` – также идентификатором группы и сеанса.



Функция `setsid` может завершиться неудачей и возвратит значение (`pid_t`) `-1`, если вызывающий процесс уже является лидером группы.

### 5.10.5. Переменные программного окружения

Программное *окружение* (`environment`) процесса – это просто набор строк, заканчивающихся нулевым символом, представленных в программе просто в виде массива указателей на строки. Эти строки называются *переменными окружения* (`environment variables`). По принятому соглашению, каждая строка окружения имеет следующую форму:

имя переменной = ее содержание

Можно напрямую использовать программное окружение процесса, добавив еще один параметр `envp` в список параметров функции `main` в программе. Следующий фрагмент программы показывает функцию `main` с параметром `envp`:

```
main(int argc, char **argv, char **envp)
{
    /* Выполнить какие-либо действия */
}
```

В качестве простого примера следующая программа просто выводит свое окружение и завершает работу:

```
/* Программа showmyenv.c – вывод окружения */
main(int argc, char **argv, char **envp)
{
    while(*envp)
        printf("%s\n", *envp++);
}
```

При запуске этой программы на одной из машин были получены следующие результаты:

```
CDPATH=...:/
HOME=/usr/keith
LOGNAME=keith
MORE=-h -s
PATH=/bin:/etc:/usr/bin:/usr/sbin:/usr/lbin
SHELL=/bin/ksh
TERM=vt100
TZ=GMTOST
```

Этот список может быть вам знакомым. Это окружение процесса командного интерпретатора (оболочки), вызвавшего программу `showmyenv`, и оно включает такие важные переменные, как `HOME` и `PATH`.

Пример показывает, что по умолчанию окружение процесса совпадает с окружением процесса, создавшего его при помощи вызова `fork` или `exec`. Поскольку окружение передается указанным способом, то можно записывать в окружении информацию, которую иначе пришлось бы задавать заново для каждого нового процесса. Переменная окружения `TERM`, в которой записан тип текущего терминала,

является хорошим примером того, насколько полезным может быть использование окружения.

Чтобы задать для процесса новое окружение, необходимо использовать для его запуска один из двух вызовов из семейства `exec`: `execle` или `execve`. Они вызываются следующим образом:

```
execle(path, arg0, arg1, ..argn, (char *)0, envp);
```

и:

```
execve(path, argv, envp);
```

Эти вызовы дублируют соответственно вызовы `execv` и `execl`. Единственное различие заключается в добавлении параметра `envp`, который является заканчивающимся нулевым символом массивом строк, задающим окружение новой программы. Следующий пример использует вызов `execle` для передачи нового программного окружения программе `showmyenv`:

```
/* Программа setmyenv.c — установка окружения программы */
```

```
main()
{
    char *argv[2], *envp[3];

    argv[0] = "showmyenv";
    argv[1] = (char *)0;

    envp[0] = "foo=bar";
    envp[1] = "bar=foo";
    envp[2] = (char *)0;

    execve("./showmyenv", argv, envp);

    perror("Ошибка вызова execve");
}
```

Хотя использование параметров, передаваемых функции `main`, является допустимым, предпочтительнее получение процессом доступа к своему окружению через глобальную переменную:

```
extern char **environ;
```

Для поиска в переменной `environ` имени переменной окружения, заданной в форме `name=string`, можно использовать стандартную библиотечную функцию `getenv`.

### Описание

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

```
char *getenv(const char *name);
```

Аргументом функции `getenv` является имя искомой переменной. При успешном завершении поиска функция `getenv` возвращает указатель на строку переменной окружения, в противном случае — указатель `NULL`. Следующий код показывает пример использования этой функции:

```
/* Найти значение переменной окружения PATH */
#include <stdlib.h>

main()
{
    printf("PATH=%s\n", getenv("PATH"));
}
```

Для изменения окружения существует парная процедура `putenv`. Она вызывается следующим образом:

```
putenv("НОВАЯ_ПЕРЕМЕННАЯ = значение");
```

В случае успеха процедура `putenv` возвращает нулевое значение. Она изменяет переменную окружения, на которую указывает глобальная переменная `environ`, но не меняет указатель `envp` в текущей функции `main`.

### 5.10.6. Текущий рабочий каталог

Как было установлено в главе 4, с каждым процессом связан текущий рабочий каталог. Первоначально текущий рабочий каталог наследуется во время создавшего процесс вызова `fork` или `exec`. Другими словами, процесс первоначально помещается в тот же каталог, что и родительский процесс.

Важным фактом является то, что текущий рабочий каталог является атрибутом отдельного процесса. Если дочерний процесс меняет каталог при помощи вызова `chdir` (определение которого приведено в главе 4), то текущий рабочий каталог родительского процесса не меняется. Поэтому стандартная команда `cd` на самом деле является «встроенной» командой оболочки, а не программой.

### 5.10.7. Текущий корневой каталог

С каждым процессом также связан корневой каталог, который используется при поиске абсолютного пути. Так же, как и текущий рабочий каталог, корневым каталогом процесса первоначально является корневой каталог его родительского процесса. Для изменения начала иерархии файловой системы для процесса в ОС UNIX существует системный вызов `chroot`:

#### Описание

```
#include <unistd.h>

int chroot(const char *path);
```

Переменная `path` указывает на путь, обозначающий каталог. В случае успешного вызова `chroot` путь `path` становится начальной точкой при поиске в путях, начинающихся с символа `/` (только для вызывающего процесса, корневой каталог системы при этом не меняется). В случае неудачи вызов `chroot` не меняет корневой каталог и возвращает значение `-1`. Для изменения корневого каталога вызывающий процесс должен иметь соответствующие права доступа.

---

**Упражнение 5.12.** Добавьте к командному интерпретатору `smallsh` команду `cd`.

---

**Упражнение 5.13.** Напишите собственную версию функции `getenv`.

### 5.10.8. Идентификаторы пользователя и группы

С каждым процессом связаны истинные идентификаторы пользователя и группы. Это всегда идентификатор пользователя и текущий идентификатор группы запустившего процесс пользователя.

Действующие идентификаторы пользователя и группы используются для определения возможности доступа процесса к файлу. Чаще всего, эти идентификаторы совпадают с истинными идентификаторами пользователя и группы. Равенство нарушается, если процесс или один из его предков имеет установленные биты доступа `set-user-id` или `set-group-id`. Например, если для файла программы установлен бит `set-user-id`, то при запуске программы на выполнение при помощи вызова `exec` действующим идентификатором пользователя становится идентификатор владельца файла, а не запустившего процесс пользователя.

Для получения связанных с процессом идентификаторов пользователя и группы существует несколько системных вызовов. Следующий фрагмент программы демонстрирует их:

```
#include <unistd.h>
main()
{
    uid_t uid, euid;
    gid_t gid, egid;

    /* Получить истинный идентификатор пользователя */
    uid = getuid();

    /* Получить действующий идентификатор пользователя */
    euid = geteuid();

    /* Получить истинный идентификатор группы */
    gid = getgid();

    /* Получить действующий идентификатор группы */
    egid = getegid();
}
```

Для задания действующих идентификаторов пользователя и группы процесса также существуют два вызова:

```
#include <unistd.h>

uid_t newuid;
gid_t newgid;
.
.
.
/* Задать действующий идентификатор пользователя */
status = setuid(newuid);
```

```
/* Задать действующий идентификатор группы */  
status = setgid(newgid);
```

Процесс, запущенный непривилегированным пользователем (то есть любым пользователем, кроме суперпользователя) может менять действующие идентификаторы пользователя и группы только на истинные.<sup>1</sup> Суперпользователю, как всегда, предоставляется полная свобода. Обе процедуры возвращают нулевое значение в случае успеха, и  $-1$  – в случае неудачи.

---

**Упражнение 5.14.** *Напишите процедуру, которая получает истинные идентификаторы пользователя и группы вызывающего процесса, а затем преобразует их в символьную форму и записывает в лог-файл.*

---

### 5.10.9. Ограничения на размер файла: вызов `ulimit`

Для каждого процесса существует ограничение на размер файла, который может быть создан при помощи системного вызова `write`. Это ограничение распространяется также на ситуацию, когда увеличивается размер существующего файла, имевшего до этого длину, меньшую максимально возможной.

Предельный размер файла можно изменять при помощи системного вызова `ulimit`.

#### Описание

```
#include <ulimit.h>  
  
long ulimit(int cmd, [long newlimit]);
```

Для получения текущего максимального размера файла можно вызвать `ulimit`, установив значение параметра `cmd` равным `UL_GETFSIZE`. Возвращаемое значение равно числу 512-байтных блоков.

Для изменения максимального размера файла можно присвоить переменной `cmd` значение `UL_SETFSIZE` и поместить новое значение максимального размера файла, также в 512-байтных блоках, в переменную `newlimit`, например:

```
if(ulimit(UL_SETFSIZE, newlimit) < 0)  
    perror("Ошибка вызова ulimit ");
```

В действительности увеличить максимальный размер файла таким способом может только суперпользователь. Процессы с идентификаторами других пользователей могут только уменьшать этот предел.

### 5.10.10. Приоритеты процессов: вызов `nice`

Система приближенно вычисляет долю процессорного времени, отводимую для работы процесса, руководствуясь значением `nice` (буквально «дружелюбный»).

---

<sup>1</sup> Современные системы, согласно спецификации SUSV2 и стандарту POSIX, должны также позволять возвращаться от истинных идентификаторов к сохраненным (*saved-set-uid*, *saved-set-gid*) действующим идентификаторам пользователя и группы. – Прим. науч. ред.

Значения `nice` находятся в диапазоне от нуля до максимального значения, зависящего от конкретной системы. Чем больше это число, тем ниже приоритет процесса. Процессы, «дружелюбно настроенные», могут понижать свой приоритет, используя системный вызов `nice`. Этот вызов имеет один аргумент, положительное число, на которое увеличивается текущее значение `nice`, например:

```
nice(5);
```

Процессы суперпользователя (и только суперпользователя) могут увеличивать свой приоритет, используя отрицательное значение параметра вызова `nice`. Вызов `nice` может пригодиться, если есть необходимость, например, вычислить число  $\pi$  с точностью до ста миллионов знаков, не изменяя существенно время реакции системы для остальных пользователей. Вызов `nice` был введен давно. Современные факультативные расширения реального времени стандарта POSIX определяют гораздо более точное управление планированием параллельной работы процессов. Но это особая тема, которая не будет затронута в данной книге.



# Глава 6. Сигналы и их обработка

## 6.1. Введение

Часто требуется создавать программные системы, состоящие не из одной монолитной программы, а из нескольких сотрудничающих процессов. Для этого может быть множество причин: одиночная программа может оказаться слишком громоздкой и не поместиться в памяти; часть требуемых функций часто уже реализована в каких-либо существующих программах; задачу может быть удобнее решать при помощи процесса-сервера, взаимодействующего с произвольным числом процессов-клиентов; есть возможность использовать несколько процессоров и т.д.

К счастью, ОС UNIX имеет развитые механизмы межпроцессного взаимодействия. В этой и следующей главах мы обсудим три наиболее популярных средства: *сигналы* (signals), *программные каналы* (pipes) и *именованные каналы* (FIFO). Данные средства, наряду с более сложными средствами, которым будут посвящены главы 8 и 10, предоставляют разработчику программного обеспечения широкий выбор средств построения многопроцессных систем.

Эта глава будет посвящена изучению первого средства – сигналам.

Рассмотрим пример запуска команды UNIX, выполнение которой, вероятно, займет много времени:

```
$ cc verybigprog.c
```

Позже становится ясным, что программа содержит ошибку, и ее компиляция не может завершиться успехом. Тогда, чтобы сэкономить время, следует прекратить выполнение команды нажатием специальной *клавиши прерывания задания* (interrupt key) терминала; обычно это клавиша **Del** или клавиатурная комбинация **Ctrl+C**. Выполнение программы прекратится, и программист вернется к приглашению ввода команды командного интерпретатора.

В действительности при этом происходит следующая последовательность событий: часть ядра, отвечающая за ввод с клавиатуры, распознает символ прерывания задания. Затем ядро посылает сигнал SIGINT всем процессам, для которых текущий терминал является управляющим терминалом. Среди этих процессов будет и экземпляр компилятора cc. Когда процесс компилятора cc получит этот сигнал, он выполнит связанное с сигналом SIGINT действие по умолчанию – завершит работу. Интересно отметить, что сигнал SIGINT посылается и процессу оболочки, тоже связанному с терминалом. Тем не менее процесс оболочки разумно игнорирует этот сигнал, поскольку он должен продолжать работу для интерпретации последующих команд. Как будет рассмотрено далее, пользовательские

программы также могут выбирать, нужно ли им перехватывать сигнал SIGINT, то есть выполнять специальную процедуру реакции на поступление сигнала, или полагаться на действие по умолчанию для данного сигнала.

Сигналы также используются ядром для обработки определенных типов критических ошибок. Например, предположим, что файл программы поврежден и содержит недопустимые машинные инструкции. При выполнении этой программы процессом, ядро определит попытку выполнения недопустимой инструкции и pošлет процессу сигнал SIGILL (здесь ILL является сокращением от *illegal*, то есть недопустимый) для завершения его работы. Получившийся диалог может выглядеть примерно так:

```
$ badprog
illegal instruction - core dumped
```

Смысл термина *core dumped* (сброс образа памяти) будет объяснен ниже.

Сигналы могут посылаться не только от ядра к процессу, но и между процессами. Проще всего показать это на примере команды *kill*. Предположим, например, что программист запускает в фоновом режиме длительную команду

```
$ cc verybigprog.c &
[1] 1098
```

а затем решает завершить ее работу. Тогда, чтобы послать процессу сигнал SIGTERM, можно использовать команду *kill*. Так же, как и сигнал SIGINT, сигнал SIGTERM завершит процесс, если в процессе не переопределена стандартная реакция на этот сигнал. В качестве аргумента команды *kill* должен быть задан идентификатор процесса:

```
$ kill 1098
Terminated
```

Сигналы обеспечивают простой метод программного прерывания работы процессов UNIX. Образно можно сравнить его с похлопыванием по плечу, отвлекающим от работы. Из-за своей природы сигналы обычно используются для обработки исключительных ситуаций, а не для обмена данными между процессами.

Процесс может выполнять три действия с сигналами, а именно:

- изменять свою реакцию на поступление определенного сигнала (изменять обработку сигналов);
- блокировать сигналы (то есть откладывать их обработку) на время выполнения определенных критических участков кода;
- посылать сигналы другим процессам.

Каждое из этих действий будет рассмотрено далее в этой главе.

### 6.1.1. Имена сигналов

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее каждому типу сигналов присвоено мнемоническое имя (например, SIGINT), которое указывает, для чего обычно используется сигнал этого



типа. Имена сигналов определены в стандартном заголовочном файле `<signal.h>` при помощи директивы препроцессора `#define`. Как и следовало ожидать, эти имена соответствуют небольшим положительным целым числам. Например, сигнал `SIGINT` обычно определяется так:

```
#define SIGINT 2 /* прерывание (rubout) */
```

Большинство типов сигналов UNIX предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу. Ниже приведен описанный в спецификации XSI полный список стандартных сигналов и их значение. Для удобства список сигналов отсортирован в алфавитном порядке. При первом чтении этот список может быть пропущен.

- ❑ `SIGABRT` – *сигнал прерывания процесса* (process abort signal). Посылается процессу при вызове им функции `abort`. В результате сигнала `SIGABRT` произойдет то, что спецификация XSI описывает как *аварийное завершение* (abnormal termination), *авост*. Следствием этого в реализациях UNIX является *сброс образа памяти* (core dump, иногда переводится как «дамп памяти») с выводом сообщения `Quit - core dumped`. Образ памяти процесса сохраняется в файле на диске для изучения с помощью отладчика;
- ❑ `SIGALRM` – *сигнал таймера* (alarm clock). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова `alarm` (или установки значения первого параметра в более редко применяющемся вызове `setitimer` равным `ITIMER_REAL`). Вызов `alarm` будет описан в разделе 6.4.2. При необходимости больше узнать о вызове `setitimer` следует обратиться к справочному руководству системы;
- ❑ `SIGBUS` – *сигнал ошибки на шине* (bus error). Этот сигнал посылается при возникновении некоторой аппаратной ошибки. Смысл ошибки на шине определяется конкретной реализацией (обычно он генерируется при попытке обращения к допустимому виртуальному адресу, для которого нет физической страницы). Данный сигнал, так же как и сигнал `SIGABRT`, вызывает аварийное завершение;
- ❑ `SIGCHLD` – *сигнал останова или завершения дочернего процесса* (child process terminated or stopped). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему сигнал `SIGCHLD`. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;
- ❑ `SIGCONT` – *продолжение работы остановленного процесса* (continue executing if stopped). Этот сигнал управления процессом, который продолжит выполнение процесса, если он был остановлен; в противном случае процесс будет игнорировать этот сигнал. Это сигнал обратный сигналу `SIGSTOP`;
- ❑ `SIGHUP` – *сигнал освобождения линии* (hangup signal). Посылается ядром всем процессам, подключенным к *управляющему терминалу* (control terminal) при

отключении терминала. (Обычно управляющий терминал группы процесса является терминалом пользователя, хотя это и не всегда так. Это понятие изучается более подробно в главе 9.) Он также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом. Это гарантирует, что если не были предприняты специальные меры, то при выходе пользователя из системы завершаются все фоновые процессы, запущенные им (подробно об этом написано в разделе 5.10);

- SIGILL – *недопустимая команда процессора* (illegal instruction). Посылается операционной системой, если процесс пытается выполнить недопустимую машинную команду. Иногда этот сигнал может возникнуть из-за того, что программа каким-либо образом повредила свой код, хотя это и маловероятно. Более вероятной представляется попытка выполнения вещественной операции, не поддерживаемой оборудованием. В результате сигнала SIGILL происходит аварийное завершение программы;
- SIGINT – *сигнал прерывания программы* (interrupt). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;
- SIGKILL – *сигнал уничтожения процесса* (kill). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса. Иногда он также посылается системой (например, при завершении работы системы). Сигнал SIGKILL – один из двух сигналов, которые не могут игнорироваться или перехватываться (то есть обрабатываться при помощи определенной пользователем процедуры);
- SIGPIPE – *сигнал о попытке записи в канал или сокет, для которых принимающий процесс уже завершил работу* (write on a pipe or socket when recipient is terminated). Программные каналы и сокеты являются другими средствами межпроцессного взаимодействия, которые обсудим в следующих главах. Там же будет рассмотрен и сигнал SIGPIPE;
- SIGPOLL – *сигнал о возникновении одного из опрашиваемых событий* (pollable event). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода. Тем не менее более удобный способ организации слежения за состояниями некоторого множества открытых файловых дескрипторов заключается в использовании системного вызова `select`, который подробно описан в главе 7;
- SIGPROF – *сигнал профилирующего таймера* (profiling time expired). Как было уже упомянуто для сигнала SIGALARM, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Сигнал SIGPROF генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством

профилирования программы. (Таймер устанавливается заданием первого параметра функции `setitimer` равным `ITIMER_PROF`);

- ❑ **SIGQUIT** – *сигнал о выходе (quit)*. Очень похожий на сигнал **SIGINT**, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. Значение клавиши выхода по умолчанию соответствует символу ASCII `FS` или **Ctrl-\**. В отличие от **SIGINT**, этот сигнал приводит к аварийному завершению и сбросу образа памяти;
- ❑ **SIGSEGV** – *обращение к некорректному адресу памяти (invalid memory reference)*. Сокращение **SEGV** в названии сигнала означает *нарушение границ сегментов памяти (segmentation violation)*. Сигнал генерируется, если процесс пытается обратиться к неверному адресу памяти. Получение сигнала **SIGSEGV** приводит к аварийному завершению процесса;
- ❑ **SIGSTOP** – *сигнал останова (stop executing)*. Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал **SIGKILL**, нельзя проигнорировать или перехватить;
- ❑ **SIGSYS** – *некорректный системный вызов (invalid system call)*. Посылается ядром, если процесс пытается выполнить некорректный системный вызов. Это еще один сигнал, приводящий к аварийному завершению;
- ❑ **SIGTERM** – *программный сигнал завершения (software termination signal)*. По соглашению, используется для завершения процесса (как и следует из его названия). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведение порядка», прежде чем посылать ему сигнал **SIGKILL**. Команда `kill` по умолчанию посылает именно этот сигнал;
- ❑ **SIGTRAP** – *сигнал трассировочного прерывания (trace trap)*. Это особый сигнал, который в сочетании с системным вызовом `ptrace` используется отладчиками, такими как `sdb`, `adb`, и `gdb`. Поскольку он предназначен для отладки, его рассмотрение в рамках данной книги не требуется. По умолчанию сигнал **SIGTRAP** приводит к аварийному завершению;
- ❑ **SIGTSTP** – *терминальный сигнал остановки (terminal stop signal)*. Этот сигнал формируется при нажатии специальной клавиши останова (обычно **Ctrl+Z**). Сигнал **SIGTSTP** аналогичен сигналу **SIGSTOP**, но его можно перехватить или игнорировать;
- ❑ **SIGTTIN** – *сигнал о попытке ввода с терминала фоновым процессом (background process attempting read)*. Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается сигнал **SIGTTIN**. Действие сигнала по умолчанию – остановка процесса;
- ❑ **SIGTTOU** – *сигнал о попытке вывода на терминал фоновым процессом (background process attempting write)*. Аналогичен сигналу **SIGTTIN**, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. И снова действие по умолчанию – остановка процесса;
- ❑ **SIGURG** – *сигнал о поступлении в буфер сокета срочных данных (high bandwidth data is available at a socket)*. Этот сигнал сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;

- SIGUSR1 и SIGUSR2 – *пользовательские сигналы* (user defined signals 1 and 2). Так же, как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя;
- SIGVTALRM – *сигнал виртуального таймера* (virtual timer expired). Как уже упоминалось для сигналов SIGALRM и SIGPROF, каждый процесс может иметь не менее трех таймеров. Последний из этих таймеров можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме. (Таймер устанавливается заданием первого параметра функции `setitimer` равным `ITIMER_VIRTUAL`);
- SIGXCPU – *сигнал о превышении лимита процессорного времени* (CPU time limit exceeded). Этот сигнал посылается процессу, если суммарное процессорное время, занятое его работой, превысило установленный предел. Действие по умолчанию – аварийное завершение;
- SIGXFSZ – *сигнал о превышении предела на размер файла* (file size limit exceeded). Этот сигнал генерируется, если процесс превысит максимально допустимый размер файла. Действие по умолчанию – аварийное завершение.

Могут встретиться и некоторые другие сигналы, но их наличие зависит от конкретной реализации системы; его не требует спецификация XSI. Большая часть этих сигналов также используется ядром для индикации ошибок, например SIGEMT – *прерывание эмулятора* (emulator trap) часто обозначает отказ оборудования и зависит от конкретной реализации.<sup>1</sup>

### 6.1.2. Нормальное и аварийное завершение

Получение большинства сигналов приводит к *нормальному завершению* (normal termination) процесса. Действие сигнала при этом похоже на неожиданный вызов процессом функции `_exit`. Статус завершения, возвращаемый при этом родительскому процессу, сообщит о причине завершения дочернего процесса. В файле `<sys/wait.h>` определены макросы, которые позволяют родительскому процессу определить причину завершения дочернего процесса (получение сигнала) и, собственно, значение сигнала. Следующий фрагмент программы демонстрирует родительский процесс, проверяющий причину завершения дочернего процесса и выводящий соответствующее сообщение:

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

.
.
if((pid=wait(&status)) == -1)
{
    perror("ошибка вызова wait");
```

<sup>1</sup> Некоторые из упомянутых сигналов могут отсутствовать в используемой системе, тогда компилятор сообщит о неизвестном мнемоническом имени. Иногда имя сигнала определено, а данный сигнал отсутствует в системе. В ряде случаев требуется определить наличие определенного сигнала на стадии выполнения программы. В этих ситуациях можно воспользоваться советом, приведенным в информативной части стандарта POSIX 1003.1: наличие поддержки сигнала сообщает вызов функции `sigaction()` с аргументами `act` и `oact`, имеющими значения `NULL`. – *Прим. науч. ред.*

```
    exit(1);
}

/* Проверка нормального завершения дочернего процесса */
if(WIFEXITED(status))
{
    exit_status = WEXITSTATUS(status);
    printf("Статус завершения %d был %d\n", pid, exit_status);
}

/* Проверка, получил ли дочерний процесс сигнал */
if(WIFSIGNALED(status))
{
    sig_no = WTERMSIG(status);
    printf("Сигнал номер %d завершил процесс %d\n", sig_no, pid);
}

/* Проверка остановки дочернего процесса */
if(WIFSTOPPED(status))
{
    sig_no = WSTOPSIG(status);
    printf("Сигнал номер %d остановил процесс %d\n", sig_no, pid);
}
```

Как уже упоминалось, сигналы SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ и SIGFPE приводят к аварийному завершению и обычно сопровождаются сбросом образа памяти на диск. Образ памяти процесса записывается в файл с именем `core` в текущем рабочем каталоге процесса (термин `core`, или *сердечник*, напоминает о временах, когда оперативная память состояла из матриц ферритовых сердечников). Файл `core` будет содержать значения всех переменных программы, регистров процессора и необходимую управляющую информацию ядра на момент завершения программы. Статус завершения процесса после аварийного завершения будет тем же, каким он был бы в случае нормального завершения из-за этого сигнала, только в нем будет дополнительно выставлен седьмой бит младшего байта. В большинстве современных систем UNIX определен макрос `WCOREDUMP`, который возвращает истинное или ложное значение в зависимости от установки этого бита в своем аргументе. Тем не менее следует учесть, что макрос `WCOREDUMP` не определен спецификацией XSI. С применением этого макроса предыдущий пример можно переписать так:

```
/* Проверка, получил ли дочерний процесс сигнал */
if(WIFSIGNALED(status))
{
    sig_no = WTERMSIG(status) ;
    printf("Сигнал номер %d завершил процесс %d\n", sig_no, pid);
    if(WCOREDUMP(status))
        printf("... создан файл дампа памяти\n");
}
```

Формат файла `core` известен отладчикам UNIX, и этот файл можно использовать для изучения состояния процесса в момент сброса образа памяти. Этим можно воспользоваться для определения точки, в которой возникает проблема.

Стоит также упомянуть функцию `abort`, которая не имеет аргументов:

```
abort();
```

Эта функция посылает вызвавшему ее процессу сигнал `SIGABRT`, вызывая его аварийное завершение, то есть сброс образа памяти. Процедура `abort` полезна в качестве средства отладки, так как позволяет процессу записать свое текущее состояние, если что-то происходит не так. Она также иллюстрирует тот факт, что процесс может послать сигнал самому себе.

## 6.2. Обработка сигналов

При получении сигнала процесс может выполнить одно из трех действий:

- выполнить действие по умолчанию. Обычно действие по умолчанию заключается в прекращении выполнения процесса. Для некоторых сигналов, например, для сигналов `SIGUSR1` и `SIGUSR2`, действие по умолчанию заключается в игнорировании сигнала.<sup>1</sup> Для других сигналов, например, для сигнала `SIGSTOP`, действие по умолчанию заключается в остановке процесса;
- игнорировать сигнал и продолжать выполнение. В больших программах неожиданно возникающие сигналы могут привести к проблемам. Например, нет смысла позволять программе останавливаться в результате случайного нажатия на клавишу прерывания, в то время как она производит обновление важной базы данных;
- выполнить определенное пользователем действие. Программист может захотеть выполнить при выходе из программы операции по «наведению порядка» (такие как удаление рабочих файлов), что бы ни являлось причиной этого выхода.

В старых версиях UNIX обработка сигналов была относительно простой. Здесь будут изучены современные процедуры управления механизмом сигналов, и хотя эти процедуры несколько сложнее, их использование дает вполне надежный результат, в отличие от устаревших методов обработки сигналов. Прежде чем перейти к примерам, следует сделать несколько пояснений. Начнем с определения *набора сигналов* (`signal set`).

### 6.2.1. Наборы сигналов

Наборы сигналов являются одним из основных параметров, передаваемых работающим с сигналами системным вызовам. Они просто задают список сигналов, которые необходимо передать системному вызову.

Наборы сигналов определяются при помощи типа `sigset_t`, который определен в заголовочном файле `<signal.h>`. Размер типа задан так, чтобы в нем мог

---

<sup>1</sup> Спецификация SUSV2 приводит для этих сигналов нормальное завершение в качестве действия по умолчанию; лучшими примерами являются сигналы `SIGCHLD` и `SIGURG`. – *Прим. науч. ред.*

поместиться весь набор определенных в системе сигналов. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур `sigemptyset` и `sigfillset` соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур `sigaddset` и `sigdelset`, соответственно добавляющих и удаляющих указанные вами сигналы.

### Описание

```
#include <signal.h>

/* Инициализация */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set) ;

/* Добавление и удаление сигналов */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

Процедуры `sigemptyset` и `sigfillset` имеют единственный параметр – указатель на переменную типа `sigset_t`. Вызов `sigemptyset` инициализирует набор `set`, исключив из него все сигналы. И наоборот, вызов `sigfillset` инициализирует набор, на который указывает `set`, включив в него все сигналы. Приложения должны вызывать `sigemptyset` или `sigfillset` хотя бы один раз для каждой переменной типа `sigset_t`.

Процедуры `sigaddset` и `sigdelset` принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, `signo`, может быть символическим именем константы, таким как `SIGINT`, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

В следующем примере создадим два набора сигналов. Первый образуется из пустого набора добавлением сигналов `SIGINT` и `SIGQUIT`. Второй набор изначально заполнен, и из него удаляется сигнал `SIGCHLD`.

```
#include <signal.h>

sigset_t mask1, mask2;
.
.
.

/* Создать пустой набор сигналов */
sigemptyset (&mask1);

/* Добавить сигналы */
sigaddset (&mask1, SIGINT);
sigaddset (&mask1, SIGQUIT);

/* Создать полный набор сигналов */
sigfillset (&mask2);
```

```
/* Удалить сигнал */
sigdelset(&mask2, SIGCHLD);
.
.
.
```

### 6.2.2. Задание обработчика сигналов: вызов *sigaction*

После определения списка сигналов можно задать определенный метод обработки сигнала при помощи процедуры *sigaction*.

#### Описание

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
              struct sigaction *oact);
```

Как вскоре увидим, структура *sigaction*, в свою очередь, также содержит набор сигналов. Первый параметр *signo* задает отдельный сигнал, для которого нужно определить действие. Чтобы это действие выполнялось, процедура *sigaction* должна быть вызвана до получения сигнала типа *signo*. Значением переменной *signo* может быть любое из ранее определенных имен сигналов, за исключением *SIGSTOP* и *SIGKILL*, которые предназначены только для остановки или завершения процесса и не могут обрабатываться по-другому.

Второй параметр, *act*, определяет обработчика сигнала *signo*. Третий параметр, *oact*, если не равен *NULL*, указывает на структуру, куда будет помещено описание старого метода обработки сигнала. Рассмотрим структуру *sigaction*, определенную в файле *<signal.h>*:

```
struct sigaction{
    void (*sa_handler)(int); /* Функция обработчика */
    sigset_t sa_mask,        /* Сигналы, которые блокируются
                               во время обработки сигнала */
    int      sa_flags;        /* Флаги, влияющие
                               на поведение сигнала */
    void (*sa_sigaction)(int, siginfo_t *, void *);
                               /* Указатель на обработчик сигнала */
};
```

Эта структура кажется сложной, но давайте рассмотрим ее поля по отдельности. Первое поле, *sa\_handler*, задает обработчик сигнала *signo*. Это поле может иметь три вида значений:

- *SIG\_DFL* – константа, сообщающая, что нужно восстановить обработку сигнала по умолчанию (для большинства сигналов это завершение процесса);
- *SIG\_IGN* – константа, означающая, что нужно *игнорировать данный сигнал* (ignore the signal). Не может использоваться для сигналов *SIGSTOP* и *SIGKILL*;
- адрес функции, принимающей аргумент типа *int*. Если функция объявлена в тексте программы до заполнения *sigaction*, то полю *sa\_handler* можно просто присвоить имя функции. Компилятор поймет, что имелся в виду ее



адрес. Эта функция будет выполняться при получении сигнала `signo`, а само значение `signo` будет передано в качестве аргумента вызываемой функции. Управление будет передано функции, как только процесс получит сигнал, какой бы участок программы при этом не выполнялся. После возврата из функции управление будет снова передано процессу и продолжится с точки, в которой выполнение процесса было прервано. Этот механизм станет понятен из следующего примера.

Второе поле, `sa_mask`, демонстрирует первое практическое использование набора сигналов. Сигналы, заданные в поле `sa_mask`, будут блокироваться во время выполнения функции, заданной полем `sa_handler`. Это не означает, что эти сигналы будут игнорироваться, просто их обработка будет отложена до завершения функции. При входе в функцию перехваченный сигнал также будет неявно добавлен к текущей маске сигналов. Все это делает механизм сигналов более надежным механизмом межпроцессного взаимодействия, чем он был в ранних версиях системы UNIX.<sup>1</sup>

Поле `sa_flags` может использоваться для изменения характера реакции на сигнал `signo`. Например, после возврата из обработчика можно вернуть обработчик по умолчанию `SIG_DFL`, установив значение поля `sa_flags` равным `SA_RESETHAND`. Если же значение поля `sa_flags` установлено равным `SA_SIGINFO`, то обработчику сигнала будет передаваться дополнительная информация. В этом случае поле `sa_handler` является избыточным, и используется последнее поле `sa_sigaction`. Передаваемая этому обработчику структура `siginfo_t` содержит дополнительную информацию о сигнале, такую как его номер, идентификатор пославшего сигнал процесса и действующий идентификатор пользователя этого процесса. Для того чтобы программа была действительно переносимой, спецификация XSI требует, чтобы процесс использовал либо поле `sa_handler`, либо поле `sa_sigaction`, но не оба эти поля сразу.

Все это достаточно трудно усвоить, поэтому рассмотрим несколько примеров. В действительности все намного проще, чем кажется.

### **Пример 1: перехват сигнала SIGINT**

Этот пример демонстрирует, как можно перехватить сигнал, а также проясняет лежащий в его основе механизм сигналов. Программа `sigex` просто связывает с сигналом `SIGINT` функцию `catchint`, а затем выполняет набор операторов `sleep` и `printf`. В данном примере определена структура `act` типа `sigaction` как `static`. Поэтому при инициализации структуры все поля, и в частности поле `sa_flags` обнуляются.

```
/* Программа sigex — демонстрирует работу sigaction */
#include <signal.h>

main()
```

---

<sup>1</sup> Тем не менее при написании сложных систем следует знать некоторые дополнительные детали механизма доставки и обработки сигналов (см. стандарт POSIX 1003.1, спецификацию SUSV2, и руководство используемой программистом системы). — *Прим. науч. ред.*

```

{
    static struct sigaction act;

/* Определение процедуры обработчика сигнала catchint */
void catchint (int);

/* Задание действия при получении сигнала SIGINT */
act.sa_handler = catchint;

/* Создать маску, включающую все сигналы */
sigfillset(&(act.sa_mask));

/* До вызова процедуры sigaction , сигнал SIGINT
 * приводил к завершению процесса (действие по умолчанию).
 */

sigaction(SIGINT, &act, NULL);
/* При получении сигнала SIGINT управление
 * будет передаваться процедуре catchint
 */
printf("Вызов sleep номер 1\n");
sleep(1);
printf("Вызов sleep номер 2\n");
sleep(1);
printf("Вызов sleep номер 3\n");
sleep(1);
printf("Вызов sleep номер 4\n");
sleep(1);
printf("Выход \n");
exit(0);
}

/* Простая функция для обработки сигнала SIGINT */
void catchint(int signo)
{
    printf("\nСигнал CATCHINT: signo=%d\n", signo);
    printf("Сигнал CATCHINT: возврат\n\n");
}

```

Сеанс обычного запуска `sigex` будет выглядеть так:

```

$ sigex
Вызов sleep номер 1
Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход

```

Пользователь может прервать выполнение программы `sigex`, нажав клавишу прерывания задания. Если она была нажата до того, как в программе `sigex` была выполнена процедура `sigaction`, то процесс просто завершит работу. Если же нажать на клавишу прерывания после вызова `sigaction`, то управление будет передано функции `catchint`:

```
$ sigex
Вызов sleep номер 1
<прерывание> (пользователь нажимает на клавишу прерывания)

Сигнал CATCHINT: signo = 2
Сигнал CATCHINT: возврат

Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход
```

Обратите внимание на то, как передается управление из тела программы процедуре `catchint`. После завершения процедуры `catchint`, управление продолжится с точки, в которой программа была прервана. Можно попробовать прервать программу `sigex` и в другом месте:

```
$ sigex
Вызов sleep номер 1
Вызов sleep номер 2
<прерывание> (пользователь нажимает на клавишу прерывания)

Сигнал CATCHINT: signo = 2
Сигнал CATCHINT: возврат

Вызов sleep номер 3
Вызов sleep номер 4
Выход
```

### **Пример 2: игнорирование сигнала SIGINT**

Для того чтобы процесс игнорировал сигнал прерывания `SIGINT`, все, что нужно для этого сделать – это заменить следующую строку в программе:

```
act.sa_handler = catchint;
на
act.sa_handler = SIG_IGN;
```

После выполнения этого оператора нажатие клавиши прерывания будет безрезультатным. Снова разрешить прерывание можно так:

```
act.sa_handler = SIG_DFL;
sigaction (SIGINT, &act, NULL);
```

Можно игнорировать сразу несколько сигналов, например:

```
act.sa_handler = SIG_IGN;
sigaction(SIGINT, &act, NULL);
sigaction(SIGQUIT, &act, NULL);
```

При этом игнорируются оба сигнала `SIGINT` и `SIGQUIT`. Это может быть полезно в программах, которые не должны прерываться с клавиатуры.

Некоторые командные интерпретаторы используют этот подход, чтобы гарантировать, что фоновые процессы не останавливаются при нажатии пользователем клавиши прерывания. Это возможно вследствие того, что игнорируемые процессом

сигналы будут игнорироваться и после вызова `exes`. Поэтому командный интерпретатор может вызвать `sigaction` для игнорирования сигналов `SIGQUIT` и `SIGINT`, а затем запустить новую программу при помощи вызова `exes`.

### **Пример 3: восстановление прежнего действия**

Как упоминалось выше, в структуре `sigaction` может быть заполнен третий параметр `oact`. Это позволяет сохранять и восстанавливать прежнее состояние обработчика сигнала, как показано в следующем примере:

```
#include <signal.h>

static struct sigaction act,oact;

/* Сохранить старый обработчик сигнала SIGTERM */
sigaction(SIGTERM, NULL, &oact);

/* Определить новый обработчик сигнала SIGTERM */
act.sa_handler = SIG_IGN;
sigaction(SIGTERM, &act, NULL);

/* Выполнить какие-либо действия */
/* Восстановить старый обработчик */
sigaction(SIGTERM, &oact, NULL);
```

### **Пример 4: аккуратный выход**

Предположим, что программа использует временный рабочий файл. Следующая простая процедура удаляет файл:

```
/* Аккуратный выход из программы */
#include <stdio.h>
#include <stdlib.h>

void g_exit(int s)
{
    unlink("tempfile");
    fprintf(stderr, "Прерывание — выход из программы\n");
    exit(1);
}
```

Можно связать эту процедуру с определенным сигналом:

```
extern void g_exit(int);
.
.
.
static struct sigaction act;
act.sa_handler = g_exit;
sigaction(SIGINT, &act, NULL);
```

Если после этого вызова пользователь нажмет клавишу прерывания, то управление будет автоматически передано процедуре `g_exit`. Можно дополнить процедуру `g_exit` другими необходимыми для завершения операциями.

### 6.2.3. Сигналы и системные вызовы

В большинстве случаев, если процессу посылается сигнал во время выполнения им системного вызова, то обработка сигнала откладывается до завершения вызова. Но некоторые системные вызовы ведут себя по-другому, и их выполнение можно прервать при помощи сигнала. Это относится к вызовам ввода/вывода (`read`, `write`, `open`, и т.д.), вызовам `wait` или `pause` (который мы обсудим в свое время). Во всех случаях, если процесс перехватывает вызов, то прерванный системный вызов возвращает значение `-1` и помещает в переменную `errno` значение `EINTR`. Такие ситуации можно обрабатывать при помощи следующего кода:

```
if( write(tfd, buf, size) < 0)
{
    if( errno == EINTR )
    {
        warn("Вызов write прерван");
        .
        .
        .
    }
}
```

В этом случае, если программа хочет вернуться к системному вызову `write`, то она должна использовать цикл и оператор `continue`. Но процедура `sigaction` позволяет автоматически повторять прерванный таким образом системный вызов. Это достигается установкой значения `SA_RESTART` в поле `sa_flags` структуры `struct sigaction`. Если установлен этот флаг, то системный вызов будет выполнен снова, и значение переменной `errno` не будет установлено.

Важно отметить, что сигналы UNIX обычно не могут накапливаться. Другими словами, в любой момент времени только один сигнал каждого типа может ожидать обработки данным процессом, хотя несколько сигналов разных типов могут ожидать обработки одновременно. Фактически то, что сигналы не могут накапливаться, означает, что они не могут использоваться в качестве полностью надежного метода межпроцессного взаимодействия, так как процесс не может быть уверен, что посланный им сигнал не будет «потерян».

---

**Упражнение 6.1.** Измените программу `smallsh` из предыдущей главы так, чтобы она обрабатывала клавиши прерывания и завершения как настоящий командный интерпретатор. Выполнение фоновых процессов не должно прерываться сигналами `SIGINT` и `SIGQUIT`. Некоторые командные интерпретаторы (а именно C-shell и Korn shell) помещают фоновые процессы в другую группу процессов. В чем преимущества и недостатки этого подхода? (В последних версиях стандарта POSIX введено накопление сигналов, но в качестве необязательного расширения.)

---

### 6.2.4. Процедуры *sigsetjmp* и *siglongjmp*

Иногда при получении сигнала необходимо вернуться на предыдущую позицию в программе. Например, может потребоваться, чтобы пользователь мог вернуться в основное меню программы при нажатии клавиши прерывания. Это можно выполнить, используя две специальные функции *sigsetjmp* и *siglongjmp*. (Существуют альтернативные функции *setjmp* и *longjmp*, но их нельзя использовать совместно с обработкой сигналов.) Процедура *sigsetjmp* «сохраняет» текущие значения счетчика команд, позиции стека, регистров процессора и маски сигналов, а процедура *siglongjmp* передает управление назад в сохраненное таким образом положение. В этом смысле процедура *siglongjmp* аналогична оператору *goto*. Важно понимать, что возврат из процедуры *siglongjmp* не происходит, так как стек возвращается в сохраненное состояние. Как будет показано ниже, при этом происходит выход из соответствующего вызова *sigsetjmp*.

#### Описание

```
#include <setjmp.h>

/* Сохранить текущее положение в программе */
int sigsetjmp(sigjmp_buf env, int savemask);

/* Вернуться в сохраненную позицию */
void siglongjmp(sigjmp_buf env, int val);
```

Текущее состояние программы сохраняется в объекте типа *sigjmp\_buf*, определенном в стандартном заголовочном файле *<setjmp.h>*. Если во время вызова *sigsetjmp* значение аргумента *savemask* не равно нулю (равно *TRUE*), то вызов *sigsetjmp* сохранит помимо основного контекста программы также текущую маску сигналов (маску блокированных сигналов и действия, связанные со всеми сигналами). Возвращаемое функцией *sigsetjmp* значение является важным: если в точку *sigsetjmp* управление переходит из функции *siglongjmp*, то она возвращает ненулевое значение – аргумент *val* вызова *siglongjmp*. Если же функция *sigsetjmp* вызывается в обычном порядке исполнения программы, то она возвращает значение 0.

Следующий пример демонстрирует технику использования этих функций:

```
/* Пример использования функций sigsetjmp и siglongjmp */

#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>

sigjmp_buf position;

main()
{
    static struct sigaction act;
    void goback(void);
```

```
.  
.   
.   
  
/* Сохранить текущее положение */  
if(sigsetjmp(position, 1) == 0)  
{  
    act.sa_handler = goback;  
    sigaction(SIGINT, &act, NULL);  
}  
  
domenu();  
.  
.  
.  
}  
  
void goback(void)  
{  
    fprintf(stderr, "\nПрерывание\n") ;  
  
    /* Вернуться в сохраненную позицию */  
    siglongjmp(position, 1);  
}
```

Если пользователь нажимает на клавишу прерывания задания после вызова `sigaction`, то управление передается в точку, положение которой было записано при помощи функции `sigsetjmp`. Поэтому выполнение программы продолжается, как если бы только что завершился соответствующий вызов `sigsetjmp`. В этом случае возвращаемое функцией `sigsetjmp` значение будет равно второму параметру процедуры `siglongjmp`.

## 6.3. Блокирование сигналов

Если программа выполняет важную задачу, такую как обновление базы данных, то может понадобиться ее защита от прерывания на время выполнения таких критических действий. Как уже упоминалось, вместо игнорирования поступающих сигналов процесс может блокировать сигналы, это будет означать, что их выполнение будет отложено до тех пор, пока процесс не завершит выполнение критического участка.

Блокировать определенные сигналы в процессе позволяет системный вызов `sigprocmask`, определенный следующим образом:

### Описание

```
#include <signal.h>  
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Параметр `how` сообщает вызову `sigpromask`, какое действие он должен выполнять. Например, этот параметр может иметь значение `SIG_MASK`, указывающее,

что с этого момента будут блокироваться сигналы, заданные во втором параметре `set`, то есть будет произведена установка маски блокирования сигналов. Третий параметр просто заполняется текущей маской блокируемых сигналов – если не нужно ее знать, просто присвойте этому параметру значение `NULL`. Поясним это на примере:

```
sigset_t set1;

.
.
.
/* Выбрать полный набор сигналов */
sigfillset(&set1);

/* Установить блокировку */
sigprocmask(SIG_SETMASK, &set1, NULL);

/* Критический участок кода .. */

/* Отменить блокировку сигналов */
sigprocmask(SIG_UNBLOCK, &set1, NULL);
```

Обратите внимание на использование для отмены блокирования сигналов параметра `SIG_UNBLOCK`. Заметим, что если использовать в качестве первого параметра `SIG_BLOCK` вместо `SIG_SETMASK`, то это приведет к *добавлению* заданных в переменной `set` сигналов к текущему набору сигналов.

Следующий более сложный пример показывает, как сначала выполняется блокирование всех сигналов во время выполнения особенно важного участка программы, а затем, при выполнении менее критического участка, блокируются только сигналы `SIGINT` и `SIGQUIT`.

```
/* Блокировка сигналов – демонстрирует вызов sigprocmask */
#include <signal.h>

main ()
{
    sigset_t set1, set2;

    /* Создать полный набор сигналов */
    sigfillset(&set1);

    /* Создать набор сигналов, не включающий
     * сигналы SIGINT и SIGQUIT.
     */
    sigfillset(&set2);
    sigdelset(&set2, SIGINT);
    sigdelset(&set2, SIGQUIT);

    /* Некритический участок кода ... */

    /* Установить блокирование всех сигналов */
    sigprocmask(SIG_SETMASK, &set1, NULL);

    /* Исключительно критический участок кода ... */
```



```
/* Блокирование двух сигналов */
sigprocmask(SIG_UNBLOCK, &set2, NULL);

/* Менее критический участок кода ... */

/* Отменить блокирование для всех сигналов */
sigprocmask(SIG_UNBLOCK, &set1, NULL);
}
```

---

**Упражнение 6.2.** *Перепишите процедуру `g_exit` в примере 4 из раздела 6.2.2 так, чтобы во время ее выполнения игнорировались сигналы `SIGINT` и `SIGQUIT`.*

---

## 6.4. Посылка сигналов

### 6.4.1. Посылка сигналов другим процессам: вызов *kill*

Процесс вызывает процедуру `sigaction` для установки реакции на поступление сигнала. Обратную операцию, посылку сигнала, выполняет системный вызов `kill`, описанный следующим образом

#### Описание

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

int kill(pid_t pid, int sig);
```

Первый параметр `pid` определяет процесс или процессы, которым посылается сигнал `sig`. Обычно `pid` является положительным числом, и в этом случае он рассматривается как идентификатор процесса. Поэтому следующий оператор

```
kill(7421, SIGTERM);
```

означает «*послать сигнал SIGTERM процессу с идентификатором 7421*». Так как процесс, посылающий сигнал `kill`, должен знать идентификатор процесса, которому предназначен сигнал, то вызов `kill` чаще всего используется для обмена между тесно связанными процессами, например, родительским и дочерним. Заметим, что процесс может послать сигнал самому себе.

Существуют некоторые ограничения, связанные с правами доступа. Чтобы можно было послать сигнал процессу, действующий или истинный идентификатор пользователя посылающего процесса должен совпадать с действующим или истинным идентификатором пользователя процесса, которому сигнал адресован. Процессы суперпользователя, как обычно, могут посылать сигналы любым другим процессам. Если непривилегированный пользователь пытается послать сигнал процессу, который принадлежит другому пользователю, то вызов `kill` завершится неудачей, вернет значение `-1` и поместит в переменную `errno` значение `EPERM`. (Другие возможные значения ошибок в переменной `errno` после неудачного вызова `kill` — это значение `ESRCH`, указывающее, что процесс с заданным идентификатором не существует, и `EINVAL`, если `sig` содержит некорректный номер сигнала.)

Параметр `pid` вызова `kill` может также принимать определенные значения, которые имеют особый смысл:

- если параметр `pid` равен нулю, то сигнал будет послан всем процессам, принадлежащим к той же группе, что и процесс, пославший сигнал, в том числе и самому процессу;
- если параметр `pid` равен `-1`, и действующий идентификатор пользователя не является идентификатором суперпользователя, то сигнал посылается всем процессам, истинный идентификатор пользователя которых равен действующему идентификатору пользователя пославшего сигнал процесса, снова включая и сам процесс, пославший сигнал;
- если параметр `pid` равен `-1` и действующий идентификатор пользователя является идентификатором суперпользователя, то сигнал посылается всем процессам, кроме определенных системных процессов (последнее исключение относится ко всем попыткам послать сигнал группе процессов, но наиболее важно это в данном случае);
- и, наконец, если параметр `pid` меньше нуля, но не равен `-1`, то сигнал посылается всем процессам, идентификатор группы которых равен модулю `pid`, включая пославший сигнал процесс, если для него также выполняется это условие.

Следующий пример – программа `synchro` создает два процесса, которые будут поочередно печатать сообщения на стандартный вывод. Они синхронизируют свою работу, посылая друг другу сигнал `SIGUSR1` при помощи вызова `kill`.

```
/* Программа synchro – пример использования вызова kill */

#include <unistd.h>
#include <signal.h>

int ntimes = 0;

main()
{
    pid_t pid, ppid;
    void p_action(int), c_action(int);
    static struct sigaction pact, cact;

    /* Задаем обработчик сигнала SIGUSR1 в родительском процессе */
    pact.sa_handler = p_action;
    sigaction(SIGUSR1, &pact, NULL);

    switch (pid = fork()){
        case -1:      /* Ошибка */
            perror("synchro");
            exit(1);

        case 0:      /* Дочерний процесс */

            /* Задаем обработчик в дочернем процессе */
            cact.sa_handler = c_action;
```

```
sigaction(SIGUSR1, &act, NULL);

/* Получаем идентификатор родительского процесса */
ppid = getppid();

/* Бесконечный цикл */
for (;;)
{
    sleep(1);
    kill(ppid, SIGUSR1);
    pause();
}

default:      /* Родительский процесс */

/* Бесконечный цикл */
for(;;)
{
    pause();
    sleep(1);
    kill(pid, SIGUSR1);
}
}
}

void p_action(int sig)
{
    printf ("Родительский процесс получил сигнал #%d\n", ++ntimes);
}

void c_action(int sig)
{
    printf("Дочерний процесс получил сигнал #%d\n", ++ntimes);
}
```

Оба процесса выполняют бесконечный цикл, приостанавливая работу до получения сигнала от другого процесса. Они используют для этого системный вызов `pause`, который просто приостанавливает работу до получения сигнала (см. раздел 6.4.3). Затем каждый из процессов выводит сообщение и, в свою очередь, посылает сигнал при помощи вызова `kill`. Дочерний процесс начинает вывод сообщений (обратите внимание на порядок операторов в каждом цикле). Оба процесса завершают работу, когда пользователь нажимает на клавишу прерывания. Диалог с программой может выглядеть примерно так:

```
$ synchro
Родительский процесс получил сигнал #1
Дочерний процесс получил сигнал #1
Родительский процесс получил сигнал #2
Дочерний процесс получил сигнал #2
< прерывание >      (пользователь нажал на клавишу прерывания)
$
```

### 6.4.2. Посылка сигналов самому процессу: вызовы *raise* и *alarm*

Функция *raise* просто посылает сигнал выполняющемуся процессу:

#### Описание

```
#include <signal.h>

int raise(int sig);
```

Вызываемому процессу посылается сигнал, определенный параметром *sig*, и в случае успеха функция *raise* возвращает нулевое значение.

Вызов *alarm* – это простой и полезный вызов, который устанавливает таймер процесса. При срабатывании таймера процессу посылается сигнал.

#### Описание

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

Переменная *secs* задает время в секундах, на которое устанавливается таймер. После истечения заданного интервала времени процессу посылается сигнал *SIGALRM*. Поэтому вызов

```
alarm(60);
```

приводит к посылке сигнала *SIGALRM* через 60 секунд. Обратите внимание, что вызов *alarm* не приостанавливает выполнение процесса, как вызов *sleep*, вместо этого сразу же происходит возврат из вызова *alarm*, и продолжается нормальное выполнение процесса, по крайней мере, до тех пор, пока не будет получен сигнал *SIGALRM*. Установленный таймер будет продолжать отсчет и после вызова *exes*, но вызов *fork* выключает таймер в дочернем процессе.

Выключить таймер можно при помощи вызова *alarm* с нулевым параметром:

```
/* Выключить таймер */
alarm(0);
```

Вызовы *alarm* не накапливаются: другими словами, если вызвать *alarm* дважды, то второй вызов отменит предыдущий. Но при этом возвращаемое вызовом *alarm* значение будет равно времени, оставшемуся до срабатывания предыдущего таймера, и его можно при необходимости записать.

Вызов *alarm* может быть полезен, если нужно ограничить время выполнения какого-либо действия. Основная идея проста: вызывается *alarm*, и процесс начинает выполнение задачи. Если задача выполняется вовремя, то таймер сбрасывается. Если выполнение задачи отнимает слишком много времени, то процесс прерывается при помощи сигнала *SIGTERM* и выполняет корректирующие действия.

Следующая функция *quickreply* использует этот подход для ввода данных от пользователя за заданное время. Она имеет один аргумент, приглашение командной строки, и возвращает указатель на введенную строку, или нулевой указатель, если после пяти попыток ничего не было введено. Обратите внимание, что после каждого напоминания пользователю функция *quickreply* посылает на терминал символ **Ctrl+G**. На большинстве терминалов и эмуляторов терминала это приводит к подаче звукового сигнала.

Функция `quickreply` вызывает процедуру `gets` из *стандартной библиотеки ввода/вывода* (Standard I/O Library). Процедура `gets` помещает очередную строку из стандартного ввода в массив `char`. Она возвращает либо указатель на массив, либо нулевой указатель в случае достижения конца файла или ошибки. Обратите внимание на то, что сигнал `SIGALRM` перехватывается процедурой обработчика прерывания `catch`. Это важно, так как по умолчанию получение сигнала `SIGALRM` приводит к завершению процесса. Процедура `catch` устанавливает флаг `timed_out`. Функция `quickreply` проверяет этот флаг, определяя таким образом, не истекло ли заданное время.

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

#define TIMEOUT    5          /* Время в секундах */
#define MAXTRIES   5          /* Число попыток */
#define LINESIZE   100        /* Длина строки */
#define CTRL_G     '\007'     /* ASCII-символ звукового сигнала */
#define TRUE       1
#define FALSE      0

/* Флаг, определяющий, истекло ли заданное время */
static int timed_out;

/* Переменная, которая будет содержать введенную строку */
static char answer[LINESIZE];

char *quickreply(char *prompt)
{
    void catch(int);
    int ntries;
    static struct sigaction act, oact;

    /* Перехватить сигнал SIGALRM и сохранить старый обработчик */
    act.sa_handler = catch;
    sigaction(SIGALRM, &act, &oact);

    for(ntries=0; ntries<MAXTRIES; ntries++)
    {
        timed_out = FALSE;
        printf("\n%s > ", prompt);

        /* Установить таймер */
        alarm(TIMEOUT);

        /* Получить введенную строку */
        gets(answer);

        /* Выключить таймер */
        alarm(0);

        /* Если флаг timed_out равен TRUE, завершить работу */
        if(!timed_out)
            break;
    }
}
```

```

/* Восстановить старый обработчик */
sigaction(SIGALRM, &oact, NULL);

/* Вернуть соответствующее значение */
return (ntries == MAXTRIES ? ((char *)0) : answer);
}

/* Выполняется при получении сигнала SIGALRM */
void catch(int sig)
{
    /* Установить флаг timed_out */
    timed_out = TRUE;

    /* Подать звуковой сигнал */
    putchar(CTRL_G);
}

```

### 6.4.3. Системный вызов *pause*

ОС UNIX также содержит дополняющий вызов `alarm` системный вызов `pause`, который определен следующим образом:

#### Описание

```

#include <unistd.h>

int pause(void);

```

Вызов `pause` приостанавливает выполнение вызывающего процесса (так что процесс при этом не занимает процессорного времени) до получения любого сигнала, например, сигнала `SIGALRM`. Если сигнал вызывает нормальное завершение процесса или игнорируется процессом, то в результате вызова `pause` будет просто выполнено соответствующее действие (завершение работы или игнорирование сигнала). Если же сигнал перехватывается, то после завершения соответствующего обработчика прерывания вызов `pause` вернет значение `-1` и поместит в переменную `errno` значение `EINTR` (объясните почему).

