М. Дансмур
Г. Дейвис

Операционная система UNIX и программирование на языке Си

РАДИО И СВЯЗЬ

М. Дансмур
Г. Дейвис

Операционная система UNIX и программирование на языке Си

Перевод с английского А. С. Богданова
Под редакцией И. Г. Шестакова

Москва
«Радио и связь»
1989

ББК 32.973
Д18
УДК 681.3.066 (420)

Редакция переводной литературы

М. Дансмур, Г. Дейвис

Д18 Операционная система UNIX и программирование на языке Си: Пер. с англ. — М.: "Радио и связь", 1989. — 192 с.: ил.

ISBN 5-256-00321-6.

Книга авторов из Великобритании является практическим руководством по программированию на языке Си в рамках оперционной системы UNIX. Обсуждается интерфейс программ пользователя с ядром операционной системы. Рассматриваются особенности файловой системы, приемы и методы программирования при работе с файлами, особенности стандартной библиотеки ввода-вывода, методы программирования взаимодействия между различными процессами, методы и средства отладки и анализа программ.

Для программистов.

Д $\frac{2404090000\text{-}147}{046\,(01)\,\text{-}89}$ 154-89 **ББК 32.973**

Производственное издание

ДАНСМУР МАРТИН, ДЕЙВИС ГЕРАЙНТ

ОПЕРАЦИОННАЯ СИСТЕМА UNIX И ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си

Заведующая редакцией О. В. Толкачева
Редактор М. Г. Коробочкина
Обложка художника Л. А. Бабаджанян
Художественный редактор А. С. Широков
Технический редактор И. Л. Ткаченко
Корректор Т. С. Власкина

ИБ № 1733

Подписано в печать с оригинала—макета 10.07.89. Формат 60x88/16. Бумага офс. № 2. Гарнитура "Универс". Печать офсетная. Усл. печ. л. 11,78. Усл. кр.-отт. 12,01. Уч.-изд. л. 12,73. Тираж 30 000 экз. Изд. № 22784. Заказ № 6788 Цена 85 к. Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО "Первая Образцовая типография" Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли.
113054 Москва, Валовая, 28

ISBN 5-256-00321-6 (рус.)
ISBN 0-333-37156-9 (англ.)

ПРЕДИСЛОВИЕ

Эта книга является практическим руководством для тех, кто постоянно программирует на языке Си под управлением операционной системы (ОС) System V. Основная ее цель — помочь всем заинтересованным пользователям как можно быстрее в полном объеме овладеть основными навыками использования ОС UNIX[1]. Надо признать, что цель эта не оригинальна, ибо книг, посвященных языку программирования Си и ОС UNIX, издано очень много, причем почти все они адресованы начинающим пользователям. Однако число программистов, уже имеющих большой опыт работы с языком Си, неуклонно растет, а решаемые ими задачи постоянно усложняются. Для таких пользователей и написана эта книга. Она будет полезна также студентам, изучающим ОС UNIX.

При изложении материала мы старались избегать отвлеченных и чисто теоретических рассуждений, отдавая предпочтение конкретным или трудным для понимания системным особенностям ОС UNIX. Читатель познакомится с набором функций, предоставляемых ОС UNIX, и научится эффективным способам их использования в программах на языке Си.

Весь приведенный в книге материал справедлив, вообще говоря, для ОС UNIX System V, тем не менее, вполне правомочно использовать его и в рамках ОС UNIX версии 7 или System III.

Мы постараемся обращать Ваше внимание на отдельные случаи и конструкции, для которых различия между этими тремя операционными системами окажутся существенными.

Мы признательны всем тем, кто оказывал нам практическую помощь при подготовке материалов, высказывал критические замечания или предоставлял необходимые вычислительные ресурсы. Мы очень благодарны Дж. Дансмур за терпение при перепечатке оригинала и исправлении грамматических ошибок. Мы благодарим за оказание неоценимой помощи при подготовке примеров и комментариев к ним Т. Гозлинга, П. Кеттла, С. Брофи и многих других.

И, наконец, мы очень признательны Х. Харту, Дж. Уилсону, Г. Керку и всем сотрудникам фирмы Logica (Великобритания) за их благожелательность и щедрое предоставление вычислительных ресурсов.

[1] Как ОС System III, так и ОС System V представляют собой версии ОС UNIX, разработанные фирмой Bell Laboratories. — *Прим. ред.*

Глава 1. ВВЕДЕНИЕ

Самым сложным шагом на пути изучения структуры и принципов функционирования современных вычислительных систем является переход от поверхностного знакомства с операционными системами разделения времени к глубокому пониманию их программного интерфейса. Эта книга поможет Вам осуществить такой переход и научиться разрабатывать не только прикладные, но и системные программы в среде ОС UNIX. Надеемся, что она окажется полезной непрерывно растущему сообществу программистов, вовлеченных в разработку прикладного программного обеспечения в среде ОС UNIX или одной из множества совместимых с ней операционных систем.

UNIX – это тщательно спроектированная, прекрасно структурированная, без неоправданных усложнений операционная система, которая предоставляет своим пользователям богатый набор самых различных функций. Кроме того, ОС UNIX все чаще принимают за основу при разработке сложных прикладных программных систем, поэтому можно считать, что она отражает современные тенденции в развитии таких систем.

На заре эпохи компьютеров каждый программный проект создавался заново, "с нуля", а это неизбежно приводило к многочисленным повторам в работе программистов. Сегодня мы понимаем, что пока программное обеспечение создается с использованием одного лишь языка ассемблера, нельзя говорить о его преемственности. Другими словами, программа, написанная в рамках одного проекта, не могла использоваться в рамках другого – ее надо было писать заново. Ситуация осложнялась еще и тем, что желая "выжать" из ЭВМ все, на что она способна, программисты делали свои программы чрезвычайно компактными, но крайне запутанными. В таких условиях реализация программного проекта становилась делом дорогим, сложным и очень хлопотным. Сегодня все обстоит совершенно иначе: основным показателем качества программного обеспечения принято считать дешевизну и мобильность[1], а не его непосредственную эффективность. Программное обеспечение, способное функционировать только на конкретной ЭВМ под управлением лишь конкретной операционной системы, сегодня не ценится; современное программное обеспечение (и системное и прикладное) должно быть разработано

[1] Под мобильностью программного обеспечения принято понимать способность последнего без каких-либо переделок функционировать на различных ЭВМ и/или под управлением различных операционных систем. – *Прим. ред.*

таким образом, чтобы перенос его на другую ЭВМ и