Следующая программа `tml` (сокращение от «tell me later» – *напомнить позднее*) использует оба вызова `alarm` и `pause` для вывода сообщения в течение заданного числа минут. Она вызывается следующим образом:

```
$ tml число_минут текст_сообщения
```

Например:

```
$ tml 10 время идти домой
```

Перед сообщением выводятся три символа **Ctrl+G** (звуковые сигналы) для привлечения внимания пользователя. Обратите внимание на создание в программе `tml` фонового процесса при помощи вызова `fork`. Фоновый процесс выполняет работу, позволяя пользователю продолжать выполнение других задач.

```

/* Программа для напоминания tml */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

```

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

#define TRUE    1
#define FALSE  0
#define BELLS  "\007\007\007"    /* Звуковой сигнал ASCII */

int alarm_flag = FALSE;

/* Обработчик сигнала SIGALRM */
void setflag(int sig)
{
    alarm_flag = TRUE;
}

main(int argc, char **argv)
{
    int nsecs,j;
    pid_t pid;
    static struct sigaction act;

    if( argc<=2 )
    {
        fprintf(stderr, "Описание: tml число_минут сообщение\n");
        exit(1);
    }

    if( (nsecs=atoi(argv[1]))*60) <= 0)
    {
        fprintf(stderr, "tml: недопустимое время\n");
        exit(2);
    }

    /* Вызов fork, создающий фоновый процесс */
    switch(pid = fork()){
        case -1:      /* Ошибка */
            perror("tml");
            exit(1);
        case 0:      /* Дочерний процесс */
            break;
        default:     /* Родительский процесс */
            printf("Процесс tml с идентификатором %d\n", pid);
            exit(0);
    }

    /* Установить обработчик таймера */
    act.sa_handler = setflag;
    sigaction(SIGALRM, &act, NULL);

    /* Установить таймер */
    alarm(nsecs);

    /* Приостановить выполнение до получения сигнала */
    pause();
}
```

```
/* Если был получен сигнал SIGALRM, вывести сообщение */
if(alarm_flag == TRUE)
{
    printf(BELLS);
    for(j = 2; j < argc; j++)
        printf("%s ",argv[j]);
    printf("\n");
}
exit(0);
}
```

Из этого примера можно получить представление о том, как работает подпрограмма `sleep`, вызывая вначале `alarm`, а затем `pause`.

---

**Упражнение 6.3.** Напишите свою версию подпрограммы `sleep`. Она должна сохранять предыдущее состояние таймера и восстанавливать его при выходе. (Посмотрите полное описание процедуры `sleep` в справочном руководстве системы.)


---

---

**Упражнение 6.4.** Перепишите программу `tml`, используя свою версию процедуры `sleep`.

---





## Глава 7. Межпроцессное взаимодействие при помощи программных каналов

Если два или несколько процессов совместно выполняют одну и ту же задачу, то они неизбежно должны использовать общие данные. Хотя сигналы и могут быть полезны для синхронизации процессов или для обработки исключительных ситуаций или ошибок, они совершенно не подходят для передачи данных от одного процесса к другому. Один из возможных путей разрешения этой проблемы заключается в совместном использовании файлов, так как ничто не мешает нескольким процессам одновременно выполнять операции чтения или записи для одного и того же файла. Тем не менее совместный доступ к файлам может оказаться неэффективным и потребует специальных мер предосторожности для избежания конфликтов.

Для решения этих проблем система UNIX обеспечивает конструкцию, которая называется *программными каналами* (pipe). (В следующих главах будут также изучены некоторые другие средства коммуникации процессов.) Программный канал (или просто канал) служит для установления односторонней связи, соединяющей один процесс с другим, и является еще одним видом обобщенного ввода/вывода системы UNIX. Как увидим далее, процесс может посылать данные в канал при помощи системного вызова `write`, а другой процесс может принимать данные из канала при помощи системного вызова `read`.

### 7.1. Каналы

#### 7.1.1. Каналы на уровне команд

Большинство пользователей UNIX уже сталкивались с конвейерами команд:

```
$ pr doc | lp
```

Этот конвейер организует совместную работу команд `pr` и `lp`. Символ `|` в командной строке сообщает командному интерпретатору, что необходимо создать канал, соединяющий стандартный вывод команды `pr` со стандартным входом команды `lp`. В результате этой команды на матричный принтер будет выведена разбитая на страницы версия файла `doc`.

Разобьем командную строку на составные части. Программа `pr` слева от символа, обозначающего канал, ничего не знает о том, что ее стандартный вывод посылается в канал. Она выполняет обычную запись в свой стандартный вывод, не

предпринимая никаких особых мер. Аналогично программа `lp` справа выполняет чтение точно так же, как если бы она получала свой стандартный ввод с клавиатуры или из обычного файла.<sup>1</sup> Результат в целом будет таким же, как при выполнении следующей последовательности команд:

```
$ pr doc > tmpfile
$ lp < tmpfile
$ rm tmpfile
```

Управление потоком в канале осуществляется автоматически и прозрачно для процесса. Поэтому, если программа `pr` будет выводить информацию слишком быстро, то ее выполнение будет приостановлено. После того как программа `lp` догонит программу `pr`, и количество данных, находящихся в канале, упадет до приемлемого уровня, выполнение программы `pr` продолжится.

Каналы являются одной из самых сильных и характерных особенностей ОС UNIX, доступных даже с уровня командного интерпретатора. Они позволяют легко соединять между собой произвольные последовательности команд. Поэтому программы UNIX могут разрабатываться как простые инструменты, осуществляющие чтение из стандартного ввода, запись в стандартный вывод и выполняющие одну, четко определенную задачу. При помощи каналов из этих основных блоков могут быть построены более сложные командные строки, например, команда

```
$ who | wc -l
```

направляет вывод программы `who` в программу подсчета числа слов `wc`, а задание параметра `-l` в программе `wc` определяет, что необходимо подсчитывать только число строк. Таким образом, в конечном итоге программа `wc` выводит число находящихся в системе пользователей (иногда нужно исключить из суммы первую строку-заголовок вывода `who`).

### 7.1.2. Использование каналов в программе

Каналы создаются в программе при помощи системного вызова `pipe`. В случае удачного завершения вызов сообщает два дескриптора файла: один для записи в канал, а другой для чтения из него. Вызов `pipe` определяется следующим образом:

#### Описание

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Переменная `filedes` является массивом из двух целых чисел, который будет содержать дескрипторы файлов, обозначающие канал. После успешного вызова `filedes[0]` будет открыт для чтения из канала, а `filedes[1]` для записи в канал.

В случае неудачи вызов `pipe` вернет значение `-1`. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа

---

<sup>1</sup> На самом деле программа имеет возможность с помощью операций управления ввода/вывода выяснить тип конечного устройства, используемого в качестве стандартного ввода/вывода (файл, терминал или канал). Некоторые стандартные утилиты UNIX ведут себя по-разному в разных ситуациях; сравните вывод `ls` на терминал и в канал. — *Прим. науч. ред.*

дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная `errno` будет содержать значение `EMFILE`), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная `errno` будет содержать значение `ENFILE`).

После создания канала с ним можно работать просто при помощи вызовов `read` и `write`. Следующий пример демонстрирует это: он создает канал, записывает в него три сообщения, а затем считывает их из канала:

```
/* Первый пример работы с каналами */
#include <unistd.h>
#include <stdio.h>

/* Эти строки заканчиваются нулевым символом */
#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main ()
{
    char inbuf[MSGSIZE];
    int p[2], j;

    /* Открыть канал */
    if(pipe(p) == -1) {
        perror("Ошибка вызова pipe") ;
        exit(1);
    }

    /* Запись в канал */
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);

    /* Чтение из канала */
    for(j = 0; j < 3; j++)
    {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }

    exit(0);
}
```

На выходе программы получим:

```
hello, world #1
hello, world #2
hello, world #3
```

Обратите внимание, что сообщения считываются в том же порядке, в каком они были записаны. Каналы обращаются с данными в порядке «*первый вошел – первым вышел*» (first-in first-out, или сокращенно FIFO). Другими словами, данные, которые помещаются в канал первыми, первыми и считываются на другом

конце канала. Этот порядок нельзя изменить, поскольку вызов `lseek` не работает с каналами.

Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми, хотя в нашем примере это и было так. Можно, например, писать в канал блоками по 512 байт, а затем считывать из него по одному символу, так

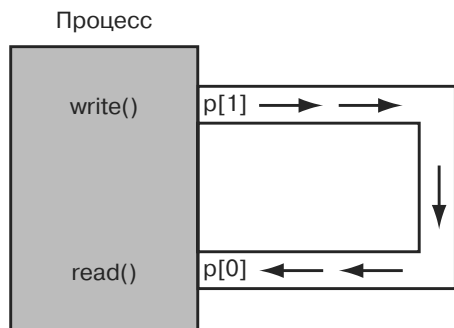


Рис. 7.1. Первый пример работы с каналами

же как и в случае обычного файла. Тем не менее, как будет показано в разделе 7.2, использование блоков фиксированного размера дает определенные преимущества.

Работа примера показана графически на рис. 7.1. Эта диаграмма позволяет более ясно представить, что процесс только посылает данные сам себе, используя канал в качестве некой разновидности механизма обратной связи. Это может показаться бессмысленным, поскольку процесс общается только сам с собой.

Настоящее значение каналов проявляется при использовании вместе с системным вызовом `fork`, тогда можно воспользоваться тем фактом, что файловые дескрипторы остаются открытыми в обоих процессах. Следующий пример демонстрирует это. Он создает канал и вызывает `fork`, затем дочерний процесс обменивается несколькими сообщениями с родительским:

```
/* Второй пример работы с каналами */
#include <unistd.h>
#include <stdio.h>

#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main ()
{
    char inbuf[MSGSIZE];
    int p[2], j;
    pid_t pid;

    /* Открыть канал */
    if(pipe(p) == -1)
    {
        perror("ошибка вызова pipe ");
        exit(1);
    }

    switch(pid = fork()){
        case -1:
            perror("ошибка вызова fork");
```

```
    exit(2);
case 0:
    /* Это дочерний процесс, выполнить запись в канал */
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);
    break;
default:
    /* Это родительский процесс, выполнить чтение из канала */
    for(j = 0; j < 3; j++)
    {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }
    wait(NULL);
}

exit(0);
}
```

Этот пример представлен графически на рис. 7.2. На нем показано, как канал соединяет два процесса. Здесь видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять запись в канал и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором `p[1]` и чтение из файла с дескриптором `p[0]`. Это создает определенную проблему. Каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

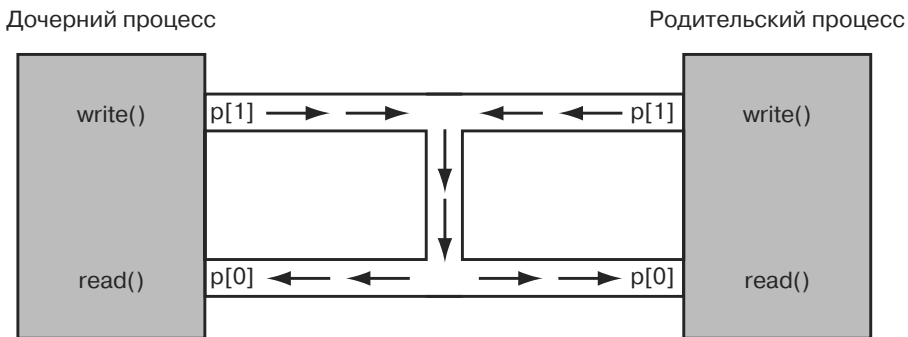


Рис. 7.2. Второй пример работы с каналом

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать неприятностей, если посылающий данные процесс закроет дескриптор файла, открытого на запись, – раздел 7.1.4 объясняет возможные последствия. Приведенные до сих пор примеры работают только потому, что принимающий

процесс в точности знает, какое количество данных он может ожидать. Следующий пример представляет собой законченное решение:

```
/* Третий пример работы с каналами */
#include <unistd.h>
#include <stdio.h>
#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main()
{
    char inbuf[MSGSIZE];
    int p[2], j;
    pid_t pid;

    /* Открыть канал */
    if(pipe(p) == -1)
    {
        perror("Ошибка вызова pipe");
        exit(1);
    }

    switch(pid = fork()){
    case -1:
        perror("Ошибка вызова fork");
        exit(2);
    case 0:
        /* Дочерний процесс, закрывает дескриптор файла,
         * открытого для чтения и выполняет запись в канал.
         */
        close(p[0]);
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
        break;
    default:
        /* Родительский процесс, закрывает дескриптор файла,
         * открытого для записи и выполняет чтение из канала.
         */
        close(p[1]);
        for(j = 0; j < 3; j++)
        {
            read(p[0], inbuf, MSGSIZE);
            printf("%s\n", inbuf);
        }
        wait(NULL);
    }

    exit(0);
}
```

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому. Эта упрощенная ситуация показана на рис. 7.3.

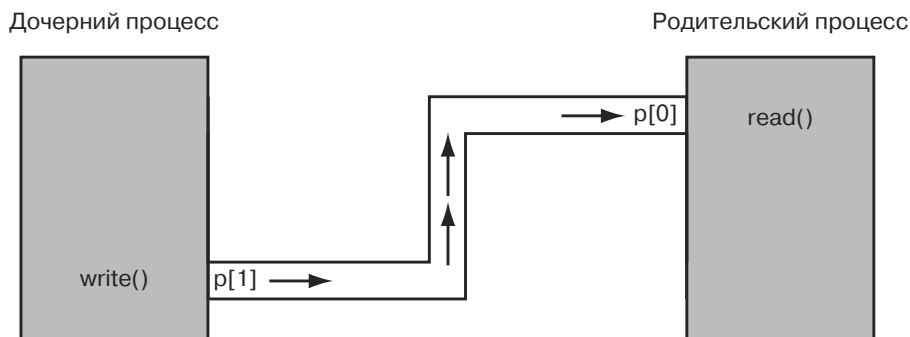


Рис. 7.3. Третий пример работы с каналами

---

**Упражнение 7.1.** В последнем примере канал использовался для установления связи между родительским и дочерним процессами. Но дескрипторы файлов канала могут передаваться и сквозь несколько вызовов `fork`. Это означает, что несколько процессов могут писать в канал и несколько процессов могут читать из него. Для демонстрации этого поведения напишите программу, которая создает три процесса, два из которых выполняют запись в канал, а один – чтение из него. Процесс, выполняющий чтение, должен выводить все получаемые им сообщения на свой стандартный вывод.

---

**Упражнение 7.2.** Для установления двусторонней связи между процессами можно создать два канала, работающих в разных направлениях. Придумайте возможный диалог между процессами и реализуйте его при помощи двух каналов.

---

### 7.1.3. Размер канала

Пока в примерах передавались только небольшие объемы данных. Важно заметить, что на практике размер буфера канала конечен. Другими словами, только определенное число байтов может находиться в канале, прежде чем следующий вызов `write` будет заблокирован. Минимальный размер, определенный стандартом POSIX, равен 512 байтам. В большинстве существующих систем это значение намного больше. При программировании важно знать максимальный размер канала для системы, так как он влияет на оба вызова `write` и `read`. Если вызов `write` выполняется для канала, в котором есть свободное место, то данные посылаются в канал, и немедленно происходит возврат из вызова. Если же в момент вызова `write` происходит переполнение канала, то выполнение процесса обычно приостанавливается до тех пор, пока место не освободится в результате выполнения другим процессом чтения из канала.

Приведенная в качестве примера следующая программа записывает в канал символ за символом до тех пор, пока не произойдет блокировка вызова `write`. Программа использует вызов `alarm` для предотвращения слишком долгого ожидания в случае, если вызов `read` никогда не произойдет. Необходимо обратить внимание на использование процедуры `fpathconf`, служащей для определения максимального числа байтов, которые могут быть записаны в канал за один прием.

```
/* Запись в канал до возникновения блокировки записи */

#include <signal.h>
#include <unistd.h>
#include <limits.h>

int count;
void alrm_action(int);

main()
{
    int p[2];
    int pipe_size;
    char c = 'x';
    static struct sigaction act;

    /* Задать обработчик сигнала */
    act.sa_handler = alrm_action;
    sigfillset(&(act.sa_mask));

    /* Создать канал */
    if(pipe(p) == -1)
    {
        perror("Ошибка вызова pipe ") ;
        exit(1);
    }

    /* Определить размер канала */
    pipe_size = fpathconf(p[0], _PC_PIPE_BUF);
    printf("Максимальный размер канала: %d байт\n", pipe_size);

    /* Задать обработчик сигнала */
    sigaction(SIGALRM, &act, NULL);

    while(1)
    {
        /* Установить таймер */
        alarm(20);

        /* Запись в канал */
        write(p[1], &c, 1),

        /* Сбросить таймер */
        alarm(0);

        if((++count % 1024) == 0)
            printf("%d символов в канале\n", count);
    }
}
```



```
/* Вызывается при получении сигнала SIGALRM */  
void alrm_action(int signo)  
{  
    printf("Запись блокируется после вывода %d символов\n", count);  
    exit(0);  
}
```

Вот результат работы программы на некоторой системе:

```
Максимальный размер канала: 32768 байт  
1024 символов в канале  
2048 символов в канале  
3072 символов в канале  
4096 символов в канале  
5120 символов в канале  
.  
.  
.  
31744 символов в канале  
32768 символов в канале  
Запись блокируется после вывода 32768 символов
```

Обратите внимание, насколько реальный предел больше, чем заданный стандартом POSIX минимальный размер канала.

Ситуация становится более сложной, если процесс пытается записать за один вызов `write` больше данных, чем может вместить даже полностью пустой канал. В этом случае ядро вначале попытается записать в канал максимально возможный объем данных, а затем приостанавливает выполнение процесса до тех пор, пока не освободится место под оставшиеся данные. Это важный момент: обычно вызов `write` для канала выполняется *неделимыми порциями* (atomically), и данные передаются ядром за одну непрерываемую операцию. Если делается попытка записать в канал больше данных, чем он может вместить, то вызов `write` выполняется поэтапно. Если при этом несколько процессов выполняют запись в канал, то данные могут оказаться беспорядочно перепутанными.

Взаимодействие вызова `read` с каналами является более простым. При выполнении вызова `read` система проверяет, является ли канал пустым. Если он пуст, то вызов `read` будет заблокирован до тех пор, пока другой процесс не запишет в канал какие-либо данные. При наличии в канале данных произойдет возврат из вызова `read`, даже если запрашивается больший объем данных, чем находится в канале.

### 7.1.4. Закрытие каналов

Что произойдет, если дескриптор файла, соответствующий одному из концов канала, будет закрыт? Возможны два случая:

- *закрывается дескриптор файла, открытого только на запись.* Если существуют другие процессы, в которых канал открыт на запись, то ничего не произойдет. Если же больше не существует процессов, которые могли бы выполнять запись в канал, и канал при этом пуст, то любой процесс, который

попытается выполнить чтение из канала, получит пустой блок данных. Процессы, которые были приостановлены и ожидали чтения из канала, продолжат свою работу, вызовы `read` вернут нулевое значение. Для процесса, выполняющего чтение, результат будет похож на достижение конца файла;

- *закрывается дескриптор файла, открытого только на чтение.* Если еще есть процессы, в которых канал открыт на чтение, то снова ничего не произойдет. Если же больше не существует процессов, выполняющих чтение из канала, то ядро посылает всем процессам, ожидающим записи в канал, сигнал `SIGPIPE`. Если этот сигнал не перехватывается в процессе, то процесс при этом завершит свою работу. Если же сигнал перехватывается, то после завершения процедуры обработчика прерывания вызов `write` вернет значение `-1`, и переменная `errno` после этого будет содержать значение `EPIPE`. Процессам, которые будут пытаться после этого выполнить запись в канал, также будет посылаться сигнал `SIGPIPE`.

### 7.1.5. Запись и чтение без блокирования

Как уже было упомянуто, при использовании и вызова `read`, и вызова `write` может возникнуть блокирование, которое иногда нежелательно. Может, например, понадобится, чтобы программа выполняла процедуру обработки ошибок или опрашивала несколько каналов до тех пор, пока не получит данные из одного из них. К счастью, есть простые способы пресечения нежелательных остановов внутри `read` и `write`.

Первый метод заключается в использовании для вызова `fstat`. Поле `st_size` в возвращаемой вызовом структуре `stat` сообщает текущее число символов, находящихся в канале. Если только один процесс выполняет чтение из канала, такой подход работает прекрасно. Если же несколько процессов выполняют чтение из канала, то за время, прошедшее между вызовами `fstat` и `read`, ситуация может измениться, если другой процесс успеет выполнить чтение из канала.

Второй метод заключается в использовании вызова `fcntl`. Помимо других выполняемых им функций этот вызов позволяет процессу устанавливать для дескриптора файла флаг `O_NONBLOCK`. Это предотвращает блокировку последующих вызовов `read` или `write`. В этом контексте вызов `fcntl` может использоваться следующим образом:

```
#include <fcntl.h>
.
.
.
if(fcntl(filedes, F_SETFL, O_NONBLOCK) == -1)
    perror("fcntl");
```

Если дескриптор `filedes` является открытым только на запись, то следующие вызовы `write` не будут блокироваться при заполнении канала. Вместо этого они будут немедленно возвращать значение `-1` и присваивать переменной `errno` значение `EAGAIN`. Аналогично, если дескриптор `filedes` соответствует выходу канала, то процесс немедленно вернет значение `-1`, если в канале нет данных, а не

приостановит работу. Так же, как и в случае вызова `write`, переменной `errno` будет присвоено значение `EAGAIN`. (Если установлен другой флаг – `O_NDELAY`, то поведение вызова `read` будет другим. Если канал пуст, то вызов вернет нулевое значение. Далее этот случай не будет рассматриваться.)

Следующая программа иллюстрирует применение вызова `fcntl`. В ней создается канал, для дескриптора чтения из канала устанавливается флаг `O_NONBLOCK`, а затем выполняется вызов `fork`. Дочерний процесс посылает сообщения родительскому, выполняющему бесконечный цикл, опрашивая канал и проверяя, поступили ли данные.

```
/* Пример использования флага O_NONBLOCK */
#include <fcntl.h>
#include <errno.h>

#define MSGSIZE 6

int parent(int *);
int child(int *);

char *msg1 = "hello";
char *msg2 = "bye!!";

main()
{
    int pfd[2];

    /* Открыть канал */
    if(pipe(pfd) == -1)
        fatal("Ошибка вызова pipe ");

    /* Установить флаг O_NONBLOCK для дескриптора p[0] */
    if(fcntl(pfd[0], F_SETFL, O_NONBLOCK) == -1)
        fatal("Ошибка вызова fcntl");

    switch(fork()){
    case -1:          /* Ошибка */
        fatal("Ошибка вызова fork");
    case 0:           /* Дочерний процесс */
        child(pfd);
    default:          /* Родительский процесс */
        parent(pfd);
    }
}

int parent(int p[2]) /* Код родительского процесса */
{
    int nread;
    char buf[MSGSIZE];
    close(p[1]);

    for(;;)
    {
        switch(nread = read(p[0], buf, MSGSIZE)){
```

```
case -1:
    /* Проверить, есть ли данные в канале */
    if(errno == EAGAIN)
    {
        printf("(канал пуст)\n");
        sleep(1);
        break;
    }
    else
        fatal("Ошибка вызова read");
case 0:
    /* Канал был закрыт */
    printf("Конец связи\n");
    exit(0);
default:
    printf("MSG=%s\n",buf);
}
}
}

int child(int p[2])
{
    int count;

    close(p[0]);

    for(count = 0; count < 3; count++)
    {
        write(p[1], msg1, MSGSIZE);
        sleep(3);
    }

    /* Послать последнее сообщение */
    write(p[1], msg2, MSGSIZE);
    exit(0);
}
```

Этот пример использует для вывода сообщений об ошибках процедуру `fatal`, описанную в предыдущей главе. Чтобы не возвращаться назад, приведем ее реализацию:

```
/* Вывести сообщение об ошибке и закончить работу */
int fatal(char *s)
{
    perror(s);
    exit(1);
}
```

Вывод программы не полностью предсказуем, так как число сообщений «канал пуст» может быть различным. На одном из компьютеров был получен следующий вывод:

```
MSG=hello
(канал пуст)
```

```
(канал пуст)
(канал пуст)
MSG=hello
(канал пуст)
(канал пуст)
(канал пуст)
MSG=hello
(канал пуст)
(канал пуст)
(канал пуст)
MSG=bye!!
Конец связи
```

### 7.1.6. Использование системного вызова *select* для работы с несколькими каналами

Для простых приложений применение неблокирующих операций чтения и записи работает прекрасно. Для работы с множеством каналов одновременно существует другое решение, которое заключается в использовании системного вызова `select`.

Представьте ситуацию, когда родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов, как показано на рис. 7.4.

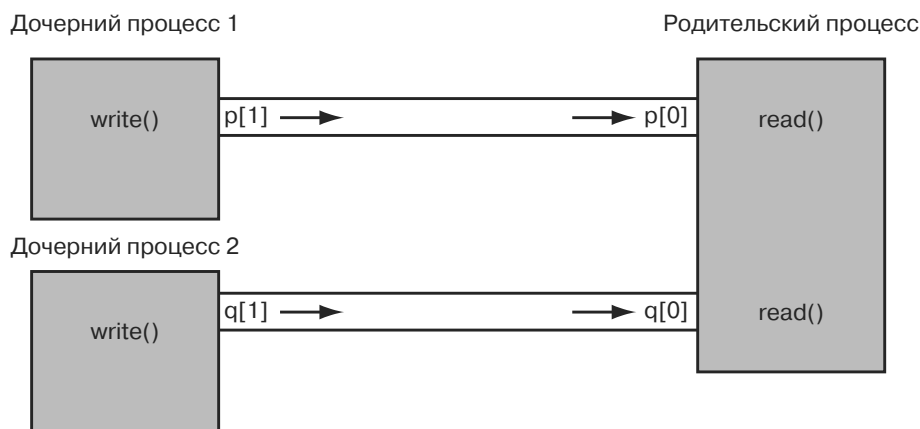


Рис. 7.4. Клиент/сервер с использованием каналов

В этом случае серверный процесс должен как-то справляться с ситуацией, когда одновременно в нескольких каналах может находиться информация, ожидающая обработки. Кроме того, если ни в одном из каналов нет ожидающих данных, то может иметь смысл приостановить работу серверного процесса до их появления, а не опрашивать постоянно каналы. Если информация поступает более чем по одному каналу, то серверный процесс должен знать обо всех таких каналах для того, чтобы работать с ними в правильном порядке (например, согласно их приоритетам).

Это можно сделать при помощи системного вызова `select` (существует также аналогичный вызов `poll`). Системный вызов `select` используется не только для

каналов, но и для обычных файлов, терминальных устройств, именованных каналов (которые будут рассмотрены в разделе 7.2) и сокетов (им посвящена глава 10). Системный вызов `select` показывает, какие дескрипторы файлов из заданных наборов готовы для чтения, записи или ожидают обработки ошибок. Иногда серверный процесс не должен совсем прекращать работу, даже если не происходит никаких событий, поэтому в вызове `select` также можно задать предельное время ожидания.

### Описание

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

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

Первый параметр `nfds` задает число дескрипторов файлов, которые могут представлять интерес для сервера. Например, если дескрипторы файлов с номерами 0, 1 и 2 присвоены потокам `stdin`, `stdout` и `stderr` соответственно, и открыты еще два файла с дескрипторами 3 и 4, то можно присвоить параметру `nfds` значение 5. Программист может определять это значение самостоятельно или воспользоваться постоянной `FD_SETSIZE`, которая определена в файле `<sys/time.h>`. Значение постоянной `FD_SETSIZE` равно максимальному числу дескрипторов файлов, которые могут быть использованы вызовом `select`.

Второй, третий и четвертый параметры вызова `select` являются указателями на *битовые маски* (bit mask), в которых каждый бит соответствует дескриптору файла. Если бит включен, то это обозначает интерес к соответствующему дескриптору файла. Набор `readfds` определяет дескрипторы, для которых сервер ожидает возможности чтения; набор `writefds` – дескрипторы, для которых ожидается возможность выполнить запись; набор `errorfds` определяет дескрипторы, для которых сервер ожидает появление ошибки или исключительной ситуации, например, по сетевому соединению могут поступить внеочередные данные. Так как работа с битами довольно неприятна и приводит к немобильности программ, существует абстрактный тип данных `fd_set`, а также макросы или функции (в зависимости от конкретной реализации системы) для работы с объектами этого типа. Вот эти макросы для работы с битами файловых дескрипторов:

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

```
/* Инициализация битовой маски, на которую указывает fdset */
void FD_ZERO(fd_set *fdset);
```

```
/* Установка бита fd в маске, на которую указывает fdset */
void FD_SET(int fd, fd_set *fdset);
```

```
/* Установлен ли бит fd в маске, на которую указывает fdset? */ int
FD_ISSET(int fd, fd_set *fdset);
```

```
/* Сбросить бит fd в маске, на которую указывает fdset */
void FD_CLR(int fd, fd_set *fdset);
```

Следующий пример демонстрирует, как отслеживать состояние двух открытых дескрипторов файлов:

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

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

int fd1, fd2;
fd_set readset;

fd1 = open("file1", O_RDONLY) ;
fd2 = open("file2", O_RDONLY) ;

FD_ZERO(&readset);
FD_SET(fd1, &readset);
FD_SET(fd2, &readset);

switch(select(5, &readset, NULL, NULL, NULL))
{
    /* Обработка ввода */ }
```

Пример очевиден, если вспомнить, что переменные `fd1` и `fd2` представляют собой небольшие целые числа, которые можно использовать в качестве индексов битовой маски. Обратите внимание на то, что аргументам `writelfds` и `errorfds` в вызове `select` присвоено значение `NULL`. Это означает, что представляет интерес только чтение из `fd1` и `fd2`.

Пятый параметр вызова `select`, `timeout`, является указателем на следующую структуру `timeval`:

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

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

Если указатель является нулевым, как в этом примере, то вызов `select` будет заблокирован до тех пор, пока не произойдет интересующее процесс событие. Если в структуре `timeout` задано нулевое время, то вызов `select` завершится немедленно (без блокирования). И, наконец, если структура `timeout` содержит ненулевое значение, то возврат из вызова `select` произойдет через заданное число секунд или микросекунд, если файловые дескрипторы неактивны.

Возвращаемое вызовом `select` значение равно `-1` в случае ошибки, нулю – после истечения временного интервала или целому числу, равному числу «интересующих» программу дескрипторов файлов. Следует сделать предостережение: при возврате из вызова `select` он переустанавливает битовые маски, на которые указывают переменные `readfds`, `writelfds` или `errorfds`, сбрасывая маску и снова задавая в ней дескрипторы файлов, содержащие искомую информацию. Поэтому необходимо сохранять копию исходных масок.<sup>1</sup>

<sup>1</sup> В некоторых реализациях вызов `select` изменяет также содержимое структуры `timeout`: оно заполняется оставшимся временем до истечения первоначально заданного интервала. Данную возможность следует учитывать при вызове `select` с нулевыми масками в качестве высокоточного аналога вызова `sleep` – тогда использование `select` в цикле может привести к неправильным результатам. – *Прим. науч. ред.*

Приведем более сложный пример, в котором используются три канала, связанные с тремя дочерними процессами. Родительский процесс должен также отслеживать стандартный ввод.

```
/* Программа server – обслуживает три дочерних процесса */
#include <sys/time.h>
#include <sys/wait.h>

#define MSGSIZE 6

char *msg1 = "hello";
char *msg2 = "bye!!";

void parent(int [][]);
int child(int []);

main()
{
    int pip[3][2];
    int i;

    /* Создает три канала связи, и порождает три процесса */
    for(i = 0; i < 3; i++)
    {
        if(pipe(pip[i]) == -1)
            fatal("Ошибка вызова pipe");

        switch(fork()){
            case -1: /* Ошибка */
                fatal("Ошибка вызова fork");
            case 0: /* Дочерний процесс */
                child(pip[i]);
        }
    }

    parent(pip);

    exit(0);
}

/* Родительский процесс ожидает сигнала в трех каналах */
void parent(int p[3][2]) /* Код родительского процесса */
{
    char buf[MSGSIZE], ch;
    fd_set set, master;
    int i;

    /* Закрывает все ненужные дескрипторы, открытые для записи */
    for(i = 0; i < 3; i++)
        close(p[i][1]);

    /* Задаёт битовые маски для системного вызова select */
    FD_ZERO(&master) ;
    FD_SET(0, &master);
```



```
for(i = 0; i < 3; i++)
    FD_SET(p[i][0], &master);

/* Лимит времени для вызова select не задан, поэтому он
 * будет заблокирован, пока не произойдет событие */
while(set = master, select(p[2][0]+1, &set,NULL,NULL,NULL) > 0)
{
    /* Нельзя забывать и про стандартный ввод,
     * то есть дескриптор файла fd=0 */
    if(FD_ISSET(0, &set))
    {
        printf("Из стандартного ввода...");
        read(0, &ch, 1);
        printf("%c\n", ch);
    }

    for(i = 0; i < 3; i++)
    {
        if(FD_ISSET(p[i][0], &set))
        {
            {
                if(read(p[i][0], buf, MSGSIZE)>0)
                {
                    printf("Сообщение от потомка%d\n", i);
                    printf("MSG=%s\n",buf);
                }
            }
        }
    }

    /* Если все дочерние процессы прекратили работу,
     * то сервер вернется в основную программу */
    if(waitpid(-1,NULL,WNOHANG) == -1)
        return;
}

int child(int p[2])
{
    int count;

    close (p[0]);

    for(count = 0; count < 2; count++)
    {
        write(p[1], msg1, MSGSIZE);
        /* Пауза в течение случайно выбранного времени */
        sleep(getpid() % 4);
    }

    /* Посылает последнее сообщение */
    write(p[1], msg2, MSGSIZE);
    exit(0);
}
```

Результат данной программы может быть таким:

Сообщение от потомка 0  
MSG=hello

Сообщение от потомка 1  
MSG=hello

Сообщение от потомка 2  
MSG=hello

*d* (пользователь нажимает клавишу **d**, а затем клавишу **Return**)  
Из стандартного ввода *d* (повторение символа **d**)  
Из стандартного ввода (повторение символа **Return**)

Сообщение от потомка 0  
MSG=hello  
Сообщение от потомка 1  
MSG=hello  
Сообщение от потомка 2  
MSG=hello

Сообщение от потомка 0  
MSG=bye  
Сообщение от потомка 1  
MSG=bye  
Сообщение от потомка 2  
MSG=bye

Обратите внимание, что в этом примере пользователь нажимает клавишу **d**, а затем символ перевода строки (Enter или Return), и это отслеживается в стандартном вводе в вызове `select`.

### 7.1.7. Каналы и системный вызов `exec`

Вспомним, как можно создать канал между двумя программами с помощью командного интерпретатора:

```
$ ls | wc
```

Как это происходит? Ответ состоит из двух частей. Во-первых, командный интерпретатор использует тот факт, что открытые дескрипторы файлов остаются открытыми (по умолчанию) после вызова `exec`. Это означает, что два файловых дескриптора канала, которые были открыты до выполнения комбинации вызовов `fork/exec`, останутся открытыми и когда дочерний процесс начнет выполнение новой программы. Во-вторых, перед вызовом `exec` командный интерпретатор соединяет стандартный вывод программы `ls` с входом канала, а стандартный ввод программы `wc` – с выходом канала. Это можно сделать при помощи вызова `fcntl` или `dup2`, как было показано в упражнении 5.10. Так как значения дескрипторов файлов, соответствующих стандартному вводу, стандартному выводу и стандартному выводу диагностики, равны 0, 1 и 2 соответственно, то можно, например, соединить стандартный вывод с другим дескриптором файла, используя вызов `dup2` следующим образом. Обратите внимание, что перед переназначением вызов `dup2` закрывает файл, представленный его вторым параметром.

```
/* Вызов fcntl будет копировать дескриптор файла "1" */
dup2(filedes, 1);
.
.
.
/* Теперь программа будет записывать свой стандартный */
/* вывод в файл, заданный дескриптором filedes */
.
.
.
```

Следующий пример, программа `join`, демонстрирует механизм каналов, задействованный в упрощенном командном интерпретаторе. Программа `join` имеет два параметра, `com1` и `com2`, каждый из которых соответствует выполняемой команде. Оба параметра в действительности являются массивами строк, которые будут переданы вызову `execvp`.

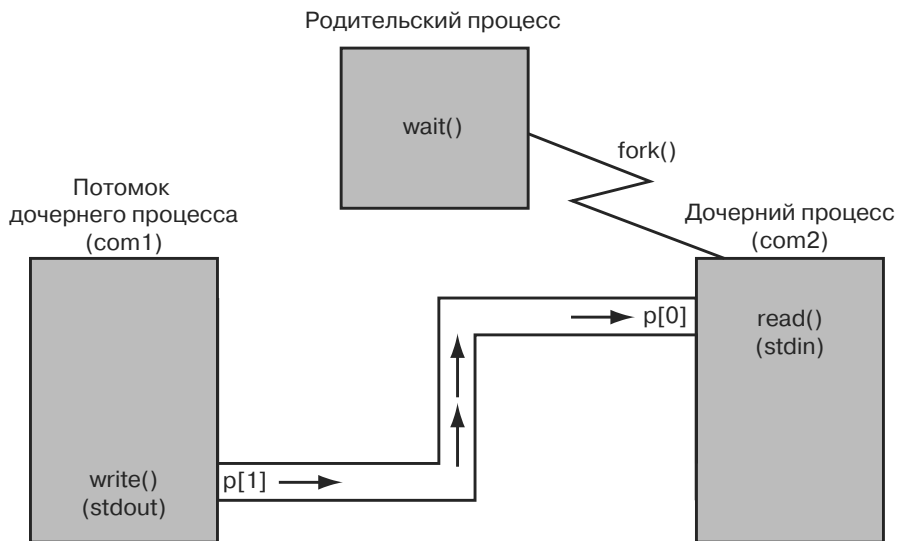


Рис. 7.5. Программа `join`

Программа `join` запустит обе программы на выполнение и свяжет стандартный вывод программы `com1` со стандартным вводом программы `com2`. Работа программы `join` изображена на рис. 7.5 и может быть описана следующей схемой (без учета обработки ошибок):

процесс порождает дочерний процесс и ожидает действий от него  
дочерний процесс продолжает работу

дочерний процесс создает канал

затем дочерний процесс порождает еще один дочерний процесс

В потомке дочернего процесса:

стандартный вывод подключается

к входу канала при помощи вызова `dup2`

ненужные дескрипторы файлов закрываются

при помощи вызова `exec` запускается программа,  
заданная параметром `'com1'`

В первом дочернем процессе:

стандартный ввод подключается

к выходу канала при помощи вызова `dup2`

ненужные дескрипторы файлов закрываются

при помощи вызова `exec` запускается программа,  
заданная параметром `'com2'`

Далее следует реализация программы `join`; она также использует процедуру `fatal`, представленную в разделе 7.1.5.

```
/* Программа join – соединяет две программы каналом */  
  
int join(char *com1[], char *com2[])  
{  
    int p[2], status;  
  
    /* Создать дочерний процесс для выполнения команд */  
    switch(fork()){  
        case -1: /* Ошибка */  
            fatal("Ошибка первого вызова fork в программе join");  
        case 0: /* Дочерний процесс */  
            break;  
        default: /* Родительский процесс */  
            wait(&status);  
            return(status);  
    }  
  
    /* Остаток процедуры, выполняемой дочерним процессом */  
  
    /* Создать канал */  
    if (pipe (p) == -1)  
        fatal("Ошибка вызова pipe в программе join");  
  
    /* Создать еще один процесс */  
    switch(fork()){  
        case -1:  
            /* Ошибка */  
            fatal("Ошибка второго вызова fork в программе join");  
        case 0:  
            /* Процесс, выполняющий запись */  
            dup2(p[1],1); /* Направить ст. вывод в канал */  
  
            close(p[0]); /* Сохранить дескрипторы файлов */  
            close(p[1]);  
  
            execvp(com1[0], com1);  
    }  
}
```

```
/* Если exesvr возвращает значение, то произошла ошибка */
fatal("Ошибка первого вызова exesvr в программе join");
default:
/* Процесс, выполняющий чтение */
dup2(p[0], 0); /* Направить ст. ввод из канала */

close(p[0]);
close(p[1]);
exesvr(com2[0], com2);
fatal("Ошибка второго вызова exesvr в программе join");
}
}
```

Эту процедуру можно вызвать следующим образом:

```
#include <stdio.h>

main()
{
char *one[4] = {"ls", "-l", "/usr/lib", NULL};
char *two[3] = {"grep", "^d", NULL};
int ret;

ret = join(one, two);
printf("Возврат из программы join %d\n", ret);
exit(0) ;
}
```

---

**Упражнение 7.3.** Как можно обобщить подход, показанный в программе `join`, для связи нескольких процессов при помощи каналов?

---

---

**Упражнение 7.4.** Добавьте возможность работы с каналами в командный интерпретатор `smallsh`, представленный в предыдущей главе.

---

---

**Упражнение 7.5.** Придумайте метод, позволяющий родительскому процессу запускать программу в качестве дочернего процесса, а затем считывать ее стандартный вывод при помощи канала. Стоит отметить, что эта идея лежит в основе процедур `open` и `pclose`, которые входят в стандартную библиотеку ввода/вывода. Процедуры `open` и `pclose` избавляют программиста от большинства утомительных деталей согласования вызовов `fork`, `exec`, `close`, `dup` или `dup2`. Эти процедуры обсуждаются в главе 11.

---

## 7.2. Именованные каналы, или FIFO

Каналы являются изящным и мощным механизмом межпроцессного взаимодействия. Тем не менее они имеют ряд недостатков.

Первый, и наиболее серьезный из них, заключается в том, что каналы могут использоваться только для связи процессов, имеющих общее происхождение, таких как родительский процесс и его потомок. Это ограничение становится очевидным при попытке разработать настоящую «серверную» программу, которая выполняется постоянно, обеспечивая системный сервис. Примерами таких программ являются серверы управления сетью и спулеры печати. В идеале клиентские процессы должны иметь возможность стартовать, подключаться к не связанному с ними серверному процессу при помощи канала, а затем снова отключаться от него. К сожалению, такую модель при помощи обычных каналов реализовать нельзя.

Второй недостаток каналов заключается в том, что они не могут существовать постоянно. Они каждый раз должны создаваться заново, а после завершения обращающегося к ним процесса уничтожаются.

Для восполнения этих недостатков существует разновидность канала, называемая *именованным каналом*, или файлом типа *FIFO* (сокращение от *first-in first-out*, то есть «первый вошел/первым вышел»). В отношении вызовов *read* и *write* именованные каналы идентичны обычным. Тем не менее, в отличие от обычных каналов, именованные каналы являются постоянными и им присвоено имя файла системы UNIX. Именованный канал также имеет владельца, размер и связанные с ним права доступа. Он может быть открыт, закрыт и удален, как и любой файл UNIX, но при чтении или записи ведет себя аналогично каналу.

Прежде чем рассматривать применение каналов FIFO на программном уровне, рассмотрим их использование на уровне команд. Для создания именованного канала используется команда *mknod*:

```
$ /etc/mknod channel p
```

Первый аргумент *channel* является именем канала FIFO (в качестве него можно задать любое допустимое имя UNIX). Параметр *p* команды *mknod* указывает, что нужно создать именованный канал. Этот параметр необходим, так как команда *mknod* также используется для создания файлов устройств.

Некоторые атрибуты вновь созданного канала FIFO можно вывести при помощи команды *ls*:

```
$ ls -l channel
prw-rw-r- 1 ben usr 0 Aug 1 21:05 channel
```

Символ *p* в первой колонке обозначает, что *channel* является файлом типа FIFO. Обратите внимание на права доступа к именованному каналу *channel* (чтение/запись для владельца и группы владельца, только чтение для всех остальных пользователей); владельца и группу владельца (*ben*, *usr*); размер (0 байт, то есть в настоящий момент канал пуст) и время создания.

При помощи стандартных команд UNIX можно выполнять чтение из канала FIFO и запись в него, например:

```
$ cat < channel
```

Если выполнить эту команду сразу же после создания именованного канала `channel`, то она «зависнет». Это происходит из-за того, что процесс, открывающий канал FIFO на чтение, по умолчанию будет заблокирован до тех пор, пока другой процесс не попытается открыть канал FIFO для записи. Аналогично процесс, пытающийся открыть канал FIFO для записи, будет заблокирован до тех пор, пока другой процесс не попытается открыть его для чтения. Это благоразумный подход, так как он экономит системные ресурсы и облегчает координацию работы программы. Вследствие этого, при необходимости создания одновременно как записывающего, так и читающего процессов, потребуется запустить один из них в фоновом режиме (или с другого терминала, или псевдотерминала `xterm` графического интерфейса), например:

```
$ cat < channel &
102
$ ls -l > channel; wait
total 17
prw-rw-r- 1 ben usr 0      Aug 1   21:05  channel
-rw-rw-r- 1 ben usr 0      Aug 1   21:06  f
-rw-rw-r- 1 ben usr 937    Jul 27  22:30  fifos
-rw-rw-r- 1 ben usr 7152   Jul 27  22:11  pipes.cont
```

Проанализируем подробнее этот результат. Содержимое каталога вначале выводится при помощи команды `ls`, а затем записывается в канал FIFO. Ожидающая команда `cat` затем считывает данные из канала FIFO и выводит их на экран. После этого процесс, выполняющий команду `cat`, завершает работу. Это происходит из-за того, что канал FIFO больше не открыт для записи, чтение из него будет безуспешным, как и для обычного канала, что команда `cat` понимает как достижение конца файла. Команда же `wait` заставляет командный интерпретатор ждать завершения команды `cat` перед тем, как снова вывести приглашение командной строки.

### 7.2.1. Программирование при помощи каналов FIFO

Программирование при помощи каналов FIFO, в основном, идентично программированию с использованием обычных каналов. Единственное существенное различие заключается в их инициализации. Вместо использования вызова `pipe` канал FIFO создается при помощи вызова `mkfifo`. В старых версиях UNIX может потребоваться использование более общего вызова `mknod`.

#### Описание

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

int mkfifo(const char *pathname, mode_t mode);
```

Системный вызов `mkfifo` создает файл FIFO с именем, заданным первым параметром `pathname`. Канал FIFO будет иметь права доступа, заданные параметром `mode` и измененные в соответствии со значением `umask` процесса.

После создания канала FIFO он должен быть открыт при помощи вызова `open`. Поэтому, например, фрагмент кода

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
.
.
mkfifo("/tmp/fifo", 0666);
.
.
fd = open("/tmp/fifo", O_WRONLY);
```

открывает канал FIFO для записи. Вызов `open` будет заблокирован до тех пор, пока другой процесс не откроет канал FIFO для чтения (конечно же, если канал FIFO уже был открыт для чтения, то возврат из вызова `open` произойдет немедленно).

Можно выполнить не блокирующий вызов `open` для канала FIFO. Для этого во время вызова должен быть установлен флаг `O_NONBLOCK` (определенный в файле `<fcntl.h>`) и один из флагов `O_RDONLY` или `O_WRONLY`, например:

```
if((fd = open("/tmp/fifo", O_WRONLY | O_NONBLOCK)) == -1)
    perror("Ошибка вызова open для канала FIFO");
```

Если не существует процесс, в котором канал FIFO открыт для чтения, то этот вызов `open` вернет значение `-1` вместо блокировки выполнения, а переменная `errno` будет содержать значение `ENXIO`. В случае же успешного вызова `open` последующие вызовы `write` для канала FIFO также будут не блокирующими.

Наступило время привести пример. Представим две программы, которые показывают, как можно использовать канал FIFO для реализации системы обмена сообщениями. Эти программы используют тот факт, что вызовы `read` или `write` для каналов FIFO, как и для программных каналов, являются неделимыми (для небольших порций данных). Если при помощи канала FIFO пересылаются сообщения фиксированного размера, то отдельные сообщения будут сохраняться, даже если несколько процессов одновременно выполняют запись в канал.

Рассмотрим вначале программу `sendmessage`, которая посылает отдельные сообщения в канал FIFO с именем `fifo`. Она вызывается следующим образом:

```
$ sendmessage 'текст сообщения 1' 'текст сообщения 2'
```

Обратите внимание на то, что каждое сообщение заключено в кавычки и поэтому считается просто одним длинным аргументом. Если не сделать этого, то каждое слово будет рассматриваться, как отдельное сообщение. Программа `sendmessage` имеет следующий исходный текст:

```
/* Программа sendmessage -- пересылка сообщений через FIFO */
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
```



```
#define MSGSIZ      63

char *fifo = "fifo";

main(int argc, char **argv)
{
    int fd, j, nwrite;
    char msgbuf[MSGSIZ+1];

    if(argc < 2)
    {
        fprintf(stderr, "Описание: sendmessage сообщение \n");
        exit(1);
    }

    /* Открыть канал fifo, установив флаг O_NONBLOCK */
    if((fd = open(fifo, O_WRONLY | O_NONBLOCK)) < 0)
        fatal("Ошибка вызова open для fifo");

    /* Посылка сообщений */
    for( j=1; j< argc; j++)
    {
        if(strlen(argv[j]) > MSGSIZ)
        {
            fprintf(stderr, "Слишком длинное сообщение %s\n",
                argv[j]);
            continue;
        }

        strcpy(msgbuf, argv[j]);

        if((nwrite = write (fd, msgbuf, MSGSIZ+1)) == -1)
            fatal("Ошибка записи сообщения");
    }
    exit(0);
}
```

И снова для вывода сообщений об ошибках использована процедура `fatal`. Сообщения посылаются блоками по 64 байта при помощи не блокируемого вызова `write`. В действительности текст сообщения ограничен 63 символами, а последний символ является нулевым.

Программа `rcvmessage` принимает сообщения при помощи чтения из канала FIFO. Она не выполняет никаких полезных действий и служит только демонстрационным примером:

```
/* Программа rcvmessage — получение сообщений из канала fifo */
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#define MSGSIZ      63

char *fifo = "fifo";
```

```
main(int argc, char **argv)
{
    int fd;
    char msgbuf[MSGSZ+1];

    /* Создать канал fifo, если он еще не существует */
    if(mkfifo(fifo, 0666) == -1 )
    {
        if(errno != EEXIST)
            fatal("Ошибка приемника: вызов mkfifo");
    }

    /* Открыть канал fifo для чтения и записи */
    if((fd = open (fifo, O_RDWR)) < 0)
        fatal("Ошибка при открытии канала fifo");

    /* Прием сообщений */
    for(;;)
    {
        if(read(fd, msgbuf, MSGSZ+1) < 0)
            fatal("Ошибка при чтении сообщения");

        /*
         * Вывести сообщение; в настоящей программе
         * вместо этого могут выполняться какие-либо
         * полезные действия.
         */

        printf("Получено сообщение:%s\n", msgbuf);
    }
}
```

Обратите внимание на то, что канал FIFO открывается одновременно для чтения и записи (при помощи задания флага `O_RDWR`). Чтобы понять, для чего это сделано, предположим, что канал FIFO был открыт только для чтения при помощи задания флага `O_RDONLY`. Тогда выполнение программы `rcvmessage` будет сразу заблокировано в момент вызова `open`. Когда после старта программы `sendmessage` в канал FIFO будет произведена запись, вызов `open` будет разблокирован, программа `rcvmessage` будет читать все посылаемые сообщения. Когда же канал FIFO станет пустым, а процесс `sendmessage` завершит работу, вызов `read` начнет возвращать нулевое значение, так как канал FIFO уже не будет открыт на запись ни в одном процессе. При этом программа `rcvmessage` войдет в бесконечный цикл. Использование флага `O_RDWR` позволяет гарантировать, что, по крайней мере, в одном процессе, то есть самом процессе программы `rcvmessage`, канал FIFO будет открыт для записи. В результате вызов `open` всегда будет блокироваться то тех пор, пока в канал FIFO снова не будут записаны данные.

Следующий диалог показывает, как можно использовать эти две программы. Программа `rcvmessage` выполняется в фоновом режиме для получения сообщений от разных процессов, выполняющих программу `sendmessage`.

```
$ rcvmessage &  
40  
$ sendmessage 'сообщение 1' 'сообщение 2'  
Получено сообщение: сообщение 1  
Получено сообщение: сообщение 2  
$ sendmessage 'сообщение номер 3'  
Получено сообщение: сообщение номер 3
```

---

**Упражнение 7.6.** Программы `sendmessage` и `rcvmessage` образуют основу простой системы обмена данными. Сообщения, посылаемые программой `rcvmessage`, могут, например, быть именами файлов, которые нужно обработать. Проблема заключается в том, что текущие каталоги программ `sendmessage` и `rcvmessage` могут быть различными, поэтому относительные пути будут восприняты неправильно. Как можно разрешить эту проблему? Можно ли создать, скажем, спулер печати в большой системе, используя только каналы FIFO?

---

---

**Упражнение 7.7.** Если программу `rcvmessage` нужно сделать настоящей серверной программой, то потребуется гарантия того, что в произвольный момент времени выполняется только одна копия сервера. Существует несколько способов достичь этого. Один из методов состоит в создании файла блокировки. Рассмотрим следующую процедуру:

```
#include <errno.h>  
#include <fcntl.h>  
  
extern int errno;  
char * lck = "/tmp/lockfile";  
  
int makelock(void)  
{  
    int fd;  
    if((fd = open(lck, O_RDWR | O_CREAT | O_EXCL, 0600)) <  
0)  
    {  
        if(errno == EEXIST)  
            exit(1);          /* Файл занят другим процессом */  
        else  
            exit(127);        /* Неизвестная ошибка */  
    }  
    /* Файл блокировки создан, выход из процедуры */  
    close(fd);  
    return(0);  
}
```

*Эта процедура использует тот факт, что вызов open осуществляется за один шаг. Поэтому, если несколько процессов пытаются выполнить процедуру makelock, одному из них это удастся первым, и он создаст файл блокировки и «заблокирует» работу остальных. Добавьте эту процедуру к программе sendmessage. При этом, если выполнение программы sendmessage завершается при помощи сигнала SIGHUP или SIGTERM, то она должна удалять файл блокировки перед выходом. Как вы думаете, почему мы использовали в процедуре makelock вызов open, а не creat?*

---



# Глава 8. Дополнительные методы межпроцессного взаимодействия

## 8.1. Введение

Используя средства из глав 6 и 7, можно осуществить основные взаимодействия между процессами. В этой главе рассматриваются усовершенствованные средства межпроцессного взаимодействия, которые позволят использовать более сложные методы программирования.

Первая и наиболее простая тема данной главы – *блокировка записей* (record locking), которая фактически является не формой прямого межпроцессного взаимодействия, а скорее – методом координирования работы процессов. (Первоначально предполагалось рассмотреть этот вопрос в одной из глав, посвященных файловому вводу/выводу, но затем было решено поместить эту тему в данную главу.) Блокировка позволяет процессу временно резервировать часть файла для исключительного использования при решении некоторых сложных задач управления базами данных. Здесь стоит сделать предупреждение: спецификация XSI определяет блокировку записей как *рекомендательную* (advisory), означающую, что она не препятствует непосредственному выполнению операций файлового ввода/вывода, а вся ответственность за проверку установленных блокировок полностью ложится на процесс.<sup>1</sup>

Другие механизмы межпроцессного взаимодействия, обсуждаемые в этой главе, являются более редкими. В общем случае эти средства описываются как *средства IPC* (IPC facilities, где сокращение IPC означает *inter-process communication* – межпроцессное взаимодействие). Этот общий термин подчеркивает общность их применения и структуры, хотя существуют три определенных типа таких средств:

- *очереди сообщений* (message passing). Они позволяют процессу посылать и принимать сообщения, под которыми понимается произвольная последовательность байтов или символов;
- *семафоры* (semaphores). По сравнению с очередями сообщений семафоры представляют собой низкоуровневый метод синхронизации процессов, малоприспособленный для передачи больших объемов данных. Их теория берет начало из работ Дейкстры (E. W. Dijkstra, 1968);
- *разделяемая память* (shared memory). Это средство межпроцессного взаимодействия позволяет двум и более процессам совместно использовать данные,

---

<sup>1</sup> В некоторых версиях UNIX есть также возможность применения *обязательных блокировок* (mandatory lock). – Прим. науч. ред.

содержащиеся в определенных сегментах памяти. Естественно, обычно данные процесса являются недоступными для других процессов. Этот механизм обычно является самым быстрым механизмом межпроцессного взаимодействия.<sup>1</sup>

## 8.2. Блокировка записей

### 8.2.1. Мотивация

На первом этапе стоит рассмотреть простой пример демонстрации того, почему блокировка записей необходима в некоторых ситуациях.

Примером будет служить известная корпорация ACME Airlines, использующая ОС UNIX в своей системе заказа билетов. Она имеет два офиса, А и В, в каждом из которых установлен терминал, подключенный к компьютеру авиакомпании. Служащие компании используют для доступа к базе данных, реализованной в виде обычного файла UNIX, программу *acmebook*. Эта программа позволяет пользователю выполнять чтение и обновление базы данных. В частности, служащий может уменьшить на единицу число свободных мест при заказе билета на определенный авиарейс.

Предположим теперь, что осталось всего одно свободное место на рейс АСМ501 в Лондон, и миссис Джонс входит в офис А; в то же самое время мистер Смит входит в офис В, и они оба заказывают место на рейс АСМ501. При этом возможна такая последовательность событий:

1. Служащий офиса А запускает программу *acmebook*. Назовем стартовавший процесс РА.
2. Сразу же после этого служащий в офисе В также запускает программу *acmebook*. Назовем этот процесс РВ.
3. Процесс РА считывает соответствующую часть базы данных при помощи системного вызова *read* и определяет, что осталось всего одно свободное место.
4. Процесс РВ выполняет чтение из базы данных сразу же после процесса РА и также определяет, что осталось одно свободное место на рейс АСМ501.
5. Процесс РА обнуляет счетчик свободных мест для рейса при помощи системного вызова *write*, изменяя соответствующую часть базы данных. Служащий в офисе А вручает билет миссис Джонс.
6. Сразу же вслед за этим процесс РВ также выполняет запись в базу данных, также записывая нулевое значение в счетчик свободных мест. Но на этот раз значение счетчика ошибочно – на самом деле оно должно было бы быть равно  $-1$ , то есть хотя процесс РА уже обновил базу данных, процесс РВ не знает об этом и спешит выполнить заказ, как если бы место было свободно.

<sup>1</sup> Три упомянутые средства часто называют *System V IPC*, поскольку впервые это семейство межпроцессных взаимодействий было введено в диалекте System V Unix. Стандарт POSIX 1003.1 их не описывает; более того, аналогичные POSIX-средства имеют другой интерфейс и семантику. Тем не менее *System V IPC* внесены в спецификацию SUSV2 как заимствованные из второй версии спецификации SVID. – Прим. науч. ред.

Вследствие этого мистер Смит также получит билет, и на самолет будет продано больше билетов, чем число свободных мест в нем.

Эта проблема возникает из-за того, что несколько процессов могут одновременно обращаться к файлу UNIX. Комплексная операция с данными файла, состоящая из нескольких вызовов `lseek`, `read` и `write`, может быть выполнена двумя или более процессами одновременно, и это, как показывает наш простой пример, будет иметь непредвиденные последствия.

Одно из решений состоит в том, чтобы разрешить процессу выполнить *блокировку* (`lock`) части файла, с которой он работает. Блокировка, которая несколько не изменяет содержимое файла, показывает другим процессам, что данные, о которых идет речь, уже используются. Это предотвращает вмешательство другого процесса во время последовательности дискретных физических операций, образующих одну комплексную операцию, или транзакцию. Этот механизм часто называют *блокировкой записи* (`record locking`), где запись означает просто произвольную часть файла. Для обеспечения корректности сама операция блокировки должна быть атомарной, чтобы она не могла пересечься с параллельной попыткой блокировки в другом процессе.

Для обеспечения нормальной работы блокировка должна выполняться централизованно. Возможно, лучше всего это возложить на ядро, хотя пользовательский процесс, выступающий в качестве агента базы данных, также может служить для этой цели. Блокировка записей на уровне ядра может выполняться при помощи уже известного нам вызова `fcntl`.

Обратите внимание, что первое издание этой книги включало также альтернативный способ блокировки записей – при помощи процедуры `lockf`. Этот подход все еще встречается во многих системах – за дополнительными сведениями следует обратиться к справочному руководству системы.

### 8.2.2. Блокировка записей при помощи вызова `fcntl`

О системном вызове управления файловым вводом/выводом `fcntl` уже упоминалось ранее. В дополнение к привычным функциям вызов `fcntl` может также использоваться для выполнения блокировки записей. Он предлагает два типа блокировки:

- *блокировка чтения* (`read locks`) – просто предотвращает установку другими процессами блокировки записи при помощи вызова `fcntl`. Несколько процессов могут одновременно выполнять блокировку чтения для одного и того же участка файла. Блокировка чтения может быть полезной, если, например, требуется предотвратить обновление данных, не скрывая их от просмотра другими пользователями;
- *блокировка записи* (`write locks`) – предотвращает установку другими процессами блокировку чтения или записи для файла. Другими словами, для заданного участка файла может существовать только одна блокировка записи одновременно. Блокировка записи может использоваться, например, для скрытия участков файла от просмотра при выполнении обновления.

Следует напомнить, что в соответствии со спецификацией XSI блокировка вызовом `fcntl` является всего лишь рекомендательной. Поэтому процессам необходимо явно согласовывать свои действия, чтобы блокировка вызовом `fcntl` была действенной (процессы не должны производить операции ввода/вывода без предварительного блокирования соответствующей области).

Для блокировки записей вызов `fcntl` используется следующим образом:

### Описание

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

```
int fcntl(int filedес, int cmd, struct flock *ldata);
```

Как обычно, аргумент `fileдес` должен быть допустимым дескриптором открытого файла. Для блокировки чтения дескриптор `fileдес` должен быть открыт при помощи флагов `O_RDONLY` или `O_RDWR`, поэтому в качестве него не подойдет дескриптор, возвращаемый вызовом `creat`. Для блокировки записи дескриптор `fileдес` должен быть открыт при помощи флагов `O_WRONLY` или `O_RDWR`.

Как уже упоминалось, параметр вызова `cmd` определяет выполняемое действие, кодируемое одним из значений, определенных в файле `<fcntl.h>`. Следующие три команды относятся к блокировке записей:

<code>F_GETLK</code>	Получить описание блокировки на основе данных, передаваемых в аргументе <code>ldata</code> . (Возвращаемая информация описывает первую блокировку, которая препятствует наложению блокировки, описанной структурой <code>ldata</code> )
<code>F_SETFLK</code>	Попытаться применить блокировку к файлу и немедленно вернуть управление, если это невозможно. Используется также для удаления активной блокировки
<code>F_SETLKW</code>	Попытаться применить блокировку к файлу и приостановить работу, если блокировка уже наложена другим процессом. Ожидание процесса внутри вызова <code>fcntl</code> можно прервать при помощи сигнала

Структура `ldata` содержит описание блокировки. Структура `flock` определена в файле `<fcntl.h>` и включает следующие элементы:

```
short  l_type;      /* Описывает тип блокировки */
short  l_whence;    /* Тип смещения, как и в вызове lseek */
off_t  l_start;     /* Смещение в байтах */
off_t  l_len;       /* Размер сегмента в байтах */
pid_t  l_pid;       /* Устанавливается командой F_GETLK */
```

Три элемента, `l_whence`, `l_start` и `l_len`, определяют участок файла, который будет заблокирован, проверен или разблокирован. Переменная `l_whence` идентична третьему аргументу вызова `lseek`. Она принимает одно из трех значений: `SEEK_SET`, `SEEK_CUR` или `SEEK_END`, обозначая, что смещение должно вычисляться от начала файла, от текущей позиции указателя чтения-записи или от конца файла. Элемент `l_start` устанавливает начальное положение участка файла по отношению к точке, заданной элементом `l_whence`. Элемент `l_len` является



длиной участка в байтах; нулевое значение обозначает участок с заданной начальной позиции до максимально возможного смещения. На рис. 8.1. показано, как это работает для случая, если значение поля `l_whence` равно `SEEK_CUR`. Структура `semid_ds` определена в файле `<sys/sem.h>`.

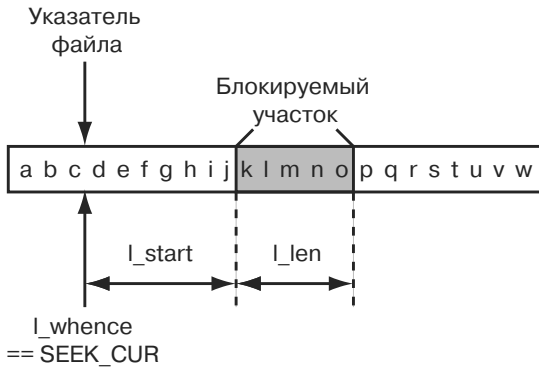


Рис. 8.1  
Параметры блокировки

Тип `l_type` определяет тип блокировки. Он может принимать одно из трех значений, определенных в файле `<fcntl.h>`:

<code>F_RDLCK</code>	Выполняется блокировка чтения
<code>F_WRLCK</code>	Выполняется блокировка записи
<code>F_UNLCK</code>	Снимается блокировка заданного участка

Поле `l_pid` существенно только при выборе команды `F_GETLK` в вызове `fcntl`. Если существует блокировка, препятствующая установке блокировки, описанной полями структуры `ldata`, то значение поля `l_pid` будет равно значению идентификатора процесса, установившего ее. Другие элементы структуры также будут переустановлены системой. Они будут содержать параметры блокировки, наложенной другим процессом.

### Установка блокировки при помощи вызова `fcntl`

Следующий пример показывает, как можно использовать вызов `fcntl` для установления блокировки записи.

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

.
.
.

struct flock my_lock;

my_lock.l_type = F_WRLCK;
my_lock.l_whence = SEEK_CUR;
my_lock.l_start = 0;
my_lock.l_len = 512;
fcntl(fd, F_SETLKW, &my_lock);
```

При этом будут заблокированы 512 байт, начиная с текущего положения указателя чтения-записи. Заблокированный участок теперь считается «зарезервированным» для исключительного использования процессом. Информация о блокировке помещается в свободную ячейку системной таблицы блокировок.

Если весь участок файла или какая-то его часть уже были заблокированы другим процессом, то вызывающий процесс будет приостановлен до тех пор, пока не будет доступен весь участок. Приостановка работы процесса может быть прервана при помощи сигнала; в частности, для задания времени ожидания может быть использован вызов `alarm`. Если ожидание не будет прервано, и, в конце концов, участок файла освободится, то процесс заблокирует его. Если происходит ошибка, например, если вызову `fcntl` передается неверный дескриптор файла или переполнится системная таблица блокировок, то будет возвращено значение `-1`.

Следующий пример – программа `lockit` открывает файл с именем `locktest` (который должен существовать) и блокирует его первые десять байт при помощи вызова `fcntl`. Затем она порождает дочерний процесс, пытающийся заблокировать первые пять байт файла; родительский процесс в это время делает паузу на пять секунд, а затем завершает работу. В этот момент система автоматически снимает блокировку, установленную родительским процессом.

```
/* Программа lockit – блокировка при помощи вызова fcntl */
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
    int fd;
    struct flock my_lock;

    /* Установка параметров блокировки записи */
    my_lock.l_type = F_WRLCK;
    my_lock.l_whence = SEEK_SET;
    my_lock.l_start = 0;
    my_lock.l_len = 10;

    /* Открыть файл */
    fd = open("locktest", O_RDWR);

    /* Заблокировать первые десять байт */
    if( fcntl(fd, F_SETLKW, &my_lock) == -1)
    {
        perror("родительский процесс: ошибка блокирования");
        exit(1);
    }

    printf("Родительский процесс: блокировка установлена\n");

    switch(fork()){
        case -1:          /* Ошибка */
            perror("Ошибка вызова fork");
            exit(1);
```

```
case 0:          /* Родительский процесс */
my_lock.l_len = 5;
if( fcntl(fd, F_SETLKW, &my_lock) == -1)
{
    perror("Дочерний процесс: установка блокировки");
    exit(1);
}
printf("Дочерний процесс: блокировка установлена\n");
printf("Дочерний процесс: выход\n");
exit(0);
}

sleep (5);

/* Выход, который автоматически снимет блокировку */
printf("Родительский процесс: выход\n");
exit(0);
}
```

Вывод программы lockit может выглядеть примерно так:

```
Родительский процесс: блокировка установлена
Родительский процесс: выход
Дочерний процесс: блокировка установлена
Дочерний процесс: выход
```

Обратите внимание на порядок, в котором выводятся сообщения. Он показывает, что дочерний процесс не может наложить блокировку до тех пор, пока родительский процесс не завершит работу и не снимет тем самым блокировку, в противном случае сообщение Дочерний процесс: блокировка установлена появилось бы вторым, а не третьим. Этот пример показывает, что блокировка, наложенная родительским процессом, также затронула и дочерний, хотя блокируемые участки файла и не совсем совпадали. Другими словами, попытка блокировки завершится неудачей, даже если участок файла лишь частично перекрывает уже заблокированный. Эта программа иллюстрирует несколько интересных моментов. Во-первых, блокировка информации не наследуется при вызове `fork`; дочерний и родительский процессы в данном примере выполняют блокировку независимо друг от друга. Во-вторых, вызов `fcntl` не изменяет положение указателя чтения-записи файла – во время выполнения обоих процессов он указывает на начало файла. В-третьих, все связанные с процессом блокировки автоматически снимаются при его завершении.

### **Снятие блокировки при помощи вызова `fcntl`**

Можно разблокировать участок файла, который был ранее заблокирован, присвоив при вызове переменной `l_type` значение `F_UNLK`. Снятие блокировки обычно используется через некоторое время после предыдущего запроса `fcntl`. Если еще какие-либо процессы собирались заблокировать освободившийся участок, то один из них прекратит ожидание и продолжит свою работу.

Если участок, с которого снимается блокировка, находится в середине большого заблокированного этим же процессом участка файла, то система создаст две

блокировки меньшего размера, исключая освобождаемый участок. Это означает, что при этом занимается еще одна ячейка в системной таблице блокировок. Вследствие этого запрос на снятие блокировки может завершиться неудачей из-за переполнения системной таблицы, хотя поначалу этого не видно.

Например, в предыдущей программе `lockit` родительский процесс снял блокировку в момент выхода из программы, но вместо этого он мог осуществить эту операцию при помощи следующего кода:

```
/* Родительский процесс снимает блокировку перед выходом */
printf("Родительский процесс: снятие блокировки\n");
my_lock.l_type = F_UNLCK;
if( fcntl(fd, F_SETLK, &my_lock) == -1)
{
    perror("ошибка снятия блокировки в родительском процессе");
    exit(1);
}
```

### ***Задача об авиакомпании ACME Airlines***

Теперь удастся разрешить конфликтную ситуацию в примере с авиакомпанией ACME Airlines. Для того, чтобы гарантировать целостность базы данных, нужно построить критический участок кода в программе `acmebook` следующим образом:

заблокировать соответствующий участок базы данных на запись

обновить участок базы данных

разблокировать участок базы данных

Если ни одна программа не обходит механизм блокировки, то запрос на блокировку гарантирует, что вызывающий процесс имеет исключительный доступ к критической части базы данных. Запрос на снятие блокировки снова делает доступной эту область для общего использования. Выполнение любой конкурирующей копии программы `acmebook`, которая пытается получить доступ к соответствующей части базы данных, будет приостановлено на стадии попытки наложения блокировки.

Текст критического участка программы можно реализовать следующим образом:

```
/* набросок процедуры обновления программы acmebook */
struct flock db_lock;

.
.
.

/* Установить параметры блокировки */
db_lock.l_type F_WRLCK;
db_lock.l_whence SEEK SET;
db_lock.l_start recstart;
db_lock.l_len RECSIZE,

.
.
.
```

```
/* Заблокировать запись в базе, выполнение приостановится, */  
/* если запись уже заблокирована */  
if( fcntl(fd, F_SETLKW, &db_lock) == -1)  
    fatal("Ошибка блокировки"),  
  
/* Код для проверки и обновления данных о заказах */  
.  
.  
.  
/* Освободить запись для использования другими процессами */  
db_lock.l_type = F_UNLCK;  
fcntl(fd, F_SETLK, &db_lock);
```

### **Проверка блокировки**

При неудачной попытке программы установить блокировку, задав параметр `F_SETLK` в вызове `fcntl`, вызов вернет значение `-1` и установит значение переменной `errno` равным `EAGAIN` или `EACCESS` (в спецификации XSI определены оба эти значения). Если блокировка уже существует, то с помощью команды `F_GETLK` можно определить процесс, установивший эту блокировку:

```
#include <unistd.h>  
#include <stdio.h>  
#include <errno.h>  
  
.  
.  
.  
  
if( fcntl(fd, F_SETLK, &alock) == 1)  
{  
    if(errno == EACCES || errno == EAGAIN)  
    {  
        fcntl(fd, F_GETLK, &b_lock);  
        fprintf(stderr, "Запись заблокирована процессом %d\n", b_lock.l_pid);  
    }  
    else  
        perror("Ошибка блокировки");  
}
```

### **Клинч**

Предположим, что два процесса, PA и PB, работают с одним файлом. Допустим, что процесс PA блокирует участок файла SX, а процесс PB – не пересекающийся с ним участок SY. Пусть далее процесс PA попытается заблокировать участок SY при помощи команды `F_SETLKW`, а процесс PB попытается заблокировать участок SX, также используя команду `F_SETLKW`. Ни одна из этих попыток не будет успешной, так как процесс PA приостановит работу, ожидая, когда процесс PB освободит участок SY, а процесс PB также будет приостановлен в ожидании освобождения участка SX процессом PA. Если не произойдет вмешательства извне, то будет казаться, что два процесса обречены вечно находиться в этом «смертельном объятии».

Такая ситуация называется *клинчем* (deadlock) по очевидным причинам. Однако UNIX иногда предотвращает возникновение клинча. Если выполнение запроса `F_SETLK` приведет к очевидному возникновению клинча, то вызов завершается неудачей, и возвращается значение `-1`, а переменная `errno` принимает значение `EDEADLK`. К сожалению, вызов `fcntl` может определять только клинч между двумя процессами, в то время как можно создать трехсторонний клинч.<sup>1</sup> Во избежание такой ситуации сложные приложения, использующие блокировки, должны всегда задавать предельное время ожидания.

Следующий пример поможет пояснить изложенное. В точке `/*A*/` программа блокирует с 0 по 9 байты файла `locktest`. Затем программа порождает дочерний процесс, который в точках, помеченных как `/*B*/` и `/*C*/`, блокирует байты с 10 по 14 и пытается выполнить блокировку байтов с 0 по 9. Из-за того, что родительский процесс уже выполнил последнюю блокировку, работа дочернего будет приостановлена. В это время родительский процесс выполняет вызов `sleep` в течение 10 секунд. Предполагается, что этого времени достаточно, чтобы дочерний процесс выполнил два вызова, устанавливающие блокировку. После того, как родительский процесс продолжит работу, он пытается заблокировать байты с 10 по 14 в точке `/*D*/`, которые уже были заблокированы дочерним процессом. В этой точке возникнет опасность клинча, и вызов `fcntl` завершится неудачей.

`/* Программа deadlock — демонстрация клинча */`

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

main()
{
    int fd;
    struct flock first_lock;
    struct flock second_lock;

    first_lock.l_type = F_WRLCK;
    first_lock.l_whence = SEEK_SET;
    first_lock.l_start = 0;
    first_lock.l_len = 10;

    second_lock.l_type = F_WRLCK;
    second_lock.l_whence = SEEK_SET;
    second_lock.l_start = 10;
    second_lock.l_len = 5;

    fd = open("locktest", O_RDWR);

    if( fcntl(fd, F_SETLKW, &first_lock) == -1)    /*A*/
        fatal("A");

    printf("A: успешная блокировка (процесс %d)\n", getpid());
```

<sup>1</sup> Клинч иногда является следствием гораздо более запутанной ситуации, если установлению блокировки препятствуют одновременно несколько других блокировок. — *Прим. науч. ред.*

```
switch(fork()){
case -1:
    /* Ошибка */
    fatal("Ошибка вызова fork");
case 0:
    /* Дочерний процесс */
    if( fcntl(fd, F_SETLKW, &second_lock) == -1)    /*B*/
        fatal("B");
    printf("B: успешная блокировка (процесс %d)\n", getpid());
    if( fcntl(fd, F_SETLKW, &first_lock) == -1)    /*C*/
        fatal("C");
    printf("C: успешная блокировка (процесс %d)\n", getpid());
    exit(0);
default:
    /* Родительский процесс */
    printf("Приостановка родительского процесса\n");
    sleep(10);
    if( fcntl(fd, F_SETLKW, &second_lock) == -1)    /*D*/
        fatal("D");
    printf("D: успешная блокировка (процесс %d)\n", getpid());
}
```

Вот пример работы этой программы:

```
A: успешная блокировка (процесс 1410)
Приостановка родительского процесса
B: успешная блокировка (процесс 1411)
D: Deadlock situation detected/avoided
C: успешная блокировка (процесс 1411)
```

В данном случае попытка блокировки завершается неудачей в точке `/*D*/`, и процедура `error` выводит соответствующее системное сообщение об ошибке. Обратите внимание, что после того, как родительский процесс завершит работу и его блокировки будут сняты, дочерний процесс сможет выполнить вторую блокировку.

Этот пример использует процедуру `fatal`, которая была применена в предыдущих главах.

---

**Упражнение 8.1.** Напишите процедуры, выполняющие те же действия, что и вызовы `read` и `write`, но которые завершатся неудачей, если уже установлена блокировка нужного участка файла. Измените аналог вызова `read` так, чтобы он блокировал читаемый участок. Блокировка должна сниматься после завершения вызова `read`.

---

**Упражнение 8.2.** Придумайте и реализуйте условную схему блокировок нумерованных логических записей файла. (Совет: можно блокировать

*участки файла вблизи максимально возможного смещения файла, даже если там нет данных. Блокировки в этом участке файла могут иметь особое значение, например, каждый байт может соответствовать определенной логической записи. Блокировка в этой области может также использоваться для установки различных флагов.)*

## 8.3. Дополнительные средства межпроцессного взаимодействия

### 8.3.1. Введение и основные понятия

ОС UNIX предлагает множество дополнительных механизмов межпроцессного взаимодействия. Их наличие дает UNIX богатые возможности в области связи между процессами и позволяет разработчику использовать различные подходы при программировании многозадачных систем. Дополнительные средства межпроцессного взаимодействия, которые будут рассмотрены, можно разбить на следующие категории:

- передача сообщений;
- семафоры;
- разделяемая память.

Эти средства широко применяются и ведут свое начало от системы UNIX System V, поэтому их иногда называют *IPC System V*. Следует заметить, что выпшеназванные дополнительные средства были определены в последних версиях стандарта POSIX.<sup>1</sup>

#### **Ключи средств межпроцессного взаимодействия**

Программный интерфейс всех трех средств *IPC System V* однороден, что отражает схожесть их реализации в ядре. Наиболее важным из общих свойств является *ключ* (key). Ключи – это числа, обозначающие объект межпроцессного взаимодействия в системе UNIX примерно так же, как имя файла обозначает файл. Другими словами, ключ позволяет ресурсу межпроцессного взаимодействия совместно использоваться несколькими процессами. Обозначаемый ключом объект может быть очередью сообщения, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип `key_t`, состав которого зависит от реализации и определяется в системном заголовочном файле `<sys/types.h>`.

Ключи не являются именами файлов и несут меньший смысл. Они должны выбираться осторожно во избежание конфликта между различными программами, в этом помогает применение дополнительной опции – «версии проекта». (Одна известная система управления базами данных использовала для ключа шестнадцатеричное значение типа `0xDB`: плохое решение, так как такое же значение мог выбрать и другой разработчик.) В ОС UNIX существует простая библиотечная функция `ftok`, которая образует ключ по указанному файлу.

<sup>1</sup> В базовом документе POSIX 1003.1 средства *IPC System V* не вводятся. – Прим. науч. ред.



### Описание

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

```
key_t ftok(const char *path, int id);
```

Эта процедура возвращает номер ключа на основе информации, связанной с файлом `path`. Параметр `id` также учитывается и обеспечивает еще один уровень уникальности – «версию проекта»; другими словами, для одного имени `path` будут получены разные ключи при разных значениях `id`. Процедура `ftok` не слишком удобна: например, если удалить файл, а затем создать другой с таким же именем, то возвращаемый после этого ключ будет другим. Она завершится неудачей и вернет значение (`key_t`) `-1` и в случае, если файл `path` не существует. Процедуру `ftok` можно применять в приложениях, использующих функции межпроцессного взаимодействия для работы с определенными файлами или при применении для генерации ключа файла, являющегося постоянной и неотъемлемой частью приложения.

### Операция *get*

Программа применяет ключ для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции вызываются при помощи операции *get*. Результатом операции *get* является его целочисленный *идентификатор* (*facility identifier*), который может использоваться при вызовах других процедур межпроцессного взаимодействия. Если продолжить аналогию с именами файлов, то операция *get* похожа на вызов `creat` или `open`, а идентификатор средства межпроцессного взаимодействия ведет себя подобно дескриптору файла. В действительности, в отличие от дескрипторов файла, идентификатор средства межпроцессного взаимодействия является уникальным. Различные процессы будут использовать одно и то же значение идентификатора для объекта межпроцессного взаимодействия.

В качестве примера рассмотрим вызов `msgget` для создания новой очереди сообщений (что представляет собой очередь сообщений, обсудим позже):

```
mqid = msgget( (key_t)0100, 0644 | IPC_CREAT | IPC_EXCL);
```

Первый аргумент вызова, `msgget`, является ключом очереди сообщений. В случае успеха процедура вернет неотрицательное значение в переменной `mqid`, которая служит идентификатором очереди сообщений. Соответствующие вызовы для семафоров и объектов разделяемой памяти называются соответственно `semget` и `shmget`.

### Другие операции

Есть еще два типа операций, которые применимы к средствам межпроцессного взаимодействия. Во-первых, это операции управления, которые используются для опроса и изменения статуса объекта ИРС, их функции выполняют вызовы `msgctl`, `semctl` и `shmctl`. Во-вторых, существуют операции, выполняющие основные функции ИРС. Для каждого из средств межпроцессного взаимодействия существует набор операций, которые будут обсуждаться ниже в соответствующих пунктах. Например, есть две операции для работы с сообщениями: операция

`msgsnd` помещает сообщение в очередь сообщений, а операция `msgrcv` считывает из нее сообщение.

### Структуры данных статуса

При создании объекта межпроцессного взаимодействия система также создает *структуру статуса средства межпроцессного взаимодействия* (IPC facility status structure), содержащую всю управляющую информацию, связанную с объектом. Для сообщений, семафоров и разделяемой памяти существуют разные типы структуры статуса. Каждый тип содержит информацию, свойственную этому средству межпроцессного взаимодействия. Тем не менее все три типа структуры статуса содержат общую структуру прав доступа. Структура прав доступа `ipc_perm` содержит следующие элементы:

```
uid_t cuid; /* Идентификатор пользователя создателя объекта */
gid_t cgid; /* Идентификатор группы создателя объекта */
uid_t uid;  /* Действующий идентификатор пользователя */
gid_t gid;  /* Действующий идентификатор группы */
mode_t umode; /* Права доступа */
```

Права доступа определяют, может ли пользователь выполнять «чтение» из объекта (получать информацию о нем) или «запись» в объект (работать с ним). Коды прав доступа образуются точно таким же образом, как и для файлов. Поэтому значение 0644 для элемента `umode` означает, что владелец может выполнять чтение и запись объекта, а другие пользователи – только чтение из него. Обратите внимание, что права доступа, заданные элементом `umode`, применяются в сочетании с действующими идентификаторами пользователя и группы (записанными в элементах `uid` и `gid`).<sup>1</sup> Очевидно также, что права на выполнение в данном случае не имеют значения. Как обычно, суперпользователь имеет неограниченные полномочия. В отличие от других конструкций UNIX, значение переменной `umask` пользователя не действует при создании средства межпроцессного взаимодействия.

### 8.3.2. Очереди сообщений

Начнем подробное рассмотрение средств межпроцессного взаимодействия с примитивов очередей сообщений.

В сущности, сообщение является просто последовательностью символов или байтов (необязательно заканчивающейся нулевым символом). Сообщения передаются между процессами при помощи *очереди сообщений* (message queues), которые можно создавать или получать к ним доступ при помощи вызова `msgget`. После создания очереди процесс может помещать в нее сообщения при помощи вызова `msgsnd`, если он имеет соответствующие права доступа. Затем другой процесс может считать это сообщение при помощи примитива `msgrcv`, который извлекает сообщение из очереди. Таким образом, обработка сообщений аналогична обмену данными при помощи вызовов чтения и записи для каналов (рассмотренном в разделе 7.1.2).

<sup>1</sup> Более точно порядок разрешения доступа к объекту IPC описан в спецификации SUSV2. – Прим. науч. ред.

Функция `msgget` определяется следующим образом:

### Описание

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

```
int msgget(key_t key, int permflags)
```

Этот вызов лучше всего представить как аналог вызова `open` или `creat`. Как уже упоминалось в разделе 8.3.1, параметр `key`, который, в сущности, является простым числом, идентифицирует очередь сообщений в системе. В случае успешного вызова, после создания новой очереди или доступа к уже существующей, вызов `msgget` вернет ненулевое целое значение, которое называется *идентификатором очереди сообщений* (message queue identifier).

Параметр `permflags` указывает выполняемое вызовом `msgget` действие, которое задается при помощи двух констант, определенных в файле `<sys/ipc.h>`; они могут использоваться по отдельности или объединяться при помощи операции побитового ИЛИ:

<code>IPC_CREAT</code>	При задании этого флага вызов <code>msgget</code> создает новую очередь сообщений для данного значения, если она еще не существует. Если продолжить аналогию с файлами, то при задании этого флага вызов <code>msgget</code> выполняется в соответствии с вызовом <code>creat</code> , хотя очередь сообщений и не будет «перезаписана», если она уже существует. Если же флаг <code>IPC_CREAT</code> не установлен и очередь с этим ключом существует, то вызов <code>msgget</code> вернет идентификатор существующей очереди сообщений
<code>IPC_EXCL</code>	Если установлен этот флаг и флаг <code>IPC_CREAT</code> , то вызов предназначен только для создания очереди сообщений. Поэтому, если очередь с ключом <code>key</code> уже существует, то вызов <code>msgget</code> завершится неудачей и вернет значение <code>-1</code> . Переменная <code>errno</code> будет при этом содержать значение <code>EEXIST</code>

При создании очереди сообщений младшие девять бит переменной `permflags` используются для задания прав доступа к очереди сообщений аналогично коду доступа к файлу. Они хранятся в структуре `ipc_perm`, создаваемой одновременно с самой очередью.

Теперь можно вернуться к примеру из раздела 8.3.1.

```
mqid = msgget( (key_t)0100, 0644 | IPC_CREAT | IPC_EXCL);
```

Этот вызов предназначен для создания (и только создания) очереди сообщений для значения ключа равного `(key_t)0100`. В случае успешного завершения вызова очередь будет иметь код доступа `0644`. Этот код интерпретируется таким же образом, как и код доступа к файлу, обозначая, что создатель очереди может отправлять и принимать сообщения, а члены его группы и все остальные могут выполнять только чтение. При необходимости для изменения прав доступа или владельца очереди может использоваться вызов `msgctl`.

### Работа с очередью сообщений: примитивы *msgsnd* и *msgrcv*

После создания очереди сообщений для работы с ней могут использоваться два следующих примитива:

#### Описание

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

int msgsnd(int mqid, const void *message, size_t size,
           int flags)

int msgrcv(int mqid, void *message, size_t size, long msg_type,
           int flags)
```

Первый из вызовов, *msgsnd*, используется для добавления сообщения в очередь, обозначенную идентификатором *mqid*.

Сообщение содержится в структуре *message* – шаблоне, определенном пользователем и имеющем следующую форму:

```
struct mmsg{
    long mtype;           /* Тип сообщения */
    char mtext[SOMEVALUE]; /* Текст сообщения */
};
```

Значение поля *mtype* может использоваться программистом для разбиения сообщений на категории. При этом значимыми являются только положительные значения; отрицательные или нулевые не могут использоваться (это будет видно из дальнейшего описания операций передачи сообщений). Массив *mtext* служит для хранения данных сообщения (постоянная *SOMEVALUE* выбрана совершенно произвольно). Длина посылаемого сообщения задается параметром *size* вызова *msgsnd* и может быть в диапазоне от нуля до меньшего из двух значений *SOMEVALUE* и максимального размера сообщения, определенного в системе.

Параметр *flsgs* вызова *msgsnd* может нести только один флаг: *IPC\_NOWAIT*. При неустановленном параметре *IPC\_NOWAIT* вызывающий процесс приостановит работу, если для отправки сообщения недостаточно системных ресурсов. На практике это произойдет, если полная длина сообщений в очереди превысит максимум, заданный для очереди или всей системы. Если флаг *IPC\_NOWAIT* установлен, тогда при невозможности послать сообщение возврат из вызова произойдет немедленно. Возвращаемое значение будет равно *-1*, и переменная *errno* будет иметь значение *EAGAIN*, означающее необходимость повторения попытки.

Вызов *msgsnd* также может завершиться неудачей из-за установленных прав доступа. Например, если ни действующий идентификатор пользователя, ни действующий идентификатор группы процесса не связаны с очередью, и установлен код доступа к очереди *0660*, то вызов *msgsnd* для этой очереди завершится неудачей. Переменная *errno* получит значение *EACCESS*.

Перейдем теперь к чтению сообщений. Для чтения из очереди, заданной идентификатором *mqid*, используется вызов *msgrcv*. Чтение разрешено, если процесс имеет права доступа к очереди на чтение. Успешное чтение сообщения приводит к удалению его из очереди.

На этот раз переменная `message` используется для хранения полученного сообщения, а параметр `size` задает максимальную длину сообщений, которые могут находиться в этой структуре. Успешный вызов возвращает длину полученного сообщения.

Параметр `msg_type` определяет тип принимаемого сообщения, он помогает выбрать нужное из находящихся в очереди сообщений. Если параметр `msg_type` равен нулю, из очереди считывается первое сообщение, то есть то, которое было послано первым. При ненулевом положительном значении параметра `msg_type` считывается первое сообщение из очереди с заданным типом сообщения. Например, если очередь содержит сообщения со значениями `mtype` 999, 5 и 1, а параметр `msg_type` в вызове `msgrcv` имеет значение 5, то считывается сообщение типа 5. И, наконец, если параметр `msg_type` имеет ненулевое отрицательное значение, то считывается первое сообщение с наименьшим значением `mtype`, которое меньше или равно модулю параметра `msg_type`. Этот алгоритм кажется сложным, но выражает простое правило: если вернуться к нашему предыдущему примеру с тремя сообщениями со значениями `mtype` 999, 5 и 1, то при значении параметра `msg_type` в вызове `msgrcv` равном -999 и троекратном вызове сообщения будут получены в порядке 1, 5, 999.

Последний параметр `flags` содержит управляющую информацию. В этом параметре могут быть независимо установлены два флага – `IPC_NOWAIT` и `MSG_NOERROR`. Флаг `IPC_NOWAIT` имеет обычный смысл – если он не задан, то процесс будет приостановлен при отсутствии в очереди подходящих сообщений, и возврат из вызова произойдет после поступления сообщения соответствующего типа. Если же этот флаг установлен, то возврат из вызова при любых обстоятельствах произойдет немедленно.

При установленном флаге `MSG_NOERROR` сообщение будет усечено, если его длина больше, чем `size` байт, без этого флага попытка чтения длинного сообщения приводит к неудаче вызова `msgrcv`. К сожалению, узнать о том, что усечение имело место, невозможно.

Этот раздел может показаться сложным: формулировка средств межпроцессного взаимодействия несколько не соответствует по своей сложности и стилю природе ОС UNIX. В действительности же процедуры передачи сообщений просты в применении и имеют множество потенциальных применений, что попробуем продемонстрировать на следующем примере.

### ***Пример передачи сообщений: очередь с приоритетами***

В этом разделе разработаем простое приложение для передачи сообщений. Его целью является реализация очереди, в которой для каждого элемента может быть задан приоритет. Серверный процесс будет выбирать элементы из очереди и обрабатывать их каким-либо образом. Например, элементы очереди могут быть именами файлов, а серверный процесс может копировать их на принтер. Этот пример аналогичен примеру использования FIFO из раздела 7.2.1.

Отправной точкой будет следующий заголовочный файл `q.h`:

```
/* q.h – заголовок для примера очереди сообщений */  
  
#include <sys/types.h>  
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
#include <string.h>
#include <errno.h>

#define QKEY      (key_t)0105 /* Ключ очереди */
#define QPERM     0660        /* Права доступа */
#define MAXOBN    50          /* Макс. длина имени объекта */
#define MAXPRIOR  10          /* Максимальный приоритет */

struct q_entry{
    long mtype;
    char mtext[MAXOBN+1];
};
```

Первая часть этого файла содержит необходимые включаемые файлы. Определение `QKEY` задает значение ключа, которое будет обозначать очередь сообщений в системе. Определение `QPERM` устанавливает связанные с очередью права доступа. Так как код доступа равен `0660`, то владелец очереди и члены его группы смогут выполнять чтение и запись. Как увидим позже, определения `MAXOBN` и `MAXPRIOR` будут налагать ограничения на сообщения, помещаемые в очередь. Последняя часть этого включаемого файла содержит определение структуры `q_entry`. Структуры этого типа будут использоваться в качестве сообщений, передаваемых и принимаемых следующими процедурами.

Первая рассматриваемая процедура называется `enter`, она помещает в очередь имя объекта, заканчивающееся нулевым символом, и имеет следующую форму:

```
/* Процедура enter – поместить объект в очередь */
#include "q.h"

int enter(char *objname, int priority)
{
    int len, s_qid;
    struct q_entry s_entry; /* Структура для хранения сообщений */

    /* Проверка длины имени и уровня приоритета */
    if( (len = strlen(objname)) > MAXOBN)
    {
        warn("Слишком длинное имя");
        return (-1);
    }

    if(priority > MAXPRIOR || priority < 0)
    {
        warn("Недопустимый уровень приоритета");
        return (-1);
    }

    /* Инициализация очереди сообщений, если это необходимо */
    if( (s_qid = init_queue()) == -1)
        return (-1);

    /* Инициализация структуры s_entry */
```

```
s_entry.mtype = (long)priority;
strncpy(s_entry.mtext, objname, MAXOBN);

/* Посылаем сообщение, выполнив ожидание, если это необходимо */
if(msgsnd(s_qid, &s_entry, len, 0) == -1)
{
    perror("Ошибка вызова msgsnd");
    return (-1);
}
else
    return (0);
}
```

Первое действие, выполняемое процедурой `enter`, заключается в проверке длины имени объекта и уровня приоритета. Обратите внимание на то, что минимальное значение переменной приоритета `priority` равно 1, так как нулевое значение приведет к неудачному завершению вызова `msgsnd`. Затем процедура `enter` «открывает» очередь, вызывая процедуру `init_queue`, реализацию которой приведем позже.

После завершения этих действий процедура формирует сообщение и пытается послать его при помощи вызова `msgsnd`. Здесь для хранения сообщения использована структура `s_entry` типа `q_entry`, и последний параметр вызова `msgsnd` равен нулю. Это означает, что система приостановит выполнение текущего процесса, если очередь заполнена (так как не задан флаг `IPC_NOWAIT`).

Процедура `enter` сообщает о возникших проблемах при помощи функции `warn` или библиотечной функции `perror`. Для простоты функция `warn` реализована следующим образом:

```
#include <stdio.h>

int warn(char *s)
{
    fprintf(stderr, "Предупреждение: %s\n", s);
}
```

В реальных системах функция `warn` должна записывать сообщения в специальный файл протокола.

Назначение функции `init_queue` очевидно. Она инициализирует идентификатор очереди сообщений или возвращает идентификатор очереди сообщений, который с ней уже связан.

```
/* Инициализация очереди — получить идентификатор очереди */
#include "q.h"

int init_queue(void)
{
    int queue_id;

    /* Попытка создания или открытия очереди сообщений */
    if( (queue_id = msgget(QKEY, IPC_CREAT | QPERM)) == -1)
```

```
perror("Ошибка вызова msgget");  
return (queue_id);  
}
```

Следующая процедура, `serve`, используется серверным процессом для получения сообщений из очереди и противоположна процедуре `enter`.

/\* Процедура `serve` – принимает и обрабатывает сообщение очереди с наивысшим приоритетом \*/

```
#include "q.h"  
  
int serve(void)  
{  
    int mlen, r_qid;  
    struct q_entry r_entry;  
  
    /* Инициализация очереди сообщений, если это необходимо */  
    if((r_qid = init_queue()) == -1)  
        return (-1);  
  
    /* Получить и обработать следующее сообщение */  
    for(;;)  
    {  
        if((mlen = msgrcv(r_qid, &r_entry, MAXOBN,  
                          (-1 * MAXPRIOR), MSG_NOERROR)) == -1)  
        {  
            perror("Ошибка вызова msgrcv");  
            return (-1);  
        }  
        else  
        {  
            /* Убедиться, что это строка */  
            r_entry.mtext[mlen]='\0';  
  
            /* Обработать имя объекта */  
            proc_obj(&r_entry);  
        }  
    }  
}
```

Обратите внимание на вызов `msgrcv`. Так как в качестве параметра типа задано отрицательное значение `(-1 * MAXPRIOR)`, то система вначале проверяет очередь на наличие сообщений со значением `mtype` равным 1, затем равным 2 и так далее, до значения `MAXPRIOR` включительно. Другими словами, сообщения с наименьшим номером будут иметь наивысший приоритет. Процедура `proc_obj` работает с объектом. Для системы печати она может просто копировать файл на принтер.



Две следующих простых программы демонстрируют взаимодействие этих процедур: программа `etest` помещает элемент в очередь, а программа `stest` обрабатывает его (в действительности она всего лишь выводит содержимое и тип сообщения).

### **Программа `etest`**

```
/* Программа etest — ввод имен объектов в очередь */
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

main(int argc, char **argv)
{
    int priority;

    if(argc != 3 )
    {
        fprintf(stderr, "Применение: %s имя приоритет\n", argv[0]);
        exit(1);
    }

    if((priority = atoi(argv[2])) <= 0 || priority > MAXPRIOR)
    {
        warn("Недопустимый приоритет");
        exit(2);
    }

    if(enter(argv[1], priority) < 0)
    {
        warn("Ошибка в процедуре enter");
        exit(3);
    }

    exit(0);
}
```

### **Программа `stest`**

```
/* Программа stest — простой сервер для очереди */
#include <stdio.h>
#include "q.h"

main()
{
    pid_t pid;

    switch(pid = fork()){
        case 0:          /* Дочерний процесс */
            serve();
            break;
        /* Сервер не существует */
        case -1:
    }
```

```

warn("Не удалось запустить сервер");
break;
default:
    printf("Серверный процесс с идентификатором %d\n", pid);
}
exit(pid != -1 ? 0 : 1);
}

int proc_obj(struct q_entry *msg)
{
    printf ("\nПриоритет: %ld имя: %s\n", msg->mtype, msg->mtext);
}

```

Ниже следует пример использования этих двух простых программ. Перед запуском программы `stest` в очередь вводятся четыре простых сообщения при помощи программы `etest`. Обратите внимание на порядок, в котором выводятся сообщения:

```

$ etest objname1 3
$ etest objname2 4
$ etest objname3 1
$ etest objname4 9
$ stest
Серверный процесс с идентификатором 2545
$
Приоритет 1 имя objname3
Приоритет 3 имя objname1
Приоритет 4 имя objname2
Приоритет 9 имя objname4

```

---

**Упражнение 8.3.** Измените процедуры `enter` и `serve` так, чтобы можно было посылать серверу управляющие сообщения. Зарезервируйте для таких сообщений единственный тип сообщения (как это повлияет на расстановку приоритетов?). Реализуйте следующие возможности:

1. Остановка сервера.
  2. Стирание всех сообщений из очереди.
  3. Стирание сообщений с заданным уровнем приоритета.
- 

### Системный вызов `msgctl`

Процедура `msgctl` служит трем целям: она позволяет процессу получать информацию о статусе очереди сообщений, изменять некоторые из связанных с очередью ограничений или удалять очередь из системы.

#### Описание

```

#include <sys/msg.h>

int msgctl(int mqid, int command, struct msqid_ds *msg_stat);

```

Переменная `mqid` должна быть допустимым идентификатором очереди. Пропуская пока параметр `command`, обратимся к третьему параметру `msg_stat`, который содержит адрес структуры `msgid_ds`. Эта структура определяется в файле `<sys/msg.h>` и содержит следующие элементы:

```
struct ipc_perm msg_perm; /* Владелец/права доступа */
msgqnum_t      msg_qnum;  /* Число сообщений в очереди */
msglen_t       msg_qbytes; /* Макс. число байтов в очереди */
pid_t          msg_lspid; /* Идентификатор процесса,
                           последним вызвавшего msgsnd */
pid_t          msg_lrpid; /* Идентификатор процесса,
                           последним вызвавшего msgrcv */
time_t         msg_stime; /* Время посл. вызова msgsnd */
time_t         msg_rtime; /* Время посл. вызова msgrcv */
time_t         msg_ctime; /* Время посл. изменения */
```

Структуры `ipc_perm`, с которыми уже встречались ранее, содержат связанную с очередью информацию о владельце и правах доступа. Типы `msgqnum_t`, `msglen_t`, `pid_t` и `time_t` зависят от конкретной системы. Переменные типа `time_t` содержат число секунд, прошедшее с 00:00 по гринвичскому времени 1 января 1970 г. (Следующий пример покажет, как можно преобразовать такие значения в удобочитаемый формат.)

Параметр `command` в вызове `msgctl` сообщает системе, какую операцию она должна выполнить. Существуют три возможных значения этого параметра, каждое из которых может быть применено к одному из трех средств межпроцессного взаимодействия. Они обозначаются следующими константами, определенными в файле `<sys/ipc.h>`.

IPC_STAT	Сообщает системе, что нужно поместить информацию о статусе объекта в структуру <code>msg_stat</code>
IPC_SET	Используется для задания значений управляющих параметров очереди сообщений, содержащихся в структуре <code>msg_stat</code> . При этом могут быть изменены только следующие поля:

```
msg_stat.msg_perm.uid
msg_stat.msg_perm.gid
msg_stat.msg_perm.mode
msg_stat.msg_qbytes
```

Операция `IPC_SET` завершится успехом только в случае ее выполнения суперпользователем или текущим владельцем очереди, заданным параметром `msg_stat.msg_perm.uid`. Кроме того, только суперпользователь может увеличивать значение `msg_qbytes` – максимальное количество байтов, которое может находиться в очереди

IPC_RMID	Эта операция удаляет очередь сообщений из системы. Она также может быть выполнена только суперпользователем или владельцем очереди. Если параметр <code>command</code> принимает значение <code>IPC_RMID</code> , то параметр <code>msg_stat</code> задается равным <code>NULL</code>
----------	---

Следующий пример, программа `show_msg`, выводит часть информации о статусе объекта очереди сообщений. Программа должна вызываться так:

```
$ show_msg значение_ключа
```

Программа `show_msg` использует библиотечную процедуру `ctime` для преобразования значений структуры `time_t` в привычную запись. (Процедура `ctime` и другие функции для работы с временными значениями будут обсуждаться в главе 12.) Текст программы `show_msg`:

```
/* Программа showmsg – выводит данные об очереди сообщений */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <time.h>

void mqstat_print(key_t, int, struct msqid_ds *);

main (int argc, char **argv)
{
    key_t mkey;
    int msq_id;
    struct msqid_ds msq_status;

    if(argc != 2)
    {
        fprintf(stderr, "Применение: showmsg значение_ключа\n");
        exit(1);
    }

    /* Получаем идентификатор очереди сообщений */
    mkey = (key_t)atoi(argv[1]);
    if(( msq_id = msgget(mkey, 0)) == -1)
    {
        perror("Ошибка вызова msgget");
        exit(2);
    }

    /* Получаем информацию о статусе */
    if(msgctl(msq_id, IPC_STAT, &msq_status) == -1)
    {
        perror("Ошибка вызова msgctl"),
        exit(3);
    }

    /* Выводим информацию о статусе */
    mqstat_print(mkey, msq_id, &msq_status)
    exit(0);
}

void mqstat_print(key_t mkey, int mqid, struct msqid_ds *mstat)
{
    printf ("\nКлюч d, msg_qid %d\n\n", mkey, mqid);
```

```
printf("%d сообщений в очереди\n\n", mstat->msg_qnum)
printf("Последнее сообщение послано процессом %d в %s\n",
      mstat->msg_lspid, ctime(&(mstat->msg_stime)));
printf("Последнее сообщение принято процессом %d в %s\n",
      mstat->msg_lrpid, ctime(&(mstat->msg_rtime)));
}
```

---

**Упражнение 8.4.** Измените процедуру `show_msg` так, чтобы она выводила информацию о владельце и правах доступа очереди сообщений.

---

**Упражнение 8.5.** Взяв за основу программу `chmod`, напишите программу `msg_chmod`, которая изменяет связанные с очередью права доступа. Очередь сообщений также должна указываться значением ее ключа.

---

### 8.3.3. Семафоры

#### Семафор как теоретическая конструкция

В информатике понятие *семафор* (semaphore) было впервые введено голландским теоретиком Е.В. Дейкстрой (E.W. Dijkstra) для решения задач синхронизации процессов. Семафор *set* может рассматриваться как целочисленная переменная, для которой определены следующие операции:

```
p(sem) или wait(sem)
    if(sem !=0)
        уменьшить sem на единицу
    else
        ждать, пока sem не станет ненулевым, затем вычесть единицу
v(sem) или signal(sem)
    увеличить sem на единицу
    if(очередь ожидающих процессов не пуста)
        продолжить выполнение первого процесса в очереди ожидания
```

Обратите внимание, что обозначения *p* и *v* происходят от голландских терминов для понятий *ожидания* (wait) и *сигнализации* (signal), причем последнее понятие не следует путать с обычными сигналами UNIX.

Действия проверки и установки в обеих операциях должны составлять одно атомарное действие, чтобы только один процесс мог изменять семафор *set* в каждый момент времени.

Формально удобство семафоров заключается в том, что утверждение

```
(начальное значение семафора
+ число операций v
- число завершившихся операций p) >= 0
```

всегда истинно. Это – *инвариант семафора* (semaphore invariant). Теоретикам нравятся такие инварианты, так как они делают возможным систематическое и строгое доказательство правильности программ.

Семафоры могут использоваться несколькими способами. Наиболее простой из них заключается в обеспечении *взаимного исключения* (mutual exclusion), когда только один процесс может выполнять определенный участок кода одновременно. Рассмотрим схему следующей программы:

```
p(sem);
```

какие-либо действия

```
v(sem);
```

Предположим далее, что начальное значение семафора *sem* равно единице. Из инварианта семафора можно увидеть, что:

(число завершенных операций *p* -  
число завершенных операций *v*)  $\leq$  начального значения семафора

или:

(число завершенных операций *p* -  
число завершенных операций *v*)  $\leq 1$

Другими словами, в каждый момент времени только один процесс может выполнять группу операторов, заключенных между определенными операциями *p* и *v*. Такая область программы часто называется *критическим участком* (critical section).

Реализация семафоров в ОС UNIX основана на этой теоретической идее, хотя в действительности предлагаемые средства являются более общими (и, возможно, чрезмерно сложными). Вначале рассмотрим процедуры *semget* и *semctl*.

### Системный вызов *semget*

#### Описание

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

```
int semget(key_t key, int nsems, int permflags);
```

Вызов *semget* аналогичен вызову *msgget*. Дополнительный параметр *nsems* задает требуемое число семафоров в наборе семафоров; это важный момент – семафорные операции в System V IPC приспособлены для работы с наборами семафоров, а не с отдельными объектами семафоров. На рис. 8.2 показан набор семафоров. Ниже увидим, что использование целого набора семафоров усложняет интерфейс процедур работы с семафорами.

Значение, возвращаемое в результате успешного вызова *semget*, является *идентификатором набора семафоров* (semaphore set identifier), который ведет себя

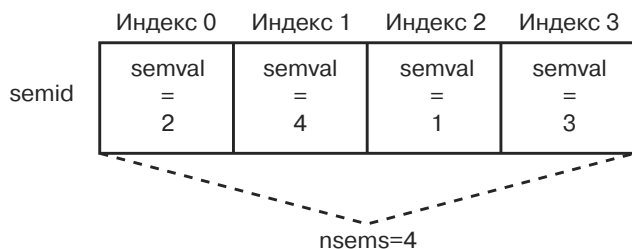


Рис. 8.2  
Набор семафоров

почти так же, как идентификатор очереди сообщений. Идентификатор набора семафоров обозначен на рис. 8.2 как `semid`. Следуя обычной практике языка C, индекс семафора в наборе может принимать значения от 0 до `nsems-1`.

С каждым семафором в наборе связаны следующие значения:

<i>semval</i>	Значение семафора, положительное целое число. Устанавливается при помощи системных вызовов работы с семафорами, то есть к значениям семафоров нельзя получить прямой доступ из программы, как к другим объектам данных
---------------	--

*sempid* Идентификатор процесса, который последним работал с семафором

<i>semcnt</i>	Число процессов, ожидающих увеличения значения семафора
---------------	---

*semzcnt* Число процессов, ожидающих обнуления значения семафора

## Системный вызов `semctl`

### Описание

```
#include <sys/sem.h>
int semctl(int semid, int sem_num, int command,
            union semun ctl_arg);
```

Из определения видно, что функция `semctl` намного сложнее, чем `msgctl`. Параметр `semid` должен быть допустимым идентификатором семафора, возвращенным вызовом `semget`. Параметр `command` имеет тот же смысл, что и в вызове `msgctl`, — задает требуемую команду. Команды распадаются на три категории: стандартные команды управления средством межпроцессного взаимодействия (такие как `IPC_STAT`); команды, которые воздействуют только на один семафор; и команды, действующие на весь набор семафоров. Все доступные команды приведены в табл. 8.1.

Таблица 8.1. Коды функций вызова *semctl*

Стандартные функции межпроцессного взаимодействия	
IPC_STAT	Поместить информацию о статусе в поле <code>ctl_arg.stat</code>
IPC_SET	Установить данные о владельце/правах доступа
IPC_RMID	Удалить набор семафоров из системы
Операции над одиночными семафорами	
(относятся к семафору <code>sem_num</code> , значение возвращается вызовом <code>semctl</code> )	
GETVAL	Вернуть значение семафора (то есть <code>setval</code> )
SETVAL	Установить значение семафора равным <code>ctl_arg.val</code>
GETPID	Вернуть значение <code>sempid</code>
GETNCNT	Вернуть <code>semncnt</code> (см. выше)
GETZCNT	Вернуть <code>semzcnt</code> (см. выше)
Операции над всеми семафорами	
GETALL	Поместить все значения <code>setval</code> в массив <code>ctl_arg.array</code>
SETALL	Установить все значения <code>setval</code> из массива <code>ctl_arg.array</code>

Параметр `sem_num` используется со второй группой возможных операций вызова `semctl` для задания определенного семафора. Последний параметр `ctl_arg` является объединением, определенным следующим образом:

```
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

Каждый элемент объединения представляет некоторый тип значения, передаваемого вызову `semctl` при выполнении определенной команды. Например, если значение `command` равно `SETVAL`, то будет использоваться элемент `ctl_arg.val`.

Одно из важных применений функции `setval` заключается в установке начальных значений семафоров, так как вызов `semget` не позволяет процессу сделать это. Приведенная в качестве примера функция `initsem` может использоваться для создания одиночного семафора и получения связанного с ним идентификатора набора семафоров. После создания семафора (если семафор еще не существовал) функция `semctl` присваивает ему начальное значение равное единице.

/\* Функция `initsem` – инициализация семафора \*/

```
#include "pv.h"

int initsem(key_t semkey)
{
    int status = 0, semid;

    if((semid = semget(semkey, 1,
        SEMPERM|IPC_CREAT|IPC_EXCL)) == -1)
    {
        if(errno == EEXIST)
            semid = semget(semkey, 1, 0);
    }
    else /* Если семафор создается */
    {
        union semun arg;
        arg.val = 1;
        status = semctl(semid, 0, SETVAL, arg);
    }

    if(semid == -1 || status == -1)
    {
        perror("Ошибка вызова initsem");
        return (-1);
    }
    /* Все в порядке */
    return (semid);
}
```



Включаемый файл `pv.h` содержит следующие определения:

```
/* Заголовочный файл для примера работы с семафорами */
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
```

```
#define SEMPERM    0600
#define TRUE      1
#define FALSE     0
```

```
typedef union _semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} semun;
```

Функция `initsem` будет использована в примере следующего раздела.

### **Операции над семафорами: вызов `semop`**

Вызов `semop` выполняет основные операции над семафорами.

#### **Описание**

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

```
int semop(int semid, struct sembuf *op_array, size_t num_ops);
```

Переменная `semid` является идентификатором набора семафоров, полученным с помощью вызова `semget`. Параметр `op_array` является массивом структур `sembuf`, определенных в файле `<sys/sem.h>`. Каждая структура `sembuf` содержит описание операций, выполняемых над семафором.

И снова основной акцент делается на операции с наборами семафоров, при этом функция `semop` позволяет выполнять группу операций как атомарную операцию. Это означает, что пока не появится возможность одновременного выполнения всех операций с отдельными семафорами набора, не будет выполнена ни одна из этих операций. Если не указано обратного, процесс приостановит работу до тех пор, пока он не сможет выполнить все операции сразу.

Рассмотрим структуру `sembuf`. Она включает в себя следующие элементы:

```
unsigned short sem_num;
    short sem_op;
    short sem_flg;
```

Поле `sem_num` содержит индекс семафора в наборе. Если, например, набор содержит всего один элемент, то значение `sem_num` должно быть равно нулю. Поле `sem_op` содержит целое число со знаком, значение которого сообщает функции `semop`, что необходимо сделать. При этом возможны три случая:

#### **Случай 1: отрицательное значение `sem_op`**

Это обобщенная форма команды для работы с семафорами `p()`, которая обсуждалась ранее. Действие функции `semop` можно описать при помощи псевдокода

следующим образом (обратите внимание, что *ABS()* обозначает модуль переменной):

```
if( semval >= ABS(sem_op) )
{
    semval = semval - ABS(sem_op)
}
else
{
    if( (sem_flg & IPC_NOWAIT) )
        немедленно вернуть -1
    else
    {
        ждать, пока semval не станет больше или равно ABS(sem_op)
        затем, как и выше, вычесть ABS(sem_op)
    }
}
```

Основная идея заключается в том, что функция *semop* вначале проверяет значение *semval*, связанное с семафором *sem\_num*. Если значение *semval* достаточно велико, то оно сразу уменьшается на указанную величину. В противном случае процесс будет ждать, пока значение *semval* не станет достаточно большим. Тем не менее, если в переменной *sem\_flg* установлен флаг *IPC\_NOWAIT*, то возврат из вызова *sem\_op* произойдет немедленно, и переменная *errno* будет содержать код ошибки *EAGAIN*.

### Случай 2: положительное значение *sem\_op*

Это соответствует традиционной операции *v()*. Значение переменной *sem\_op* просто прибавляется к соответствующему значению *semval*. Если есть процессы, ожидающие изменения значения этого семафора, то они могут продолжить выполнение, если новое значение семафора удовлетворит их условия.

### Случай 3: нулевое значение *sem\_op*

В этом случае вызов *sem\_op* будет ждать, пока значение семафора не станет равным нулю; значение *semval* этим вызовом не будет изменяться. Если в переменной *sem\_flg* установлен флаг *IPC\_NOWAIT*, а значение *semval* еще не равно нулю, то функция *semop* сразу же вернет сообщение об ошибке.

### Флаг *SEM\_UNDO*

Это еще один флаг, который может быть установлен в элементе *sem\_flg* структуры *sembuf*. Он сообщает системе, что нужно автоматически «отменить» эту операцию после завершения процесса. Для отслеживания всей последовательности таких операций система поддерживает для семафора целочисленную переменную *semadj*. Важно понимать, что переменная *semadj* связана с процессами, и для разных процессов один и тот же семафор будет иметь различные значения *semadj*. Если при выполнении операции *semop* установлен флаг *SEM\_UNDO*, то значение переменной *sem\_num* просто вычитается из значения *semadj*. При этом важен знак переменной *sem\_num*: значение *semadj* уменьшается, если значение *sem\_num*

положительное, и увеличивается, если оно отрицательное. После выхода из процесса система прибавляет все значения `semadj` к соответствующим семафорам и, таким образом, сводит на нет эффект от всех вызовов `semop`. В общем случае флаг `SEM_UNDO` должен быть всегда установлен, кроме тех случаев, когда значения, устанавливаемые процессом, должны сохраняться после завершения процесса.

### **Пример работы с семафорами**

Теперь продолжим пример, который начали с процедуры `initsem`. Он содержит две процедуры `p()` и `v()`, реализующие традиционные операции над семафорами. Сначала рассмотрим `p()`:

```
/* Процедура p.c – операция p для семафора */
#include "pv.h"

int p (int semid)
{
    struct sembuf p_buf,
    p_buf.sem_num = 0;
    p_buf.sem_op = -1;
    p_buf.sem_fig = SEM_UNDO;

    if(semop(semid, &p_buf, 1) == -1)
    {
        perror("Ошибка операции p(semid)");
        exit(1);
    }
    return (0);
}
```

Обратите внимание на то, что здесь использован флаг `SEM_UNDO`. Теперь рассмотрим текст процедуры `v()`.

```
/* Процедура v.c – операция v для семафора */
#include "pv.h"

int v(int semid)
{
    struct sembuf v_buf;
    v_buf.sem_num = 0;
    v_buf.sem_op = 1;
    v_buf.sem_fig = SEM_UNDO;

    if(semop(semid, &v_buf, 1) == 1)
    {
        perror("Ошибка операции v(semid)");
        exit(1);
    }
    return (0);
}
```

Можно продемонстрировать использование этих довольно простых процедур для реализации взаимного исключения. Рассмотрим следующую программу:

```
/* Программа testsem – проверка процедур работы с семафорами.*/  
  
#include "pv.h"  
  
void handlesem(key_t skey);  
  
main()  
{  
    key_t semkey = 0x200;  
    int i;  
  
    for(i = 0; i < 3; i++)  
    {  
        if(fork() == 0)  
            handlesem(semkey);  
    }  
}  
  
void handlesem(key_t skey)  
{  
    int semid;  
    pid_t pid = getpid();  
  
    if((semid = initsem(skey)) < 0)  
        exit(1);  
  
    printf("\nПроцесс %d перед критическим участком\n", pid);  
    p(semid);  
  
    printf("Процесс %d выполняет критический участок\n", pid);  
  
    /* В реальной программе здесь выполняется нечто осмысленное.*/  
    sleep(10);  
  
    printf("Процесс %d покинул критический участок\n", pid);  
    v(semid),  
  
    printf("Процесс %d завершает работу\n", pid);  
    exit(0);  
}
```

Программа testsem порождает три дочерних процесса, которые используют вызовы p() и v() для того, чтобы в каждый момент времени только один из них выполнял критический участок. Запуск программы testsem может дать следующий результат:

```
Процесс 799 перед критическим участком  
Процесс 799 выполняет критический участок  
Процесс 800 перед критическим участком  
Процесс 801 перед критическим участком  
Процесс 799 покинул критический участок
```

Процесс 801 выполняет критический участок  
Процесс 799 завершает работу  
Процесс 801 покинул критический участок  
Процесс 801 завершает работу  
Процесс 800 выполняет критический участок  
Процесс 800 покинул критический участок  
Процесс 800 завершает работу

### 8.3.4. Разделяемая память

Операции с разделяемой памятью позволяют двум и более процессам совместно использовать область физической памяти (общеизвестно, что обычно области данных любых двух программ совершенно отделены друг от друга). Чаще всего разделяемая память является наиболее производительным механизмом межпроцессного взаимодействия.

Для того, чтобы сегмент памяти мог использоваться совместно, он должен быть сначала создан при помощи системного вызова `shmget`. После создания сегмента разделяемой памяти процесс может подключаться к нему при помощи вызова `shmat` и затем использовать его для своих частных целей. Когда этот сегмент памяти больше не нужен, процесс может отключиться от него при помощи вызова `shmdt`.

#### Системный вызов `shmget`

Сегменты разделяемой памяти создаются при помощи вызова `shmget`.

#### Описание

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

```
int shmget(key_t key, size_t size, int permflags);
```

Этот вызов аналогичен вызовам `msgget` и `semget`. Наиболее интересным параметром вызова является `size`, который задает требуемый минимальный размер (в байтах) сегмента памяти. Параметр `key` является значением ключа сегмента памяти, параметр `permflags` задает права доступа к сегменту памяти и, кроме того, может содержать флаги `IPC_CREAT` и `IPC_EXCL`.

#### Операции с разделяемой памятью: вызовы `shmat` и `shmdt`

Сегмент памяти, созданный вызовом `shmget`, является участком *физической* памяти и не находится в *логическом* пространстве данных процесса. Для использования разделяемой памяти текущий процесс, а также все другие процессы, взаимодействующие с этим сегментом, должны явно подключать этот участок памяти к логическому адресному пространству при помощи вызова `shmat`:

#### Описание

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

```
void *shmat(int shmid, const void *daddr, int shmflags);
```

Вызов `shmat` связывает участок памяти, обозначенный идентификатором `shmid` (который был получен в результате вызова `shmget`) с некоторым допустимым адресом логического адресного пространства вызывающего процесса. Этот адрес является значением, возвращаемым вызовом `shmat` (в языке C такие адреса данных обычно представляются типом `void *`).

Параметр `daddr` позволяет программисту до некоторой степени управлять выбором этого адреса. Если этот параметр равен `NULL`, то участок подключается к первому доступному адресу, выбранному системой. Это наиболее простой случай использования вызова `shmat`. Если параметр `daddr` не равен `NULL`, то участок будет подключен к содержащемуся в нем адресу или адресу в ближайшей окрестности в зависимости от флагов, заданных в аргументе `shmflags`. Этот вариант сложнее, так как при этом необходимо знать расположение программы в памяти.

Аргумент `shmflag` может содержать два флага, `SHM_RDONLY` и `SHM_RND`, определенные в заголовочном файле `<sys/shm.h>`. При задании флага `SHM_RDONLY` участок памяти подключается только для чтения. Флаг `SHM_RND` определяет, если это возможно, способ обработки в вызове `shmat` ненулевого значения `daddr`.

В случае ошибки вызов `shmat` вернет значение:

```
(void *)-1
```

Вызов `shmdt` противоположен вызову `shmat` и отключает участок разделяемой памяти от логического адресного пространства процесса (это означает, что процесс больше не может использовать его). Он вызывается очень просто:

```
retval = shmdt(memptr);
```

Возвращаемое значение `retval` является целым числом и равно 0 в случае успеха и -1 – в случае ошибки.

*Системный вызов `shmctl`*

### Описание

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

```
int shmctl(int shmid, int command, struct shmid_ds *shm_stat);
```

Этот вызов в точности соответствует вызову `msgctl`, и параметр `command` может, наряду с другими, принимать значения `IPC_STAT`, `IPC_SET` и `IPC_RMID`. В следующем примере этот вызов будет использован с аргументом `command` равным `IPC_RMID`.

### **Пример работы с разделяемой памятью: программа `shmcopy`**

В этом разделе создадим простую программу `shmcopy` для демонстрации практического использования разделяемой памяти. Программа `shmcopy` просто копирует данные со своего стандартного ввода на стандартный вывод, но позволяет избежать лишних простоев в вызовах `read` и `write`. При запуске программы `shmcopy` создаются два процесса, один из которых выполняет чтение, а другой – запись, и которые совместно используют два буфера, реализованные в виде сегментов разделяемой памяти. Когда первый процесс считывает данные в первый

буфер, второй записывает содержимое второго буфера, и наоборот. Так как чтение и запись выполняются одновременно, пропускная способность возрастает. Этот подход используется, например, в программах, которые выводят информацию на ленточный накопитель.

Для согласования двух процессов (чтобы записывающий процесс не писал в буфер до тех пор, пока считывающий процесс его не заполнит) будем использовать два семафора. Почти во всех программах, использующих разделяемую память, требуется дополнительная синхронизация, так как механизм разделяемой памяти не содержит собственных средств синхронизации.

Программа `shmcopy` использует следующий заголовочный файл `share_ex.h`:

```
/* Заголовочный файл для примера работы с разделяемой памятью */

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMKEY1    (key_t)0x10 /* Ключ разделяемой памяти */
#define SHMKEY2    (key_t)0x15 /* Ключ разделяемой памяти */
#define SEMKEY     (key_t)0x20 /* Ключ семафора */

/* Размер буфера для чтения и записи */
#define SIZ 5*BUFSIZ

/* В этой структуре будут находиться данные и счетчик чтения */
struct databuf {
    int d_nread;
    char d_buf[SIZ];
};

typedef union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} semun;
```

Напомним, что постоянная `BUFSIZ` определена в файле `<stdio.h>` и задает оптимальный размер порций данных при работе с файловой системой. Шаблон `databuf` показывает структуру, которая связывается с каждым сегментом разделяемой памяти. В частности, элемент `d_nread` позволит процессу, выполняющему чтение, передавать другому, осуществляющему запись, через участок разделяемой памяти число считанных символов.

Следующий файл содержит процедуры для инициализации двух участков разделяемой памяти и набора семафоров. Он также содержит процедуру `remobj`, которая удаляет различные объекты межпроцессного взаимодействия в конце работы программы. Обратите внимание на способ вызова `shmat` для подключения участков разделяемой памяти к адресному пространству процесса.

```
/* Процедуры инициализации */

#include "share_ex.h"
#define IFLAGS (IPC_CREAT | IPC_EXCL)
#define ERR ((struct databuf *) -1)

static int shmid1, shmid2, semid;

void getseg(struct databuf **p1, struct databuf **p2)
{
    /* Создать участок разделяемой памяти */
    if((shmid1 = shmget(SHMKEY1, sizeof(struct databuf),
        0600 | IFLAGS)) == -1)
        fatal("shmget");
    if((shmid2 = shmget(SHMKEY2, sizeof(struct databuf),
        0600 | IFLAGS)) == -1)
        fatal("shmget");

    /* Подключить участки разделяемой памяти */
    if((*p1 = (struct databuf *)shmat(shmid1,0,0)) == ERR)
        fatal ("shmat");
    if((*p2 = (struct databuf *)shmat(shmid2,0,0)) == ERR)
        fatal ("shmat");
}

int getsem(void)      /* Получить набор семафоров */
{
    union semun x;
    x.val = 0;

    /* Создать два набора семафоров */
    if((semid = semget(SEMKEY, 2, 0600 | IFLAGS)) == -1)
        fatal("semget");

    /* Задать начальные значения */
    if(semctl(semid, 0, SETVAL, x) == -1)
        fatal("semctl");
    if(semctl(semid, 1, SETVAL, x) == -1)
        fatal("semctl");
    return(semid);
}

/* Удалить идентификаторы разделяемой памяти
и идентификатор набора семафоров */
void remobj(void)
{
    if(shmctl(shmid1, IPC_RMID, NULL) == -1)
        fatal("shmctl");
    if(shmctl(shmid2, IPC_RMID, NULL) == -1)
        fatal("shmctl");
    if(semctl(semid, 0, IPC_RMID, NULL) == -1)
        fatal("semctl");
}
```



Ошибки в этих процедурах обрабатываются при помощи процедуры `fatal`, которая использовалась в предыдущих примерах. Она просто вызывает `perorr`, а затем `exit`.

Ниже следует функция `main` для программы `strcopy`. Она вызывает процедуры инициализации, а затем создает процесс для чтения (родительский) и для записи (дочерний). Обратите внимание на то, что именно выполняющий запись процесс вызывает процедуру `remobj` при завершении программы.

```
/* Программа shmcopy — функция main */

#include "share_ex.h"
main()
{
    int semid;
    pid_t pid;
    struct databuf *buf1, *buf2;

    /* Инициализация набора семафоров */
    semid = getsem();

    /* Создать и подключить участки разделяемой памяти */
    getseg(&buf1, &buf2);

    switch(pid = fork()){
    case -1:
        fatal("fork");
    case 0: /* дочерний процесс */
        writer(semid, buf1, buf2);
        remobj();
        break;
    default: /* Родительский процесс */
        reader(semid, buf1, buf2);
        break;
    }

    exit(0);
}
```

Функция `main` создает объекты межпроцессного взаимодействия до вызова `fork`. Обратите внимание на то, что адреса, определяющие сегменты разделяемой памяти (которые находятся в переменных `buf1` и `buf2`), будут заданы в обоих процессах.

Процедура `reader` принимает данные со стандартного ввода, то есть из дескриптора файла 0, и является первой функцией, представляющей интерес. Ей передается идентификатор набора семафоров в параметре `semid` и адреса двух участков разделяемой памяти в переменных `buf1` и `buf2`.

```
/* Процедура reader — выполняет чтение из файла */

#include "share_ex.h"
/* Определения процедур p() и v() для двух семафоров */
struct sembuf p1 = {0,-1,0}, p2 = {1,-1,0};
```

```

struct sembuf v1 = {0,1,0}, v2 = {1,1,0};

void reader(int semid, struct databuf *buf1,
            struct databuf *buf2)
{
    for(;;)
    {
        /* Считать в буфер buf1 */
        buf1->d_nread = read(0, buf1->d_buf, SIZ);

        /* Точка синхронизации */
        semop(semid, &v1, 1);
        semop(semid, &p2, 1);

        /* Чтобы процедура writer не была приостановлена */
        if(buf1->d_nread <=0)
            return;

        buf2->d_nread = read(0, buf2->d_buf, SIZ);

        semop(semid, &v2, 1);
        semop(semid, &p1, 1);

        if(buf2->d_nread <=0)
            return;
    }
}

```

Структуры `sembuf` просто определяют операции `p()` и `v()` для набора из двух семафоров. Но на этот раз они используются не для блокировки критических участков кода, а для синхронизации процедур, выполняющих чтение и запись. Процедура `reader` использует операцию `v2` для сообщения о том, что она завершила чтение и ожидает, вызвав `semop` с параметром `p1`, пока процедура `writer` не сообщит о завершении записи. Это станет более очевидным при описании процедуры `writer`. Возможны другие подходы, включающие или четыре бинарных семафора, или семафоры, имеющие более двух значений.

Последней процедурой, вызываемой программой `shmcopy`, является процедура `writer`:

```

/* Процедура writer — выполняет запись */

#include "share_ex.h"

extern struct sembuf p1, p2; /* Определены в reader.c */
extern struct sembuf v1, v2; /* Определены в reader.c */

void writer(int semid, struct databuf *buf1,
            struct databuf *buf2)
{
    for(;;)
    {
        semop(semid, &p1, 1);
        semop(semid, &v2, 1);
    }
}

```

```
if(buf1->d_nread <= 0)
    return;

write(1, buf1->d_buf, buf1->d_nread);

semop(semid, &p2, 1);
semop(semid, &v1, 1);

if(buf2->d_nread <= 0)
    return;

write(1, buf2->d_buf, buf2->d_nread);
}
}
```

И снова следует обратить внимание на использование набора семафоров для согласования работы процедур reader и writer. На этот раз процедура writer использует операцию v2 для сигнализации и ждет p1. Важно также отметить, что значения buf1->d\_nread и buf2->d\_nread устанавливаются процессом, выполняющим чтение.

После компиляции можно использовать программу shmcopy при помощи подобной команды:

```
$ shmcopy < big > /tmp/big
```

---

**Упражнение 8.6.** Усовершенствуйте обработку ошибок и вывод сообщений в программе shmcopy (в особенности для вызовов read и write). Сделайте так, чтобы программа shmcopy принимала в качестве аргументов имена файлов в форме команды cat. Какие последствия возникнут при прерывании программы shmcopy? Можете ли вы улучшить поведение программы?

---

---

**Упражнение 8.7.** Придумайте систему передачи сообщений, использующую разделяемую память. Измерьте ее производительность и сравните с производительностью стандартных процедур передачи сообщений.

---

### 8.3.5. Команды ipcs и ipcrm

Существуют две команды оболочки для работы со средствами межпроцессного взаимодействия. Первая из них – команда ipcs, которая выводит информацию о текущем статусе средства межпроцессного взаимодействия. Вот простой пример ее применения:

```
$ ipcs
```

```
IPC status from /dev/kmem as of Wed Feb 26 18:31:31 1998
T ID KEY MODE OWNER GROUP
Message Queues:
Shared Memory:
Semaphores:
s 10 0x00000200 --ra----- keith users
```

Другая команда `ipcrm` используется для удаления средства межпроцессного взаимодействия из системы (если пользователь является его владельцем или суперпользователем), например, команда

```
$ ipcrm -s 0
```

удаляет семафор, связанный с идентификатором 0, а команда

```
$ ipcrm -S 200
```

удаляет семафор со значением ключа равным 200.

За дополнительной информацией о возможных параметрах этих команд следует обратиться к справочному руководству системы.



## Глава 9. Терминал

### 9.1. Введение

Когда пользователь взаимодействует с программой при помощи терминала, происходит намного больше действий, чем может показаться на первый взгляд. Например, если программа выводит строку на терминальное устройство, то она вначале обрабатывается в разделе ядра, которое будем называть *драйвером терминала* (terminal driver). В зависимости от значения определенных флагов состояния системы строка может передаваться буквально или как-то изменяться драйвером. Одно из обычных изменений заключается в замене символов `line-feed` (перевод строки) или `newline` (новая строка) на последовательность из двух символов `carriage-return` (возврат каретки) и `newline`. Это гарантирует, что каждая строка всегда будет начинаться с левого края экрана терминала или открытого окна.

Аналогично драйвер терминала обычно позволяет пользователю редактировать ошибки в строке ввода при помощи текущих символов `erase` (стереть) и `kill` (уничтожить). Символ `erase` удаляет последний напечатанный символ, а символ `kill` – все символы до начала строки. Только после того, как вид строки устроит пользователя и он нажмет на клавишу **Return** (ввод), драйвер терминала передаст строку программе.

Но это еще не все. После того, как выводимая строка достигает терминала, аппаратура терминала может либо просто вывести ее на экран, либо интерпретировать ее как *escape-последовательность* (escape-sequence), посланную для управления экраном. В результате, например, может произойти не вывод сообщения, а очистка экрана.

На рис. 9.1 более ясно показаны различные компоненты связи между компьютером и терминалом.

Эта связь включает четыре элемента:

- *программы (A)*. Программа генерирует выходные последовательности символов и интерпретирует входные. Она может взаимодействовать с терминалом при помощи системных вызовов (`read` или `write`), стандартной библиотеки ввода/вывода или специального библиотечного пакета, разработанного для управления экраном. Разумеется, в конечном счете весь ввод/вывод будет осуществляться при помощи вызовов `read` и `write`, так как высокоуровневые библиотеки могут вызывать только эти основные примитивы;
- *драйвер терминала (B)*. Основная функция драйвера терминала заключается в передаче данных от программы к периферийному устройству и наоборот.

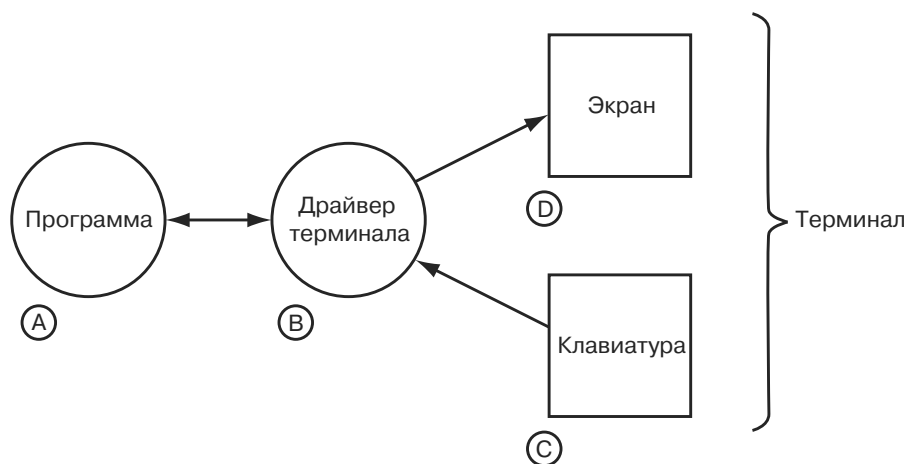


Рис. 9.1. Связь между процессом UNIX и терминалом

В самом ядре UNIX терминал обычно состоит из двух основных программных компонентов – *драйвера устройства* (device driver) и *дисциплины линии связи* (line discipline).

Драйвер устройства является низкоуровневым программным обеспечением, написанным для связи с определенным аппаратным обеспечением, которое позволяет компьютеру взаимодействовать с терминалом. На самом деле чаще всего драйверы устройств нужны для того, чтобы работать с разными типами аппаратного обеспечения. Над этим нижним слоем надстроены средства, которые полагаются на то, что основные свойства, поддерживаемые драйвером устройства, являются общими независимо от аппаратного обеспечения. Кроме этой основной функции передачи данных, драйвер терминала будет также выполнять некоторую логическую обработку входных и выходных данных, преобразуя одну последовательность символов в другую. Это осуществляется дисциплиной линии связи. Она также может обеспечивать множество функций для помощи конечному пользователю, таких как редактирование строки ввода. Точная обработка и преобразование данных зависят от флагов состояния, которые хранятся в дисциплине линии связи для каждого порта терминала. Они могут устанавливаться при помощи группы системных вызовов, которая будет рассмотрена в следующих разделах;

- *клавиатура и экран (C и D)*. Эти два элемента представляют сам терминал и подчеркивают его двойственную природу. Узел (C) означает клавиатуру терминала и служит источником ввода. Узел (D) представляет экран терминала и выступает в качестве назначения вывода. Программа может получить доступ к терминалу и как к устройству ввода, и как к устройству вывода при помощи общего имени терминала, и, в конечном счете, единственного дескриптора файла. Для того чтобы это было возможно, дисциплина линии связи имеет отдельные очереди ввода и вывода для каждого терминала. Эта схема показана на рис. 9.2.

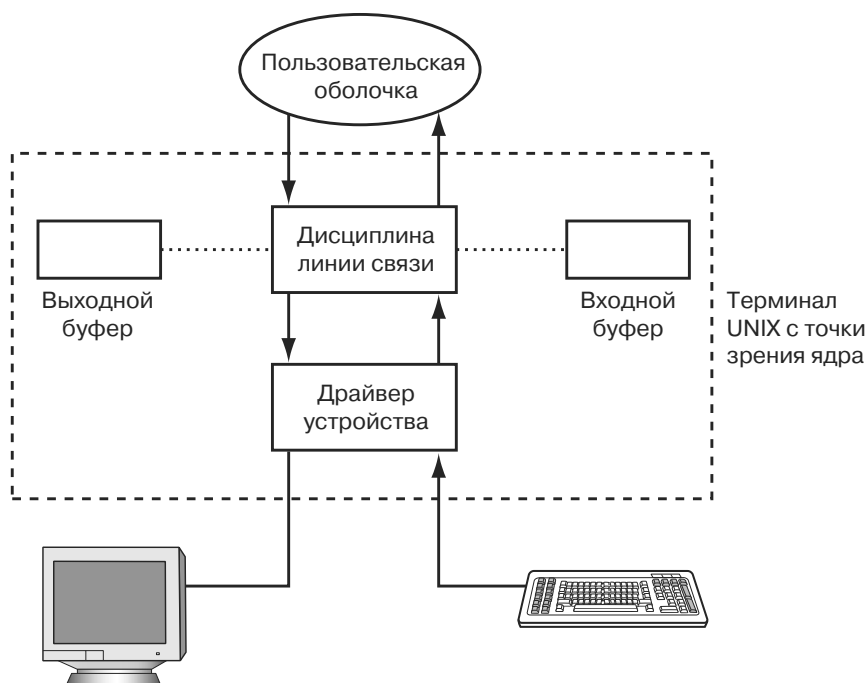


Рис. 9.2. Реализация терминала

До сих пор предполагалось, что подключенное к терминалу периферийное устройство является стандартным дисплеем. Вместе с тем периферийное устройство может быть принтером, плоттером, сетевым адаптером или даже другим компьютером. Тем не менее, независимо от природы периферийного устройства, оно может служить и в качестве источника, и в качестве назначения для входного и выходного потоков данных соответственно.

Эта глава будет в основном посвящена узлам (А) и (В) схемы. Другими словами, будет рассмотрено взаимодействие между программой и драйвером устройства на уровне системных вызовов. Не будем касаться совершенно отдельного вопроса работы с экраном, поскольку драйвер терминала не принимает участия в создании соответствующих ескаре-последовательностей, управляющих экраном.

Перед тем как продолжить дальше, следует сделать два предостережения. Во-первых, будут рассматриваться только «обычные» терминалы, а не графические, построенные на оконных системах X Window System или MS Windows. Для них характерны свои проблемы, которых касаться не будем. Во-вторых, работа с терминалом в UNIX является областью, печально известной своей несовместимостью. Тем не менее спецификация XSI обеспечивает стандартный набор системных вызовов. Именно на них и сфокусируем внимание.

## 9.2. Терминал UNIX

Как уже упоминалось в главе 4, терминалы обозначаются файлами устройств (из-за природы терминалов они рассматриваются как символьные устройства).

Вследствие этого доступ к терминалам, а точнее к портам терминалов, обычно можно получить при помощи имен файлов в каталоге `dev`. Типичные имена терминалов могут быть такими:

```
/dev/console  
/dev/tty01  
/dev/tty02  
/dev/tty03  
...
```

Обозначение `tty` является широко используемым в UNIX синонимом терминала.

Из-за универсальности понятия файла UNIX к терминалам можно получить доступ при помощи стандартных примитивов доступа к файлам, таких как `read` или `write`. Права доступа к файлам сохраняют свои обычные значения и поэтому управляют доступом к терминалам в системе. Чтобы эта схема работала, владелец терминала меняется при входе пользователя в систему, при этом все пользователи являются владельцами терминала, за которым они работают.

Обычно процессу не нужно явно открывать файл терминала для взаимодействия с пользователем. Это происходит из-за того, что его стандартный ввод и вывод, если они не переопределены, будут по умолчанию связаны с терминалом пользователя. Поэтому, если предположить, что стандартный вывод не назначен в файл, то следующий фрагмент кода приведет к выводу данных на экран терминала:

```
#define FD_STDOUT 1  
.  
.  
.  
write(FD_STDOUT, mybuffer, somesize);
```

В традиционном окружении UNIX терминалы, обеспечивающие вход в систему, обычно первоначально открываются при старте системы программой управления процессами `init`. Дескрипторы файла терминала передаются потомкам программы `init`, и, в конечном итоге, каждый процесс пользовательской оболочки будет наследовать три дескриптора файла, связанные с терминалом пользователя. Эти дескрипторы будут представлять стандартный ввод, стандартный вывод и стандартный вывод диагностики оболочки. Они в свою очередь передаются всем запущенным из оболочки программам.

### 9.2.1. Управляющий терминал

При обычных обстоятельствах терминал, связанный с процессом при помощи его стандартных дескрипторов файлов, является *управляющим терминалом* (`control terminal`) этого процесса и его сеанса. Управляющий терминал является важным атрибутом процесса, который определяет обработку генерируемых с клавиатуры прерываний. Например, если пользователь нажимает текущую клавишу прерывания, то все процессы, которые считают терминал своим управляющим терминалом, получают сигнал `SIGINT`. Управляющие терминалы, как и другие атрибуты процесса, наследуются при вызове `fork`. (Более конкретно, терминал становится



управляющим терминалом для сеанса, когда его открывает лидер сеанса, при условии, что терминал еще не связан с сеансом и лидер сеанса еще не имеет управляющего терминала. Вследствие этого процесс может разорвать свою связь с управляющим терминалом, изменив свой сеанс при помощи вызова `setsid`. Этот аспект был рассмотрен в главе 5, хотя, возможно, там это было несколько преждевременно. Теперь же следует получить понимание того, как процесс `init` инициализирует систему при старте.)

Если процесс должен получить доступ к своему управляющему терминалу независимо от состояния его стандартных дескрипторов файлов, то можно использовать имя файла

```
/dev/tty
```

которое всегда интерпретируется как определяющее текущий управляющий терминал процесса. Следовательно, терминал, обозначаемый в действительности этим файлом, различается для разных процессов.

### 9.2.2. Передача данных

Основной задачей драйвера терминала является передача данных между процессом и его терминальным устройством. На самом деле это достаточно сложное требование, так как пользователь может печатать символы в любое время, даже во время вывода. Чтобы лучше понять эту ситуацию, вернемся к рис. 9.1 и представим, что по путям от (C) к (B) и от (B) к (D) одновременно передаются данные. Напомним, что программа, представленная на схеме узлом (A), может выполнять только последовательные вызовы `read` или `write`.

Поскольку в каждый момент времени программа может обрабатывать только один поток данных терминала, то для одновременного управления двумя потоками дисциплина линии связи запоминает входные и выходные данные во внутренних буферах. Входные данные передаются программе пользователя, когда он выполняет вызов `read`. Входные символы могут быть потеряны при переполнении поддерживаемых ядром буферов или если число символов, поступивших на терминал, превысит ограничение `MAX_INPUT`, определенное в файле `<limits.h>`. Это предельное значение обычно равно 255, что достаточно велико для того, чтобы потери данных при обычном использовании были достаточно редки. Тем не менее не существует способа определить, что данные были потеряны, так как система просто молча отбрасывает лишние символы.

Ситуация с выводом несколько проще. Каждый вызов `write` для терминала помещает символы в очередь вывода. Если очередь переполняется, то следующий вызов `write` будет «заблокирован» (то есть процесс будет приостановлен) до тех пор, пока очередь вывода не уменьшится до нужного уровня.

### 9.2.3. Эхо-отображение вводимых символов и опережающий ввод с клавиатуры

Поскольку терминалы используются для взаимодействия между людьми и компьютерными программами, драйвер терминала UNIX поддерживает множество дополнительных средств, облегчающих жизнь пользователям.

Возможно, самым элементарным из этих дополнительных средств является «эхо», то есть отображение вводимых с клавиатуры символов на экране. Оно позволяет увидеть на экране символ «А», когда вы печатаете «А» на клавиатуре. Подключенные к системам UNIX терминалы обычно работают в *полнодуплексном* (full-duplex) режиме; это означает, что за эхо-отображение символов на экране отвечает система UNIX, а не терминал. Следовательно, при наборе символа он вначале передается терминалом системе UNIX. После его получения дисциплина линии связи сразу же помещает его копию в очередь вывода терминала. Затем символ выводится на экран терминала. Если вернуться к рис. 9.1, то увидим, что символ при этом пересылается по пути от (С) к (В), а затем сразу же направляется по пути от (В) к (D). Все может произойти до того, как программа (А) успеет считать символ. Это приводит к интересному явлению, когда символы, вводимые с клавиатуры в то время, когда программа осуществляет вывод на экран, отображаются в середине вывода. В операционных системах некоторых семейств (не UNIX) отображение вводимых символов на экране может подавляться до тех пор, пока программа не сможет прочесть их.

#### **9.2.4. Канонический режим, редактирование строки и специальные символы**

Терминал может быть настроен на самые разные режимы в соответствии с типом работающих с ним программ; работа этих режимов поддерживается соответствующей дисциплиной линии связи. Например, экранному редактору может понадобиться максимальная свобода действий, поэтому он может перевести терминал в режим *прямого доступа* (raw mode), при котором дисциплина линии связи просто передает символы программе по мере их поступления, без всякой обработки.

Вместе с тем ОС UNIX не была бы универсальной программной средой, если бы всем программам приходилось бы иметь дело с деталями управления терминалом. Поэтому стандартная дисциплина линии связи обеспечивает также режим работы, специально приспособленный для простого интерактивного использования на основе построчного ввода. Этот режим называется *каноническим режимом терминала* (canonical mode) и используется командным интерпретатором, редактором `ed` и аналогичными программами.

В каноническом режиме драйвер терминала выполняет специальные действия при нажатии специальных клавиш. Многие из этих действий относятся к редактированию строки. Вследствие этого, если терминал находится в каноническом режиме, то ввод в программе осуществляется целыми строками (подробнее об этом ниже).

Наиболее часто используемой в каноническом режиме клавишей редактирования является клавиша `erase`, нажатие которой приводит к стиранию предыдущего символа в строке. Например, следующий клавиатурный ввод

```
$ whp<erase>o
```

за которым следует перевод строки, приведет к тому, что драйвер терминала передаст командному интерпретатору строку *who*. Если терминал настроен правильно, то символ *p* должен быть стерт с экрана.

В качестве символа `erase` пользователем может быть задано любое значение ASCII. Наиболее часто для этих целей используется символ ASCII `backspace` (возврат).

На уровне оболочки проще всего поменять этот символ при помощи команды `stty`, например

```
$ stty erase "^h"
```

задает в качестве символа `erase` комбинацию **Ctrl+H**, которая является еще одним именем для символа возврата. Обратите внимание, что зачастую можно набрать строку `"^h"`, а не нажимать саму комбинацию клавиш **Ctrl+H**.

Ниже следует систематическое описание других символов, определенных в спецификации XSI и имеющих особое значение для оболочки в каноническом режиме. Если не указано иначе, их точные значения могут быть установлены пользователем или администратором системы.

**kill**      Приводит к стиранию всех символов до начала строки. Поэтому входящая последовательность

```
$ echo < kill > who
```

за которой следует новая строка, приводит к выполнению команды `who`. По умолчанию значение символа **kill** равно **Ctrl+?**, часто также используются комбинации клавиш **Ctrl+X** и **Ctrl+U**. Можно поменять символ **kill** при помощи команды:

```
$ stty kill <НОВЫЙ_СИМВОЛ>
```

**intr**      Символ прерывания. Если пользователь набирает его, то программе, выполняющей чтение с терминала, а также всем другим процессам, которые считают терминал своим управляющим терминалом, посылается сигнал `SIGINT`. Такие программы, как командный интерпретатор, перехватывают этот сигнал, поскольку по умолчанию сигнал `SIGINT` приводит к завершению программы. Одно из значений для символа **intr** по умолчанию равно символу ASCII **delete** (удалить), который часто называется **DEL**. Вместо него может также использоваться символ **Ctrl+C**. Для изменения текущего значения символа **intr** на уровне оболочки используйте команду:

```
$ stty intr <НОВЫЙ_СИМВОЛ>
```

**quit**      Обратитесь к главе 6 за описанием обработки сигналов **quit**. Если пользователь набирает этот символ, то группе процессов, связанной с терминалом, посылается сигнал `SIGQUIT`. Командный интерпретатор перехватывает этот сигнал, остальные пользовательские процессы также имеют возможность перехватить и обработать этот сигнал. Как уже описывалось в главе 6, действием этого сигнала по умолчанию является сброс образа памяти процесса на диск и аварийное завершение, сопровождаемое выводом сообщения «Quit – core dumped». Обычно символ **quit** – это символ ASCII **FS**, или **Ctrl+\\**. Его можно изменить, выполнив команду:

```
$ stty quit <НОВЫЙ_СИМВОЛ>
```

**eof** Этот символ используется для обозначения окончания входного потока с терминала (для этого он должен быть единственным символом в начале новой строки). Стандартным начальным значением для него является символ ASCII **eot**, известный также как **Ctrl+D**. Его можно изменить, выполнив команду:

```
$ stty eof <новый_символ>
```

**nl** Это обычный разделитель строк. Он всегда имеет значение ASCII символа **line-feed** (перевод строки), который соответствует символу **newline** (новая строка) в языке C. Он не может быть изменен или установлен ользователем. На терминалах, которые посылают вместо символа **line-feed** символ **carriage-return** (возврат каретки), дисциплина линии связи может быть настроена так, чтобы преобразовывать возврат каретки в перевод строки

**eol** Еще один разделитель строк, который действует так же, как и **nl**. Обычно не используется и поэтому имеет по умолчанию значение символа ASCII **NULL**

**stop** Обычно имеет значение **Ctrl+S** и в некоторых реализациях может быть изменен пользователем. Используется для временной приостановки записи вывода на терминал. Этот символ особенно полезен при применении старомодных терминалов, так как он может использоваться для того, чтобы приостановить вывод прежде, чем он исчезнет за границей экрана терминала

**start** Обычно имеет значение **Ctrl+Q**. Может ли он быть изменен, также зависит от конкретной реализации. Он используется для продолжения вывода, приостановленного при помощи комбинации клавиш **Ctrl+S**. Если комбинация **Ctrl+S** не была нажата, то **Ctrl+Q** игнорируется

**susp** Если пользователь набирает этот символ, то группе процессов, связанной с терминалом, посылается сигнал **SIGTSTP**. При этом выполнение группы приоритетных процессов останавливается, и она переводится в фоновый режим. Обычное значение символа **suspend** равно **Ctrl+Z**. Его также можно изменить, выполнив команду:

```
$ stty susp <новый_символ>
```

Специальное значение символов **erase**, **kill**, и **eof** можно отменить, набрав перед ними символ обратной косой черты (**\**). При этом связанная с символом функция не выполняется, и символ посылается программе без изменений. Например, строка

```
aa\<erase> b <erase> c
```

приведет к тому, что программе, выполняющей чтение с терминала, будет передана строка `aa\<erase> c`.

### 9.3. Взгляд с точки зрения программы

До сих пор изучались функции драйвера терминала, относящиеся к пользовательскому интерфейсу. Теперь же мы рассмотрим их с точки зрения программы, использующей терминал.

### 9.3.1. Системный вызов *open*

Вызов *open* может использоваться для открытия дисциплины линии связи терминала так же, как и для открытия обычного дискового файла, например:

```
fd = open("/dev/tty0a", O_RDWR);
```

Однако при попытке открыть терминал возврата из вызова не произойдет до тех пор, пока не будет установлено соединение. Для терминалов с модемным управлением это означает, что возврат из вызова не произойдет до тех пор, пока не будут установлены сигналы управления модемом и не получен сигнал «детектирования несущей», что может потребовать значительного времени либо вообще не произойти.

Следующая процедура использует вызов *alarm* (представленный в главе 6) для задания интервала ожидания, если возврат из вызова *open* не произойдет за заданное время:

```
/* Процедура ttyopen — вызов open с интервалом ожидания */

#include <stdio.h>
#include <signal.h>

#define TIMEOUT 10
#define FALSE 0
#define TRUE 1

static int timeout = FALSE;
static char *termname;

static void settimeout(void)
{
    fprintf(stderr, "Превышено время ожидания %s\n", termname);
    timeout = TRUE;
}

int ttyopen(char *filename, int flags)
{
    int fd = -1;
    static struct sigaction act, oact;

    termname = filename;

    /* Установить флаг таймаута */
    timeout = FALSE;

    /* Установить обработчик сигнала SIGALRM */
    act.sa_handler = settimeout;
    sigfillset(&(act.sa_mask));
    sigaction(SIGALRM, &act, &oact);

    alarm(TIMEOUT);

    fd = open(filename, flags);

    /* Сброс установок */
    alarm(0);
}
```

```
sigaction(SIGALRM, &oact, &act);  
return (timeout ? -1 : 0);  
}
```

### 9.3.2. Системный вызов *read*

При использовании файла терминального устройства вместо обычного дискового файла изменения больше всего затрагивают работу системного вызова *read*. Это особенно проявляется, если терминал находится в каноническом режиме, устанавливаемом по умолчанию для обычного интерактивного использования. В этом режиме основной единицей ввода становится строка. Следовательно, программа не может считать символы из строки, пока пользователь не нажмет на клавишу **Return**, которая интерпретируется системой как переход на новую строку. Важно также, что после ввода новой строки всегда происходит возврат из вызова *read*, даже если число символов в строке меньше, чем число символов, запрошенное вызовом *read*. Если вводится только **Return** и системе посылается пустая строка, то соответствующий вызов *read* вернет значение 1, так как сам символ новой строки тоже доступен программе. Поэтому нулевое значение, как и обычно, может использоваться для определения конца файла (то есть ввода символа **eof**).

Использование вызова *read* для чтения из терминала в программе *io* уже рассматривалось в главе 2. Тем не менее эта тема нуждается в более подробном объяснении, поэтому рассмотрим следующий вызов:

```
nread = read(0, buffer, 256);
```

При извлечении стандартного ввода процесса из обычного файла вызов интерпретируется просто: если в файле более 256 символов, то вызов *read* вернет в точности 256 символов в массиве *buffer*. Чтение из терминала происходит в несколько этапов – чтобы продемонстрировать это, обратимся к рис. 9.3, который показывает взаимодействие между программой и пользователем в процессе вызова *read*.

Эта схема иллюстрирует возможную последовательность действий, инициированных чтением с терминала с помощью вызова *read*. Для каждого шага изображены два прямоугольника. Верхний из них показывает текущее состояние строки ввода на уровне драйвера терминала; нижний, обозначенный как буфер чтения, показывает данные, доступные для чтения процессом в настоящий момент. Необходимо подчеркнуть, что схема показывает только логику работы с точки зрения пользовательского процесса. Кроме того, обычная реализация драйвера терминала использует не один, а два буфера (очереди), что, впрочем, не намного усложняет представленную схему.

Шаг 1 представляет ситуацию в момент выполнения программой вызова *read*. В этот момент пользователь уже напечатал строку *echo*, но поскольку строка еще не завершена символом перевода строки, то в буфере чтения нет данных, и выполнение процесса приостановлено.

На шаге 2 пользователь напечатал *q*, затем передумал и набрал символ **erase** для удаления символа из строки ввода. Эта часть схемы подчеркивает, что



```
/* Программа read_demo - вызов read для терминала */  
  
#include sys/types.h  
#define SMALLSZ 10  
  
main(int argc, char **argv)  
{  
    ssize_t nread;  
    char smallbuf[SMALLSZ+1];  
  
    while((nread = read(0, smallbuf, SMALLSZ)) > 0)  
    {  
        smallbuf[nread] = "\0";  
        printf("nread: %d %s\n", nread, smallbuf);  
    }  
}
```

Если подать на вход программы следующий терминальный ввод

```
1  
1234  
Это более длинная строка  
<EOF>
```

то получится такой диалог:

```
1  
nread:2 1  
1234  
nread:5 1234  
Это более длинная строка  
nread 10 Это более  
nread 10 длинная с  
nread 6 трока
```

Обратите внимание, что для чтения самой длинной строки требуется несколько последовательных операций чтения. Заметим также, что значения счетчика `nread` включают также символ перехода на новую строку. Это не показано для простоты изложения.

Что происходит, если терминал не находится в каноническом режиме? В таком случае для полного управления процессом ввода программа должна устанавливать дополнительные параметры состояния терминала. Это осуществляется при помощи семейства системных вызовов, которые будут описаны позже.

---

**Упражнение 9.1.** Попробуйте запустить программу `read_demo`, перенаправив ее ввод на чтение файла.

---

### 9.3.3. Системный вызов `write`

Этот вызов намного проще в отношении взаимодействия с терминалом. Единственный важный момент заключается в том, что вызов `write` будет блокироваться при переполнении очереди вывода терминала. Программа продолжит работу,



только когда число символов в очереди станет меньше некоторого заданного порогового уровня.

### 9.3.4. Утилиты `ttyname` и `isatty`

Теперь представим две полезных утилиты, которые будем использовать в следующих примерах. Утилита `ttyname` возвращает имя терминального устройства, связанного с дескриптором открытого файла, а утилита `isatty` возвращает значение 1 (то есть *истинно* в терминах языка С), если дескриптор файла описывает терминальное устройство, и 0 (*ложно*) – в противном случае.

#### Описание

```
#include <unistd.h>

char* ttyname(int filedес);

int isatty(int filedес);
```

В обоих случаях параметр `fileдес` является дескриптором открытого файла. Если дескриптор `fileдес` не соответствует терминалу, то утилита `ttyname` вернет значение `NULL`. Иначе возвращаемое утилитой `ttyname` значение указывает строку в статической области памяти, которая переписывается при каждом вызове утилиты `ttyname`.

Следующий пример – процедура `what_tty` выводит имя терминала, связанного с дескриптором файла, если это возможно:

```
/* Процедура what_tty – выводит имя терминала */

void what_tty(int fd)
{
    if(isatty(fd))
        printf("fd %d =>> %s\n", fd, ttyname(fd));
    else
        printf("fd %d не является терминалом! \n", fd);
}
```

---

**Упражнение 9.2.** Измените процедуру `ttyopen` предыдущего раздела так, чтобы она возвращала дескриптор файла только для терминалов, а не для дисковых файлов или других типов файлов. Для выполнения проверки используйте утилиту `isatty`. Существуют ли еще какие-либо способы сделать это?

---

### 9.3.5. Изменение свойств терминала: структура `termios`

На уровне оболочки пользователь может вызвать команду `stty` для изменения свойств дисциплины линии связи терминала. Программа может сделать практически то же самое, используя структуру `termios` вместе с соответствующими функциями. Обратите внимание, что в более старых системах для этого использовался системный вызов `ioctl` (сокращение от *I/O control* – управление вводом/выводом), его применение было описано в первом издании этой книги. Вызов

`ioctl` предназначен для более общих целей и теперь разделен на несколько конкретных вызовов. Совокупность этих вызовов обеспечивает общий программный интерфейс ко всем асинхронным портам связи, независимо от свойств их оборудования.

Структуру `termios` можно представлять себе как объект, способный описать общее состояние терминала в соответствии с набором флагов, поддерживаемым системой для любого терминального устройства. Точное определение структуры `termios` будет вскоре рассмотрено. Структуры `termios` могут заполняться текущими установками терминала при помощи вызова `tcgetattr`, определенного следующим образом:

### Описание

```
#include <termios.h>
```

```
int tcgetattr(int ttyfd, struct termios *tsaved);
```

Эта функция сохраняет текущее состояние терминала, связанного с дескриптором файла `ttyfd` в структуре `tsaved` типа `termios`. Параметр `ttyfd` должен быть дескриптором файла, описывающим терминал.

### Описание

```
#include <termios.h>
```

```
int tcsetattr(int ttyfd, int actions,
              const struct termios *tnew);
```

Вызов `tcsetattr` установит новое состояние дисциплины связи, заданное структурой `tnew`. Второй параметр вызова `tcsetattr`, переменная `actions`, определяет, как и когда будут установлены новые атрибуты терминала. Существует три возможных варианта, определенных в файле `<termios.h>`:

<code>TCSANOW</code>	Немедленное выполнение изменений, что может вызвать проблемы, если в момент изменения флагов драйвер терминала выполняет вывод на терминал
<code>TCSADRAIN</code>	Выполняет ту же функцию, что и <code>TCSANOW</code> , но перед установкой новых параметров ждет опустошения очереди вывода
<code>TCSAFLUSH</code>	Аналогично <code>TCSADRAIN</code> ждет, пока очередь вывода не опустеет, а затем также очищает и очередь ввода перед установкой для параметров дисциплины линии связи значений, заданных в структуре <code>tnew</code>

Следующие две функции используют описанные вызовы. Функция `tsave` сохраняет текущие параметры, связанные с управляющим терминалом процесса, а функция `tback` восстанавливает последний набор сохраненных параметров. Флаг `saved` используется для предотвращения восстановления установок функцией `tback`, если перед этим не была использована функция `tsave`.

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

```
#include <termios.h>
```

```
#define SUCCESS 0
#define ERROR (-1)

/* Структура tsaved будет содержать параметры терминала */
static struct termios tsaved;

/* Равно TRUE если параметры сохранены */
static int saved = 0;
int tsave(void)
{
    if(isatty(0) && tcgetattr(0,&tsaved) >= 0)
    {
        saved = 1;
        return (SUCCESS);
    }
    return (ERROR);
}

int tback(void); /* Восстанавливает состояние терминала */
{
    if( !isatty(0) || !saved)
        return (ERROR);

    return tcsetattr(0, TCSAFLUSH, &tsaved);
}
```

Между этими двумя процедурами может быть заключен участок кода, который временно изменяет состояние терминала, например:

```
#include <stdio.h>

main()
{
    if(tsave() == 1)
    {
        fprintf(stderr, "Невозможно сохранить параметры терминала\n");
        exit(1);
    }
    /* Интересующий нас участок */

    tback();
    exit(0);
}
```

### **Определение структуры *termios***

Теперь изучим состав структуры *termios*. Определение структуры *termios* находится в заголовочном файле *<termios.h>* и содержит следующие элементы:

```
tcflag_t    c_iflag;      /* Режимы ввода */
tcflag_t    c_oflag;      /* Режимы вывода */
tcflag_t    c_cflag;      /* Управляющие режимы */
tcflag_t    c_lflag;      /* Режимы дисц. линии связи */
cc_t        c_cc[NCCS];   /* Управляющие символы */
```

Проще всего рассматривать структуру, начав с ее последнего элемента *c\_cc*.

### Массив `c_cc`

Символы редактирования строки, которые рассматривались в разделе 9.2.4, находятся в массиве `c_cc`. Их позиции в этом массиве задаются константами, определенными в файле `<termios.h>`. Все определенные в спецификации XSI значения приведены в табл. 9.1. Размер массива определяется константой `NCCS`, также определенной в файле `<termios.h>`.

Следующий фрагмент программы показывает, как можно изменить значение символа **quit** для терминала, связанного со стандартным вводом (дескриптор файла со значением 0):

```
struct termios tdes;

/* Получить исходные настройки терминала */
tcgetattr(0, &tdes);

tdes.c_cc[VQUIT] = "\031"; /* CTRL-Y */

/* Изменить установки терминала */
tcsetattr(0, TCSAFLUSH, &tdes);
```

Таблица 9.1. Коды управляющих символов

Константа	Значение
VINTR	Клавиша прерывания (Interrupt key)
VQUIT	Клавиша завершения (Quit key)
VERASE	Символ стирания (Erase character)
VKILL	Символ удаления строки (Kill character)
VEOF	Символ конца файла (EOF character)
VEOL	Символ конца строки (End of line marker – необязательный)
VSTART	Символ продолжения передачи данных (Start character)
VSTOP	Символ остановки передачи данных (Stop character)
VSUSP	Символ временной приостановки выполнения (Suspend character)

Этот пример иллюстрирует наиболее безопасный способ изменения состояния терминала. Сначала нужно получить текущее состояние терминала. Далее следует изменить только нужные параметры, не трогая остальные. И, наконец, изменить состояние терминала при помощи модифицированной структуры `termios`. Как уже было упомянуто, сохранять исходные значения полезно также для восстановления состояния терминала перед завершением программы, иначе нестандартное состояние терминала может оказаться сюрпризом для остальных программ.

### Поле `c_cflag`

Поле `c_cflag` содержит параметры, управляющие состоянием порта терминала. Обычно процессу не нужно изменять значения поля `c_cflag` своего управляющего терминала. Изменения этого поля могут понадобиться специальным коммуникационным приложениям, или программам, открывающим коммуникационные

линии для дополнительного оборудования, например, для принтера. Значения поля `c_cflag` образуются объединением при помощи операции ИЛИ битовых констант, определенных в файле `<termios.h>`. В общем случае каждая константа представляет бит поля `c_cflag`, который может быть установлен или сброшен. Не будем объяснять назначение всех битов этого поля (за полным описанием обратитесь к справочному руководству системы). Тем не менее есть четыре функции, которые позволяют опрашивать и устанавливать скорость ввода и вывода, закодированную в этом поле, не беспокоясь о правилах манипулирования битами.

### Описание

```
#include <termios.h>

/* Установить скорость ввода */
int cfsetispeed(struct termios *tdes, speed_t speed);

/* Установить скорость вывода */
int cfsetospeed(struct termios *tdes, speed_t speed);

/* Получить скорость ввода */
speed_t cfgetispeed(const struct termios *tdes);

/* Получить скорость вывода */
speed_t cfgetospeed(const struct termios *tdes);
```

Следующий пример устанавливает в структуре `tdes` значение скорости терминала равное 9600 бод. Постоянная `B9600` определена в файле `<termios.h>`.

```
struct termios tdes;

/* Получает исходные настройки терминала */
tcgetattr(0, &tdes);

/* Изменяет скорость ввода и вывода */
cfsetispeed(&tdes, B9600);
cfsetospeed(&tdes, B9600);
```

Конечно, эти изменения не будут иметь эффекта, пока не будет выполнен вызов `tcsetattr`:

```
tcsetattr(0, TCSAFLUSH, &tdes);
```

Следующий пример устанавливает режим контроля четности, напрямую устанавливая необходимые биты:

```
tdes.c_cflag |= (PARENB | PARODD);
tcsetattr(0, TCSAFLUSH, &tdes);
```

В этом примере установка флага `PARENB` включает проверку четности. Установленный флаг `PARODD` сообщает, что ожидаемый контроль – контроль нечетности. Если флаг `PARODD` сброшен и установлен флаг `PARENB`, то предполагается, что используется контроль по четности. (Термин *четность*, *parity*, относится к использованию битов проверки при передаче данных. Для каждого символа задается один такой бит. Это возможно благодаря тому, что набор символов ASCII занимает только семь бит из восьми, используемых для хранения символа на большинстве

компьютеров. Значение бита проверки может использоваться для того, чтобы полное число битов в байте было либо четным, либо нечетным. Программист также может полностью выключить проверку четности.)

### Поле `c_iflag`

Поле `c_iflag` в структуре `termios` служит для общего управления вводом с терминала. Не будем рассматривать все возможные установки, а выберем из них только наиболее общие.

Три из связанных с этим полем флага связаны с обработкой символа возврата каретки. Они могут быть полезны в терминалах, которые посылают для обозначения конца строки последовательность, включающую символ возврата каретки **CR**. ОС UNIX, конечно же, ожидает в качестве символа конца строки символ **LF** (line feed) или символ перевода строки ASCII, символ **NL** (new line). Следующие флаги могут исправить ситуацию:

INLCR	Преобразовывать символ новой строки в возврат каретки
IGNCR	Игнорировать возврат каретки
ICRNL	Преобразовывать возврат каретки в символ новой строки

Три других поля `c_iflag` связаны с управлением потоком данных:

IXON	Разрешить старт/стопное управление выводом
IXANY	Продолжать вывод при нажатии любого символа
IXOFF	Разрешить старт/стопное управление вводом

Флаг `IXON` дает пользователю возможность управления выводом. Если этот флаг установлен, то пользователь может прервать вывод, нажав комбинацию клавиш **Ctrl+S**. Вывод продолжится после нажатия комбинации **Ctrl+Q**. Если также установлен параметр `IXANY`, то для возобновления приостановленного вывода достаточно нажатия любой клавиши, хотя для остановки вывода должна быть нажата именно комбинация клавиш **Ctrl+S**. Если установлен флаг `IXOFF`, то система сама пошлет терминалу символ остановки (как обычно, **Ctrl+S**), когда буфер ввода будет почти заполнен. После того как система будет снова готова к приему данных, для продолжения ввода будет послана комбинация символов **Ctrl+Q**.

### Поле `c_oflag`

Поле `c_oflag` позволяет управлять режимом вывода. Наиболее важным флагом в этом поле является флаг `OPOST`. Если он не установлен, то выводимые символы передаются без изменений. В противном случае символы подвергаются обработке, заданной остальными флагами, устанавливаемыми в поле `c_oflag`. Некоторые из них вызывают подстановку символа возврата каретки (**CR**) при выводе на терминал:

ONLCR	Преобразовать символ возврата каретки ( <b>CR</b> ) в символ возврата каретки ( <b>CR</b> ) и символ перевода строки ( <b>NL</b> )
ONCRNL	Преобразовать символ возврата каретки ( <b>CR</b> ) в символ перевода строки ( <b>NL</b> )
ONOCR	Не выводить символ возврата каретки ( <b>CR</b> ) в нулевом столбце
ONLRET	Символ перевода строки ( <b>NL</b> ) выполняет функцию символа возврата каретки ( <b>CR</b> )

Если установлен флаг `ONLCR`, то символы перевода строки **NL** преобразуются в последовательность **CR+NL** (символ возврата каретки и символ перевода строки). Это гарантирует, что каждая строка будет начинаться с левого края экрана. И наоборот, если установлен флаг `OCRNL`, то символ возврата каретки будет преобразовываться в символ перевода строки. Установка флага `ONRLET` сообщает драйверу терминала, что для используемого терминала символы перевода строки будут автоматически выполнять и возврат каретки. Если установлен флаг `ONOCR`, то символ возврата каретки не будет посылаться при выводе строки нулевой длины.

Большинство остальных флагов поля `c_oflag` относятся к задержкам в передаче, связанным с интерпретацией специальных символов, таких как перевод строки, табуляция, перевод страницы и др. Эти задержки учитывают время позиционирования указателя знакоместа, где должен быть выведен следующий символ на экране или принтере. Подробное описание этих флагов должно содержать справочное руководство системы.

### Поле `c_flag`

Возможно, наиболее интересным элементом структуры `termios` для программиста является поле `c_flag`. Оно используется текущей дисциплиной линии связи для управления функциями терминала. Это поле содержит следующие флаги:

<code>ICANON</code>	Канонический построчный ввод
<code>ISIG</code>	Разрешить обработку прерываний
<code>IEXTEN</code>	Разрешить дополнительную (зависящую от реализации) обработку вводимых символов
<code>ECHO</code>	Разрешить отображение вводимых символов на экране
<code>ECHOE</code>	Отображать символ удаления как возврат-пробел-возврат
<code>ECHOK</code>	Отображать новую строку после удаления строки
<code>ECHONL</code>	Отображать перевод строки
<code>NOFLSH</code>	Отменить очистку буфера ввода после прерывания
<code>TOSTOP</code>	Посылать сигнал <code>SIGTTOU</code> при попытке вывода фонового процесса

Если установлен флаг `ICANON`, то включается канонический режим работы терминала. Как уже было видно выше, это позволяет использовать символы редактирования строки в процессе построчного ввода. Если флаг `ICANON` не установлен, то терминал находится в режиме прямого доступа (*raw mode*), который чаще всего используется полноэкранными программами и коммуникационными пакетами. Вызовы `read` будут при этом получать данные непосредственно из очереди ввода. Другими словами, основной единицей ввода будет одиночный символ, а не логическая строка. Программа при этом может считывать данные по одному символу (что обязательно для экранных редакторов) или большими блоками фиксированного размера (что удобно для коммуникационных программ). Но для того чтобы полностью управлять поведением вызова `read`, программист должен задать еще два дополнительных параметра. Это параметр `VMIN`, наименьшее число символов, которые должны быть приняты до возврата из вызова `read`, и параметр `VTIME`, максимальное время ожидания для вызова `read`. Оба параметра записываются

в массиве `c_cc`. Это важная тема, которая будет подробно изучена в следующем разделе. А пока просто обратим внимание на то, как в следующем примере сбрасывается флаг `ICANON`.

```
#include <termios.h>

struct termios tdes;
.
.
.
tcgetattr(0, &tdes);

tdes.c_lflag &= ~ICANON;

tcsetattr(0, TCSAFLUSH, &tdes);
```

Если установлен флаг `ISIG`, то разрешается обработка клавиш прерывания (**intr**) и аварийного завершения (**quit**). Обычно это позволяет пользователю завершить выполнение программы. Если флаг `ISIG` не установлен, то проверка не выполняется, и символы **intr** и **quit** передаются программе без изменений.

Если установлен флаг `ECHO`, то символы будут отображаться на экране по мере их набора. Сброс этого флага полезен для процедур проверки паролей и программ, которые используют клавиатуру для особых функций, например, для перемещения курсора или команд экранного редактора.

Если одновременно установлены флаги `ECHOE` и `ECHO`, то символ удаления будет отображаться как последовательность символов **backspace-space-backspace** (возврат-пробел-возврат). При этом последний символ на терминале немедленно стирается с экрана, и пользователь видит, что символ действительно был удален. Если флаг `ECHOE` установлен, а флаг `ECHO` нет, то символ удаления будет отображаться как **space-backspace**, тогда при его вводе будет удаляться символ в позиции курсора алфавитно-цифрового терминала.

Если установлен флаг `ECHONL`, то перевод строки будет всегда отображаться на экране, даже если отображение символов отключено, что может быть полезным при выполнении самим терминалом локального отображения вводимых символов. Такой режим часто называется *полудуплексным режимом* (half-duplex mode).

Последним флагом, заслуживающим внимания в этой группе флагов, является флаг `NOFLSH`, который подавляет обычную очистку очередей ввода и вывода при нажатии клавиш **intr** и **quit** и очистку очереди ввода при нажатии клавиши **susp**.

---

**Упражнение 9.3.** Напишите программу `ttystate`, которая выводит текущее состояние терминала, связанного со стандартным вводом. Эта программа должна использовать имена констант, описанных в этом разделе (`ICANON`, `ECHOE`, и т.д.). Найдите в справочном руководстве системы полный список этих имен.

---



**Упражнение 9.4.** *Напишите программу `ttyset`, которая распознает выходной формат программы `ttystate` и настраивает терминал, связанный с ее стандартным выводом в соответствии с описанным состоянием. Есть ли какая-то польза от программ `ttystate` и `ttyset`, вместе или по отдельности?*

### 9.3.6. Параметры `MIN` и `TIME`

Параметры `MIN` и `TIME` имеют значение только при выключенном флаге `ICANON`. Они предназначены для тонкой настройки управления вводом данных. Параметр `MIN` задает минимальное число символов, которое должен получить драйвер терминала для возврата из вызова `read`. Параметр `TIME` задает значение максимального интервала ожидания; этот параметр обеспечивает еще один уровень управления вводом с терминала. Время ожидания измеряется десятymi долями секунды.

Значения параметров `MIN` и `MAX` находятся в массиве `c_cc` структуры `termios`, описывающей состояние терминала. Их индексы в массиве определяются постоянными `VMIN` и `VTIME` из файла `<termios.h>`. Следующий фрагмент программы показывает, как можно задать их значения:

```
#include <termios.h>

struct termios tdes;
int ttyfd;

/* Получает текущее состояние */
tcgetattr(ttyfd, &tdes);

tdes.c_lflag &= ~ICANON; /* Отключает канонический режим */
tdes.c_cc[VMIN] = 64;    /* В символах */
tdes.c_cc[VTIME] = 2;    /* В десятых долях секунды */

tcsetattr(0, TCSAFLUSH, &tdes);
```

Константы `VMIN` и `VTIME` обычно имеют те же самые значения, что и постоянные `VEOF` и `VEOL`. Это означает, что параметры `MIN` и `TIME` занимают то же положение, что и символы `eof` и `eol`. Следовательно, при переключении из канонического в неканонический режим нужно обязательно задавать значения параметров `MIN` и `TIME`, иначе может наблюдаться странное поведение терминала. (В частности, если символу `eof` соответствует комбинация клавиш **Ctrl+D**, то программа будет читать ввод блоками по четыре символа.) Аналогичная опасность возникает при возврате в канонический режим.

Существуют четыре возможных комбинации параметров `MIN` и `TIME`:

- *оба параметра `MIN` и `TIME` равны нулю.* При этом возврат из вызова `read` обычно происходит немедленно. Если в очереди ввода терминала присутствуют символы (напомним, что попытка ввода может быть осуществлена в любой момент времени), то они будут помещены в буфер процесса. Поэтому, если программа переводит свой управляющий терминал в режим прямого

доступа при помощи сброса флага ICANON и оба параметра MIN и TIME равны нулю, то вызов

```
nread = read(0, buffer, SOMESZ);
```

вернет произвольное число символов от нуля до SOMESZ в зависимости от того, сколько символов находится в очереди в момент выполнения вызова;

- *параметр MIN больше нуля, а параметр TIME равен нулю.* В этом случае таймер не используется. Вызов read завершится только после того, как будут считаны MIN символов. Это происходит даже в том случае, если вызов read запрашивал меньше, чем MIN символов.

В самом простом варианте параметр MIN равен единице, а параметр TIME – нулю, что приводит к возврату из вызова read после получения каждого символа из линии терминала. Это может быть полезно при чтении с клавиатуры терминала, хотя могут возникнуть проблемы с клавишами, посылающими последовательности из нескольких символов;

- *параметр MIN равен нулю, а параметр TIME больше нуля.* В этом случае параметр MIN не используется. Таймер запускается сразу же после выполнения вызова read. Возврат из вызова read происходит, как только вводится первый символ. Если заданный интервал времени истекает (то есть проходит время, заданное в параметре TIME в десятых долях секунды), то вызов read возвращает нулевые символы;

- *оба параметра MIN и TIME больше нуля.* Это, возможно, наиболее полезный и гибкий вариант. Таймер запускается после получения первого символа, а не при входе в вызов read. Если MIN символов будут получены до истечения заданного интервала времени, то происходит возврат из вызова read. Если таймер срабатывает раньше, то в программу пользователя возвращаются только символы, находящиеся при этом в очереди ввода. Этот режим работы полезен при поступлении ввода пакетами, которые посылаются в течение коротких интервалов времени. При этом упрощается программирование и уменьшается число необходимых системных вызовов. Такой режим также полезен, например, при работе с функциональными клавишами, которые посылают при нажатии на них сразу несколько символов.

### 9.3.7. Другие системные вызовы для работы с терминалом

Есть несколько дополнительных системных вызовов для работы с терминалом, позволяющих программисту до некоторой степени управлять очередями ввода и вывода, поддерживаемыми драйвером терминала. Эти вызовы определены следующим образом.

#### Описание

```
#include <termios.h>

int tcflush(int ttyfd, int queue);

int tcdrain(int ttyfd);

int tcflow(int ttyfd, int actions);

int tcsendbrk(int ttyfd, int duration);
```

Вызов `tcflush` очищает заданную очередь. Если параметр `queue` имеет значение `TCIFLUSH` (определенное в файле `<termios.h>`), то очищается очередь ввода. Это означает, что все символы в очереди ввода сбрасываются. Если параметр `queue` имеет значение `TCOFLUSH`, то очищается очередь вывода. При значении `TCIOFLUSH` параметра `queue` очищаются и очередь ввода, и очередь вывода.

Вызов `tcdrain` приводит к приостановке работы процесса до тех пор, пока текущий вывод не будет записан в терминал `ttyfd`.

Вызов `tcflow` обеспечивает старт/стопное управление драйвером терминала. Если параметр `actions` равен `TCOOFF`, то вывод приостанавливается. Он может быть возобновлен при помощи еще одного вызова `tcflow` со значением параметра `actions` равным `TCOON`. Вызов `tcflow` также может использоваться для посылки драйверу терминала специальных символов `START` и `STOP`, это происходит при задании значения параметра `actions` равного `TCIOFF` или `TCION` соответственно. Специальные символы `START` и `STOP` служат для приостановки и возобновления ввода с терминала.

Вызов `tcsendbrk` используется для посылки сигнала прерывания сеанса связи, которому соответствует посылка нулевых битов в течение времени, заданного параметром `duration`. Если параметр `duration` равен 0, то биты посылаются в течение не менее четверти секунды и не более полсекунды. Если параметр `duration` не равен нулю, то биты будут посылаться в течение некоторого промежутка времени, длительность которого зависит от значения параметра `duration` и конкретной реализации.

### 9.3.8. Сигнал разрыва соединения

В главе 6 упоминалось, что сигнал разрыва соединения `SIGHUP` посылается членам сеанса в момент завершения работы лидера сеанса (при условии, что он имеет управляющий терминал). Этот сигнал также имеет другое применение в средах, в которых соединение между компьютером и терминалом может быть разорвано (тогда пропадает сигнал несущей в линии терминала). Это может происходить, например, если терминалы подключены через телефонную сеть или с помощью некоторых типов локальных сетей. В этих обстоятельствах драйвер терминала должен послать сигнал `SIGHUP` всем процессам, которые считают данный терминал своим управляющим терминалом. Если этот сигнал не перехватывается, то он приводит к завершению работы программы. (В отличие от сигнала `SIGINT`, сигнал `SIGHUP` обычно завершает и процесс оболочки. В результате пользователь автоматически отключается от системы при нарушении его связи с системой – такой подход необходим для обеспечения безопасности.)

Обычно программисту не нужно обрабатывать сигнал `SIGHUP`, так как он служит нужным целям. Тем не менее может потребоваться перехватывать его для выполнения некоторых операций по «наведению порядка» перед выходом из программы; вот как это можно сделать:

```
#include <signal.h>
```

```
void hup_action();  
static struct sigaction act;
```

```

.
.
.
act.sa_handler=hup_action;
sigaction(SIGHUP, &act, NULL);

```

Этот подход используется некоторыми редакторами, сохраняющими редактируемый файл и отсылающими пользователю сообщение перед выходом. Если сигнал SIGHUP полностью игнорируется (установкой значения `act.sa_handler` равного SIG\_IGN) и терминал разрывает соединение, то следующие попытки чтения из терминала будут возвращать 0 для обозначения «конца файла».

## 9.4. Псевдотерминалы

Еще одно применение модуля дисциплины линии связи заключается в поддержке работы так называемого *псевдотерминала* (pseudo terminal), применяемого для организации дистанционного доступа через сеть. Псевдотерминал предназначен для обеспечения соединения терминала одного компьютера с командным интерпретатором другого компьютера. Пример такого соединения приведен на рис. 9.4. На нем пользователь подключен к компьютеру А (клиенту), но использует командный интерпретатор на компьютере В (сервере). Стрелки на схеме показывают направление пересылки вводимых с клавиатуры символов. Здесь схема несколько упрощена за счет исключения стеков сетевых протоколов на клиентской и серверной системе.

Когда пользователь подключается к командному интерпретатору на другом компьютере (обычно при помощи команды `rlogin`), то локальное терминальное соединение должно быть изменено. По мере того как данные считываются с локального терминала, они должны без изменений передаваться через модуль дисциплины линии связи клиентскому процессу `rlogin`, выполняющемуся на локальном компьютере. Поэтому дисциплина линии связи на локальном компьютере должна работать в режиме прямого доступа. Следующий фрагмент должен напомнить, как включается такой режим:

```

#include <termio.h>

struct termios attr;
.
.
.
/* Получает текущую дисциплину линии связи */
tcgetattr(0, &attr);

/* Возврат из вызова read разрешен только после считывания одного символа */
attr.c_cc[VMIN] = 1;
attr.c_cc[VTIME] = 0;
attr.c_lflag &= ~(ISIG|ECHO|ICANON);

/* Устанавливает новую дисциплину линии связи */
tcsetattr(0, TCSAFLUSH, &attr);

```

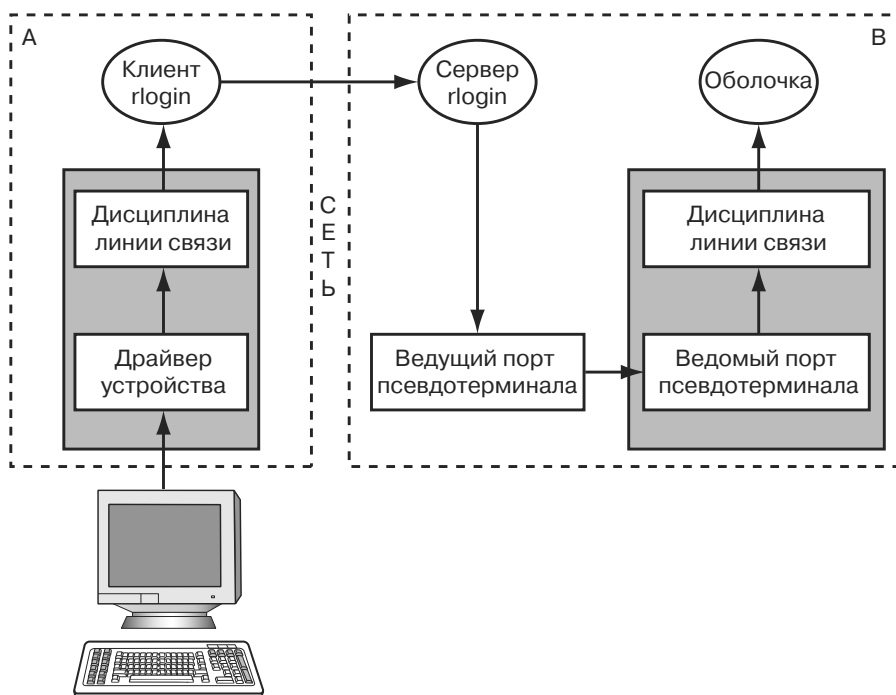


Рис. 9.4. Удаленный вход в систему в ОС UNIX

Теперь процесс клиента `rlogin` может передавать данные по сети в исходном виде.

Когда сервер (В) получает первоначальный запрос на вход в систему, он выполняет вызовы `fork` и `exec`, порождая новый командный интерпретатор. С новым командным интерпретатором не связан управляющий терминал, поэтому создается псевдотерминал, который имитирует обычный драйвер устройства терминала. Псевдотерминал, (*pseudo tty*), действует подобно двустороннему каналу и просто позволяет двум процессам передавать данные. В рассматриваемом примере псевдотерминал связывает командный интерпретатор с соответствующим сетевым процессом. Псевдотерминал является парой устройств, которые называются ведущим и ведомым устройствами, или портами псевдотерминала. Сетевой процесс открывает ведущий порт устройства псевдотерминала, а затем выполняет запись и чтение. Процесс командного интерпретатора открывает ведомый порт, а затем работает с ним (через дисциплину линии связи). Данные, записываемые в ведущий порт, попадают на вход ведомого порта, и наоборот. В результате пользователь на клиентском компьютере (А) как бы напрямую обращается к командному интерпретатору, который на самом деле выполняется на сервере (В). Аналогично, по мере того как данные выводятся командным интерпретатором на сервере, они обрабатываются дисциплиной линии связи на сервере (которая работает в каноническом режиме) и затем передаются без изменений клиентскому терминалу, не подвергаясь модификации со стороны дисциплины линии связи клиента.

Хотя средства, при помощи которых выполняется инициализация псевдотерминалов, были улучшены в новых версиях ОС UNIX и спецификации XSI, они все еще остаются довольно громоздкими. Система UNIX обеспечивает конечное число псевдотерминалов, и процесс командного интерпретатора должен открыть следующий доступный псевдотерминал. В системе SVR4 это выполняется при помощи открытия устройства `/dev/ptmx`, которое определяет и открывает первое неиспользуемое ведущее устройство псевдотерминала. С каждым ведущим устройством связано ведомое устройство. Для того, чтобы предотвратить открытие ведомого устройства другим процессом, открытие устройства `/dev/ptmx` также блокирует соответствующее ведомое устройство.

```
#include <fcntl.h>
int mfd;
.
.
.
/* Открывает псевдотерминал -
 * получает дескриптор файла главного устройства */
if( (mfd = open("/dev/ptmx", O_RDWR)) == -1)
{
    perror("Ошибка при открытии главного устройства");
    exit(1);
}
```

Перед тем как открыть и «разблокировать» ведомое устройство, необходимо убедиться, что только один процесс с соответствующими правами доступа сможет выполнять чтение из устройства и запись в него. Функция `grantpt` изменяет режим доступа и идентификатор владельца ведомого устройства в соответствии с параметрами связанного с ним главного устройства. Функция `unlockpt` снимает флаг, блокирующий ведомое устройство (то есть делает его доступным). Далее нужно открыть ведомое устройство. Но его имя пока еще не известно. Функция `ptsname` возвращает имя ведомого устройства, связанного с заданным ведущим устройством, которое обычно имеет вид `/dev/pts/pttyXX`. Следующий фрагмент демонстрирует последовательность необходимых действий:

```
#include <fcntl.h>

int mfd, sfd;
char *slavenm;
.
.
.
/* Открываем ведущее устройство, как и раньше */
if( (mfd = open("/dev/ptmx", O_RDWR)) == -1)
{
    perror("Ошибка при открытии ведущего устройства. ");
    exit(1);
}

/* Изменяем права доступа ведомого устройства */
```

```
if(grantpt(mfd) == -1)
{
    perror("Невозможно разрешить доступ к ведомому устройству");
    exit(1);
}

/* Разблокируем ведомое устройство, связанное с mfd */
if(unlockpt(mfd) == -1)
{
    perror("Невозможно разблокировать ведомое устройство");
    exit(1);
}

/* Получаем имя ведомого устройства и затем открыть его */
if( (slavenm = ptsname(mfd)) == NULL )
{
    perror("Невозможно получить имя ведомого устройства");
    exit(1);
}
if( (sfd = open(slavenm, O_RDWR)) == -1)
{
    perror("Ошибка при открытии ведомого устройства");
    exit(1);
}
```

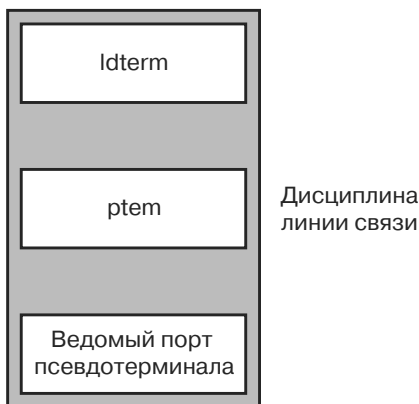
Теперь, когда получен доступ к драйверу устройства псевдотерминала, нужно установить для него дисциплину линии связи. До сих пор дисциплина линии связи рассматривалась как единое целое, тогда как в действительности она состоит из набора внутренних модулей ядра, известных как модули *STREAM*. Стандартная дисциплина линии связи псевдотерминала состоит из трех модулей: *ldterm* (модуль дисциплины линии связи терминала), *ptem* (модуль эмуляции псевдотерминала) и ведомой части псевдотерминала. Вместе они работают как настоящий терминал. Эта конфигурация показана на рис. 9.5.

Для создания дисциплины линии связи нужно «вставить» дополнительные модули *STREAM* в ведомое устройство. Это достигается при помощи многоцелевой функции *ioctl*, например:

```
/*
 * Заголовочный файл stropts.h содержит интерфейс STREAMS
 * и определяет макрокоманду I_PUSH, используемую в качестве
 * второго аргумента функции ioctl().
 */

#include <stropts.h>
.
.
.

/* Открываем ведущее и ведомое устройства, как и раньше */
/* Вставляем два модуля в ведомое устройство */
ioctl(sfd, I_PUSH, "ptem");
ioctl(sfd, I_PUSH, "ldterm");
```



*Рис. 9.5  
Дисциплина линии связи в виде модулей  
STREAM для устройства псевдотерминала*

Обратимся теперь к главному примеру, программе `tscript`, которая использует псевдотерминал в пределах одного компьютера для перехвата вывода командного интерпретатора в процессе интерактивного сеанса, не влияя на ход этого сеанса. (Эта программа аналогична команде UNIX `script`.) Данный пример можно расширить и для дистанционного входа через сеть.

## 9.5. Пример управления терминалом: программа `tscript`

Программа `tscript` устроена следующим образом: при старте она выполняет вызовы `fork` и `exec` для запуска пользовательской оболочки. Далее все данные, записываемые на терминал оболочкой, сохраняются в файле, при этом оболочка ничего об этом не знает и продолжает вести себя так, как будто она полностью управляет дисциплиной линии связи и, следовательно, терминалом. Логическая структура программы `tscript` показана на рис. 9.6.

Основные элементы схемы:

`tscript` Первый запускаемый процесс. После инициализации псевдотерминала и дисциплины линии связи этот процесс использует вызовы `fork` и `exec` для создания оболочки `shell`. Теперь программа `tscript` играет две роли. Первая состоит в чтении из настоящего терминала и записи всех данных в порт ведущего устройства псевдотерминала. (Все данные, записываемые в ведущее устройство псевдотерминала, непосредственно передаются на ведомое устройство псевдотерминала.) Вторая роль состоит в чтении вывода программы оболочки `shell` при помощи псевдотерминала и копировании этих данных на настоящий терминал и в выходной файл

`shell` Пользовательская оболочка. Перед запуском процесса `shell` модули дисциплины линии связи STREAM вставляются в ведомое устройство. Стандартный ввод, стандартный вывод и стандартный вывод диагностики оболочки перенаправляются в ведомое устройство псевдотерминала



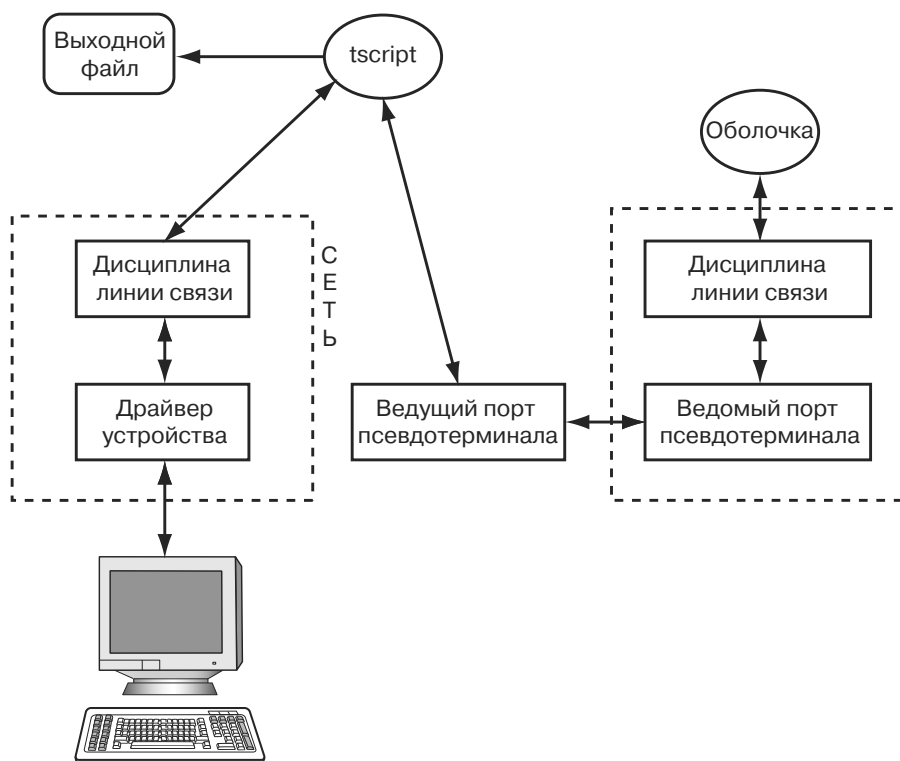


Рис. 9.6. Использование псевдотерминала в программе tscript

В примере используется заголовочный файл `tscript.h`. Он содержит следующие определения:

```

/* tscript.h - заголовочный файл для примера tscript */
#include <stropts.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stream.h>
#include <sys/ptms.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <termio.h>
#include <signal.h>

/* Внутренние функции */
void catch_child(int);
void runshell(int);
void script(int);
int ptyopen(int *,int *);

extern struct termios dattr;
  
```

Основная программа `tscript` приведена ниже. Первая задача программы состоит в установке обработчика сигнала `SIGCHLD` и в открытии псевдотерминала. Затем программа создает процесс `shell`. И, наконец, вызывается процедура `script`. Эта процедура отслеживает два потока данных: ввод с клавиатуры (стандартный ввод), который она передает ведущему устройству псевдотерминала, и ввод с ведущего устройства псевдотерминала, передаваемый на стандартный вывод и записываемый в выходной файл.

```
/* Программа tscript - управление терминалом */
#include "tscript.h"
struct termios dattr;

main()
{
    struct sigaction act;
    int          mfd, sfd;
    char          buf[512];

    /* Сохраняем текущие установки терминала */
    tcgetattr(0, &dattr);

    /* Открываем псевдотерминал */
    if (pttyopen(&mfd, &sfd) == -1)
    {
        perror("Ошибка при открытии псевдотерминала");
        exit(1);
    }

    /* Устанавливаем обработчик сигнала SIGCHLD */
    act.sa_handler = catch_child;
    sigfillset(&(act.sa_mask));
    sigaction(SIGCHLD, &act, NULL);

    /* Создаем процесс оболочки */
    switch(fork()){
    case -1: /* Ошибка */
        perror("Ошибка порождения дочернего процесса для вызова оболочки");
        exit(2);
    case 0: /* Дочерний процесс */
        close(mfd);
        runshell(sfd);
    default: /* Родительский процесс */
        close(sfd);
        script(mfd);
    }
}
```

Основная программа использует четыре процедуры. Первая из них называется `catch_child`. Это обработчик сигнала `SIGCHLD`. При получении сигнала `SIGCHLD` процедура `catch_child` восстанавливает атрибуты терминала и завершает работу.

```
void catch_child(int signo)
{
```

```
tcsetattr(0, TCSAFLUSH, &dattr);
exit(0);
}
```

Вторая процедура, `pttyopen`, открывает псевдотерминал.

```
int pttyopen(int *masterfd, int *slavefd)
{
    char *slavenm;

    /* Открываем псевдотерминал -
     * Получаем дескриптор файла ведущего устройства */
    if( (*masterfd = open("/dev/ptmx", O_RDWR)) == -1)
        return (-1);

    /* Изменяем права доступа для ведомого устройства */
    if(grantpt(*masterfd) == -1)
    {
        close(*masterfd);
        return (-1);
    }

    /* Разблокируем ведомое устройство, связанное с mfd */
    if(unlockpt(*masterfd) == -1)
    {
        close(*masterfd);
        return (-1);
    }

    /* Получаем имя ведомого устройства и затем открываем его */
    if( (slavenm = ptsname(*masterfd)) == NULL )
    {
        close(*masterfd);
        return (-1);
    }

    if( (*slavefd = open(slavenm, O_RDWR)) == -1)
    {
        close(*masterfd);
        return (-1);
    }

    /* Создаем дисциплину линии связи */
    if( ioctl(*slavefd, I_PUSH, "ptem") == -1)
    {
        close(*masterfd);
        close(*slavefd);
        return (-1);
    }

    if( ioctl(*slavefd, I_PUSH, "ldterm") == -1)
    {
        close(*masterfd);
        close(*slavefd);
    }
}
```

```
    return (-1);  
}  
  
return (1);  
}
```

Третья процедура – процедура `runshell`. Она выполняет следующие задачи:

- вызывает `setgrp`, чтобы оболочка выполнялась в своей группе процессов. Это позволяет оболочке полностью управлять обработкой сигналов, в особенности в отношении управления заданиями;
- вызывает системный вызов `dup2` для перенаправления дескрипторов `stdin`, `stdout` и `stderr` на дескриптор файла ведомого устройства. Это особенно важный шаг;
- запускает оболочку при помощи вызова `exec`, которая выполняется до тех пор, пока не будет прервана пользователем.

```
void runshell(int sfd)  
{  
    setpgrp();  
  
    dup2(sfd, 0);  
    dup2(sfd, 1);  
    dup2(sfd, 2);  
  
    execl("/bin/sh", "sh", "-i", (char *)0);  
}
```

Теперь рассмотрим саму процедуру `script`. Первым действием процедуры `script` является изменение дисциплины линии связи так, чтобы она работала в режиме прямого доступа. Это достигается получением текущих атрибутов терминала и изменением их при помощи вызова `tcsetattr`. Затем процедура `script` открывает файл `output` и использует системный вызов `select` (обсуждавшийся в главе 7) для обеспечения одновременного ввода со своего стандартного ввода и ведущего устройства псевдотерминала. Если данные поступают со стандартного ввода, то процедура `script` передает их без изменений ведущему устройству псевдотерминала. При поступлении же данных с ведущего устройства псевдотерминала процедура `script` записывает эти данные в терминал пользователя и в файл `output`.

```
void script(int mfd)  
{  
    int          nread, ofile;  
    fd_set       set, master;  
    struct       termios attr;  
    char         buf[512];  
  
    /* Переводим дисциплину линии связи в режим прямого доступа */  
    tcgetattr(0, &attr);  
  
    attr.c_cc[VMIN] = 1;
```

```
attr.c_cc[VTIME] = 0;
attr.c_lflag &= ~(ISIG|ECHO|ICANON);
tcsetattr(0, TCSAFLUSH, &attr);

/* Открываем выходной файл */
ofile = open("output", O_CREAT|O_WRONLY|O_TRUNC, 0666);

/* Задаем битовые маски для системного вызова select */
FD_ZERO(&master);
FD_SET(0, &master);
FD_SET(mfd, &master);

/* Вызов select осуществляется без таймаута,
 * и будет заблокирован до наступления события */
while(set=master, select(mfd+1, &set, NULL, NULL, NULL) > 0)
{
    /* Проверяем стандартный ввод */
    if(FD_ISSET(0, &set))
    {
        nread = read(0, buf, 512);
        write(mfd, buf, nread);
    }

    /* Проверяем главное устройство */
    if(FD_ISSET(mfd, &set))
    {
        nread = read(mfd, buf, 512);
        write(ofile, buf, nread);
        write(1, buf, nread);
    }
}
```

Следующий сеанс демонстрирует работу программы tscript. Комментарии, обозначенные символом #, показывают, какая из оболочек выполняется в данный момент.

```
$ ./tscnpt

$ ls -l tscript      # работает новая оболочка
-rwxr-xr-x  1 spate   fcf 6984 Jan 22 21:57 tscript

$ head -2 /etc/passwd # выполняется в новой оболочке
root:x:0:1:0000-Admin(0000):/:/bin/ksh
daemon:x:1:1:0000-Admin(0000):/:/

$ exit              # выход из новой оболочки

$ cat output        # работает исходная оболочка
-rwxr-xr-x  1 spate   fcf 6984 Jan 22 21:57 tscript
root:x:0:1:0000-Admin(0000):/:/bin/ksh
daemon:x:1:1:0000-Admin(0000):/:/
```

---

**Упражнение 9.5.** Добавьте к программе обработку ошибок и возможность задания в качестве параметра имени выходного файла. Если имя не задано, используйте по умолчанию имя `output`.

---

---

**Упражнение 9.6.** Эквивалентная стандартная программа UNIX `script` позволяет задать параметр `-a`, который указывает на необходимость дополнения файла `output` (содержимое файла не уничтожается). Реализуйте аналогичную возможность в программе `tscrip`.

---



# Глава 10. Сокеты

## 10.1. Введение

В предыдущих главах был изучен ряд механизмов межпроцессного взаимодействия системы UNIX. В настоящее время пользователями и разработчиками часто используются и сетевые среды, предоставляющие возможности технологии клиент/сервер. Такой подход позволяет совместно использовать данные, дисковое пространство, периферийные устройства, процессорное время и другие ресурсы. Работа в сетевой среде по технологии клиент/сервер предполагает взаимодействие процессов, находящихся на клиентских и серверных системах, разделенных средой передачи данных.

Исторически сетевые средства UNIX развивались двумя путями. Разработчики Berkeley UNIX создали в начале 80-х годов известный и широко применяемый интерфейс сокетов, а разработчики System V выпустили в 1986 г. Transport Level Interface (интерфейс транспортного уровня, сокращенно TLI). Раздел сетевого программирования в документации по стандарту X/Open часто называют спецификацией XTI. Спецификация XTI включает и интерфейс сокетов, и интерфейс TLI. Основные понятия являются общими для обеих реализаций, но интерфейс TLI использует намного больше структур данных, и его реализация гораздо сложнее, чем реализация интерфейса сокетов. Поэтому в этой главе будет рассмотрен хорошо известный и испытанный интерфейс сокетов. Они обеспечивают простой программный интерфейс, применимый как для связи процессов на одном компьютере, так и для связи процессов через сети. Целью сокетов является обеспечение средства межпроцессного взаимодействия для двустороннего обмена сообщениями между двумя процессами независимо от того, находятся ли они на одном или на разных компьютерах.

Глава будет посвящена краткому ознакомлению с основными понятиями и средствами работы с сокетами. При необходимости продолжить их изучение следует обратиться к более подробному руководству (например, книге «Advanced Programming in the UNIX Environment», Stevens, W.R., 1992) или к специалисту по сетевому программированию.

Кроме этого, необходимо отметить, что во время компоновки программ может понадобиться подключение сетевых библиотек (нужно задать в командной строке компилятора `cc` параметр `-lsocket` или некоторые другие параметры) – за рекомендациям обратитесь к справочному руководству системы.

## 10.2. Типы соединения

Если процессам нужно передать данные по сети, они могут выбрать для этого один из двух способов связи. Процесс, которому нужно посылать неформатированный, непрерывный поток символов одному и тому же абоненту, например, процессу удаленного входа в систему, может использовать *модель соединения* (connection oriented model) или *виртуальное соединение* (virtual circuit). В других же случаях (например, если серверу нужно разослать сообщение клиентам, не проверяя его доставку) процесс может использовать *модель дейтаграмм* (connectionless oriented model). При этом процесс может посылать сообщения (*дейтаграммы*) по произвольным адресам через один и тот же сокет без предварительного установления связи с этими адресами. Термин *дейтаграмма* (datagram) обозначает пакет пользовательского сообщения, посылаемый через сеть. Для облегчения понимания приведем аналогию: модель виртуальных соединений напоминает телефонную сеть, а модель дейтаграмм – пересылку писем по почте. Поэтому в последнем случае нельзя быть абсолютно уверенным, что сообщение дошло до адресата, а если необходимо получить ответ на него, то нужно указать свой обратный адрес на конверте. Модель соединений будет более подходящей при необходимости получения тесного взаимодействия между системами, когда обмен сообщениями и подтверждениями происходит в определенном порядке. Модель без соединений является более эффективной и лучше подходит в таких случаях, как рассылка широковещательных сообщений большому числу компьютеров.

Для того чтобы взаимодействие между процессами на разных компьютерах стало возможным, они должны быть связаны между собой как на аппаратном уровне при помощи сетевого оборудования – кабелей, сетевых карт и различных устройств маршрутизации, так и на программном уровне при помощи стандартного набора сетевых протоколов. Протокол представляет собой просто набор правил, в случае сетевого протокола – набор правил обмена сообщениями между компьютерами. Поэтому в системе UNIX должны существовать наборы правил для обеих моделей – как для модели соединений, так и для модели дейтаграмм. Для модели соединений используется *протокол управления передачей* (Transmission Control Protocol, сокращенно TCP), а для модели дейтаграмм – *протокол пользовательских дейтаграмм* (User Datagram Protocol, сокращенно UDP).<sup>1</sup>

## 10.3. Адресация

Чтобы процессы могли связаться по сети, должен существовать механизм определения *сетевого адреса* (network address) компьютера, на котором находится другой процесс. В конечном счете адрес определяет физическое положение компьютера в сети. Обычно адреса состоят из нескольких частей, соответствующих

<sup>1</sup> Протокол UDP не гарантирует доставку дейтаграмм; кроме того, может быть нарушен исходный порядок сообщений, допускается также случайное дублирование дейтаграмм. Приложения, использующие протокол UDP, должны обеспечивать контроль данных на прикладном уровне, для этого может потребоваться организация подтверждения доставки и повторной пересылки данных. – *Прим. науч. ред.*



различным уровням сети. Далее будут затронуты только те вопросы, без ответов на которые не обойтись при программировании с использованием сокетов.

### 10.3.1. Адресация Internet

Сейчас почти во всех глобальных сетях применима *адресация IP* (сокращение от Internet Protocol – межсетевой протокол, протокол сети Интернет).

Адрес IP состоит из четырех десятичных чисел, разделенных точками, например:

```
197.124.10.1
```

Эти четыре числа содержат достаточную информацию для определения сети назначения, а также компьютера в этой сети; собственно, термин Internet и означает «сеть сетей».

Сетевые вызовы UNIX не могут работать с IP адресами в таком формате. На программном уровне IP адреса хранятся в структуре типа `in_addr_t`. Обычно программистам не нужно знать внутреннее представление этого типа, так как для преобразования IP адреса в структуру типа `in_addr_t` предназначена процедура `inet_addr`.

#### Описание

```
#include <arpa/inet.h>
```

```
in_addr_t inet_addr(const char *ip_address);
```

Процедура `inet_addr` принимает IP адрес в форме строки вида `1.2.3.4` и возвращает адрес в виде структуры соответствующего типа. Если вызов процедуры завершается неудачей из-за неверного формата IP адреса, то возвращаемое значение будет равно (`in_addr_t`) – 1, например:

```
in_addr_t server;
```

```
server = inet_addr("197.124.10.1");
```

Для того чтобы процесс мог ссылаться на адрес своего компьютера, в заголовочном файле `<netinet.h>` определена постоянная `INADDR_ANY`, содержащая локальный адрес компьютера в формате `in_addr_t`.

### 10.3.2. Порты

Кроме адреса компьютера, клиентская программа должна иметь возможность подключения к нужному серверному процессу. Серверный процесс ждет подключения к заданному *номеру порта* (port number). Поэтому клиентский процесс должен выполнить запрос на подключение к определенному порту на заданном компьютере. Если продолжить аналогию с пересылкой писем по почте, то это равносильно дополнению адреса номером комнаты или квартиры.

Некоторые номера портов по соглашению считаются отведенными для стандартных сервисов, таких как `ftp` или `rlogin`. Эти номера портов записаны в файле `/etc/services`. В общем случае порты с номерами, меньшими 1024, считаются

зарезервированными для системных процессов UNIX. Все остальные порты доступны для пользовательских процессов.

## 10.4. Интерфейс сокетов

Для хранения информации об адресе и порте адресата (абонента) существуют стандартные структуры. Обобщенная структура адреса сокета определяется в заголовочном файле `<sys/socket.h>` следующим образом:

```
struct sockaddr{
    sa_family_t sa_family;    /* Семейство адресов */
    char sa_data[];          /* Адрес сокета */
};
```

Эта структура называется *обобщенным сокетом* (generic socket), так как в действительности применяются различные типы сокетов в зависимости от того, используются ли они в качестве средства межпроцессного взаимодействия на одном и том же компьютере или для связи процессов через сеть. Сокеты для связи через сеть имеют следующую форму:

```
#include <netinet/in.h>

struct sockaddr_in{
    sa_family_t      sin_family;    /* Семейство адресов */
    in_port_t        sin_port;      /* Номер порта */
    struct in_addr    sin_addr;      /* IP-адрес */
    unsigned char     sin_zero[8];  /* Поле выравнивания */
};
```

### 10.4.1. Создание сокета

При любых моделях связи клиент и сервер должны создать *абонентские точки* (transport end points), или *сокеты*, которые являются дескрипторами, используемыми для установки связи между процессами в сети. Они создаются при помощи системного вызова `socket`.

#### Описание

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

int socket(int domain, int type, int protocol);
```

Параметр `domain` определяет коммуникационный домен, в котором будет использоваться сокет. Например, значение `AF_INET` определяет, что будет использоваться домен Internet. Интерес может представлять также другой домен, `AF_UNIX`, который используется, если процессы находятся на одном и том же компьютере.

Параметр `type` определяет тип создаваемого сокета. Значение `SOCK_STREAM` указывается при создании сокета для работы в режиме виртуальных соединений, а значение `SOCK_DGRAM` – для работы в режиме пересылок дейтаграмм. Последний параметр `protocol` определяет используемый протокол. Этот параметр обычно задается равным нулю, при этом по умолчанию сокет типа `SOCK_STREAM` будет использовать протокол TCP, а сокет типа `SOCK_DGRAM` – протокол UDP. Оба данных

протокола являются стандартными протоколами UNIX. Поэтому виртуальное соединение часто называют TCP-соединением, а пересылку дейтаграмм – работой с UDP-сокетами.

Системный вызов `socket` обычно возвращает неотрицательное целое число, которое является дескриптором файла сокета, что позволяет считать механизм сокетов разновидностью обобщенного файлового ввода/вывода UNIX.

## 10.5. Программирование в режиме TCP-соединения

Для того чтобы продемонстрировать основные системные вызовы для работы с сокетами, рассмотрим пример, в котором клиент посылает серверу поток строчных символов через TCP-соединение. Сервер преобразует строчные символы в прописные и посылает их обратно клиенту. В следующих разделах этой главы приведем тот же самый пример, но использующий сокеты UDP-протокола.

Сначала составим план реализации серверного процесса:

```
/* Серверный процесс */

/* Включает нужные заголовочные файлы */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    int sockfd;

    /* Устанавливает абонентскую точку сокета */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit(1);
    }

    /*
    "Связывание" адреса сервера с сокетом

    Ожидание подключения

    Цикл
    установка соединения
    создание дочернего процесса для работы с соединением
    если это дочерний процесс,
        то нужно в цикле принимать данные от клиента и посылать ему ответы
    */
}
```

План клиентского процесса выглядит следующим образом:

```
/* Клиентский процесс */

/* Включает нужные заголовочные файлы */
```

```
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    int sockfd;

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit(1);
    }

    /* Соединяет сокет с адресом серверного процесса */
    /* В цикле посылает данные серверу и принимает от него ответы */
}
```

Далее будем постепенно превращать эти шаблоны в настоящие программы, начиная с реализации сервера.

### 10.5.1. Связывание

Системный вызов `bind` связывает сетевой адрес компьютера с идентификатором сокета.

#### Описание

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *address,
        size_t addrlen);
```

Первый параметр, `sockfd`, является дескриптором файла сокета, созданным с помощью вызова `socket`, а второй – указателем на обобщенную структуру адреса сокета. В рассматриваемом примере данные пересылаются по сети, поэтому в действительности в качестве этого параметра будет задан адрес структуры `sockaddr_in`, содержащей информацию об адресе нашего сервера. Последний параметр содержит размер указанной структуры адреса сокета. В случае успешного завершения вызова `bind` он возвращает значение 0. В случае ошибки, например, если сокет для этого адреса уже существует, вызов `bind` возвращает значение -1. Переменная `errno` будет иметь при этом значение `EADDRINUSE`.

### 10.5.2. Включение приема TCP-соединений

После выполнения связывания с адресом и перед тем, как какой-либо клиент сможет подключиться к созданному сокету, сервер должен включить прием соединений. Для этого служит вызов `listen`.

#### Описание

```
#include <sys/socket.h>
int listen(int sockfd, int queue_size);
```

Параметр `sockfd` имеет то же значение, что и в предыдущем вызове. В очереди сервера может находиться не более `queue_size` запросов на соединение. (Спецификация XSI определяет минимальное ограничение сверху на длину очереди равное пяти.)

### 10.5.3. Прием запроса на установку TCP-соединения

Когда сервер получает от клиента запрос на соединение, он должен создать новый сокет для работы с новым соединением. Первый же сокет используется только для установки соединения. Дополнительный сокет создается при помощи вызова `accept`, принимающего очередное соединение.

#### Описание

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

int accept(int sockfd, struct sockaddr *address,
           size_t *add_len);
```

Системному вызову `accept` передается дескриптор сокета, для которого ведется прием соединений. Возвращаемое значение соответствует идентификатору нового сокета, который будет использоваться для связи. Параметр `address` заполняется информацией о клиенте. Так как связь использует соединение, адрес клиента знать не обязательно, поэтому можно присвоить параметру `address` значение `NULL`. Если значение `address` не равно `NULL`, то переменная, на которую указывает параметр `add_len`, первоначально должна содержать размер структуры адреса, заданной параметром `address`. После возврата из вызова `accept` переменная `*add_len` будет содержать реальный размер записанной структуры.

После подстановки вызовов `bind`, `listen` и `accept` текст программы сервера примет вид:

```
/* Серверный процесс */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

int newsockfd;

main()
{
    int sockfd;

    /* Инициализация сокета Internet с номером порта 7000
     * и локальным адресом, заданным в постоянной INADDR_ANY */
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
```

```

perror("Ошибка вызова socket");
exit(1);
}

/* Связывает адрес с сокетом */
if ( bind(sockfd, (struct sockaddr *)&server, SIZE) == -1)
{
    perror("Ошибка вызова bind");
    exit(1);
}

/* Включает прием соединений */
if ( listen(sockfd, 5) == -1 )
{
    perror("Ошибка вызова listen");
    exit(1);
}

for ( ; ;)
{
    /* Принимает очередной запрос на соединение */
    if ( (newsockfd = accept(sockfd, NULL, NULL)) == -1)
    {
        perror("Ошибка вызова accept");
        continue;
    }
    /*
    Создает дочерний процесс для работы с соединением.
    Если это дочерний процесс,
    то в цикле принимает данные от клиента
    и посылает ему ответы.
    */
}
}

```

Обратите внимание на то, что сервер использует константу `INADDR_ANY`, соответствующую адресу локального компьютера.

Теперь имеется серверный процесс, способный переходить в режим приема соединений и принимать запросы на установку соединений. Рассмотрим, как клиент должен обращаться к серверу.

#### 10.5.4. Подключение клиента

Для выполнения запроса на подключение к серверному процессу клиент использует системный вызов `connect`.

##### Описание

```

#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *address,
            size_t add_len);

```

Первый параметр `csockfd` является дескриптором сокета клиента и не имеет отношения к дескриптору сокета на сервере. Параметр `address` указывает на структуру, содержащую адрес сервера, параметр `add_len` определяет размер используемой структуры адреса.

Продолжая составление рассматриваемого примера, запишем следующий вариант текста программы клиента:

```
/* Клиентский процесс */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    struct sockaddr_in server = {AF_INET, 7000};

    /* Преобразовывает и записывает IP адрес сервера */
    server.sin_addr.s_addr = inet_addr("206.45.10.2");

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit(1);
    }

    /* Соединяет сокет с сервером */
    if ( connect(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {
        perror("Ошибка вызова connect");
        exit(1);
    }

    /* Обмен данными с сервером */
}
```

Адрес сервера преобразуется в нужный формат при помощи вызова `inet_addr`. Адреса известных компьютеров локальной сети обычно можно найти в файле `/etc/hosts`.

### 10.5.5. Пересылка данных

Теперь уже освоена процедура установления соединения между клиентом и сервером. Для сокетов типа `SOCK_STREAM` и клиент, и сервер получают дескрипторы файлов, которые могут использоваться для чтения или записи. В большинстве случаев для этого годятся обычные вызовы `read` и `write`. Если же необходимо задавать дополнительные параметры пересылки данных по сети, то можно использовать два новых системных вызова – `send` и `recv`. Эти вызовы имеют

схожий интерфейс и ведут себя точно так же, как вызовы `read` и `write`, если их четвертый аргумент равен нулю.

### Описание

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

ssize_t recv(int sockfd, void *buffer, size_t length,
             int flags);

ssize_t send(int sockfd, const void *buffer, size_t length,
             int flags);
```

Вызов `recv` имеет четыре параметра: дескриптор файла `filedes`, из которого читаются данные, буфер `buffer`, в который они помещаются, размер буфера `length` и поле флагов `flags`.

Параметр `flags` указывает дополнительные опции получения данных. Его возможные значения определяются комбинациями следующих констант:

<code>MSG_PEEK</code>	Процесс может просматривать данные, не «получая» их
<code>MSG_OOB</code>	Обычные данные пропускаются. Процесс принимает только срочные данные, например, сигнал прерывания
<code>MSG_WAITALL</code>	Возврат из вызова <code>recv</code> произойдет только после получения всех данных

При аргументе `flags` равном нулю вызов `send` работает точно так же, как и вызов `write`, пересылая массив данных буфера `buffer` в сокет `sockfd`. Параметр `length` задает размер массива данных. Аналогично вызову `recv` параметр `flags` определяет опции передачи данных. Его возможные значения определяются комбинациями следующих констант:

<code>MSG_OOB</code>	Передать <i>срочные</i> (out of band) данные
<code>MSG_DONTROUTE</code>	При передаче сообщения игнорируются условия маршрутизации протокола более низкого уровня. Обычно это означает, что сообщение посылается по прямому, а не по самому быстрому маршруту (самый быстрый маршрут не обязательно прямой и может зависеть от текущего распределения нагрузки сети)

Теперь с помощью этих вызовов можно реализовать обработку данных на серверной стороне:

```
/* Серверный процесс */

main()
{
    /* Приведенная выше инициализация сокета */
    .
    .
    .
    char c;
```



```
for (;;)
{
    /* Принимает запрос на установку соединения */
    if ( (newsockfd = accept(sockfd, NULL, NULL)) == -1)
    {
        perror("Ошибка вызова accept");
        continue;
    }

    /* Создает дочерний процесс для работы с соединением */
    if ( fork() == 0)
    {
        /* Принимает данные */
        while (recv(newsockfd, &c, 1, 0) > 0)
        {
            /* Преобразовывает строчный символ в прописной */
            c = toupper(c);
            /* Пересылает символ обратно */
            send(newsockfd, &c, 1, 0);
        }
    }
}
```

Напомним, что использование вызова `fork` позволяет серверу обслуживать несколько клиентов. Цикл работы клиентского процесса может быть реализован так:

```
/* Клиентский процесс */
main()
{
    int sockfd;
    char c, rc;

    /* Приведенная выше инициализация сокета и запрос
     * на установку соединения */
    .
    .
    .
    /* Обмен данными с сервером */
    for(rc = '\n';;)
    {
        if (rc == '\n')
            printf("Введите строчный символ\n");
        c = getchar();
        send(sockfd, &c, 1, 0);
        recv(sockfd, &rc, 1, 0);
        printf("%c", rc);
    }
}
```

### 10.5.6. Закрытие TCP-соединения

При работе с сокетами важно корректно реагировать на завершение работы абонентского процесса. Так как сокет является двусторонним механизмом связи, то нельзя предсказать заранее, когда произойдет разрыв соединения – во время чтения или записи. Поэтому нужно учитывать оба возможных варианта.

Если процесс пытается записать данные в оборванный сокет при помощи вызова `write` или `send`, то он получит сигнал `SIGPIPE`, который может быть перехвачен соответствующим обработчиком сигнала. При чтении обрыв диагностируется проще.

В случае разорванной связи вызов `read` или `recv` возвращает нулевое значение. Поэтому для вызовов `read` и `recv` необходимо всегда проверять возвращаемое значение, чтобы не заиклиться при приеме данных.

Закрываются сокеты так же, как и обычные дескрипторы файлового ввода/вывода, – при помощи системного вызова `close`. Для сокета типа `SOCK_STREAM` ядро гарантирует, что все записанные в сокет данные будут переданы принимающему процессу. Это может вызвать блокирование операции закрытия сокета до тех пор, пока данные не будут доставлены. (Если сокет имеет тип `SOCK_DGRAM`, то сокет закрывается немедленно.)

Теперь можно привести полный текст примера клиента и сервера, добавив в серверный процесс обработку сигналов и вызов `close` в обе программы. В данном случае эти меры могут показаться излишними, но в реальном клиент/серверном приложении обязательна надежная обработка всех исключительных ситуаций. Приведем окончательный текст программы сервера:

```
/* Серверный процесс */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

#define SIZE sizeof(struct sockaddr_in)

void catcher(int sig);
int newsockfd;

main()
{
    int sockfd;
    char c;
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};
    static struct sigaction act;

    act.sa_handler = catcher;
    sigfillset(&(act.sa_mask));
    sigaction(SIGPIPE, &act, NULL);

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
```

```
perror ("Ошибка вызова socket");
exit(1);
}

/* Связывает адрес с сокетом */
if ( bind(sockfd, (struct sockaddr *)&server, SIZE) == -1)
{
    perror("Ошибка вызова bind");
    exit(1);
}

/* Включает прием соединений */
if ( listen(sockfd, 5) == -1 )
{
    perror("Ошибка вызова listen");
    exit(1);
}

for (;;)
{
    /* Принимает запрос на соединение */
    if ( (newsockfd = accept(sockfd, NULL, NULL)) == -1)
    {
        perror("Ошибка вызова accept");
        continue;
    }

    /* Создает дочерний процесс для работы с соединением */
    if ( fork() == 0)
    {
        while (recv(newsockfd, &c, 1, 0) > 0)
        {
            c = toupper(c);
            send(newsockfd, &c, 1, 0)
        }
        /* После того, как клиент прекратит передачу данных,
           сокет может быть закрыт и дочерний процесс
           завершает работу */
        close(newsockfd);
        exit(0);
    }

    /* В родительском процессе newsockfd не нужен */
    close(newsockfd);
}

void catcher(int sig)
{
    close(newsockfd);
    exit(0);
}
```

## И клиента:

```
/* Клиентский процесс */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c, rc;
    struct sockaddr_in server = {AF_INET, 7000};

    /* Преобразовывает и сохраняет IP адрес сервера */
    server.sin_addr.s_addr = inet_addr("197.45.10.2");

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit (1);
    }

    /* Соединяет сокет с сервером */
    if ( connect(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {
        perror("Ошибка вызова connect");
        exit(1);
    }

    /* Цикл обмена данными с сервером */
    for(rc = '\n';;)
    {
        if (rc == '\n')
            printf("Введите строчный символ\n");
        c = getchar();
        send(sockfd, &c, 1, 0);
        if(recv(sockfd, &rc, 1, 0)>0)
            printf("%c", rc);
        else
        {
            printf("Сервер не отвечает\n");
            close(sockfd);
            exit(1);
        }
    }
}
```

---

**Упражнение 10.1.** Запустите приведенную программу сервера и несколько клиентских процессов. Что произойдет после того, как все клиентские процессы завершат работу?

---

**Упражнение 10.2.** Измените код программ так, чтобы после того, как все клиентские процессы завершат свою работу, сервер также завершал работу после заданного промежутка времени, если не поступят новые запросы на соединение.

---

**Упражнение 10.3.** Измените код программ так, чтобы два взаимодействующих процесса выполнялись на одном и том же компьютере. В этом случае сокет должен иметь коммуникационный домен AF\_UNIX.

---

## 10.6. Программирование в режиме пересылок UDP-дейтаграмм

Перепишем теперь пример, используя модель дейтаграмм. Основное отличие будет заключаться в том, что дейтаграммы (UDP-пакеты), передаваемые между клиентом и сервером, могут достигать точки назначения в произвольном порядке. К тому же, как уже упоминалось, протокол UDP не гарантирует доставку пакетов. При работе с UDP-сокетами процесс клиента должен также сначала создать сокет и связать с ним свой локальный адрес при помощи вызова `bind`. После этого процесс сможет использовать этот сокет для отправки и приема UDP-пакетов. Чтобы послать сообщение, процесс должен знать адрес назначения, который может быть как конкретным адресом, так и шаблоном, называемым «широковещательным адресом» и обозначающим сразу несколько компьютеров.

### 10.6.1. Прием и передача UDP-сообщений

Для сокетов UDP есть два новых системных вызова – `sendto` и `recvfrom`.

Параметр `sockfd` в обоих вызовах задает связанный с локальным адресом сокет, через который принимаются и передаются пакеты.

#### Описание

```
ssize_t recvfrom(int sockfd, void *message, size_t length,
                 int flags, struct sockaddr *send_addr,
                 size_t *add_len);

ssize_t sendto(int sockfd, const void *message, size_t length,
               int flags, const struct sockaddr *dest_addr,
               size_t dest_len);
```

Если параметр `send_addr` равен `NULL`, то вызов `recvfrom` работает точно так же, как и вызов `recv`. Параметр `message` указывает на буфер, в который помещается принимаемая дейтаграмма, а параметр `length` задает число байтов, которые

должны быть считаны в буфер. Параметр `flags` принимает те же самые значения, что и в вызове `recv`. Два последних параметра помогают установить двустороннюю связь с помощью UDP-сокета. В структуру `send_addr` будет помещена информация об адресе и порте, откуда пришел прочитанный пакет. Это позволяет принимающему процессу направить ответ пославшему пакет процессу. Последний параметр является указателем на целочисленную переменную типа `size_t`, в которую помещается длина записанного в структуру `send_addr` адреса.

Вызов `sendto` противоположен вызову `recvfrom`. В этом вызове параметр `dest_addr` задает адрес узла сети и порт, куда должно быть передано сообщение, а параметр `dest_len` определяет длину адреса.

Адаптируем пример для модели дейтаграммных посылок.

```
/ * Сервер */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c;

    /* Локальный серверный порт */
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};

    /* Структура, которая будет содержать адрес */
    /* клиентского процесса */
    struct sockaddr_in client;
    int client_len = SIZE;

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit(1);
    }

    /* Связывает локальный адрес с сокетом */
    if ( bind(sockfd, (struct sockaddr *)&server, SIZE)==-1)
    {
        perror("Ошибка вызова bind");
        exit(1);
    }

    /* Бесконечный цикл ожидания очередного пакета данных */
    for( ; ;)
    {
        /* Принимает сообщение и записывает адрес клиента */
```

```
if(recvfrom(sockfd, &c, 1, 0,
            &client, &client_len)==-1)
{
    perror("Сервер: ошибка при приеме");
    continue;
}

c = toupper (c);

/* Посылает сообщение обратно */
if( sendto(sockfd, &c, 1, 0, &client, client_len) == -1)
{
    perror ("Сервер: ошибка при передаче");
    continue;
}

}
}
```

### Новый текст клиента:

```
/* Клиентский процесс */

#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c;

    /* Локальный порт на клиенте */
    struct sockaddr_in client = {AF_INET, INADDR_ANY, INADDR_ANY};

    /* Адрес удаленного сервера */
    struct sockaddr_in server = {AF_INET, 7000};

    /* Преобразовывает и записывает IP-адрес */
    server.sin_addr.s_addr = inet_addr("197.45.10.2");

    /* Создает сокет */
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("Ошибка вызова socket");
        exit(1);
    }

    /* Связывает локальный адрес с сокетом */
    if ( bind(sockfd, (struct sockaddr *)&client, SIZE) == -1)
    {
        perror("Ошибка вызова bind");
        exit(1);
    }
}
```

```
}

/* Считывает символ с терминала */
while( read(0, &c, 1) != 0)
{
    /* Передает символ серверу */
    if( sendto(sockfd, &c, 1, 0, &server, SIZE) == -1)
    {
        perror("Клиент: ошибка передачи");
        continue;
    }

    /* Принимает вернувшееся сообщение */
    if(recv(sockfd, &c, 1, 0) == -1)
    {
        perror("Клиент: ошибка приема");
        continue;
    }

    write (1, &c, 1);
}
}
```

---

**Упражнение 10.4.** Запустите сервер и несколько клиентов. Как сервер определяет, от какого клиента он принимает сообщение?

---

## 10.7. Различия между двумя моделями

Обсудим различия между двумя реализациями рассматриваемого примера с точки зрения техники программирования.

В обеих моделях сервер должен создать сокет и связать свой локальный адрес с этим сокетом. В модели TCP-соединений серверу следует после этого включить прием соединений. В модели UDP-сокеты этот шаг не нужен, зато на клиента возлагается больше обязанностей.

С точки зрения клиента – в модели TCP-соединений достаточно простого подключения к серверу. В модели UDP-сокеты клиент должен создать сокет и связать свой локальный адрес с этим сокетом.

И, наконец, для передачи данных обычно используются различные системные вызовы. Системные вызовы `sendto` и `recvfrom` могут использоваться в обеих моделях, но все же они обычно используются в UDP-модели, чтобы сервер мог получить информацию об отправителе и отправить обратно ответ.





# Глава 11. Стандартная библиотека ввода/вывода

## 11.1. Введение

В последних главах книги рассмотрим некоторые из стандартных библиотек процедур системы UNIX (а также большинства сред поддержки языка C в других операционных системах).

Начнем с изучения очень важной стандартной библиотеки ввода/вывода, образующей основную часть стандартной библиотеки C, поставляемой со всеми системами UNIX. Читатели кратко ознакомились со стандартным вводом/выводом во второй главе и уже встречались с некоторыми из входящих в его состав процедур, например, процедурами `getchar` и `printf`.

Основная цель стандартной библиотеки ввода/вывода состоит в предоставлении эффективных, развитых и переносимых средств доступа к файлам. Эффективность процедур, образующих библиотеку, достигается за счет обеспечения механизма автоматической буферизации, который не видим для пользователя и минимизирует число действительных обращений к файлам и число выполняемых низкоуровневых системных вызовов. Библиотека предлагает широкий выбор функций, таких как форматированный вывод и преобразование данных. Процедуры стандартного ввода/вывода являются переносимыми, так как они не привязаны к особым свойствам системы UNIX и на самом деле являются частью независимого от UNIX стандарта ANSI языка C. Любой полноценный компилятор языка C предоставляет доступ к стандартной библиотеке ввода/вывода независимо от используемой операционной системы.

## 11.2. Структура FILE

Процедуры *буферизованного ввода/вывода* идентифицируют открытые файлы (каналы, сокет, устройства и другие объекты) при помощи указателя на структуру типа `FILE`. Процедуры этого семейства также называют процедурами *стандартного ввода/вывода*, так как они содержатся в стандартной библиотеке языка C. Указатель на объект `FILE` часто называется также *поток*ом ввода/вывода и является аналогом файловых дескрипторов базового ввода/вывода.

Определение структуры `FILE` находится в стандартном заголовочном файле `<stdio.h>`. Следует отметить, что программисту нет необходимости знать устройство структуры `FILE`, более того, ее определение различно в разных системах.

Все данные, считываемые из файла или записываемые в файл, передаются через буфер структуры `FILE`. Например, стандартная процедура вывода сначала будет лишь заполнять символ за символом буфер. Только после заполнения буфера очередной вызов библиотечной процедуры вывода автоматически запишет его содержимое в файл вызовом `write`. Эти действия не видимы для пользовательской программы. Размер буфера составляет `BUFSIZ` байтов. Постоянная `BUFSIZ` определена в файле `<stdio.h>` и, как уже описывалось во второй главе, обычно задает размер блоков на диске. Как правило, ее значение равно 512 или 1024 байта.

Аналогично процедура ввода извлекает данные из буфера, связанного со структурой `FILE`. Как только буфер опустеет, для его заполнения автоматически считывается еще один фрагмент файла. Эти действия также не видимы для пользовательской программы.

Механизм буферизации стандартной библиотеки ввода/вывода гарантирует, что данные всегда считываются и записываются блоками стандартного размера. В результате число обращений к файлам и число внутренних системных вызовов поддерживаются на оптимальном уровне. Но поскольку вся буферизация скрыта внутри процедур ввода/вывода, программист может их использовать для чтения или записи произвольных порций данных, даже по одному байту. Поэтому программа может быть составлена исходя из требований простоты и наглядности, а проблему общей эффективности ввода/вывода решат стандартные библиотеки. Далее увидим, что стандартные библиотечные процедуры также обеспечивают простые в использовании средства форматирования. Поэтому для большинства приложений стандартный ввод/вывод является предпочтительным методом доступа к файлам.

## 11.3. Открытие и закрытие потоков: процедуры `fopen` и `fclose`

### Описание

```
#include <stdio.h>

FILE * fopen(const char *filename, const char *type);

int fclose(FILE *stream);
```

Библиотечные процедуры `fopen` и `fclose` являются эквивалентами вызовов `open` и `close`. Процедура `fopen` открывает файл, заданный параметром `filename`, и связывает с ним структуру `FILE`. В случае успешного завершения процедура `fopen` возвращает указатель на структуру `FILE`, идентифицирующую открытый файл, объект `FILE *` также часто называют открытым *потоком* ввода/вывода (эта структура `FILE` является элементом внутренней таблицы). Процедура `fclose` закрывает файл, заданный параметром `stream`, и, если этот файл использовался для вывода, также сбрасывает на диск все данные из внутреннего буфера.

В случае неудачи процедура `fopen` возвращает нулевой указатель `NULL`, определенный в файле `<stdio.h>`. В этом случае, так же как и для вызова `open`, переменная `errno` будет содержать код ошибки, указывающий на ее причину.

Второй параметр процедуры `fopen` указывает на строку, определяющую режим доступа. Она может принимать следующие основные значения:

- `r` Открыть файл `filename` только для чтения. (Если файл не существует, то вызов завершится неудачей и процедура `fopen` вернет нулевой указатель `NULL`)
- `w` Создать файл `filename` и открыть его только для записи. (Если файл уже существует, то он будет усечен до нулевой длины)
- `a` Открыть файл `filename` только для записи. Все данные будут добавляться в конец файла. Если файл не существует, он создается

Файл может быть также открыт для обновления, то есть программа может выполнять чтение из файла и запись в него. Другими словами, программа может одновременно выполнять для файла и операции ввода, и операции вывода без необходимости открывать его заново. В то же время из-за механизма буферизации такой ввод/вывод будет более ограниченным, чем режим чтения/записи, поддерживаемый вызовами `read` и `write`. В частности, после вывода нельзя осуществить ввод без вызова одной из стандартных процедур ввода/вывода `fseek` или `rewind`. Эти процедуры изменяют положение внутреннего указателя чтения/записи и обсуждаются ниже. Аналогично нельзя выполнить вывод после ввода без вызова процедур `fseek` или `rewind` или процедуры ввода, которая перемещает указатель в конец файла. Режим обновления обозначается символом `+` в конце аргумента, передаваемого процедуре `fopen`. Вышеприведенные режимы можно дополнить следующим образом:

- `r+` Открыть файл `filename` для чтения и записи. Если файл не существует, то вызов снова завершится неудачей
- `w+` Создать файл `filename` и открыть его для чтения и записи. (Если файл уже существует, то он будет усечен до нулевой длины)
- `a+` Открыть файл `filename` для чтения и записи. При записи данные будут добавляться в конец файла. Если файл не существует, то он создается

В некоторых системах для доступа к двоичным, а не текстовым файлам, к строке также нужно добавлять символ `b`, например, `rb`.

Если файл создается при помощи процедуры `fopen`, для него обычно устанавливается код доступа `0666`. Это позволяет всем пользователям выполнять чтение из файла и запись в него. Эти права доступа по умолчанию могут быть изменены установкой ненулевого значения атрибута процесса `umask`. (Системный вызов `umask` был изучен в главе 3.)

Следующий пример программы показывает использование процедуры `fopen` и ее связь с процедурой `fclose`. При этом, если файл `indata` существует, то он открывается для чтения, а файл `outdata` создается (или усекается до нулевой длины, если он существует). Процедура `fatal` предназначена для вывода сообщения об ошибке, ее описание было представлено в предыдущих главах. Она просто передает свой аргумент процедуре `perror`, а затем вызывает `exit` для завершения работы программы.

```
#include <stdio.h>
char *iname = "indata";
char *outname = "outdata";
main()
{
    FILE *inf, *outf;

    if( (inf = fopen(iname, "r")) == NULL)
        fatal("Невозможно открыть входной файл");
    if( (outf = fopen(outname, "w")) == NULL)
        fatal("Невозможно открыть выходной файл");

    /* Выполняются какие-либо действия */

    fclose(inf);
    fclose(outf);

    exit(0);
}
```

На самом деле, в данном случае оба вызова `fclose` не нужны. Дескрипторы, связанные с файлами `inf` и `outf`, будут автоматически закрыты при завершении работы процесса, и вызов `exit` автоматически сбросит данные из буфера процедуры `outf` на диск, записав их в файл `outdata`.

С процедурой `fclose` тесно связана процедура `fflush`:

### **Описание**

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

Выполнение этой процедуры приводит к сбросу на диск содержимого буфера вывода, связанного с потоком `stream`. Другими словами, данные из буфера записываются в файл немедленно, независимо от того, заполнен буфер или нет. Это гарантирует, что содержимое файла на диске будет соответствовать тому, как он выглядит с точки зрения процесса. (Процесс считает, что данные записаны в файл с того момента, как они оказываются в буфере, поскольку механизм буферизации прозрачен.) Любые данные из буфера ввода этим вызовом предусмотрительно отбрасываются.

Поток `stream` остается открытым после завершения процедуры `flush`. Как и процедура `fclose`, процедура `fflush` возвращает постоянную EOF в случае ошибки и нулевое значение – в случае успеха. (Значение постоянной EOF задано в файле `<stdio.h>` равным `-1`. Оно обозначает конец файла, но может также использоваться для обозначения ошибок.)

## **11.4. Посимвольный ввод/вывод: процедуры `getc` и `putc`**

### **Описание**

```
#include <stdio.h>
int getc(FILE *inf);
int putc(int c, FILE *outf);
```

Наиболее простыми из процедур стандартной библиотеки ввода/вывода являются процедуры `getc` и `putc`. Процедура `getc` возвращает очередной символ из входного потока `inf`. Процедура `putc` помещает символ, обозначенный параметром `c`, в выходной поток `outf`.

В обеих процедурах символ `c` имеет тип `int`, а не `char`, что позволяет процедурам использовать наборы 16-битовых «широких» символов. Это также позволяет процедуре `getc` возвращать значение `-1`, находящееся вне диапазона возможных значений типа `unsigned char`. Постоянная `EOF` используется процедурой `getc` для обозначения того, что либо достигнут конец файла, либо произошла ошибка. Процедура `putc` также может возвращать значение `EOF` в случае ошибки.

Следующий пример является новой версией процедуры `copyfile`, представленной в главе 2; в данном случае вместо использования вызовов `read` и `write` используются процедуры `getc` и `putc`:

```
#include <stdio.h>

/* Скопировать файл f1 в файл f2
 * при помощи стандартных процедур ввода/вывода */
int copyfile(const char *f1, const char *f2)
{
    FILE *inf, *outf;
    int c;

    if( (inf = fopen(f1, "r")) == NULL)
        return (-1);

    if( (outf = fopen(f2, "w")) == NULL)
    {
        fclose(inf);
        return (-2);
    }

    while( (c = getc(inf)) != EOF)
        putc(c, outf);

    fclose(inf);
    fclose(outf);
    return (0);
}
```

Копирование выполняет внутренний цикл `while`. Снова обратите внимание на то, что переменная `c` имеет тип `int`, а не `char`.

Здесь следует сделать предостережение: процедуры `getc` и `putc` могут не быть функциями. Если они реализованы в виде макросов, то их поведение может быть странным, когда им передаются сложные аргументы. В частности, выражения `getc(*f++)` и `putc(c, *f++)` в этом случае дадут неверный результат. Реализация в виде макроса, конечно же, повышает эффективность за счет устранения процесса вызова функции. Но для полноты функционального набора гарантируется наличие функциональных аналогов `fgetc` и `fputc`. Эти функции редко используются, но они незаменимы, если имя функции нужно передать в качестве параметра другой функции.

**Упражнение 11.1.** В упражнениях 2.4 и 2.5 мы описали программу `count`, которая выводит число символов, слов и строк во входном файле. (Напомним, что слово определялось, как любая последовательность алфавитно-цифровых символов или одиночный пробельный символ.) Перепишите программу `count`, используя процедуру `getc`.

**Упражнение 11.2.** Используя процедуру `getc`, напишите программу, выводящую статистику распределения символов в файле, то есть число раз, которое встречается в файле каждый символ. Один из способов сделать это состоит в использовании массива целых чисел типа `long`, который будет содержать счетчики числа символов, а затем рассматривать значение каждого символа в качестве индекса увеличиваемого счетчика массива. Убедитесь, что ваша программа работает корректно, даже если тип `char` в вашей системе определен как `signed` (то есть значения символов могут быть отрицательными). Программа также должна рисовать простую гистограмму полученного распределения при помощи процедур `printf` и `putc`.

## 11.5. Возврат символов в поток: процедура `ungetc`

### Описание

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

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

Процедура `ungetc` возвращает символ `c` в поток `stream`. Это всего лишь логическая операция. Входной файл не будет при этом изменяться. В случае успешного завершения процедуры `ungetc` символ `c` будет следующим символом, который будет считан процедурой `getc`. Гарантируется возврат только одного символа. В случае неудачной попытки вернуть символ `c` процедура `ungetc` возвращает значение `EOF`. Попытка вернуть сам символ `EOF` должна всегда завершаться неудачей. Но это обычно не представляет проблемы, так как все последующие вызовы процедуры `getc` после достижения конца файла приведут к возврату символа `EOF`.

Обычно процедура `ungetc` используется для восстановления исходного состояния входного потока после чтения лишнего символа для проверки условия. Следующая процедура `getword` применяет это простой подход для ввода строки, которая содержит либо непрерывную последовательность алфавитно-цифровых символов, либо одиночный нетекстовый символ. Конец файла кодируется возвращенным значением `NULL`. Процедура `getword` принимает в качестве аргумента указатель на структуру `FILE`. Она использует для проверки два макроса, определенные в стандартном заголовочном файле `<ctype.h>`. Первый из них, `isspace`, определяет, является ли символ пробельным символом, таким как символ пробела, табуляции или перевода строки. Вторым, `isalnum`, проверяет, является ли символ алфавитно-цифровым, то есть цифрой или буквой.

```
#include <stdio.h>

/* В этом файле определены макрокоманды isspace и isalnum */
#include <ctype.h>

#define MAXТОК    256

static char inbuf[MAXТОК+1] ;

char *getword(FILE *inf)
{
    int c, count = 0;

    /* Удаляет пробельные символы */
    do{
        c = getc(inf);
    } while ( isspace(c) );

    if(c == EOF)
        return (NULL);

    if( !isalnum(c)) /* Символ не является алфавитно-цифровым */
        inbuf[count++] = c;

    else
    {
        /* Сборка "слова" */
        do{
            if(count < MAXТОК)
                inbuf[count++] = c;

            c = getc (inf);
        } while( isalnum(c));
        ungetc(c, inf); /* Необходимо вернуть символ */
    }

    inbuf[count] = '\0'; /* Нулевой символ в конце строки */
    return (inbuf);
}
```

Если подать на вход программы следующий ввод

Это данные  
на входе  
программы!!!

то процедура getword вернет следующую последовательность строк:

Это  
данные  
на  
входе  
программы  
!  
!  
!

**Упражнение 11.3.** Измените процедуру `getword` так, чтобы она распознавала также числа, которые могут начинаться со знака минус или плюс и могут содержать десятичную точку.

## 11.6. Стандартный ввод, стандартный вывод и стандартный вывод диагностики

Стандартная библиотека ввода/вывода обеспечивает три структуры `FILE`, связанные со стандартным вводом, стандартным выводом и стандартным выводом диагностики. (Еще раз напомним, что не следует путать эти потоки с одноименными дескрипторами ввода/вывода 0, 1 и 2.) Эти стандартные структуры `FILE` не требуют открытия и задаются предопределенными указателями:

<code>stdin</code>	Соответствует стандартному вводу
<code>stdout</code>	Соответствует стандартному выводу
<code>stderr</code>	Соответствует стандартному выводу диагностики

Следующий вызов получает очередной символ из структуры `stdin`, которая так же, как и дескриптор файла со значением 0, по умолчанию соответствует клавиатуре:

```
inchar = getc(stdin);
```

Так как ввод и вывод через потоки `stdin` и `stdout` используются очень часто, для удобства определены еще две процедуры – `getchar` и `putchar`. Процедура `getchar` возвращает очередной символ из `stdin`, а процедура `putchar` выводит символ в `stdout`. Они аналогичны процедурам `getc` и `putc`, но не имеют аргументов.

Следующая программа `io2` использует процедуры `getchar` и `putchar` для копирования стандартного ввода в стандартный вывод:

```
/* Программа io2 – копирует stdin в stdout */  
  
#include <stdio.h>  
  
main()  
{  
    int c;  
  
    while( (c = getchar() ) != EOF)  
        putchar(c);  
}
```

Программа `io2` ведет себя почти аналогично приведенному ранее примеру – программе `io` из главы 2.

Так же, как `getc` и `putc`, `getchar` и `putchar` могут быть макросами. Фактически `getchar()` часто просто определяется как `getc(stdin)`, а `putchar()` – как `putc(stdout)`.

Структура `stderr` обычно предназначена для вывода сообщений об ошибках, поэтому вывод в `stderr` обычно не буферизуется. Другими словами, символ,



который посылается в `stderr`, будет немедленно записан в файл или устройство, соединенное со стандартным выводом диагностики. При включении отладочной печати в код для тестирования рекомендуется выполнять вывод в `stderr`. Вывод в `stdout` буферизуется и может появиться через несколько шагов после того, как он в действительности произойдет. (Вместо этого можно использовать процедуру `fflush(stdout)` после каждого вывода для записи всех сообщений из буфера `stdout`.)<sup>1</sup>

---

**Упражнение 11.4.** При помощи стандартной команды `time` сравните производительность программы `io2` и программы `io`, разработанной в главе 2. Измените исходную версию программы `io` так, чтобы она использовала вызовы `read` и `write` для посимвольного ввода и вывода. Снова сравните производительность полученной программы и программы `io2`.

---

**Упражнение 11.5.** Перепишите программу `io2` так, чтобы она более соответствовала команде `cat`. В частности, сделайте так, чтобы она выводила на экран содержимое файлов, заданных в качестве аргументов командной строки. При отсутствии аргументов она должна принимать ввод из `stdin`.

---

## 11.7. Стандартные процедуры опроса состояния

Для опроса состояния структуры `FILE` существует ряд простых процедур. Они, например, позволяют программе определять, вернула ли процедура ввода, такая как `getc`, символ EOF из-за того, что достигнут конец файла или в результате возникновения ошибки. Эти процедуры описаны ниже:

### Описание

```
#include <stdio.h>

int ferror(FILE *stream);

int feof(FILE *stream);

void clearerr(FILE *stream);

int fileno(FILE *stream);
```

Функция `ferror` является предикатом, который возвращает ненулевое значение (то есть значение *истинно* в языке C), если в потоке `stream` возникла ошибка во время последнего запроса на ввод или вывод. Ошибка может возникать в результате вызова примитивов доступа к файлам (`read`, `write` и др.) внутри процедуры стандартного ввода/вывода. Если же функция `ferror` возвращает нулевое

---

<sup>1</sup> Потоки `stdin`, `stdout` и `stderr` не следует закрывать: это может привести к аварийному завершению процесса, так как соответствующие структуры `FILE` часто размещены в статической, а не динамической памяти. — Прим. науч. ред.

значение (то есть *ложно*), значит, ошибок не было. Функция `ferror` может использоваться следующим образом:

```
if(ferror(stream)) { /* Обработка ошибок */
}
else { /* Ошибок нет */
}
```

Функция `feof` является предикатом, возвращающим ненулевое значение, если для потока `stream` достигнут конец файла. Возврат нулевого значения просто означает, что этого еще не произошло.

Функция `clearerr` используется для сброса индикаторов ошибки и флага достижения конца файла для потока `stream`. При этом гарантируется, что последующие вызовы функций `ferror` и `feof` для этого файла вернут нулевое значение, если за это время не произошло что-нибудь еще. Очевидно, что функция `clearerr` бывает необходима редко.

Функция `fileno` является вспомогательной и не связана с обработкой ошибок. Она возвращает целочисленный дескриптор файла, содержащийся в структуре `FILE`, на которую указывает параметр `stream`. Это может быть полезно, если нужно передать какой-либо процедуре дескриптор файла, а не идентификатора потока `FILE`. Однако не следует использовать процедуру `fileno` для смешивания вызовов буферизованного и небуферизованного ввода/вывода. Это почти неизбежно приведет к хаосу.

Следующий пример – процедура `egetc` использует функцию `ferror`, чтобы отличить ошибку от достижения конца файла при возврате процедурой `getc` значения EOF.

```
/* Процедура egetc – getc с проверкой ошибок */
#include <stdio.h>

int egetc(FILE *stream)
{
    int c;
    c = getc(stream);
    if(c == EOF)
    {
        if(ferror(stream))
        {
            fprintf(stderr, "Фатальная ошибка: ошибка ввода \n");
            exit(1);
        }
        else
            fprintf(stderr, "Предупреждение: EOF\n");
    }
    return (c);
}
```

Обратите внимание на то, что все описанные в этом разделе функции обычно реализуются в виде макросов и на них распространяются уже упомянутые выше предупреждения.

## 11.8. Построчный ввод и вывод

Существует также набор простых процедур для ввода и вывода строк (под которыми понимается последовательность символов, завершаемая символом перевода строки). Эти процедуры удобно использовать в интерактивных программах, которые выполняют чтение с клавиатуры и вывод на экран терминала. Основные процедуры для ввода строк называются `gets` и `fgets`.

### Описание

```
#include <stdio.h>

char * gets(char *buf);

char * fgets(char *buf, int nsize, FILE *inf);
```

Процедура `gets` считывает последовательность символов из потока стандартного ввода (`stdin`), помещая все символы в буфер, на который указывает аргумент `buf`. Символы считываются до тех пор, пока не встретится символ перевода строки или конца файла. Символ перевода строки `newline` отбрасывается, и вместо него в буфер `buf` помещается нулевой символ, образуя завершенную строку. В случае возникновения ошибки или при достижении конца файла возвращается значение `NULL`.

Процедура `fgets` является обобщенной версией процедуры `gets`. Она считывает символы из потока `inf` в буфер `buf` до тех пор, пока не будет считано `nsize-1` символов или не встретится раньше символ перевода строки `newline` или не будет достигнут конец файла. В процедуре `fgets` символы перевода строки `newline` не отбрасываются, а помещаются в конец буфера (это позволяет вызывающей функции определить, в результате чего произошел возврат из процедуры `fgets`). Как и процедура `gets`, процедура `fgets` возвращает указатель на буфер `buf` в случае успеха и `NULL` – в противном случае.

Процедура `gets` является довольно примитивной. Так как она не знает размер передаваемого буфера, то слишком длинная строка может привести к возникновению внутренней ошибки в процедуре. Чтобы избежать этого, можно использовать процедуру `fgets` (для стандартного ввода `stdin`).

Следующая процедура `yesno` использует процедуру `fgets` для получения положительного или отрицательного ответа от пользователя; она также вызывает макрос `isspace` для пропуска пробельных символов в строке ответа:

```
/* Процедура yesno – получить ответ от пользователя */

#include <stdio.h>
#include <ctype.h>

#define YES      1
#define NO       0
#define ANSWSZ   80

static char *pdefault = "Наберите 'y' (YES), или 'n' (NO)";
static char *error = "Неопределенный ответ";

int yesno(char *prompt)
{
    char buf[ANSWSZ], *p_use, *p;
```

```
/* Выводит приглашение, если оно не равно NULL
 * Иначе использует приглашение по умолчанию pdefault */
p_use = (prompt != NULL) ? prompt : pdefault;

/* Бесконечный цикл до получения правильного ответа */
for(;;)
{
    /* Выводит приглашение */
    printf("%s > ", p_use);

    if( fgets(buf, ANSWSZ, stdin) == NULL )
        return EOF;

    /* Удаляет пробельные символы */
    for(p = buf; isspace(*p); p++)
        ;

    switch(*p){
        case 'Y':
        case 'y':
            return (YES);
        case 'N':
        case 'n':
            return (NO);
        default:
            printf("\n%s\n", error);
    }
}
}
```

В этом примере предполагается, что `stdin` связан с терминалом. Как можно сделать эту процедуру более безопасной?

Обратными процедурами для `gets` и `fgets` будут соответственно процедуры `puts` и `fputs`.

### Описание

```
#include <stdio.h>

int puts(const char *string);

int fputs(const char *string, FILE *outf);
```

Процедура `puts` записывает все символы (кроме завершающего нулевого символа) из строки `string` на стандартный вывод (`stdout`). Процедура `fputs` записывает строку `string` в поток `outf`. Для обеспечения совместимости со старыми версиями системы процедура `puts` добавляет в конце символ перевода строки, процедура же `fputs` не делает этого. Обе функции возвращают в случае ошибки значение `EOF`.

Следующий вызов процедуры `puts` приводит к выводу сообщения `Hello, world` на стандартный вывод, при этом автоматически добавляется символ перевода строки `newline`:

```
puts("Hello, world");
```

## 11.9. Ввод и вывод бинарных данных: процедуры fread и fwrite

### Описание

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t nitems,
             FILE *inf);

size_t fwrite(const void *buffer, size_t size, size_t nitems,
             FILE *outf);
```

Эти две полезные процедуры обеспечивают ввод и вывод произвольных текстовых данных. Процедура `fread` считывает `nitems` объектов данных из входного файла, соответствующего потоку `inf`. Считанные байты будут помещены в массив `buffer`. Каждый считанный объект представляется последовательностью байтов длины `size`. Возвращаемое значение дает число успешно считанных объектов.

Процедура `fwrite` является точной противоположностью процедуры `fread`. Она записывает данные из массива `buffer` в поток `outf`. Массив `buffer` содержит `nitems` объектов, размер которых равен `size`. Возвращаемое процедурой значение дает число успешно записанных объектов.

Эти процедуры обычно используются для чтения и записи содержимого произвольных структур данных языка С. При этом параметр `size` часто содержит конструкцию `sizeof`, которая возвращает размер структуры в байтах.

Следующий пример показывает, как все это работает. В нем используется шаблон структуры `dict_elem`. Экземпляр этой структуры может представлять собой часть записи простой базы данных. Используя терминологию баз данных, структура `dict_elem` представляет собой запись, или атрибут, базы данных. Мы поместили определение структуры `dict_elem` в заголовочный файл `dict.h`, который выглядит следующим образом:

```
/* dict.h – заголовочный файл для writedict и readdict */
#include <stdio.h>

/* Структура dict_elem – элемент данных */
/* Соответствует полю базы данных */

struct dict_elem{
    char d_name[15]; /* Имя элемента словаря */
    int d_start;     /* Начальное положение записи */
    int d_length;    /* Длина поля */
    int d_type;      /* Обозначает тип данных */
};

#define ERROR    (-1)
#define SUCCESS  0
```

Не вдаваясь в смысл элементов структуры, введем две процедуры `writedict` и `readdict`, которые соответственно выполняют запись и чтение массива структур `dict_elem`. Файлы, создаваемые при помощи этих двух процедур, можно рассматривать как простые словари данных для записей в базе данных.

Процедура `writedict` имеет два параметра, имя входного файла и адрес массива, структур `dict_elem`. Предполагается, что этот список заканчивается первой структурой массива, в которой элемент `d_length` равен нулю.

```
#include "dict.h"

int writedict(const char *dictname, struct dict_elem *elist)
{
    int j;
    FILE *outf;

    /* Открывает входной файл */
    if( (outf = fopen(dictname, "w")) == NULL)
        return ERROR;

    /* Вычисляет размер массива */
    for( j = 0; elist[j].d_length != 0; j++)
        ;

    /* Записывает список структур dict_elem */
    if( fwrite((void *)elist, sizeof(struct dict_elem), j, outf) < j)
    {
        fclose(outf);
        return ERROR;
    }

    fclose(outf);
    return SUCCESS;
}
```

Обратите внимание, что адрес массива `elist` приводится к указателю на тип `void` при помощи оператора `(void *)`. Другой важный момент заключается в использовании `sizeof(struct dict_elem)` для сообщения процедуре `fwrite` размера структуры `dict_elem` в байтах.

Процедура `readdict` использует процедуру `fread` для считывания списка структур из файла. Она имеет три параметра: указатель на имя файла словаря `indictname`, указатель `inlist` на массив структур `dict_elem`, в который будет загружен список структур из файла, и размер массива `maxlength`.

```
struct dict_elem *readdict(const char *indictname,
                           struct dict_elem *inlist,
                           int maxlength)
{
    int i;
    FILE *inf;

    /* Открывает входной файл */
    if( (inf = fopen(indictname, "r")) == NULL)
        return NULL;

    /* Считывает структуры dict_elem из файла */
    for( i = 0; i < maxlength - 1; i++)
        if( fread((void *)&inlist[i], sizeof(struct dict_elem),
```

```
    1, inf) < 1)
    break;

fclose (inf);

/* Обозначает конец списка */
inlist[i].d_length = 0;

/* Возвращает начало списка */
return inlist;
}
```

И снова обратите внимание на приведение типа и использование конструкции `sizeof`.

Необходимо сделать важную оговорку. Бинарные данные, записываемые в файл при помощи процедуры `fwrite`, отражают внутреннее представление данных в системной памяти. Так как это представление зависит от архитектуры компьютера и различается порядком байтов в слове и выравниванием слов, то данные, записанные на одном компьютере, могут не читаться на другом, если не предпринять специальные усилия для того, чтобы они были записаны в машинно-независимом формате. По тем же причинам почти всегда бессмысленно выводить значения адресов и указателей.

И последний момент: можно было бы получить практически тот же результат, напрямую используя вызовы `read` или `write`, например:

```
write(fd, (void *)ptr, sizeof(struct dict_elem));
```

Основное преимущество версии, основанной на стандартной библиотеке ввода/вывода, снова заключается в ее лучшей эффективности. Данные при этом будут читаться и записываться большими блоками, независимо от размера структуры `dict_elem`.

---

**Упражнение 11.6.** Представленные версии процедур `writedict` и `readdict` работают с файлами словаря, которые могут содержать только один тип записей. Измените их так, чтобы в одном файле можно было хранить информацию о нескольких типах записей. Другими словами, нужно, чтобы файл словаря мог содержать несколько независимых именованных списков структур `dict_elem`. (Совет: включите в начало файла «заголовок», содержащий информацию о числе записей и типе полей.)

---

## 11.10. Произвольный доступ к файлу: процедуры `fseek`, `rewind` и `ftell`

Стандартная библиотека ввода/вывода содержит процедуры для позиционирования `fseek`, `rewind` и `ftell`, которые позволяют программисту перемещать указатель файла, а также опрашивать его текущее положение. Они могут использоваться только для потоков, допускающих произвольный доступ (в число которых, например, не входят терминалы).

**Описание**

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int direction);

void rewind(FILE *stream);

long ftell(FILE *stream);
```

Процедура `fseek` аналогична низкоуровневой функции `lseek`, она устанавливает указатель файла, связанный с потоком `stream`, изменяя позицию следующей операции ввода или вывода. Параметр `direction` определяет начальную точку, от которой отсчитывается новое положение указателя. Если значение этого параметра равно `SEEK_SET` (обычно 0), то отсчет идет от начала файла; если оно равно `SEEK_CUR` (обычно 1), то отсчет идет от текущего положения; для значения `SEEK_END` (обычно 2) отсчет ведется от конца файла.

Процедура `rewind(stream)` равносильна оператору:

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

Другими словами, она устанавливает указатель чтения/записи на начало файла.

Процедура `ftell` сообщает текущее положение указателя в файле – число байтов от начала файла (началу файла соответствует нулевая позиция).

## 11.11. Форматированный вывод: семейство процедур `printf`

**Описание**

```
#include <stdio.h>

/* Аргументы arg1 .. произвольного типа */

int printf(const char *fmt, arg1, arg2 ... argn);

int fprintf(FILE *outf, const char *fmt, arg1, arg2 ... argn);

int sprintf(char *string, const char *fmt, arg1, arg2 ... argn);
```

Каждая из этих процедур получает строку формата вывода `fmt` и переменное число аргументов произвольного типа (обозначенных как `arg1`, `arg2` и т.д.), используемых для формирования выходной строки вывода. В выходную строку выводится информация из параметров `arg1 ... argn` согласно формату, заданному аргументом `fmt`. В случае процедуры `printf` эта строка затем копируется в `stdout`. Процедура `fprintf` направляет выходную строку в файл `outf`. Процедура `sprintf` вывода не производит, а копирует строку в символьный массив, заданный указателем `string`. Процедура `sprintf` также автоматически добавляет в конец строки нулевой символ.

Строка формата `fmt` похожа на строки, задающие формат вывода языка Fortran. Она состоит из обычных символов, которые копируются без изменений, и набора *спецификаций формата* (conversion specifications). Это подстроки, которые начинаются с символа `%` (если нужно напечатать сам символ процента, то нужно записать два таких символа: `%%`).



Для каждого из аргументов `arg1`, `arg2` и др. должна быть задана своя спецификация формата, которая указывает тип соответствующего аргумента и способ его преобразования в выходную последовательность символов ASCII.

Прежде чем обсудить общую форму этих спецификаций, рассмотрим пример, демонстрирующий использование формата процедуры `printf` в двух простых случаях. В первом из них нет других аргументов, кроме строки `fmt`. Во втором есть один параметр форматирования: целочисленная переменная `iarg`.

```
int iarg = 34;
.
.
.
printf("Hello, world!\n");
printf("Значение переменной iarg равно %d\n", iarg);
```

Так как в первом вызове нет аргументов, которые нужно было бы преобразовать, то в строке формата не заданы спецификации формата. Этот оператор просто приводит к выводу сообщения

```
Hello, world!
```

на стандартный вывод, за которым следует символ перевода строки (символ `\n` в строке интерпретируется в языке C как символ перевода строки). Второй оператор `printf` содержит еще один аргумент `iarg` и поэтому в строке формата есть спецификация `%d`. Это сообщает процедуре `printf`, что дополнительный аргумент является целым числом, которое должно быть выведено в десятичной форме (поэтому используется символ `d`). Вывод этого оператора будет выглядеть так:

```
Значение переменной iarg равно 34
```

Приведем возможные типы спецификаций (кодов) формата:

#### *Целочисленные форматы*

- `%d` Как уже было видно из примеров, это общеупотребительный код формата для значений типа `int`. Если значение является отрицательным, то будет автоматически добавлен знак минуса
- `%u` Аргумент имеет тип `unsigned int` и будет выводиться в десятичной форме
- `%o` Аргумент имеет тип `unsigned int` и будет выводиться как восьмеричное число без знака
- `%x` Аргумент имеет тип `unsigned int` и будет выводиться как шестнадцатеричное число без знака. В качестве дополнительных шестнадцатеричных цифр будут использоваться символы `a`, `b`, `c`, `d`, `e` и `f`. Если задан код `%X`, то будут использоваться символы `A`, `B`, `C`, `D`, `E` и `F`
- `%ld` Аргумент имеет тип `long` со знаком и будет выводиться в десятичной форме. Можно также использовать спецификации `%lo`, `%lu`, `%lx`, `lX`

#### *Форматы вещественных чисел*

- `%f` Аргумент имеет тип `float` или `double` и будет выводиться в стандартной десятичной форме
- `%e` Аргумент имеет тип `float` или `double` и будет выводиться в экспоненциальной форме, принятой в научных приложениях. Для обозначения

экспоненты будет использоваться символ `e`. Если задана спецификация `%E`, то будет использоваться символ `E`

- `%g` Это объединение спецификаций `%e` и `%f`. Аргумент имеет тип `float` или `double`. В зависимости от величины числа, оно будет выводиться либо в обычном формате, либо в формате экспоненциальной записи (как для спецификации `%e`). Если задана спецификация `%G`, то экспонента будет обозначаться, как при задании спецификации `%E`

### Форматирование строк и символов

- `%c` Аргумент имеет тип `char` и будет выводиться без изменений, даже если он является «непечатаемым» символом. Численное значение символа можно вывести, используя код формата для целых чисел. Это может понадобиться при невозможности отображения символа на терминале
- `%s` Соответствующий аргумент считается строкой (то есть указателем на массив символов). Содержимое строки передается дословно в выходной поток. Строка должна заканчиваться нулевым символом

Следующий пример, процедура `warnuser`, демонстрирует использование кодов `%c` и `%s`. Она использует процедуру `fprintf` для вывода предупреждения на стандартный вывод диагностики – поток `stderr`. Если `stderr` соответствует терминалу, то процедура также пытается подать три звуковых сигнала, послав символ **Ctrl+G** (символ ASCII **BEL**, который имеет шестнадцатеричное значение `0x07`). Эта процедура использует функцию `isatty`, определяющую, соответствует ли дескриптор файла терминалу, и процедуру `fileno`, возвращающую дескриптор файла, связанный с потоком. Функция `isatty` является стандартной функцией UNIX, представленной в главе 9, а процедура `fileno` является частью стандартной библиотеки ввода/вывода и описана в разделе 11.7.

```
/* Процедура warnuser – вывод сообщения и звукового сигнала */  
  
#include <stdio.h>  
  
/* Этот код на большинстве терминалов вызывает */  
/* подачу звукового сигнала */  
const char bel = 0x07;  
  
void warnuser(const char *string)  
{  
    /* Это терминал?? */  
    if(isatty(fileno(stderr)))  
        fprintf(stderr, "%c%c%c", bel, bel, bel);  
    fprintf(stderr, "Предупреждение: %s\n", string);  
}
```

### Задание ширины поля и точности

Спецификации формата могут также включать информацию о минимальной *ширине* (`width`) поля, в котором выводится аргумент, и *точности* (`precision`). В случае целочисленного аргумента под точностью понимается максимальное число выводимых цифр. Если аргумент имеет тип `float` или `double`, то точность задает

число цифр после десятичной точки. Для строчного аргумента этот параметр определяет число символов, которые будут взяты из строки.

Значения ширины поля и точности находятся в спецификации формата сразу же после знака процента и разделены точкой, например, спецификация

```
%10.5d
```

означает: вывести соответствующий аргумент типа `int` в поле шириной 10 символов; если аргумент имеет меньше пяти цифр, то дополнить его спереди нулями до пяти знаков. Спецификация

```
%.5f
```

означает: вывести соответствующий аргумент типа `float` или `double` с точностью до пяти десятичных знаков после запятой. Этот пример также показывает, что можно опускать параметр ширины поля. Аналогично можно задавать только ширину поля, поэтому спецификация

```
%10s
```

показывает: вывести соответствующую строку в поле длиной не менее 10 символов.

Во всех приведенных примерах вывод будет выравниваться по правой границе заданного поля. Для выравнивания по левой границе нужно задать знак минус сразу же за символом процента. Например, спецификация

```
%-30s
```

означает, что соответствующий строчный аргумент будет выведен в левой части поля шириной не менее 30 символов.

Может случиться, что ширина спецификации формата не может быть вычислена до запуска программы. В этом случае можно заменить спецификацию ширины поля символом (\*). Тогда процедура `printf` будет ожидать для этого кода дополнительный аргумент типа `int`, определяющий ширину поля. Поэтому выполнение оператора

```
int width, iarg;  
.  
.  
.  
printf("%*d", width, iarg);
```

приведет к тому, что целочисленная переменная `iarg` будет выведена в поле шириной `width`.

### **Комплексный пример**

Число возможных комбинаций различных форматов огромно, поэтому для экономии места в одну программу были включены сразу несколько примеров. Функция `atan` является стандартной функцией арктангенса из математической библиотеки UNIX. (Поэтому для компиляции примера нужно задать параметр `-lm` в командной строке компилятора `cc` для подключения математической библиотеки при компоновке.)

```
/* Программа cram — демонстрация процедуры printf */  
#include <stdio.h>
```

```
#include <math.h>

main()
{
    char *weekday = "Воскресенье";
    char *month = "Сентября";
    char *string = "Hello, world";

    int i = 11058;
    int day = 15, hour = 16, minute = 25;

    /* Выводим дату */
    printf("Дата %s, %d %s, %d:%.2d\n",
           weekday, day, month, hour, minute);

    /* Перевод строки */
    putchar ('\n');

    /* Демонстрация различных комбинаций ширины поля и точности */
    printf(">>%s<<\n", string);
    printf(">>%14s<<\n", string);
    printf(">>%-14s<<\n", string);
    printf(">>%14.5s<<\n", string);
    printf(">>%-14.5s<<\n", string);

    putchar ('\n');

    /* Выводим число i в разных форматах */
    printf("%d, %u, %o, %x, %X\n", i, i, i, i, i);

    /* Выводим число пи с точностью 5 знаков после запятой */
    printf("пи равно %.5f\n", 4*atan(1.0));
}
```

Программа генерирует следующий вывод:

```
Дата Воскресенье, 15 Сентября, 16:25
>>Hello, world<<
>> Hello, world<<
>>Hello, world <<
>>           Hello<<
>>Hello      <<

11058, 11058, 25462, 2b32, 2B32
пи равно 3.14159
```

### Специальные символы

Спецификации формата вывода могут быть еще более сложными и содержать дополнительные символы, одним из которых является знак #. Он должен задаваться сразу же за значением ширины поля. Для спецификаций формата беззнаковых целых чисел, включающих коды формата o, x и X, это приводит к выводу префикса 0, 0x и 0X соответственно. Поэтому такой фрагмент программы

```
int arg = 0xFF;
printf("В восьмеричной форме, %#o\n", arg);
```

приведет к выводу строки:

В восьмеричной форме, 0377

Для вывода вещественных чисел задание знака # приведет к выводу десятичной точки, даже если задано нулевое число знаков после запятой.

В спецификации может также содержаться знак плюса (+) для принудительного вывода символа + даже для положительных чисел. (Этот символ имеет смысл только для вывода целых чисел со знаком или вещественных чисел.) Знак + располагается в спецификации на особом месте, следуя сразу же после знака минус, который обозначает выравнивание влево, или после знака процента, если знак минуса отсутствует. Следующие строки кода

```
float farg = 57.88;
printf("Значение farg равно <%+10.2f>\n", farg);
```

приведут к выводу:

Значение farg равно <+57.88>

Обратите внимание на комбинацию символов минус и плюс. Можно также заменить символ + пробелом. В этом случае процедура printf выведет на месте знака плюс пробел. Это позволяет правильно выравнивать таблицы, содержащие положительные и отрицательные числа.

### ***Процедура sprintf***

Прежде всего нужно отметить еще один момент, касающийся процедуры sprintf. Дело в том, что не следует думать о процедуре sprintf как о процедуре вывода. На самом деле она представляет собой наиболее гибкую из библиотечных процедур, работающих со строками и преобразующих форматы данных. Следующий текст показывает использование этой функции:

```
/* Процедура genkey — генерация ключа базы данных */
/* Длина ключа всегда будет равна 20 символам */

#include <stdio.h>
#include <string.h>

char * genkey(char *buf, const char *suppcode, long orderno)
{
    /* Проверяет размер ключа */
    if(strlen(suppcode) != 10)
        return (NULL);

    sprintf(buf, "%s_%.9ld", suppcode, orderno);

    return (buf);
}
```

Тогда вызов процедуры genkey

```
printf("%s\n", genkey(buf, "abcdefghij", 12));
```

выведет такую строку ключа:

abcdefghij\_000000012

## 11.12. Форматированный ввод: семейство процедур `scanf`

### Описание

```
#include <stdio.h>
/* Все параметры ptr1 .. ptrn являются указателями.
 * Переменные, на которые они указывают, могут
 * иметь произвольный тип.
 */

int scanf(const char *fmt, ptr1, ptr2, ... ptrn);
int fscanf(FILE *inf, const char *fmt, ptr1, ptr2 ... ptrn);
int sscanf(const char *string, const char *fmt,
           ptr1, ptr2 ... ptrn);
```

Процедуры семейства `scanf` противоположны по смыслу процедурам семейства `printf`. Все они принимают ввод из файла (или из строки в случае процедуры `sscanf`), декодируют его в соответствии с информацией формата `fmt` и помещают полученные данные в переменные, заданные указателями от `ptr1` до `ptrn`. Указатель файла перемещается на число обработанных символов.

Процедура `scanf` всегда выполняет чтение из `stdin`; процедура `fscanf` выполняет чтение из потока `inf`; а процедура `sscanf` выделяется в этом семействе процедур тем, что декодирует строку `string` и не осуществляет ввода данных. Поскольку последняя процедура работает со строкой в памяти, то она особенно полезна, если некоторую строку ввода нужно анализировать несколько раз.

Строка формата `fmt` имеет ту же структуру, что и строка формата процедуры `printf`. Например, следующий оператор считывает очередное целое число из потока стандартного ввода:

```
int inarg;
scanf("%d", &inarg);
```

Важно, что функции `scanf` передается адрес переменной `inarg`. Это связано с тем, что в языке С можно передавать параметры только по значению, а не по ссылке. Поэтому, если нужно, чтобы процедура `scanf` изменяла переменную, которая находится в вызывающей процедуре, следует передать указатель, содержащий адрес этой переменной. Можно очень легко забыть про символ `&`, что приведет к ошибке записи в память. Новичкам также приходится бороться с искушением помещать знак `&` перед всеми указателями, такими как имена символьных массивов.

В общем случае строка формата процедуры `scanf` может содержать:

- *пробельные символы*, то есть пробелы, символы табуляции, перевода строки и страницы. Обычно они соответствуют любым пробельным символам с текущей позиции во входном потоке, до первого не пробельного символа;
- *обычные, не пробельные символы*. Они должны точно совпадать с соответствующими символами во входном потоке;
- *спецификации формата*. Как упоминалось ранее, они в основном аналогичны спецификациям, используемым в процедуре `printf`.

Следующий пример показывает использование процедуры scanf с несколькими переменными различных типов:

```
/* Демонстрационная программа для процедуры scanf */  
  
#include <stdio.h>  
main()  
{  
    int i1, i2;  
    float flt;  
    char str1[10], str2[10];  
  
    scanf("%2d %2d %f %s %s", &i1, &i2, &flt, str1, str2);  
    .  
    .  
    .  
}
```

Первые две спецификации в строке формата сообщают процедуре scanf, что она должна считать два целых числа (в десятичном формате). Так как в обоих случаях ширина поля равна двум символам, предполагается, что первое число должно находиться в двух считанных первыми символах, а второе – в двух следующих (в общем случае ширина поля обозначает максимальное число символов, которое может занимать значение). Спецификации %f соответствует переменная типа float. Спецификация %s означает, что ожидается строка, ограниченная пробельными символами. Поэтому, если подать на вход программы последовательность

```
11 12 34.07  
keith ben
```

то в результате получится следующее:

- переменная i1 будет иметь значение 11
- переменная i2 будет иметь значение 12
- переменная flt будет иметь значение 34.07
- строка str1 будет содержать значение keith
- строка str2 будет содержать значение ben

Обе строки будут заканчиваться нулевым символом. Обратите внимание, что переменные str1 и str2 должны иметь достаточно большую длину, чтобы в них поместились вводимые строки и нулевой символ в конце. Нельзя передавать процедуре scanf неинициализированный указатель.

Если задана спецификация формата %s, то предполагается, что строка должна быть ограничена пробельными символами. Для считывания строки целиком, включая пробельные символы, необходимо использовать код формата %c. Например, оператор

```
scanf ("%10c", s1);
```

считает любые 10 символов из входного потока и поместит их в массив символов s1. Так как код формата c соответствует пробельным символам, то для получения следующего не пробельного символа должна использоваться спецификация %1s, например:

```
/* Считать 2 символа, начиная с первого не пробельного */  
scanf("%1s%1c", &c1, &c2);
```

Другим способом задания формата строчных данных, не имеющим аналога в формате процедуры `printf`, является *шаблон* (scan set). Это последовательность символов, заключенных в квадратные скобки: [ и ]. Входное поле составляется из максимальной последовательности символов, которые входят в шаблон (в этом случае пробельные символы не игнорируются и не попадают в поле, если они не являются частью шаблона). Например, оператор

```
scanf("%[ab12]%s", str1, str2);
```

при задании входной строки

```
2bbaa1other
```

поместит в строку `str1` значение `2bbaa1`, а в строку `str2` – значение `other`.

Существует несколько соглашений, используемых при создании шаблона, которые должны быть знакомы пользователям программ `grep` или `ed`. Например, диапазон символов задается строкой вида `a-z`, то есть `[a-d]` равносильно `[abcd]`. Если символ тире (`-`) должен входить в шаблон, то он должен быть первым или последним символом. Аналогично, если в шаблон должна входить закрывающая квадратная скобка `]`, то она должна быть первым символом после открывающей квадратной скобки `[`. Если первым символом шаблона является знак `^`, то при этом выбираются только символы, *не* входящие в шаблон.

Для присваивания переменных типа `long int` или `double` после символа процента в спецификации формата должен находиться символ `l`. Это позволяет процедуре `scanf` определять размер параметра, с которым она работает. Следующий фрагмент программы показывает, как можно считать из входного потока переменные обоих типов:

```
long l;  
double d;  
  
scanf("%ld %lf", &l, &d);
```

Другая ситуация часто возникает, если входной поток содержит больше данных, чем необходимо. Для обработки этого случая спецификация формата может содержать символ `(*)` сразу же после символа процента, обозначающий данное поле лишним. В результате поле ввода, соответствующее спецификации, будет проигнорировано. Вызов

```
scanf("%d %s %*d %s", &ivar, string);
```

для строки ввода

```
131 cat 132 mat
```

приведет к присвоению переменной `ivar` значения `131`, пропуску следующих двух полей, а затем присвоению строке `string` значения `mat`.

И, наконец, какое значение возвращают функции семейства `scanf`? Они обычно возвращают число преобразованных и присвоенных полей. Возвращаемое значение может быть равно нулю в случае несоответствия строки формата и вводимых



данных. Если ввод прекращается до первого успешно (или неуспешно) введенного поля, то возвращается значение EOF.<sup>1</sup>

---

**Упражнение 11.7.** *Напишите программу, выводящую в шестнадцатеричной и восьмеричной форме свои аргументы, которые должны быть десятичными целыми числами.*

---

---

**Упражнение 11.8.** *Напишите программу `savematrix`, которая должна сохранять в файле матрицу целых чисел произвольного размера в удобочитаемом формате, и программу `readmatrix`, которая загружает матрицу из файла. Используйте для этого только процедуры `fprintf` и `fscanf`. Постарайтесь свести к минимуму число пробельных символов (пробелов, символов табуляции и др.) в файле. Совет: используйте для задания формата записи в файл символ переменной ширины (\*).*

---

## 11.13. Запуск программ при помощи библиотеки стандартного ввода/вывода

Стандартная библиотека ввода/вывода содержит несколько процедур для запуска одних программ из других. Основной из них является уже известная процедура `system`.

### Описание

```
#include <stdlib.h>

int system(const char *comstring);
```

Процедура `system` выполняет команду, заданную строкой `comstring`. Вначале она создает дочерний процесс, который, в свою очередь, осуществляет вызов `exec` для запуска стандартного командного интерпретатора UNIX с командной строкой `comstring`. В это время процедура `system` в первом процессе выполняет вызов `wait`, гарантируя тем самым, что выполнение продолжится только после того, как запущенная команда завершится. Возвращаемое после этого значение `retval` содержит статус выхода командного интерпретатора, по которому можно определить, было ли выполнение программы успешным или нет. В случае неудачи любого из вызовов `fork` или `exec` значение переменной `retval` будет равно `-1`.

Поскольку в качестве посредника выступает командный интерпретатор, строка `comstring` может содержать любую команду, которую можно набрать на

---

<sup>1</sup> Опытные программисты не советуют использовать функцию `fscanf` (`scanf`) для ввода данных, кроме случаев простого интерактивного ввода. Вместо функции `fscanf` (`scanf`) предлагается использовать комбинацию вызовов `fgets` (`gets`) и `sscanf`. Недостаток функции `fscanf` (`scanf`) в данном случае состоит в том, что при случайном нарушении формата строки вводимых данных эта функция может перейти к чтению следующей строки, поскольку функция `fscanf` (`scanf`) не отличает символ окончания строки от других разделителей полей. В такой ситуации сложно обработать ошибку входных данных корректно, а кроме того, ввод следующей строки тоже будет нарушен. — Прим. науч. ред.

терминале. Это позволяет программисту воспользоваться такими преимуществами командного интерпретатора, как перенаправление ввода/вывода, поиск файлов в пути и т.д. Следующий оператор использует процедуру `system` для создания подкаталога при помощи программы `mkdir`:

```
if( (retval = system("mkdir workdir")) != 0)
    fprintf (stderr, "Процедура system вернула значение
        %d\n", retval);
```

Остановимся на некоторых важных моментах. Во-первых, процедура `system` в вызывающем процессе будет игнорировать сигналы `SIGINT` и `SIGQUIT`. Это позволяет пользователю прерывать выполнение команды, не затрагивая родительский процесс. Во-вторых, команда, выполняемая процедурой `system`, будет наследовать из вызывающего процесса некоторые открытые дескрипторы файлов. В частности, стандартный ввод команды будет получен из того же источника, что и в родительском процессе. При вводе из файла могут возникнуть проблемы, если процедура `system` используется для запуска интерактивной программы, так как ввод программы также будет производиться из файла. Следующий фрагмент программы показывает одно из возможных решений этой задачи. В ней используется вызов `fcntl` для того, чтобы гарантировать, что стандартный ввод, обозначаемый дескриптором файла `0`, соответствовал бы управляющему терминалу процесса `/dev/tty`. В примере используется составленная ранее функция `fatal`.

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

.
.
.
int newfd, oldfd;
.
.
.
/* Скопируем дескриптор файла стандартного ввода */
if( (oldfd = fcntl(0, F_DUPFD, 0)) == -1)
    fatal("Ошибка вызова fcntl");

/* Открываем управляющий терминал */
if( (newfd = open("/dev/tty", O_RDONLY)) == -1)
    fatal("Ошибка вызова open");

/* Закрываем стандартный ввод */
close (0);

/* Создаем новый стандартный ввод */
if( (fcntl(newfd, F_DUPFD, 0) != 0)
    fatal("Проблема при вызове fcntl");

close(newfd);

/* Запускаем интерактивный редактор */
if(system("ed newfile") == -1)
    fatal("Ошибка вызова system");
```

```
/* Восстанавливаем прежний стандартный ввод */
close (0);
if(fcntl(oldfd, F_DUPFD, 0) != 0)
    fatal ("Проблема при вызове fcntl");
close(oldfd);
.
.
.
```

Процедура `system` имеет один серьезный недостаток. Он не позволяет программе получать доступ к выводу запускаемой программы. Для этого можно использовать две другие процедуры из стандартной библиотеки ввода/вывода: `popen` и `pclose`.

### Описание

```
#include <stdio.h>

FILE * popen(const char *comstring, const char *type);

int pclose(FILE *strm);
```

Как и процедура `system`, процедура `popen` создает дочерний процесс командного интерпретатора для запуска команды, заданной параметром `comstring`. Но, в отличие от процедуры `system`, она также создает канал между вызывающим процессом и командой, а затем возвращает структуру `FILE`, связанную с этим каналом. Если значение параметра `type` равно `w`, то программа может выполнять запись в стандартный ввод при помощи структуры `FILE`. Если же значение параметра `type` равно `r`, то программа сможет выполнять чтение из стандартного вывода программы. Таким образом, процедура `popen` представляет простой и понятный метод взаимодействия с другой программой.

Для закрытия потока, открытого при помощи процедуры `popen`, должна всегда использоваться процедура `pclose`. Она будет ожидать завершения команды, после чего вернет статус ее завершения.

Следующий пример, процедура `getlist`, использует процедуру `popen` и команду `ls` для вывода списка элементов каталога. Каждое имя файла затем помещается в двухмерный массив символов, адрес которого передается процедуре `getlist` в качестве параметра.

```
/* getlist - процедура для получения списка файлов в каталоге */

#include <stdio.h>
#include <string.h>

#define MAXLEN    255 /* Максимальная длина имени файла */
#define MAXCMD    100 /* Максимальная длина команды */
#define ERROR    (-1)
#define SUCCESS    0

int getlist(char *namepart, char dirnames[][MAXLEN+1],
            int maxnames)
{
    char cmd[MAXCMD+1], in_line[MAXLEN+2];
    int i;
    FILE *lsf;
```

```

/* Основная команда */
strcpy(cmd, "ls ");

/* Дополнительные параметры команды */
if(namepart != NULL)
    strncat(cmd, namepart, MAXCMD - strlen(cmd));

if(( lsf = popen(cmd, "r")) == NULL) /* Запускаем команду */
    return (ERROR);

for(i =0; i < maxnames; i++)
{
    if(fgets(in_line, MAXLEN+2, lsf) == NULL)
        break;

    /* Удаляем символ перевода строки */
    if(in_line[strlen(in_line)-1] == '\n')
        in_line[strlen(in_line)-1] = '\0';

    strcpy(&dirnames[i][0], in_line);
}
if(i < maxnames)
    dirnames[i][0] = '\0';

pclose (lsf);
return(SUCCESS);
}

```

Процедура `getlist` может быть вызвана следующим образом:

```
getlist("*.c", namebuf, 100);
```

при этом в переменную `namebuf` будут помещены имена всех программ `C` в текущем каталоге.

Следующий пример разрешает обычную проблему, с которой часто сталкиваются администраторы UNIX: как быстро «освободить» терминал, который был заблокирован какой-либо программой, например, неотлаженной программой, работающей с экраном. Программа `unfreeze` принимает в качестве аргументов имя терминала и список программ. Затем она запускает команду вывода списка процессов `ps` при помощи процедуры `popen` для получения списка связанных с терминалом процессов и выполняет поиск указанных программ в этом списке процессов. Далее программа `unfreeze` запрашивает разрешение пользователя на завершение работы каждого из процессов, удовлетворяющих критерию.

Программа `ps` сильно привязана к конкретной системе. Это связано с тем, что она напрямую обращается к ядру (через специальный файл, представляющий образ системной памяти) для получения системной таблицы процессов. На системе, использованной при разработке этого примера, команда `ps` имеет синтаксис

```
$ ps -t ttyname
```

где `ttyname` является именем специального файла терминала в каталоге `/dev`, например, `tty01`, `console`, `pts/8` и др. Выполнение этой команды `ps` дает следующий вывод:

```
PID TTY TIME  COMMAND
29  co  0:04  sh
39  co  0:49  vi
42  co  0:00  sh
43  co  0:01  ps
```

Первый столбец содержит идентификатор процесса. Второй – имя терминала, в данном случае co соответствует консоли. В третьем столбце выводится суммарное время выполнения процесса. В последнем, четвертом, столбце выводится имя выполняемой программы. Обратите внимание на первую строку, которая является заголовком. В программе unfreeze, текст которой приведен ниже, нам требуется ее пропустить.

```
/* Программа unfreeze – освобождение терминала */
```

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

#define    LINESZ    50
#define    SUCCESS  0
#define    ERROR    (-1)

main(int argc, char **argv)
{
    /* Инициализация этой переменной зависит от вашей системы */
    static char *pspart = "ps -t ";
    static char *fmt = "%d %*s %*s %s";

    char comline[LINESZ], inbuf[LINESZ], header[LINESZ] ;
    char name[LINESZ] ;
    FILE *f;
    int killflag =0, j;
    pid_t pid;

    if(argc <= 2)
    {
        fprintf(stderr, "Синтаксис: %s терминал программа ...\n",
            argv[0]);
        exit(1);
    }

    /* Сборка командной строки */
    strcpy(comline, pspart);
    strcat(comline, argv[1]);

    /* Запуск команды ps */
    if((f = popen(comline, "r")) == NULL)
    {
        fprintf(stderr, "%s:невозможно запустить команду ps \n",
            argv[0]);
        exit(2);
    }
}
```

```
}

/* Получаем первую строку от ps и игнорировать ее */
if( fgets(header, LINESZ, f) == NULL)
{
    fprintf(stderr, "%s:нет вывода от ps?\n", argv[0]);
    exit(3);
}

/* Ищем программу, которую нужно завершить */
while (fgets (inbuf, LINESZ, f) !=NULL)
{
    if(sscanf(inbuf, fmt, &pid, name) < 2)
        break;

    for(j = 2; j < argc; j++)
    {
        if(strcmp(name, argv[j]) == 0)
        {
            if(dokill(pid, inbuf, header) == SUCCESS)
                killflag++;
        }
    }
}

/* Это предупреждение, а не ошибка */
if(!killflag)
    fprintf(stderr, "%s: работа программы не завершена %s\n",
            argv[0], argv[1]);

exit(0);
}
```

Ниже приведена реализация процедуры `dokill`, вызываемой программой `unfreeze`. Обратите внимание на использование процедуры `scanf` для чтения первого не пробельного символа (вместо нее можно было бы использовать и функцию `yesno`, представленную в разделе 11.8).

```
/* Получаем подтверждение, затем завершаем работу программы */
int dokill(pid_t procid, const char *line, const char *hd)
{
    char c;

    printf("\nНайден процесс, выполняющий заданную программу : \n");
    printf("\t%s\t%s\n", hd, line);
    printf("Нажмите 'y' для завершения процесса %d\n", procid);
    printf("\nYes\No? > ");

    /* Ввод первого не пробельного символа */
    scanf("%1s", &c);

    if(c == 'y' || c == 'Y')
    {
        kill(procid, SIGKILL);
    }
}
```

```
    return(SUCCESS);  
}  
return(ERROR);  
}
```

---

**Упражнение 11.9.** Напишите свою версию процедуры `getcwd`, которая возвращает строку с именем текущего рабочего каталога. Назовите вашу программу `wdir`. Совет: используйте стандартную команду `pwd`.

---

**Упражнение 11.10.** Напишите программу `arrived`, которая запускает программу `who` при помощи процедуры `popen` для проверки (с 60-секундными интервалами), находится ли в системе пользователи из заданного списка. Список пользователей должен передаваться программе `arrived` в качестве аргумента командной строки. При обнаружении кого-нибудь из перечисленных в списке пользователей, программа `arrived` должна выводить сообщение. Программа должна быть достаточно эффективной, для этого используйте вызов `sleep` между выполнением проверок. Программа `who` должна быть описана в справочном руководстве системы.

---

## 11.14. Вспомогательные процедуры

Этот раздел будет посвящен краткому описанию различных дополнительных процедур стандартной библиотеки ввода/вывода. Более подробная информация содержится в справочном руководстве системы.

### 11.14.1. Процедуры `freopen` и `fdopen`

#### Описание

```
#include <stdio.h>  
  
FILE * freopen(const char *filename, const char *type,  
              FILE *oldstream);  
  
FILE * fdopen(int filedес, const char *type);
```

Процедура `freopen` закрывает поток `oldstream`, а затем открывает его для ввода из файла `filename`. Параметр `type` определяет режим доступа к новой структуре `FILE` и принимает те же значения, что и аналогичный аргумент процедуры `fopen` (строки `r`, `w` и др. ). Процедура `freopen` обычно используется для перенаправления `stdin`, `stdout` или `stderr`, например:

```
if(freopen("new.input", "r", stdin) == NULL)  
    fatal("Невозможно перенаправить stdin");
```

Процедура `fdopen` связывает новую структуру `FILE` с целочисленным дескриптором файла `filedes`, полученным при выполнении одного из системных вызовов `creat`, `open`, `pipe` или `dup2`.

В случае ошибки обе процедуры возвращают `NULL`.

## 11.14.2. Управление буфером: процедуры *setbuf* и *setvbuf*

### Описание

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

```
void setbuf(FILE *stream, char *buf1);
```

```
int setvbuf(FILE *stream, char *buf2, int type, rsize_t size);
```

Эти процедуры позволяют программисту в некоторой степени управлять буферами потоков. Они должны использоваться после открытия файла, но до первых операций чтения или записи.


Процедура *setbuf* подставляет буфер *buf1* вместо буфера, выделяемого стандартной библиотекой ввода/вывода. Размер сегмента памяти, на который указывает параметр *buf1*, должен быть равен константе *BUFSIZ*, определенной в файле *<stdio.h>*.

Процедура *setvbuf* позволяет осуществлять более тонкое управление буферизацией, чем процедура *setbuf*. Параметр *buf2* задает адрес нового буфера, а параметр *size* – его размер. Если вместо адреса буфера передается значение *NULL*, то используется буферизация по умолчанию. Параметр *type* в процедуре *setvbuf* определяет метод буферизации потока *stream*. Он позволяет настроить поток для использования с конкретным типом устройства, например, для дисковых файлов или терминальных устройств. Возможны три значения *type*, они определены в файле *<stdio.h>*:

- \_IOFBF* Поток файла буферизуется полностью. Этот режим включен по умолчанию для всех потоков ввода/вывода, не связанных с терминалом. Данные при этом будут записываться или считываться блоками размером *BUFSIZ* байтов для обеспечения максимальной эффективности
- \_IOLBF* Вывод буферизируется построчно, и буфер сбрасывается при записи символа перевода строки. Он также очищает буфер вывода при его заполнении или при поступлении запроса на ввод. Этот режим включен по умолчанию для терминалов и служит для поддержки интерактивного использования
- \_IOBNF* Ввод и вывод не буферизируются. В этом случае параметры *buf2* и *size* игнорируются. Этот режим иногда необходим, например, для записи диагностических сообщений в файл протокола

Обратите внимание, что при задании недопустимого значения любого из параметров *type* или *size* процедура *setvbuf* возвращает ненулевое значение. В случае успеха возвращается 0.





# Глава 12. Разные дополнительные системные вызовы и библиотечные процедуры

## 12.1. Введение

В последней главе будут рассмотрены несколько системных вызовов и некоторые полезные библиотечные процедуры, которые не соответствовали тематике всех предыдущих глав, такие как управление памятью, работа с временными значениями, предикаты типов символов, функции работы со строками.

## 12.2. Управление динамическим распределением памяти

Каждая из программ, рассмотренных ранее, использовала структуры данных, определенные при помощи стандартных объявлений языка C, например:

```
struct something x, y, *z, a[20];
```

Другими словами, расположение данных в наших примерах определялось во время компиляции. Однако многие вычислительные задачи удобнее решать при помощи динамического создания и уничтожения структур данных, а это значит, что расположение данных в программе определяется окончательно только во время выполнения программы. Семейство библиотечных функций ОС UNIX `malloc` (от *memory allocation* – выделение памяти) позволяет создавать объекты в области *динамической памяти* на стадии выполнения программы, которая в англоязычной литературе часто называется *heap* – «куча», «кипа» (книг). Функция `malloc` определяется следующим образом:

### Описание

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

```
void *malloc(size_t nbytes);
```

В результате этого вызова функция `malloc` обычно возвращает указатель на участок памяти размером `nbytes`. При этом программа получает указатель на массив байтов, которые она может использовать по своему усмотрению. Если памяти недостаточно и система не может выделить запрошенный вызовом `malloc` объем памяти, то вызов возвращает нулевой указатель.

Обычно вызов `malloc` используется для выделения памяти под одну или несколько структур данных, например:

```
struct item *p;  
  
p = (struct item *) malloc(sizeof(struct item));
```

В случае успеха вызов `malloc` создает новую структуру `item`, на которую ссылается указатель `p`. Заметим, что возвращаемое вызовом `malloc` значение приводится к соответствующему типу указателя. Это помогает избежать вывода предупреждений компилятором или такими программами, как `lint`. Приведение типа в данном случае целесообразно, поскольку вызов `malloc` реализован так, чтобы отводить память под объекты любого типа при условии, что запрашивается достаточный для хранения объекта ее объем. Такие задачи, как выравнивание по границе слова, решаются самим алгоритмом функции `malloc`. Обратите внимание на то, что размер структуры `item` задается при помощи конструкции `sizeof`, которая возвращает размер объекта в байтах.

Функция `free` противоположна по своему действию функции `malloc` и возвращает отведенную память, позволяя использовать ее повторно. Функции `free` передается указатель, который был получен во время вызова `malloc`:

```
struct item *ptr;  
.  
.  
.  
  
ptr = (struct item *)malloc( sizeof(struct item) );  
  
/* Выполнить какие-либо действия .. */  
  
free( (void *)ptr);
```

После подобного вызова функции `free` освобожденный участок памяти, на который указывает `ptr`, уже нельзя использовать, так как функция `malloc` может позднее снова его отдать процессу (целиком или частично). Очень важно, чтобы функции `free` передавался указатель, который был получен от функции `malloc` или от одной из функций `calloc` и `realloc`, принадлежащих тому же семейству. Если указатель не удовлетворяет этому требованию, то почти наверняка возникнут ошибки в механизме распределения динамической памяти, которые приведут к некорректной работе программы или к ее аварийному завершению. Неправильное применение функции `free` — очень часто встречающаяся и трудноуловимая ошибка.<sup>1</sup>

Еще две функции в семействе `malloc` непосредственно связаны с выделением памяти. Первой из них является функция `calloc`.

### Описание

```
#include <stdlib.h>  
  
void * calloc(size_t nelem, size_t nbytes);
```

<sup>1</sup> Для отладки программ, интенсивно использующих динамическую память, существуют специальные библиотеки, подменяющие стандартный механизм процедур семейства `malloc`. Эти библиотеки менее производительны, зато выполняют функцию «раннего предупреждения» подобных ошибок, то есть незамедлительно фиксируют нарушения правил работы с динамической памятью. — *Прим. науч. ред.*

Функция `calloc` отводит память под массив из `nitem` элементов, размер каждого из которых равен `nbytes`. Она обычно используется следующим образом:

```
/* Выделить память под массив структур */  
  
struct item *aptr;  
.  
.  
.  
aptr = (struct item *) calloc(nitem, sizeof(struct item));
```

В отличие от функции `malloc`, память, отводимая функцией `calloc`, заполняется нулями, что приводит к задержке при ее выполнении, но может быть полезно в тех случаях, когда такая инициализация необходима.

Последней функцией выделения памяти из семейства `malloc` является функция `realloc`.

### **Описание**

```
#include <stdlib.h>  
  
void * realloc(void *oldptr, size_t newsize);
```

Функция `realloc` используется для изменения размера блока памяти, на который указывает параметр `oldptr` и который был получен ранее в результате вызова `malloc`, `calloc` или `realloc`. При этом блок памяти может переместиться, и возвращаемый указатель задает новое положение его начала. После изменения размера блока сохраняется содержимое его части, соответствующей меньшему из старого и нового размеров.

### **Пример использования функции `malloc`: связанные списки**

В информатике существует множество типов динамических структур данных. Одним из классических примеров таких структур является связный список, в котором группа одинаковых объектов связывается в единый логический объект. В этом разделе составляется простой пример, использующий односвязный список для демонстрации применения функций семейства `malloc`.

```
/* list.h – заголовочный файл для примера */  
#include <stdio.h>  
#include <stdlib.h>  
  
/* Определение основной структуры */  
  
typedef struct list_member {  
    char *m_data;  
    struct list_member *m_next;  
} MEMBER;  
  
/* Определение функций */  
MEMBER *new_member(char *);  
void add_member(MEMBER **head, MEMBER *newmem);  
void free_list(MEMBER **head);  
void printlist(MEMBER *);
```

Оператор `typedef` вводит тип `MEMBER`, имеющий два поля. Первое поле, `m_data`, в экземпляре типа `MEMBER` будет указывать на произвольную строку. Второе поле, `m_next`, указывает на другой объект типа `MEMBER`. При определении поля `m_next` пришлось использовать запись `struct list_member`, а не `MEMBER *m_next` из-за требований языка C.

Каждый элемент в связанном списке структур типа `MEMBER` будет указывать на следующий в списке объект `MEMBER`, то есть если известен один элемент списка, то следующий можно найти при помощи указателя `m_next`. Поскольку на каждый объект `MEMBER` в списке ссылается только один указатель, список можно обходить только в одном направлении. Такие списки называются *односвязными* (singly linked). Если бы был определен еще один указатель `m_prev`, то список можно было бы обходить и в обратном направлении, и в этом случае он был бы *двусвязным* (doubly linked).

Адрес начала, или головы, списка обычно записывается в отдельном указателе, определенном следующим образом:

```
MEMBER *head = (MEMBER *)0;
```

Конец списка обозначается нулевым значением поля `m_next` последнего элемента списка.

На рис. 12.1 показан простой список из трех элементов. Его начало обозначено указателем `head`.

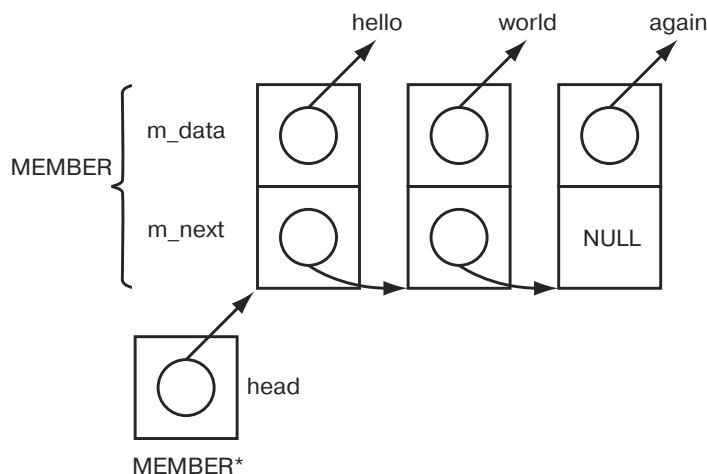


Рис. 12.1  
Связный список  
объектов `MEMBER`

Теперь представим небольшой набор процедур для работы с этими структурами. Первая функция называется `new_member`. Она отводит память под структуру `MEMBER` с помощью вызова `malloc`. Обратите внимание, что указателю `m_next` присвоено нулевое значение, в данном случае обозначенное как `(MEMBER *)0`. Это связано с тем, что функция `malloc` не обнуляет выделяемую память. Поэтому при создании структуры `MEMBER` поле `m_text` может содержать ложный, но правдоподобный адрес.

```
/* Функция new_member — выделяем память для нового элемента */
```

```
#include <string.h>
#include "list.h"
```

```
MEMBER * new_member(char *data)
{
    MEMBER *newmem;

    if((newmem=(MEMBER *)malloc(sizeof(MEMBER)))==(MEMBER *)0)
        fprintf(stderr, "new_member: недостаточно памяти\n");
    else
    {
        /* Выделяем память для копирования данных */
        newmem->m_data = (char *)malloc(strlen (data)+1);

        /* Копируем данные в структуру */
        strcpy(newmem->m_data, data);
        /* Обнуляем указатель в структуре */
        newmem->m_next = (MEMBER *)0;
    }
    return (newmem);
}
```

Следующая процедура `add_member` добавляет в список, на который указывает `head`, новый элемент `MEMBER`. Как видно из текста процедуры, элемент `MEMBER` добавляется всегда в начало списка.

```
/* Процедура add_member – добавляет новый элемент MEMBER */
#include "list.h"

void add_member(MEMBER **head, MEMBER *newmem)
{
    /* Эта простая процедура вставляет новый
     * элемент в начало списка
     */
    newmem->m_next = *head;
    *head = newmem;
}
```

Последняя процедура – `free_list`. Она принимает указатель на начало списка `*head` и освобождает память, занятую всеми структурами `MEMBER`, образующими список. Она также обнуляет указатель `*head`, гарантируя, что в указателе `*head` не содержится прежнее значение (иначе при случайном использовании `*head` могла бы возникать трудноуловимая ошибка).

```
/* Процедура free_list – освобождает занятую списком память */
#include "list.h"

void free_list(MEMBER **head)
{
    MEMBER *curr, *next;

    for(curr = *head; curr != (MEMBER *)0; curr = next)
    {
        next = curr->m_next;
        /* Освобождает память, занятую данными */
        free((void *)curr->m_data);
    }
}
```

```
/* Освобождает память, отведенную под структуру списка */
free((void *)curr);
}

/* Обнуляет указатель на начало списка */
*head = (MEMBER *)0;
}
```

Следующая простая программа использует описанные процедуры. Она создает односвязный список, изображенный на рис. 12.1, а затем удаляет его. Обратите внимание на способ обхода списка процедурой `printlist`. Используемый в ней цикл `for` типичен для программ, в которых применяются связные списки.

```
/* Программа для проверки процедур работы со списком */
#include "list.h"

char *strings[] = {"again", "world", "Hello"};
main()
{
    MEMBER *head, *newm;
    int j;

    /* Инициализация списка */
    head = (MEMBER *)0;

    /* Добавляем элементы в список */
    for(j = 0; j < 3; j++)
    {
        newm = new_member(strings[j]);
        add_member(&head, newm);
    }

    /* Выводим элементы списка */
    printlist(head);

    /* Удаляем список */
    free_list(&head);

    /* Выводим элементы списка */
    printlist(head);
}

/* Функция обхода и вывода содержимого списка */
void printlist(MEMBER *listhead)
{
    MEMBER *m;

    printf("\nСодержимое списка:\n");

    if(listhead == (MEMBER *)0)
        printf("\t(пусто)\n");
    else
        for(m = listhead; m != (MEMBER *)0; m = m->m_next)
            printf("\t%s\n", m->m_data);
}
```

Следует обратить внимание и на то, как в начале программы был инициализирован список присвоением указателю `head` значения (`MEMBER *`)`0`. Это важно, так как иначе список мог оказаться заполненным «мусором», что неизбежно привело бы к ошибочной работе или к аварийному завершению программы.

Рассматриваемая программа должна выдать такой результат:

Содержимое списка:

```
Hello  
world  
again
```

Содержимое списка:

(пусто)

### **Вызовы `brk` и `sbrk`**

Для полноты изложения необходимо упомянуть вызовы `brk` и `sbrk`. Это базовые низкоуровневые вызовы UNIX для динамического выделения памяти. Они изменяют размер сегмента данных процесса или, если быть более точным, смещают верхнюю границу сегмента данных процесса. Вызов `brk` устанавливает абсолютное значение границы сегмента, а вызов `sbrk` — ее относительное смещение. В большинстве ситуаций для выделения динамической памяти рекомендуется использовать функции семейства `malloc`, а не эти вызовы.

---

**Упражнение 12.1.** Односвязный список нашего примера может использоваться для реализации стека, в котором первым используется последний добавленный элемент. Процедура `add_member` будет соответствовать операции вставки (`push`) данных в стек. Напишите процедуру, реализующую обратную операцию извлечения (`pop`) данных из стека за счет удаления первого элемента списка.

---

---

**Упражнение 12.2.** Напишите программу, использующую функции семейства `malloc` для выделения памяти для целого числа, массива из трех переменных типа `int` и массива указателей на переменные типа `char`.

---

## **12.3. Ввод/вывод с отображением в память и работа с памятью**

При выполнении процессом ввода/вывода больших объемов данных с диска, возникают накладные расходы, связанные с двойным копированием данных сначала с диска во внутренние буферы ядра, а потом из этих буферов в структуры данных процесса. Ввод/вывод с отображением в память увеличивает скорость доступа к файлу, напрямую «отображая» дисковый файл в пространство памяти процесса. Как уже было видно на примере работы с разделяемой памятью в разделе 8.3.4, работа с данными в адресном пространстве процесса является наиболее эффективной формой доступа. (Но хотя использование ввода/вывода с отображением

в память и приводит к ускорению доступа к файлам, его интенсивное использование может уменьшить объем памяти, доступный другим средствам, использующим память процесса.)

Вскоре будут рассмотрены системные вызовы `mmap` и `munmap`. Но вначале опишем несколько простых процедур для работы с блоками памяти.

### Описание

```
#include <string.h>
void * memset(void *buf, int character, size_t size);
void * memcpy(void *buf1, const void *buf2, size_t size);
void * memmove(void *buf1, const void *buf2, size_t size);
int memcmp(const void *buf1, const void *buf2, size_t size);
void * memchr(const void *buf, int character, size_t size);
```

Для инициализации массивов данных можно использовать процедуру `memset`, записывающую значения `character` в первые `size` байтов массива памяти `buf`.

Для прямого копирования одного участка памяти в другой можно использовать любую из процедур `memcpy` или `memmove`. Обе эти функции перемещают `size` байт памяти, начиная с адреса `buf1` в участок, начинающийся с адреса `buf2`. Разница между этими двумя функциями состоит в том, что функция `memmove` гарантирует, что если области источника и адресата копирования `buf1` и `buf2` перекрываются, то при перемещении данные не будут искажены. Для этого функция `memmove` вначале копирует сегмент `buf2` во временный массив, а затем копирует данные из временного массива в сегмент `buf1` (или использует более разумный алгоритм).

Функция `memcmp` работает аналогично функции `strcmp`. Если первые `size` байтов `buf1` и `buf2` совпадают, то функция `memcmp` вернет значение 0.

Функция `memchr` проверяет первые `size` байтов `buf` и возвращает либо адрес первого вхождения символа `character`, либо значение `NULL`. Процедуры `memset`, `memcpy` и `memmove` в случае успеха возвращают значение первого параметра.

### Системные вызовы `mmap` и `mmapr`

Вернемся к отображению файлов в память. Ввод/вывод с отображением в память реализуется при помощи системного вызова `mmap`. Вызов `mmap` работает со страницами памяти, и отображение файла в памяти будет выровнено по границе страницы. Вызов `mmap` определяется следующим образом:

### Описание

```
#include <sys/mman.h>
void * mmap(void *address, size_t length, int protection,
            int flags, int filedес, off_t offset);
```

Файл должен быть заранее открыт при помощи системного вызова `open`. Полученный дескриптор файла используется в качестве параметра `fileдес` в вызове `mmap`.

Первый аргумент `address` вызова `mmap` позволяет программисту задать начало отображаемого файла в адресном пространстве процесса. Как и для вызова



shmat, рассмотренного в разделе 8.3.4, в этом случае программе нужно знать расположение данных и кода процесса в памяти. Из соображений безопасности и переносимости лучше, конечно, позволить выбрать начальный адрес системе, и это можно сделать, присвоив параметру `address` значение 0. При этом возвращаемое вызовом `mmap` значение является адресом начала отображения. В случае ошибки вызов `mmap` вернет значение `(void *)-1`.

Параметр `offset` определяет смещение в файле, с которого начинается отображение данных. Обычно нужно отображать в память весь файл, поэтому параметр `offset` часто равен 0, что соответствует началу файла. Если параметр `offset` не равен нулю, то он должен быть кратен размеру страницы памяти.

Число отображаемых в память байтов файла задается параметром `length`. Если этот параметр не кратен размеру страницы памяти, то `length` байтов будут взяты из файла, а оставшаяся часть страницы будет заполнена нулями.

Параметр `protection` определяет, можно ли выполнять чтение, запись или выполнение содержимого адресного пространства отображения. Параметр `protection` может быть комбинацией следующих значений, определенных в файле `<sys/mman.h>`:

<code>PROT_READ</code>	Разрешено выполнять чтение данных из памяти
<code>PROT_WRITE</code>	Разрешено выполнять запись данных в память
<code>PROT_EXEC</code>	Разрешено выполнение кода, содержащегося в памяти
<code>PROT_NONE</code>	Доступ к памяти запрещен

Значение параметра `protection` не должно противоречить режиму, в котором открыт файл.

Параметр `flags` влияет на доступность изменений отображенных в память данных файла для просмотра другими процессами. Наиболее полезны следующие значения этого параметра:

<code>MAP_SHARED</code>	Все изменения в области памяти будут видны в других процессах, также отображающих файл в память, изменения также записываются в файл на диске
<code>MAP_PRIVATE</code>	Изменения в области памяти не видны в других процессах и не записываются в файл

После завершения процесса отображение файла автоматически отменяется. Чтобы отменить отображение файла в память до завершения программы, можно использовать системный вызов `munmap`.

### Описание

```
#include <sys/mman.h>
int munmap(void *address, size_t length);
```

Если был задан флаг `MAP_SHARED`, то в файл вносятся все оставшиеся изменения, при флаге `MAP_PRIVATE` все изменения отбрасываются.

Следует иметь в виду, что эта функция только отменяет отображение файла в память, но не закрывает файл. Файл требуется закрыть при помощи системного вызова `close`.

Следующий пример повторяет программу `copyfile`, последний вариант которой был рассмотрен в разделе 11.4. Эта программа открывает входной и выходной файлы и копирует один из них в другой. Для простоты опущена часть процедур обработки ошибок.

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

main(int argc, char **argv)
{
    int input, output;
    size_t filesize;
    void *source, *target;
    char endchar = '\0';

    /* Проверка числа входных параметров */
    if(argc != 3)
    {
        fprintf(stderr, "Синтаксис: copyfile источник цель\n");
        exit(1);
    }

    /* Открываем входной и выходной файлы */
    if((input = open(argv[1], O_RDONLY)) == -1)
    {
        fprintf(stderr, "Ошибка при открытии файла %s\n", argv[1]);
        exit(1);
    }

    if( (output = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1)
    {
        close(input);
        fprintf(stderr, "Ошибка при открытии файла %s\n", argv[2]);
        exit(2);
    }

    /* Создаем второй файл с тем же размером, что и первый */
    filesize = lseek(input, 0, SEEK_END);
    lseek(output, filesize - 1, SEEK_SET);
    write(output, &endchar, 1);

    /* Отображаем в память входной и выходной файлы */
    if( (source = mmap(0, filesize, PROT_READ, MAP_SHARED, input,
        0)) == (void *)-1)
    {
        fprintf(stderr, "Ошибка отображения файла 1 в память\n");
        exit(1);
    }

    if( (target = mmap(0, filesize, PROT_WRITE, MAP_SHARED, output,
        0)) == (void *)-1)
```

```
{
    fprintf(stderr, "Ошибка отображения файла 1 в память \n");
    exit(2);
}

/* Копирование */
memcpy(target, source, filesize);

/* Отменяем отображение обоих файлов */
munmap(source, filesize);
munmap(target, filesize);

/* Закрываем оба файла */
close(input);
close(output);
exit(0);
}
```

Конечно, файлы были бы автоматически закрыты при завершении программы. Вызовы `munmap` включены для полноты изложения.

## 12.4. Время

В ОС UNIX существует группа процедур для установки и получения системного времени. Время в системе измеряется как число секунд, прошедших с 00:00:00 по Гринвичу с 1 января 1970 г., и размер переменной для хранения этого числа должен быть не меньше формата `long`. Поэтому для хранения системного времени в файле `<sys/types.h>` определяется тип `time_t`.

Основным вызовом этой группы является системный вызов `time`, который возвращает текущее время в стандартном формате времени системы UNIX.

### Описание

```
#include <time.h>

time_t time(time_t *now);
```

После этого вызова переменная `now` будет содержать время в системном формате. Возвращаемое вызовом `time` значение также содержит текущее время, но оно обычно не используется.

Человеку сложно представлять время в виде большого числа секунд, поэтому в ОС UNIX существует набор библиотечных процедур для перевода системного времени в более понятную форму. Наиболее общей является процедура `ctime`, которая преобразует вывод вызова `time` в строку из 26 символов, например, выполнение следующей программы

```
#include <time.h>
main ()
{
    time_t timeval;
    time(&timeval);
```

```
printf("Текущее время %s\n", ctime(&timeval));
exit(0);
}
```

дает примерно такой вывод:

Текущее время Tue Mar 18 00:17:06 1998

С функцией `ctime` связан набор процедур, использующих структуры типа `tm`. Тип структуры `tm` определен в заголовочном файле `<time.h>` и содержит следующие элементы:

```
int tm_sec;      /* Секунды */
int tm_min;      /* Минуты */
int tm_hour;     /* Часы от 0 до 24 */
int tm_mday;     /* Дни месяца от 1 до 31 */
int tm_mon;      /* Месяц от 0 до 11 */
int tm_year;     /* Год минус 1900 */
int tm_wday;     /* День недели Воскресенье = 0 */
int tm_yday;     /* День года 0-365 */
int tm_isdst;    /* Флаг летнего времени только для США */
```

Назначение всех элементов очевидно. Некоторые из процедур, использующих эту структуру, описаны ниже.

### **Описание**

```
#include <time.h>

struct tm * localtime(const time_t *timeval);

struct tm * gmtime(const time_t *timeval);

char * asctime(const struct tm *tptr);

time_t mktime(struct tm *tptr);

double difftime(time_t time1, time_t time2);
```

Процедуры `localtime` и `gmtime` конвертируют значение, полученное в результате вызова `time`, в структуру `tm` и возвращают локальное время и стандартное гринвичское время соответственно. Например, программа

```
/* Программа tm — демонстрация структуры tm */

#include <sys/types.h>
#include <time.h>
main ()
{
    time_t t;
    struct tm *tp;

    /* Получаем системное время */
    time (&t);

    /* Получаем структуру tm */
    tp = localtime (&t);
```

```
printf("Время %d:%d:%d\n", tp->tm_hour, tp->tm_min,  
        tp->tm_sec);  
exit(0);  
}
```

ВЫВОДИТ сообщение

Время 1:13:23

Процедура `asctime` преобразует структуру `tm` в строку в формате вывода процедуры `ctime`, а процедура `mktime` преобразует структуру `tm` в соответствующее ей системное время (число секунд). Процедура `difftime` возвращает разность между двумя значениями времени в секундах.

---

**Упражнение 12.3.** Напишите свою версию процедуры `asctime`.

---

**Упражнение 12.4.** Напишите функцию `weekday`, возвращающую 1 для рабочих дней и 0 – для выходных. Напишите обратную функцию `weekend`. Эти функции также должны предоставлять возможность задания выходных дней.

---

**Упражнение 12.5.** Напишите процедуры, возвращающие разность между двумя значениями, полученными при вызове `time` в днях, месяцах, годах и секундах. Не забывайте про високосные годы!

---

## 12.5. Работа со строками и символами

Библиотеки UNIX предоставляют богатые возможности для работы со строчными или символьными данными. Они достаточно полезны и поэтому также заслуживают упоминания.

### 12.5.1. Семейство процедур *string*

Некоторые из этих хорошо известных процедур уже были показаны в предыдущих главах книги, например, процедуры `strcat` и `strcpy`. Ниже следует подробный список процедур этого семейства.

#### Описание

```
#include <string.h>  
  
char * strcat(char *s1, const char *s2);  
char * strncat(char *s1, const char *s2, size_t length);  
  
int strcmp(const char *s1, const char *s2);  
int strncmp(const char *s1, const char *s2, size_t length);  
int strcasecmp(const char *s1, const char *s2);  
int strncasecmp(const char *s1, const char *s2, size_t length);  
  
char *strcpy(char *s1, const char *s2);
```

```
char *strncpy(char *s1, const char *s2, size_t length);
char *strdup(const char *s1);

size_t strlen(const char *s1);

char * strchr(const char *s1, int c);
char * strrchr(const char *s1, int c);
char * strstr(const char *s1, const char *s2);
char * strpbrk(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);

char * strtok(char *s1, const char *s2); /* Первый вызов */
char * strtok(NULL, const char *s2); /* Последующие вызовы */
```

Процедура `strcat` присоединяет строку `s2` к концу строки `s1`. Процедура `strncat` делает то же самое, но добавляет при этом не более `length` символов. Обе процедуры возвращают указатель на строку `s1`. Пример использования процедуры `strcat`:

```
strcat(fileprefix, ".dat");
```

Если переменная `fileprefix` первоначально содержала строку `file`, то после выполнения процедуры она будет содержать строку `file.dat`. Следует отметить, что процедура `strcat` изменяет строку, на которую указывает ее первый аргумент. Таким же свойством обладают и процедуры `strncat`, `strcpy`, `strncpy` и `strtok`. Так как процедура `C` не может определить размер передаваемого ей массива, программист должен убедиться, что размер первого аргумента этих процедур достаточно велик, чтобы вместить результат выполнения соответствующей операции.

Процедура `strcmp` сравнивает две строки `s1` и `s2`. Если возвращаемое значение положительно, то это означает, что строка `s1` лексикографически «больше», чем строка `s2`, в соответствии с порядком расположения символов в наборе символов ASCII. Если оно отрицательно, то это означает, что строка `s1` «меньше», чем строка `s2`. Если же возвращаемое значение равно нулю, то строки совпадают. Процедура `strncmp` аналогична процедуре `strcmp`, но сравнивает только первые `length` символов. Процедуры `strcasestr` и `strcasenchr` выполняют те же проверки, но игнорируют регистр символов. Пример использования процедуры `strcmp`:

```
if(strcmp(token, "print") == 0)
{
    /* Обработать ключевое слово print */
}
```

Процедура `strcpy` подобна процедуре `strcat`. Она копирует содержимое строки `s2` в строку `s1`. Процедура `strncpy` копирует в точности `length` символов, отбрасывая ненужные символы (что означает, что строка `s1` может не заканчиваться нулевым символом) или записывая нулевые символы вместо недостающих символов строки `s2`. Процедура `strdup` возвращает указатель на копию строки `s1`. Возвращаемый процедурой `strdup` указатель может быть передан функции `free`, так как память выделяется при помощи функции `malloc`.

Процедура `strlen` просто возвращает длину строки `s1`. Другими словами, она возвращает число символов в строке `s1` до нулевого символа, обозначающего ее конец.

Процедура `strchr` возвращает указатель на первое вхождение символа `c` (который передается в параметре типа `int`) в строке `s1` или `NULL`, если символ в строке не обнаружен. Процедура `strstr` возвращает адрес первого вхождения подстроки `s2` в строке `s1` (или `NULL`, если подстрока не найдена). Процедура `strrchr` точно так же ищет последнее вхождение символа `c`. В главе 4 процедура `strchr` была использована для удаления пути из полного маршрутного имени файла:

```
/* Выделяем имя файла из полного маршрутного имени */  
filename = strchr(pathname, '/');
```

Процедура `strbrk` возвращает указатель на первое вхождение в строке `s1` любого символа из строки `s2` или нулевой указатель, если таких вхождений нет.

Процедура `strspn` возвращает длину префикса строки `s1`, который состоит только из символов, содержащихся в строке `s2`. Процедура `strcspn` возвращает длину префикса строки `s1`, который не содержит ни одного символа из строки `s2`.

И, наконец, процедура `strtok` позволяет программе разбить строку `s1` на лексические единицы (лексемы). В этом случае строка `s2` содержит символы, которые могут разделять лексемы (например, пробелы, символы табуляции и перевода строки). Во время первого вызова, для которого первый аргумент равен `s1`, указатель на строку `s1` запоминается, и возвращается указатель на первую лексему. Последующие вызовы, для которых первый аргумент задается равным `NULL`, возвращают следующие лексемы из строки `s1`. Когда лексем в строке больше не останется, возвращается нулевой указатель.

### 12.5.2. Преобразование строк в числовые значения

Стандарт ANSI C определяет два набора функций для преобразования строк в числовые величины:

#### Описание

```
#include <stdlib.h>  
  
/* Преобразование строки в целое число */  
long int strtol(const char *str, char **endptr, int base);  
long int atol(const char *str);  
int atoi(const char *str);  
  
/* Преобразование строки в вещественное число */  
double strtod(const char *str, char **endptr);  
double atof(const char *str);
```

Функции `atoi`, `atol` и `atof` преобразуют строку числовой константы в число формата `int`, `long` и `double` соответственно. В настоящее время эти функции устарели и заменены функциями `strtol` и `strtod`.

Функции `strtod` и `strtol` намного более надежны. Обе функции удаляют все пробельные символы из начала строки `str` и все нераспознанные символы в конце

строки (включая нулевой символ) и записывают указатель на полученную строку в переменную `endptr`, если значение аргумента `endptr` не равно нулю. Последний параметр функции `strtol – base` может иметь любое значение между 0 и 36, при этом строка конвертируется в целое число с основанием `base`.

### 12.5.3. Проверка и преобразование символов

В ОС UNIX существуют два полезных набора макросов и функций для работы с символами, которые определены в файле `<ctype.h>`. Первый набор, называемый семейством `ctype`, предназначен для проверки одиночных символов. Это макросы-предикаты возвращают 1 (*истинно*), если условие выполняется, и 0 (*ложно*) – в противном случае. Например, макрос `isalpha` проверяет, является ли символ буквой, то есть, лежит ли он в диапазонах `a-z` или `A-Z`:

```
#include <ctype.h>
int c;
.
.
.
/* Макрос "isalpha" из набора ctype */
if( isalpha(c) )
{
    /* Обработываем букву */
}
else
    warn("Символ не является буквой");
```

Обратите внимание на то, что аргумент `c` имеет тип `int`. Ниже следует полный список макросов `ctype`:

<code>isalpha(c)</code>	Является ли <code>c</code> буквой?
<code>isupper(c)</code>	Является ли <code>c</code> прописной буквой?
<code>islower(c)</code>	Является ли <code>c</code> строчной буквой?
<code>isdigit(c)</code>	Является ли <code>c</code> цифрой (0–9)?
<code>isxdigit(c)</code>	Является ли <code>c</code> шестнадцатеричной цифрой?
<code>isalnum(c)</code>	Является ли <code>c</code> буквой или цифрой?
<code>isspace(c)</code>	Является ли <code>c</code> пробельным символом; то есть одним из символов: пробел, табуляция, возврат каретки, перевод строки, перевод страницы или вертикальная табуляция?
<code>ispunct(c)</code>	Является ли <code>c</code> знаком препинания?
<code>isprint(c)</code>	Является ли <code>c</code> печатаемым знаком? Для набора символов ASCII это означает любой символ в диапазоне от пробела (040) до тильды ( ~ или 0176)
<code>isgraph(c)</code>	Является ли <code>c</code> печатаемым знаком, но не пробелом?
<code>isctrl(c)</code>	Является ли <code>c</code> управляющим символом? В качестве управляющего символа рассматривается символ удаления ASCII и все символы со значением меньше 040
<code>isascii(c)</code>	Является ли <code>c</code> символом ASCII? Обратите внимание, что для любого целочисленного значения, кроме значения символа EOF,



определенного в файле `<stdio.h>`, которое передается другим процедурам семейства `ctype`, это условие должно выполняться. (Включение символа `EOF` позволяет использовать макрокоманды из семейства `ctype` в процедурах типа `getc`)

Другой набор утилит для работы с символами предназначен для простых преобразований символов. Например, функция `tolower` переводит прописную букву в строчную.

```
#include <ctype.h>
int newc, c;
.
.
.
/* Перевод прописной буквы в строчную */
/* Например, перевод буквы 'A' в 'a' */

newc = tolower(c);
```

Если `c` является прописной буквой, то она преобразуется в строчную, иначе возвращается исходный символ. Другие функции и макросы (которые могут быть объединены под заголовком `conv` в справочном руководстве системы) включают в себя:

<code>toupper(c)</code>	Функция, преобразующая букву <code>c</code> в прописную, если она является строчной
<code>toascii(c)</code>	Макрос, преобразующий целое число в символ ASCII за счет отбрасывания лишних битов
<code>_toupper(c)</code>	Быстрая версия <code>toupper</code> , выполненная в виде макроса и не выполняющая проверки того, является ли символ строчной буквой
<code>_tolower(c)</code>	Быстрая версия <code>tolower</code> , выполненная в виде макроса, аналогичная макросу <code>_toupper</code>

## 12.6. Дополнительные средства

В книге основное внимание было уделено вызовам и процедурам, которые позволяют дать основы системного программирования для ОС UNIX, и читатель к этому времени уже должен располагать средствами для решения большого числа задач. Конечно же, ОС UNIX является системой с богатыми возможностями, поэтому существует еще много процедур, обеспечивающих выполнение специальных функций. Этот последний раздел просто привлекает внимание к некоторым из них. Попробуйте найти недостающую информацию в справочном руководстве системы.

### 12.6.1. Дополнение о сокетах

Сокеты являются мощным и популярным способом взаимодействия между процессами разных компьютеров (в главе 10 они достаточно подробно были описаны). При необходимости получить больше информации по этой теме следует обратиться к специальной литературе. В качестве первого шага можно изучить

следующие процедуры, которые позволяют получить информацию о сетевом окружении:<sup>1</sup>

gethostent	getservbyname
gethostbyaddr	getservbyport
gethostbyname	getservent

### 12.6.2. Потоки управления

*Потоки управления*, или *нити*, (threads) являются облегченной версией процессов – поддерживающие их системы позволяют выполнять одновременно несколько таких потоков в рамках одного процесса, при этом все потоки могут работать с данными процесса. Их применение может быть очень ценным для повышения производительности и интерактивности определенных классов задач, но техника программирования многопоточковых программ является довольно сложной. Многие версии UNIX поддерживали различные подходы к организации потоков выполнения, но теперь выработана стандартная модель (*POSIX threads*), включенная в стандарт POSIX и пятую версию спецификации XSI. Интерфейс функций работы с потоками управления может быть описан в справочном руководстве системы под заголовком pthread\_.

### 12.6.3. Расширения режима реального времени

В последнее время в стандарт POSIX были введены определенные расширения для режима реального времени. Как и потоки управления, это специализированная и сложная область, и часто ядро UNIX обладает достаточными возможностями для решения большинства задач реального времени. Специфические требования для реализации работы в режиме реального времени включают в себя:

- организацию очереди сигналов и дополнительные средства для работы с сигналами (см. sigwaitinfo, sigtimedwait, sigqueue);
- управление приоритетом и политикой планирования процессов и потоков (см. процедуры, начинающиеся с sched\_);
- дополнительные средства для работы с таймерами, асинхронным и синхронным вводом/выводом;
- альтернативы рассмотренным процедурам передачи сообщений, интерфейсам семафоров и разделяемой памяти (попробуйте найти процедуры, начинающиеся с mq\_, sem\_ и shm\_).

### 12.6.4. Получение параметров локальной системы

В книге были рассмотрены некоторые процедуры, сообщающие системные ограничения (например, pathconf). Есть также и другие процедуры, служащие для той же цели:

sysconf	Обеспечивает доступ к конфигурационным параметрам, находящимся в файлах <limits.h> и <unistd.h>
---------	---

<sup>1</sup> Полезно также знать о процедуре setsockopt, управляющей параметрами соединения. – Прим. науч. ред.

uname	Возвращает указатель на структуру <code>utsname</code> , содержащую название операционной системы, имя узла, которое может использоваться системой в сети для установления связи, а также номер версии системы UNIX
getpwent	Это семейство процедур обеспечивает доступ к данным из файла паролей <code>/etc/passwd</code> . Все следующие вызовы возвращают указатель на структуру <code>passwd</code> , определенную в файле <code>&lt;pwd.h&gt;</code> : <pre>getpwnam(const char *username); getpwuid(uid_t uid); getpwent(void);</pre>
getgrent	Это семейство процедур связано с доступом к файлу описания групп <code>/etc/group</code>
getrlimit	Обеспечивает доступ к предельным значениям системных ресурсов, таких как память или доступное дисковое пространство
getlogin	
cuserid	Получает имя пользователя для текущего процесса

### 12.6.5. Интернационализация

Многие версии ОС UNIX могут поддерживать различные национальные среды – позволяют учитывать языковые среды, менять порядок сортировки строк, задавать символы денежных единиц, форматы чисел и т. д. Попробуйте найти процедуры `setlocale` или `catopen`. Справочное руководство системы может содержать дополнительные сведения по этой теме под заголовком `environ`.

### 12.6.6. Математические функции


ОС UNIX предоставляет обширную библиотеку математических функций для программирования научных и технических приложений. Для применения некоторых из этих функций необходимо подключать заголовочный файл `<math.h>`, который включает определения функций, некоторых важных констант (таких как  $e$  или  $\pi$ ) и структур, относящихся к обработке ошибок. Для использования большинства функций, которых коснемся ниже, потребуется также подключить математическую библиотеку заданием специальной опции компоновки программы, например:

```
cc -o mathprog mathprog.c -lm
```

Перечислим математические функции, описание которых можно найти в руководствах UNIX и спецификации XSI:

abs	Возвращает модуль целого числа. Входит в стандартную библиотеку языка C, поэтому при компоновке не нужно подключать математическую библиотеку
cbrt	Находит кубический корень числа
div	Вычисляет частное и остаток целочисленной операции деления
drand48	Набор функций для генерации псевдослучайных чисел (см. также более простую функцию <code>rand</code> )

erf	Статистическая функция ошибки, Гауссов колокол (не следует путать эту функцию с обработкой ошибок в программе)
exp/log	Набор экспоненциальных и логарифмических функций
floor	Набор функций для выделения дробной части и усечения числа до целого значения (см. также <code>ceil</code> )
frexp	Процедуры для выделения мантиссы и показателя вещественного числа
gamma	Интегральная гамма-функция
hypot	Функция Евклидова расстояния. Полезна для демонстрации детям практической пользы компьютеров
sinh	Набор гиперболических функций
sqrt	Извлекает квадратный корень из числа
trig	Обозначает группу тригонометрических функций: <code>sin</code> , <code>cos</code> , <code>tan</code> , <code>asin</code> , <code>acos</code> , <code>atan</code> и <code>atan2</code> . Каждой из функций может быть посвящен отдельный раздел справочного руководства



# Приложение 1. Коды ошибок переменной `errno` и связанные с ними сообщения

## Введение

Как уже упоминалось в главе 2, ОС UNIX обеспечивает набор стандартных кодов ошибок и сообщений, описывающих ошибки. Ошибки генерируются при неудачном завершении системных вызовов. С каждым типом ошибки системного вызова связан номер ошибки, мнемонический код (константа или `enum` описание, имеющее значение номера ошибки) и строка сообщения. Эти объекты можно использовать в программе, если включить заголовочный файл `errno.h`.

В случае возникновения ошибки системный вызов устанавливает новое значение переменной `errno`. Почти всегда системный вызов сообщает об ошибке, возвращая вызывающему процессу в качестве результата величину `-1`. После этого можно проверить соответствие значения переменной `errno` мнемоническим кодам, определенным в файле `errno.h`, например:

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

pid_t pid;
.
.
.

if((pid = fork()) == -1)
{
    if(errno == EAGAIN)
        fprintf(stderr, "Превышен предел числа процессов \n");
    else
        fprintf(stderr, "Другая ошибка\n");
}
```

Внешний массив `sys_errlist` является таблицей сообщений об ошибках, выводимых процедурой `perror`. Переменная `errno` может использоваться в качестве индекса этого массива, если нужно вручную получить системное сообщение об ошибке. Внешняя целочисленная переменная `sys_nerr` задает текущий размер таблицы `sys_errlist`. Для получения индекса в массиве следует всегда проверять, что значение переменной `errno` меньше значения `sys_nerr`, так как

новые номера ошибок могут вводиться с опережением соответствующего пополнения таблицы системных сообщений.

## Список кодов и сообщений об ошибках

Ниже приведен список сообщений об ошибках системных вызовов. Он основан на информации выпуска 4.2 стандарта X/Open System Interfaces Standard (стандарта системных интерфейсов X/Open). Каждый пункт списка озаглавлен мнемоническим сокращением имени ошибки, определенным соответствующим кодом ошибки в файле `errno.h`, и содержит системное сообщение об ошибке из таблицы `sys_errlist`, а также краткое ее описание. Обратите внимание, что текст сообщения об ошибке может меняться в зависимости от установки параметра `LC_MESSAGES` текущей *локализации* (locale).

E2BIG	<i>Слишком длинный список аргумента</i> (Argument list too long). Чаще всего означает, что вызову <code>exec</code> передается слишком длинный (согласно полному числу байтов) список аргументов
EACCES	<i>Нет доступа</i> (Permission denied). Произошла ошибка, связанная с отсутствием прав доступа. Может происходить при выполнении системных вызовов <code>open</code> , <code>link</code> , <code>creat</code> и аналогичных им. Может также генерироваться вызовом <code>exec</code> , если отсутствует доступ на выполнение
EADDRINUSE	<i>Адрес уже используется</i> (Address in use). Означает, что запрошенный программой адрес уже используется
EADDRNOTAVAIL	<i>Адрес недоступен</i> (Address not available). Эта ошибка может возникать, если программа запрашивает адрес, который уже используется процессом
EAFNOSUPPORT	<i>Семейство адресов не поддерживается</i> (Address family not supported). При использовании интерфейса вызова сокетов было задано семейство адресов, которое не поддерживается системой
EAGAIN	<i>Ресурс временно недоступен, следует повторить попытку позже</i> (Resource temporarily unavailable, try again later). Обычно означает переполнение некоторой системной таблицы. Эта ошибка может генерироваться вызовом <code>fork</code> (если слишком много процессов) и вызовами межпроцессного взаимодействия (если слишком много объектов одного из типов межпроцессного взаимодействия)
EALREADY	<i>Соединение устанавливается</i> (Connection already in progress). Означает отказ при попытке установления соединения из-за того, что этот сокет уже находится в состоянии установления соединения
EBADF	<i>Недопустимый дескриптор файла</i> (Bad file descriptor). Файловый дескриптор не соответствует открытому файлу

	или же установленный режим доступа (только для чтения или только для записи) не позволяет выполнить нужную операцию. Генерируется многими вызовами, например, <code>read</code> и <code>write</code>
EBADMSG	<i>Недопустимое сообщение</i> (Bad message). (Данная ошибка связана с архитектурой модулей STREAM.) Генерируется, если системный вызов получает сообщение потока, которое не может прочитать. Может, например, генерироваться, если вызов <code>read</code> был выполнен для чтения в обход модуля STREAM, и в результате было получено управляющее сообщение STREAM, а не сообщение с данными
EBUSY	<i>Занято устройство или ресурс</i> (Device or resource busy). Может генерироваться, например, если вызов <code>rmdir</code> пытается удалить каталог, который используется другим процессом
ECHILD	<i>Нет дочерних процессов</i> (No child processes). Был выполнен вызов <code>wait</code> или <code>waitpid</code> , но соответствующий дочерний процесс не существует
ECONNABORTED	<i>Соединение разорвано</i> (Connection aborted). Сетевое соединение было разорвано по неопределенной причине
ECONNREFUSED	<i>Отказ в установке соединения</i> (Connection refused). Очередь запросов переполнена или ни один процесс не принимает запросы на установку соединения
ECONNRESET	<i>Сброс соединения</i> (Connection reset). Соединение было закрыто другим процессом
EDEADLK	<i>Предотвращен клинч</i> (Resource deadlock would occur). В случае успешного выполнения вызова произошел бы клинч (то есть ситуация, когда два процесса ожидали бы действия друг от друга). Эта ошибка может возникать в результате вызовов <code>fcntl</code> и <code>lockf</code>
EDESTADDRREQ	<i>Требуется адрес назначения</i> (Destination request required). При выполнении операции с сокетом был опущен адрес назначения
EDOM	<i>Ошибка диапазона</i> (Domain error). Ошибка пакета математических процедур. Означает, что аргумент функции выходит за границы области определения этой функции. Может возникать во время выполнения функций семейств <code>trig</code> , <code>exp</code> , <code>gamma</code> и др.
EDQUOT	<i>Зарезервирован</i> (Reserved)
EEXIST	<i>Файл уже существует</i> (File exists). Файл уже существует, и это препятствует успешному завершению вызова. Может возникать во время выполнения вызовов <code>link</code> , <code>mkdir</code> , <code>mkfifo</code> , <code>shmget</code> и <code>open</code>

EFAULT	<i>Недопустимый адрес</i> (Bad address). Генерируется системой после ошибки защиты памяти. Обычно означает, что был задан некорректный адрес памяти. Не все системы могут отслеживать ошибки памяти и сообщать о них процессам
EFBIG	<i>Слишком большой файл</i> (File too large). При попытке увеличить размер файла было превышено максимальное значение размера файла для процесса (установленное вызовом <code>ulimit</code> ) или общесистемное максимальное значение размера файла
ENOTUNREACH	<i>Компьютер недоступен</i> (Host is unreachable). Генерируется сетью, если компьютер выключен или недоступен для маршрутизатора
EIDRM	<i>Идентификатор удален</i> (Identifier removed). Означает, что идентификатор межпроцессного взаимодействия, например, идентификатор разделяемой памяти, был удален из системы при помощи команды <code>ipcrm</code>
EILSEQ	<i>Недопустимая последовательность байтов</i> (Illegal byte sequence). Недопустимый символ (не все возможные «широки» символы являются допустимыми). Эта ошибка может возникать во время вызова <code>fprintf</code> или <code>fscanf</code>
EINPROGRESS	<i>Соединение в процессе установки</i> (Connection in progress). Означает, что вызов запроса соединения принят и будет выполнен. Данный код выставляется при вызове <code>connect</code> с флагом <code>O_NONBLOCK</code>
EINTR	<i>Прерванный вызов функции</i> (Interrupted function call). Возвращается при поступлении сигнала во время выполнения системного вызова. (Возникает только во время выполнения некоторых вызовов – обратитесь к документации системы)
EINVAL	<i>Недопустимый аргумент</i> (Invalid argument). Означает, что системному вызову был передан недопустимый параметр или список параметров. Может генерироваться вызовами <code>fcntl</code> , <code>sigaction</code> , некоторыми процедурами межпроцессного взаимодействия, а также математическими функциями
EIO	<i>Ошибка ввода/вывода</i> (I/O error). Во время ввода/вывода произошла физическая ошибка
EISCONN	<i>Сокет подключен</i> (Socket is connected). Этот сокет уже соединен
EISDIR	<i>Это каталог</i> (Is a directory). Была выполнена попытка открыть каталог для записи. Эта ошибка генерируется вызовами <code>open</code> , <code>read</code> или <code>rename</code>
ELOOP	<i>Слишком много уровней символьных ссылок</i> (Too many levels of symbolic links). Возвращается, если системе



	приходится обойти слишком много символьных ссылок при попытке найти файл в каталоге. Эта ошибка может генерироваться любым системным вызовом, принимающим в качестве параметра имя файла
EMFILE	<i>Слишком много открытых процессом файлов</i> (Too many open files in a process). Происходит в момент открытия файла и означает, что процессом открыто максимально возможное число файлов, заданное постоянной OPEN_MAX в файле <limits.h>
EMLINK	<i>Слишком много ссылок</i> (Too many links). Генерируется вызовом link, если с файлом связано максимально возможное число жестких ссылок, заданное постоянной LINK_MAX в файле <limits.h>
EMSGSIZE	<i>Слишком большое сообщение</i> (Message too large). Генерируется в сети, если посланное сообщение слишком велико, чтобы поместиться во внутреннем буфере приемника
EMULTIHOP	<i>Зарезервирован</i> (Reserved)
ENAMETOOLONG	<i>Слишком длинное имя файла</i> (Filename too long). Может означать, что имя файла длиннее NAME_MAX или полное маршрутное имя файла превышает значение PATH_MAX. Выставляется любым системным вызовом, принимающим в качестве параметра имя файла или полное маршрутное имя
ENETDOWN	<i>Сеть не работает</i> (Network is down)
ENETUNREACH	<i>Сеть недоступна</i> (Network unreachable). Путь к указанной сети недоступен
ENFILE	<i>Переполнение таблицы файлов</i> (File table overflow). Генерируется вызовами, которые возвращают дескриптор открытого файла (такими, как creat, open и pipe). Это означает, что внутренняя таблица ядра переполнена, и нельзя открыть новые дескрипторы файлов
ENOBUFS	<i>Нет места в буфере</i> (No buffer space is available). Относится к сокетам. Это сообщение об ошибке выводится, если при выполнении любого из вызовов, работающих с сокетами, система не способна нарастить буферы данных
ENODATA	<i>Сообщение отсутствует</i> (No message available). (Данная ошибка связана с архитектурой модулей STREAM.) Возвращается вызовом read, если в модуле STREAM нет сообщений
ENODEV	<i>Устройство не существует</i> (No such device). Была сделана попытка выполнить недопустимый системный вызов для устройства (например, чтение для устройства, открытого только для записи)

ENOENT	<i>Файл или каталог не существует</i> (No such file or directory). Эта ошибка происходит, если файл, заданный полным маршрутным именем (например, при выполнении вызова <code>open</code> ), не существует или не существует один из каталогов в пути
ENOEXEC	<i>Ошибка формата Exec</i> (Exec format error). Формат запускаемой программы не является допустимым форматом исполняемой программы. Эта ошибка возникает во время вызова <code>exec</code>
ENOLCK	<i>Нет свободных блокировок</i> (No locks available). Больше нет свободных блокировок, которые можно было бы установить при помощи вызова <code>fcntl</code>
ENOLINK	<i>Зарезервирован</i> (Reserved)
ENOMEM	<i>Нет места в памяти</i> (Not enough space). Ошибка нехватки памяти происходит, если процессу требуется больше памяти, чем может обеспечить система. Может генерироваться вызовами <code>exec</code> , <code>fork</code> и процедурами <code>brk</code> и <code>sbrk</code> , которые используются библиотекой управления динамической памятью
ENOMSG	<i>Нет сообщений нужного типа</i> (No message of the desired type). Возвращается, если вызов <code>msgrcv</code> не может найти сообщение нужного типа в очереди сообщений
ENOPROTOOPT	<i>Протокол недоступен</i> (Protocol not available). Запрошенный протокол не поддерживается системным вызовом <code>socket</code>
ENOSPC	<i>Исчерпано свободное место на устройстве</i> (No space left on device). Устройство заполнено, и увеличение размера файла или создание элемента каталога невозможно. Может генерироваться вызовами <code>write</code> , <code>creat</code> , <code>open</code> , <code>mknod</code> или <code>link</code> .
ENOSR	<i>Нет ресурсов потоков</i> (No streams resources). (Данная ошибка связана с архитектурой модулей STREAM.) Временное состояние, о котором сообщается, если ресурсы памяти модуля STREAM не позволяют в данный момент передать сообщение
ENOSTR	<i>Это не STREAM</i> (not a STREAM). (Данная ошибка связана с архитектурой модулей STREAM) Возвращается, если функция работы с модулем STREAM, такая как операция «push» функции <code>ioctl</code> , вызывается для устройства, которое не является устройством, представленным модулями STREAM
ENOSYS	<i>Функция не реализована</i> (Function not implemented). Означает, что запрошен системный вызов, не реализованный в данной версии системы

ENOTCONN	<i>Сокет не подключен (Socket not connected).</i> Эта ошибка генерируется, если для неподключенного сокета выполняется вызов <code>sendmsg</code> or <code>rcvmsg</code>
ENOTDIR	<i>Это не каталог (Not a directory).</i> Возникает, если путь не представляет имя каталога. Может устанавливаться вызовами <code>chdir</code> , <code>mkdir</code> , <code>link</code> и многими другими
ENOTEMPTY	<i>Каталог не пуст (Directory not empty).</i> Возвращается, например, вызовом <code>rmdir</code> , если делается попытка удалить непустой каталог
ENOTSOCK	<i>Это не сокет (Not a socket).</i> Дескриптор файла, используемый в вызове для работы с сетью, например, вызове <code>connect</code> , не является дескриптором сокета
ENOTTY	<i>Не символьное устройство (Not a character device).</i> Был выполнен вызов <code>ioctl</code> для открытого файла, который не является символьным устройством
ENXIO	<i>Устройство или адрес не существует (No such device or address).</i> Происходит, если выполняется попытка получить доступ к несуществующему устройству или адресу устройства. Эта ошибка может возникать при доступе к отключенному устройству
EOPNOTSUPP	<i>Операция не поддерживается сокетом (Operation not supported on a socket).</i> Связанное с сокетом семейство адресов не поддерживает данной функции
EOVERFLOW	<i>Значение не может уместиться в типе данных (Value too large to be stored in the data type)</i>
EPERM	<i>Запрещенная операция (Operation not permitted).</i> Означает, что процесс пытался выполнить действие, разрешенное только владельцу файла или суперпользователю ( <code>root</code> )
EPIPE	<i>Разрыв связи в канале (Broken pipe).</i> Устанавливается вызовом <code>write</code> и означает, что была выполнена попытка осуществить запись в канал, который не открыт на чтение ни одним процессом. Обычно при этом процесс, выполняющий запись в канал, прерывается при помощи сигнала <code>SIGPIPE</code> . Код ошибки <code>EPIPE</code> устанавливается, только если сигнал <code>SIGPIPE</code> перехватывается, игнорируется или блокируется
EPROTO	<i>Ошибка протокола (Protocol error).</i> Эта ошибка зависит от устройства и означает, что была получена ошибка протокола
EPROTONOSUPPORT	<i>Протокол не поддерживается (Protocol not supported).</i> Возвращается системным вызовом <code>socket</code> , если семейство адресов не поддерживается системой
EPROTOTYPE	<i>Тип сокета не поддерживается (Socket type not supported).</i> Возвращается вызовом <code>socket</code> , если заданный тип

	протокола, такой как SOCK_DGRAM, не поддерживается системой
ERANGE	<i>Результат слишком велик или слишком мал</i> (Result too large or too small). Эта ошибка возникает при вызове математических функций и означает, что возвращаемое функцией значение не может быть представлено на процессоре компьютера
EROFS	<i>Файловая система доступна только для чтения</i> (Read-only file system). Была выполнена попытка осуществить вызов write или изменить элемент каталога для файловой системы, которая была смонтирована в режиме только для чтения
ESPIPE	<i>Некорректное позиционирование</i> (Illegal seek). Для канала была предпринята попытка вызова lseek
ESRCH	<i>Процесс не существует</i> (No such process). Задан несуществующий процесс. Генерируется вызовом kill
ESTALE	<i>Зарезервирован</i> (Reserved)
ETIME	<i>Таймаут вызова ioctl для модуля STREAM</i> (ioctl timeout on a STREAM). (Данная ошибка связана с архитектурой модулей STREAM.) Показывает, что истекло время ожидания вызова ioctl для модуля ядра STREAM. Это может означать, что период ожидания нужно увеличить
ETIMEDOUT	<i>Истекло время ожидания соединения</i> (Connection timed out). Когда процесс пытается установить соединение с другой системой, то заданное время ожидания может истечь, если эта система не включена или перегружена запросами на установку соединения
ETXTBSY	<i>Файл программного кода занят</i> (Text file busy). Если ошибка генерируется вызовом exes, то это означает, что была выполнена попытка запустить на выполнение исполняемый файл, открытый для записи. Если же она генерируется вызовом, возвращающим дескриптор файла, то была сделана попытка открыть на запись файл программы, которая в данный момент выполняется
EWOLDBLOCK	<i>Операция привела бы к блокировке</i> (Operation would block). Эта ошибка возвращается, если дескриптор ввода/вывода был открыт как не блокируемый и был выполнен запрос записи или чтения, который в обычных условиях был бы заблокирован. В соответствии со спецификацией XSI код ошибки EWOLDBLOCK должен иметь то же значение, что и EAGAIN
EXDEV	<i>Попытка создания ссылки между устройствами</i> (Cross-device link). Возникает, если выполняется попытка связать при помощи вызова link файлы в разных файловых системах



## Приложение 2. История UNIX

ОС UNIX начала свою историю с того момента, как в 1969 г. Кен Томсон (Ken Thomson) и Деннис Ричи (Dennis Ritchie) с коллегами начали работу над ней на «стоящем в углу свободном компьютере PDP-7». С тех пор ОС UNIX нашла применение в сфере бизнеса и образования, а также стала основой ряда международных стандартов.

Кратко перечислим некоторые вехи истории UNIX:

- ❑ *шестая редакция системы*, 1975 (Sixth Edition, или Version 6). Первый широко известный в техническом сообществе коммерческий вариант и основа первой версии Berkeley UNIX;
- ❑ *Xenix* (1980). Версия от Microsoft – один из коммерческих вариантов UNIX с измененным названием, возникших в начале восьмидесятых годов;
- ❑ *System V* (1983–92). Одна из наиболее важных версий от создателя UNIX – корпорации AT&T. Наследница Version 7 и последовавшей за ней System III;
- ❑ *Berkeley UNIX* (версия 4.2 в 1984 г., 4.4 в 1993 г.). Разработанная в университете Berkeley, эта версия также была одной из наиболее важных версий UNIX и ввела в семейство UNIX много новых средств;
- ❑ *POSIX* (с 1988 г. и далее). Ключевой набор стандартов комитета IEEE (см. ниже), сыгравший важную роль в развитии UNIX;
- ❑ *Open Portability Guides* (XPG3 в 1990 г., XPG4.2 в 1994 г.). Практическая спецификация, объединившая целый ряд основных стандартов и рецептов использования UNIX систем. Консорциум X/Open в конце концов приобрел торговую марку UNIX.

Кроме того, существовало множество коммерческих взаимодействий, таких как передача торговой марки UNIX от компании AT&T к компании Novell, а затем консорциуму X/Open, а также слияние организаций X/Open и Open Software Foundation. Однако это достаточно упрощенный взгляд на историю UNIX. На самом деле дерево семейства UNIX является очень сложным.

### Основные стандарты

В книге упомянуты следующие стандарты:

#### **SVID**

Название стандарта SVID является сокращением от AT&T System V Interface Definition (определение интерфейса ОС System V). Первоначально был разработан весной 1985 г. для описания интерфейса версии System V операционной системы UNIX. Стандарт SVID имеет ряд версий, последней из которых является

третья редакция, вышедшая в 1989 г. Первое издание этой книги основывалось на стандарте SVID.

### **ANSI C**

Комитет ANSI определяет стандарты различных информационных технологий. Наиболее интересным для системных программистов UNIX является стандарт языка ANSI C.

### **IEEE/POSIX**

Институт электротехники и радиоэлектроники (Institute of Electrical and Electronics Engineers, сокращенно IEEE) разрабатывает в числе прочих стандарт интерфейса переносимой операционной системы (Portable Operating Systems Interface, сокращенно POSIX), который непосредственно базируется на ОС UNIX. Этот стандарт был впервые опубликован в 1988 г. и с тех пор несколько раз обновлялся. С темой данной книги наиболее тесно связаны следующие стандарты:

- стандарт IEEE 1003.1-1990, идентичный стандарту ISO POSIX-1 – ISO/IEC 9945-1, 1990, полное название которого: Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C language] (Информационные технологии – стандарт интерфейса переносимой операционной системы – часть 1. Системный интерфейс для прикладных программ [язык C]);
- стандарт IEEE 1003.2-1992, идентичный стандарту ISO POSIX-2 – ISO/IEC 9945-2, 1993, полное название которого: Information Technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities (Информационные технологии – стандарт интерфейса переносимой операционной системы – часть 2. Командный интерпретатор и утилиты).

Позже к стандарту были добавлены расширения и добавления, включая стандарты 1003.1b-1993, 1003.1c-1995, 1003.li-1995, охватывающие такие темы, как потоки управления и расширения реального времени.

### **X/Open (в настоящее время Open Group)**

Группа X/Open Group объединила вышеупомянутые стандарты с другими; эти стандарты, вместе взятые, получили название спецификации Common Application Environment (общей среды приложений, сокращенно CAE). Спецификация CAE охватывает как системные, так и сетевые интерфейсы.

С момента первой публикации книги в июле 1985 г. в стандарты был введен ряд изменений. Текст книги лучше всего соответствует Issue 4 Version 2 X/Open CAE Specification, System Interface and Headers (четвертому выпуску второй версии спецификации CAE консорциума X/Open, системные интерфейсы и заголовки), вышедшей в августе 1994 г. Эта спецификация представляет собой базовый документ X/Open. Пятое издание спецификации CAE, вышедшее в 1997 г., включает некоторые из недавно введенных в стандарт POSIX средств работы с потоками управления в режиме реального времени, а также других наработок из практики промышленного использования. Эти обновления также вошли в Version 2 of the Single UNIX Specification (вторую версию единой спецификации ОС UNIX) консорциума Open Group и в описание продукта UNIX98.



## **Библиография и дополнительные источники**

Bach, M.J. «The Design of the UNIX Operating System». Prentice-Hall, 1986.

Curry, D.A. «UNIX System Programming for SVR4». Sebastopol, CA: O'Reilly & Associates, Inc., 1996.

Dijkstra, E.W. «Co-operating Sequential Processes» in Genugs, F. (ed.) «Programming Languages». Academic Press, 1968.

Galmeister, B.O. «POSIX.4: Programming for the Real World». Sebastopol, CA: O'Reilly & Associates, Inc., 1995.

Kernighan, B.W. and Pike, R. «The UNIX Programming Environment». Prentice-Hall, 1984.

Kernighan, B.W. and Ritchie, D. «The C Programming Language». Prentice-Hall, 1978.

Northrup, C.J. «Programming with UNIX Threads». New York, NY: Wiley, 1996.

Stevens, W.R. «Advanced Programming in the UNIX Environment». Reading, MA: Addison-Wesley, 1992.

Разные авторы. «The Bell System Technical Journal (Computer Science and Systems)». AT&T Bell Laboratories. July-August, 1978.

Разные авторы. «AT&T Bell Laboratories Technical Journal (Computer Science and Systems). The UNIX System». AT&T Bell Laboratories. July-August, 1984.



# Алфавитный указатель

## А

Абонентские точки 274  
Авост 145  
Адрес  
    сетевой 272

## Б

Блокировка  
    записи 197, 199  
    чтения 199

## В

Взаимное исключение 222

## Д

Дамп памяти 145  
Дейтаграмма 285  
Дисциплина линии связи 238  
Драйвер  
    терминала 237  
    устройства 238

## З

Завершение  
    аварийное 145  
    нормальное 148  
Загрузочная область 96

## И

Идентификатор  
    egid 60, 63  
    euid 60, 62, 66

gid 60  
IPC 209  
ruid 60, 66  
группы 59  
    действующий 60, 63  
    процессов 135  
набора семафоров 222  
очереди сообщений 211  
пользователя 59  
    действующий 60, 62, 66  
    истинный 60, 66  
процесса 108  
сеанса 136  
Индексный дескриптор 96

## К

Канал 143, 169  
    именованный 143, 190  
Канонический режим терминала 242  
Каталог  
    домашний 80  
    корневой 80  
    текущий рабочий 81  
Клавиша  
    прерывания задания 143  
Клинч 206  
Ключ 208  
Код завершения 33  
Команда  
    df 102  
    ipcrm 236



- ipcs 235
- mknod 190
- script 264
- Критический участок программы 222

## Л

- Лексема 128

## М

- Магическое число 111
- Макрос
  - WCOREDUMP 149
  - WIFEXITED 123
- Маска
  - блокированных сигналов 160
  - создания файла 64
- Межпроцессное взаимодействие 197
- Модель
  - дейтаграмм 272
  - соединения 272
- Модуль
  - дисциплины линии связи терминала 263
  - эмуляции псевдотерминала 263
- Монтируемые тома 96

## Н

- Номер
  - индексного дескриптора 73, 82
  - порта 273
  - устройства
    - младший 100
    - старший 100

## О

- Общесистемные ограничения 104
- Очередь сообщений 210

## П

- Переменная
  - environ 139
  - errno 57
- Переменные окружения 137
- PATH 113
- Подкаталог 80
- Пользователь
  - root 60
  - группа 60
  - суперпользователь 60
- Последовательность
  - escape-последовательность 237
- Поток управления 338
- Право
  - выполнения файла 61
  - записи в файл 60
  - чтения файла 60
- Программа
  - addx 77
  - copyfile 330
  - docommand 117
  - etest 217
  - fsck 100
  - fsys 102
  - init 105
  - io 51
  - io2 296
  - join 187
  - list 95
  - lockit 202
  - lookout 75
  - lookup 103
  - mkfs 97, 100
  - myecho 114
  - proc\_file 118
  - rcvmessage 193
  - read\_demo 247

runls 111  
runls2 112  
runls3 115  
sendmessage 192  
setmyenv 138  
shmcopy 230  
show\_msg 220  
showmyenv 137  
sigex 153  
smallsh 126  
spawn 108  
status 122  
stest 217  
strcopy 233  
synchro 162  
tml 166  
tscript 264  
unfreeze 317

Программное окружение 137

Программный  
канал 143, 169  
именованный 143

Процедура  
filedata 74  
find\_entry 89

Процесс  
cron 136  
группа процессов 135  
демон 136  
дочерний 106  
зомби 126  
лидер 135  
родительский 106

Псевдотерминал 260

## Р

Режим  
полнодуплексный 242  
полудуплексный 256

## С

Сброс образа памяти 145  
Семафор 221  
инвариант 221  
Семейство вызовов ехес 109  
Сигнал 143  
SIGABRT 145, 150  
SIGALRM 164  
SIGINT 153  
SIGPIPE 178  
SIGTERM 164  
блокирование сигнала 159  
маска блокированных сигналов 160  
набор сигналов 150

Системный вызов

\_exit 121  
accept 277  
access 66  
alarm 164  
bind 276  
brk 327  
chdir 91  
chmod 67, 77  
chown 68  
chroot 139  
close 36, 282  
closedir 87  
connect 278  
creat 35  
execl 138  
execv 111  
execve 109, 138  
exit 120  
fcntl 48, 178, 199  
fork 106  
fstat 72, 178  
fsync 97

- getpgrp 135
- getpid 124, 133
- getrlimit 93
- getsid 136
- ioctl 249
- kill 161
- link 69, 83
- lseek 43
- mkdir 86
- mkfifo 191
- mknod 191
- mmap 328
- msgctl 218
- msgrcv 212
- msgsnd 212
- munmap 329
- nice 142
- open 31, 65, 245
- opendir 87
- pause 166
- pipe 170
- poll 181
- raise 164
- read 36, 246
- readdir 88
- readlink 72
- recv 279
- recvfrom 285
- remove 47
- rename 71
- rewinddir 88
- rmdir 87
- sbrk 327
- select 181
- semctl 222
- semget 222
- semop 225
- send 279
- sendto 285
- setpgid 135
- shmctl 230
- shmdt 230
- shmget 229
- sigprocmask 159
- socket 274
- stat 72
- symlink 71
- sync 97
- tcdrain 259
- tcflow 259
- tcflush 259
- tcgetattr 250
- tcsendbrk 259
- tcsetattr 250
- time 331
- ulimit 141
- unlink 47, 69, 83
- wait 115, 122
- waitpid 124
- write 39
- Сокет 274
- Спецификация формата 304
- Список
  - двусвязный 324
  - односвязный 324
- Средства
  - IPC 197
- Ссылка
  - жесткая 69
  - символьная 71
- Стандартная библиотека ввода/  
вывода 165
- Стандартный
  - ввод 50
  - вывод 50
  - диагностики 50, 53

Статус  
завершения 120

Структура  
FILE 289  
message 212  
sembuf 225  
statvfs 101  
termios 249  
tm 332

Суперблок 96

Счетчик  
ссылки 69

## Т

Таблица  
блочных устройств 100  
символьных устройств 100

Терминал  
управляющий 136, 145, 240

## У

Указатель  
ввода/вывода 38  
файла 38

Устройства  
блочные 99  
прямого доступа 100  
символьные 99

Утилита  
isatty 249  
ttyname 249

## Ф

Файл  
core 149  
FIFO 143  
блочного устройства 99  
владелец файла 59

дескриптор файла 30  
код доступа 61  
права доступа 34, 60  
символьного устройства 100  
специальный 79, 96  
устройства 96, 98

Файловая система 71, 95  
демонтирование 96  
монтирование 96

Флаг  
close-on-exec 120

Функция  
abort 150  
asctime 333  
atexit 121  
atof 335  
atoi 335  
atol 335  
calloc 323  
clearerr 298  
ctime 331, 332  
difftime 333  
execlp 112  
execvp 112  
fclose 290  
fdopen 319  
feof 298  
ferror 297  
fflush 292  
fgetc 293  
fgets 299  
fileno 298  
fopen 290  
fpathconf 103  
fprintf 304  
fputc 293  
fputs 300  
fread 301

free 322  
freopen 319  
fscsnf 310  
fseek 304  
fstatvfs 101  
ftell 304  
ftw 93  
fwrite 301  
getc 293  
getchar 127, 296  
getcwd 92  
getenv 138  
gets 299  
gmtime 332  
itoa 134  
localtime 332  
longjmp 158  
malloc 321  
memchr 328  
memcmp 328  
memcpy 328  
memmove 328  
memset 328  
mktime 333  
msgget 211  
pathconf 103  
pclose 315  
perror 57, 111, 115  
popen 315  
printf 56, 304  
putc 293  
putchar 296  
putenv 139  
puts 300  
realloc 323  
rewind 304  
scanf 310  
setbuf 320

setjmp 158  
setvbuf 320  
sigaction 152  
siglongjmp 158  
sigsetjmp 158  
sleep 168  
sprintf 304, 309  
sscanf 310  
statvfs 101  
strbrk 335  
strcasecmp 334  
strcasencmp 334  
strcat 334  
strchr 335  
strcmp 334  
strcomp 90  
strcpy 334  
strdup 334  
strlen 90, 335  
strncat 334  
strrchr 335  
strspn 335  
strstr 335  
strtod 335  
strtok 335  
strtol 335  
system 117, 313  
ungetc 294

## Ч

Четность 253

Число

    магическое 111

## А

Abnormal termination 145

abort 150

accept 277

access 66  
alarm 164  
asctime 333  
atof 335  
atoi 335  
atol 335

## B

bind 276  
Block device 99  
    switch 100  
Bootstrap area 96  
brk 327

## C

calloc 323  
Canonical terminal mode 242  
Character device 99  
    switch 100  
chdir 91  
Child process 106  
chmod 67, 77  
chown 68  
chroot 139  
clearerr 298  
close 36, 282  
close-on-exec 120  
closedir 87  
connect 278  
Control terminal 145  
Controlling terminal 136  
Conversion specifications 304  
core 149  
Core dump 145  
creat 35  
Critical section 222  
ctime 331, 332

## D

Daemon process 136  
Deadlock 206  
Demount 96  
Device  
    block device 99  
    character device 99  
    driver 238  
    file 98  
    number  
        major 100  
        minor 100  
difftime 333  
Directory  
    current working 81  
    home 80  
    root 80

## E

E2BIG 342  
EACCES 342  
EADDRINUSE 342  
EADDRNOTAVAIL 342  
EAFNOSUPPORT 342  
EAGAIN 342  
EALREADY 342  
EBADF 342  
EBADMSG 343  
EBUSY 343  
ECHILD 343  
ECONNABORTED 343  
ECONNREFUSED 343  
ECONNRESET 343  
EDEADLK 343  
EDESTADDRREQ 343  
EDOM 343  
EDQUOT 343

EEXIST 343  
EFAULT 344  
EFBIG 344  
EHOSTUNREACH 344  
EIDRM 344  
EILSEQ 344  
EINPROGRESS 344  
EINTR 344  
EINVAL 344  
EIO 344  
EISCONN 344  
EISDIR 344  
ELOOP 344  
EMFILE 345  
EMLINK 345  
EMSGSIZE 345  
EMULTIHOP 345  
ENAMETOOLONG 345  
ENETDOWN 345  
ENETUNREACH 345  
ENFILE 345  
ENOBUFS 345  
ENODATA 345  
ENODEV 345  
ENOENT 346  
ENOEXEC 346  
ENOLCK 346  
ENOLINK 346  
ENOMEM 346  
ENOMSG 346  
ENOPROTOOPT 346  
ENOSPC 346  
ENOSR 346  
ENOSTR 346  
ENOSYS 347  
ENOTCONN 347  
ENOTDIR 347

ENOTEMPTY 347  
ENOTSOCK 347  
ENOTTY 347  
environ 139  
Environment 137  
    variable 137  
ENXIO 347  
EOPNOTSUPP 347  
EOVERFLOW 347  
EPERM 347  
EPIPE 347  
EPROTO 347  
EPROTONOSUPPORT 348  
EPROTOTYPE 348  
ERANGE 348  
EROFS 348  
errno 57  
Escape-sequence 237  
ESPIPE 348  
ESRCH 348  
ESTALE 348  
ETIME 348  
ETIMEDOUT 348  
ETXTBSY 348  
EWOULDBLOCK 348  
exec 109  
execle 138  
execlp 112  
execv 111  
execve 138  
execvp 112  
exit 120  
Exit status 33, 120

## F

Facility identifier 209  
fclose 290

fcntl 48, 199  
fdopen 319  
feof 298  
ferror 297  
fflush 292  
fgetc 293  
fgets 299  
FIFO 143, 171, 190  
FILE 289  
File  
    access permissions 34  
    core 149  
    creation mask 64  
    descriptor 30  
    device file 96  
    mode 61  
    permissions 60  
    pointer 38  
    special file 79, 96  
    system 71, 95  
fileno 298  
First-in first-out 171  
fopen 290  
fork 106  
fpathconf 103  
fprintf 304  
fputc 293  
fputs 300  
fread 301  
free 322  
freopen 319  
fsat 72  
fscsnf 310  
fseek 304  
fstatvfs 101  
fsync 97  
ftell 304  
ftw 93

Full-duplex 242  
fwrite 301

## G

getc 293  
getchar 127, 296  
getcwd 92  
getenv 138  
getpgrp 135  
getpid 124, 133  
gets 299  
getsid 136  
gmtime 332  
Group-id 59  
    effective 60, 63  
    real 60  
Groups 60

## H

Half-duplex mode 256  
Hard link 69  
heap 321

## I

init 105  
Inode 96  
    number 73, 82  
    structure 83  
Inter-process communication 197  
Interrupt key 143  
ioctl 249  
IPC  
    идентификатор 209  
IPC facilities 197  
ipcrm 236  
ipcs 235  
isatty 249  
itoa 134



**K**

Key 208  
kill 161  
ldterm 263

**L**

Leader 135  
Line discipline 238  
Link  
    count 69  
    hard 69  
    symbolic 71  
link 69, 83  
localtime 332  
Lock 199  
Locking 197  
longjmp 158  
lseek 43

**M**

Magic number 111  
malloc 321  
memchr 328  
memcmp 328  
memcpy 328  
memmove 328  
memset 328  
Message queue 210  
    identifier 211  
mkdir 86  
mkfifo 191  
mknod 191  
mktime 333  
mmap 328  
Model  
    connection oriented 272  
    connectionless oriented 272

mount 96  
Mountable volumes 96  
msgctl 218  
msgget 211  
msgrcv 212  
msgsnd 212  
munmap 329  
Mutual exclusion 222

**N**

Network address 272  
nice 142  
Normal termination 148

**O**

open 31, 65  
opendir 87

**P**

Parent process 106  
Parity 253  
PATH 113  
pathconf 103  
pause 166  
pclose 315  
Permission 60  
    execute 61  
    read 60  
    search 85  
    write 60  
perror 57, 115  
Pipe 143, 169  
    named 190  
pipe 170  
Pointer  
    file 38  
    read-write 38

poll 181  
popen 315  
Port number 273  
printf 56, 304  
Process  
    group 135  
    group-id 135  
process-id 108  
ptem 263  
putc 293  
putchar 296  
putenv 139  
puts 300

## R

raise 164  
Raw device 100  
read 36  
readdir 88  
readlink 72  
realloc 323  
Record locking 197, 199  
recv 279  
recvfrom 285  
remove 47  
rename 71  
rewind 304  
rewinddir 88  
rmdir 87

## S

sbrk 327  
scanf 310  
script 264  
select 181  
Semaphore 221  
    invariant 221  
    set identifier 222

semctl 222  
semget 222  
semop 225  
send 279  
sendto 285  
Session-id 136  
Set-group-id 62, 68  
Set-user-id 62, 68  
setbuf 320  
setjmp 158  
setpgid 135  
setvbuf 320  
shmctl 230  
shmdt 230  
shmget 229  
SIGABRT 145, 150  
sigaction 152  
SIGALRM 145  
SIGBUS 145  
SIGCHLD 145  
SIGCONT 145  
SIGHUP 145  
SIGILL 146  
SIGINT 146, 153  
SIGKILL 146  
siglongjmp 158  
Signal 143–150  
    set 150  
SIGPIPE 146  
SIGPOLL 146  
sigprocmask 159  
SIGPROF 146  
SIGQUIT 147  
SIGSEGV 147  
sigsetjmp 158  
SIGSTOP 147  
SIGSTP 147  
SIGTERM 147

SIGTRAP 147  
SIGTTIN 147  
SIGTTOU 147  
SIGURG 147  
SIGUSR1 148  
SIGUSR2 148  
SIGVTALRM 148  
SIGXCPU 148  
SIGXFSZ 148  
sleep 168  
socket 274  
sprintf 304, 309  
sscanf 310  
Standard  
    error 50  
    input 50  
    output 50  
Standard I/O Library 165  
stat 72  
statvfs 101  
stderr 296  
stdin 296  
stdout 296  
strbrk 335  
strcasecmp 334  
strcasencmp 334  
strcat 334  
strchr 335  
strcmp 334  
strcomp 90  
strcpy 334  
strdup 334  
strlen 90, 335  
strncat 334  
strrchr 335  
strspn 335  
strstr 335

strtod 335  
strtok 335  
strtol 335  
Subdirectory 80  
Superblock 96  
symlink 71  
sync 97  
system 313  
System-wide limits 104

## T

tcdrain 259  
tcflow 259  
tcflush 259  
tcgetattr 250  
TCP 272  
tcsendbrk 259  
tcsetattr 250  
Terminal  
    control terminal 240  
    driver 237  
    pseudo terminal 260  
Termination  
    abnormal 145  
    normal 148  
termios 249  
thread 338  
time 331  
time\_t 331  
Token 128  
Transport end points 274  
ttyname 249

## U

UDP 272  
ulimit 141  
ungetc 294

unlink 47, 69, 83

User-id 59

effective 60, 62, 66

real 60, 66

## V

Virtual circuit 272

## W

wait 115, 122

waitpid 124

WCOREDUMP 149

WIFEXITED 123

write 39

## Z

Zombie 126

# ВНИМАНИЕ!

## Вы можете стать активным участником процесса книгоиздания в России!

Заполните предлагаемую анкету и укажите, книги на какие темы Вас интересуют.

При полном заполнении анкеты Вы получаете возможность в течение 6 месяцев пользоваться **10% скидкой** при приобретении компьютерной и радиотехнической литературы, имеющейся в нашем Internet-магазине, выпускаемой как «ДМК», так и другими издательствами (всего 700 наименований). Для этого достаточно получить логин и пароль, которые Вы будете использовать при входе в Internet-магазин на сайте [www.dmk.ru](http://www.dmk.ru). Узнать логин и пароль Вы можете, позвонив по тел. 369-33-60 или прислав письмо по электронной почте ([info@dmk.ru](mailto:info@dmk.ru)) после отправки анкеты по адресу: 105023, Москва, пл. Журавлева, д. 2/8, оф. 400.

1. Где Вы приобрели эту книгу? \_\_\_\_\_  
(в магазине (адрес), на рынке, у знакомых)

2. Вы приобрели эту книгу за \_\_\_\_ руб. Это очень дорого ☐ приемлемо ☐ дешево ☐

3. Оцените по 5-балльной системе:

	1	2	3	4	5
а) качество выполнения иллюстраций	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
б) качество изложения материала	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
в) актуальность рассмотренных тем	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
г) общее впечатление от книги	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

4. С какой целью Вы приобрели эту книгу?

а) книга необходима Вам по работе	<input type="checkbox"/>
б) для самостоятельного изучения предмета	<input type="checkbox"/>
в) почитать для общего развития	<input type="checkbox"/>

5. Если Вы обнаружили какие-либо ошибки или неточности в книге, пожалуйста, перечислите их ниже (с указанием номеров страниц) \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Предлагаем Вам  
принять участие в подготовке наших будущих изданий.  
Для этого ответьте на нижеследующие вопросы:**

Наиболее актуальные, по Вашему мнению, темы компьютерной и радиотехнической литературы. Если Вы затрудняетесь точно определить тему, воспользуйтесь списком примерных тем (см. на обороте) \_\_\_\_\_

\_\_\_\_\_

Основные аспекты, которые, с Вашей точки зрения, должны быть подробно рассмотрены в книге \_\_\_\_\_

\_\_\_\_\_

Примерный объем (число страниц) \_\_\_\_\_

Уровень читателя (начинающий, опытный пользователь, ...) \_\_\_\_\_

Приемлемая для Вас цена \_\_\_\_\_ руб.

**Мы приглашаем авторов к сотрудничеству.  
Пишите по адресу: [info@dmk.ru](mailto:info@dmk.ru). Факс: 369-78-74**

**СДЕЛАЙТЕ ПРИЯТНОЕ ВАШИМ ДРУЗЬЯМ –  
ПРИГЛАСИТЕ ИХ С СОБОЙ НА ПРОГУЛКУ В НАШ INTERNET-МАГАЗИН на сайте  
[www.dmk.ru](http://www.dmk.ru)**

# Темы книг по компьютерам и электронике

Укажите в анкете любую из нижеперечисленных тем  
или предложите свой вариант

---

Учебник (в помощь пользователю)	Самоучитель IBM PC Шифрование (алгоритмы) Алгоритмы сжатия (компрессии) данных Спецификации AGP, PCI и т.п. Локальные сети Материнские платы Пакеты компьютерной верстки (Adobe FrameMaker 5.5, Corel Ventura 8.0, MS Publisher) WinFaxPro и другие «маленькие помощники» Совместная работа Macintosh+Windows Системы и протоколы безопасных транзакций через Internet, ID-systems DVD E-commerce, E-marketing, E-business Novell Netware 5.0
Для программиста	C++, Pascal, Delphi, FoxPro, ... Отладка программ (дебаггеры) Программирование игр для PC Программирование для приложений распознавания речи OpenGL, DirectX – программирование приложений VBA for Office 2000 Oracle 8i, Solaris, Clarion 5.0 Программы компании «1С» ASP, HTML, XML, Java, JavaScript SQL, SQL Server 2000
Операционные системы	Windows, UNIX, Linux, EPOC, Palm OS, Mac OS, ...
Компьютерная графика и анимация	Adobe inDesign, Adobe Photoshop 5.0, 3D MAX, Fractal Design Painter 5.0, LightWave, Corel Draw 9.0, Maya 2.5, SoftImage 3D, Macromedia Flash, Bryce ...
В помощь проектировщику	Пакеты схемотехнического моделирования и проектирования печатных плат (Workbench, Accel Eda 14.0, ORCAD, ...) Различные пакеты САПР (Mechanical Desktop, Real Architect 3.8, ...) Системы архитектурного проектирования (ArchiCAD, ...) CASE-средства (ErWin, BpWin, Rational Rose)
В помощь радиолюбителю	Электронные охранные устройства Справочник по радиолампам Автомобильная электроника Акустика и акустические системы Книги для начинающих радиолюбителей Микроконтроллеры и ОЭВМ (программирование) Радиолюбительские конструкции Спутниковое телевидение
Ремонт бытовой и радиоаппаратуры	Модернизация и ремонт ПК Ремонт устройств бытовой техники Мини-АТС Охранные системы для автомобилей Справочник по бытовой радиотехнике и радиодеталям СВ радиосвязь Миноискатели Радиопередатчики СВЧ схемотехника

---



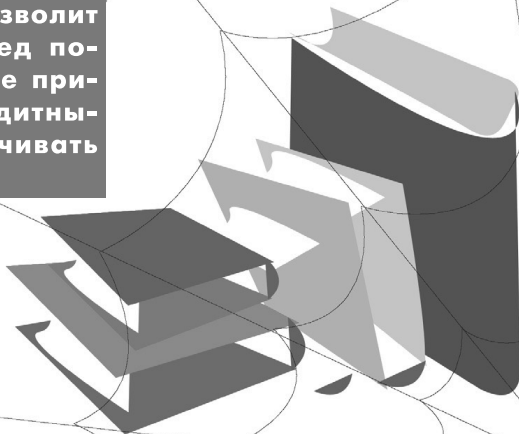
## ИЗДАТЕЛЬСТВО «ДМК» ПРЕДОСТАВЛЯЕТ ВАМ

возможность приобрести интересующие Вас книги, посвященные компьютерным технологиям и радиоэлектронике, самым быстрым и удобным способом. Для этого Вам достаточно всего лишь посетить Internet-магазин «ДМК» по адресу **www.dmk.ru**. Вашему вниманию будет представлен самый полный перечень книг по программированию, компьютерному дизайну, проектированию, ремонту радиоаппаратуры, выпущенных в нашем и других издательствах. В Internet-магазине Вы сможете приобрести любые издания, не отходя от домашнего компьютера: оформите заказ, воспользовавшись готовым бланком, и мы доставим Вам книги в самый короткий срок по почте или с курьером.

Internet-магазин на **www.dmk.ru**

- экономит Ваше время, позволяя заказать любые книги в любом количестве, не выходя из дома;
- избавляет Вас от лишних расходов: мы предлагаем компьютерную и радиотехническую литературу по ценам значительно ниже, чем в магазинах;
- дает возможность легко и быстро оформить заказ на книги — как новинки, так и издания прошлых лет, пользующиеся постоянным спросом.

Если Вы живете в Москве, то доставка с курьером позволит Вам увидеть книгу перед покупкой. При этом Вам не придется пользоваться кредитными картами или оплачивать почтовые услуги.



Кейт Хэвиленд, Дайна Грэй  
Бен Салама

# Системное программирование в UNIX

## Руководство программиста по разработке ПО

Главный редактор	<i>Захаров И. М.</i>
Научный редактор	<i>Самборский Д. В.</i>
Литературный редактор	<i>Ютлиб О. В.</i>
Технический редактор	<i>Габышев А. В.</i>
Верстка	<i>Татаринов А. Ю.</i>
Графика	<i>Бахарев А. А.</i>
Дизайн обложки	<i>Антонов А. И.</i>

Гарнитура «Петербург». Печать офсетная.  
Усл. печ. л. 28. Тираж 5000. Зак. №

Издательство «ДМК Пресс», 105023, Москва, пл. Журавлева, д 2/8.

Отпечатано в полном соответствии  
с качеством предоставленных диапозитивов  
в ППП «Типография «Наука»  
121099, Москва, Шубинский пер., 6.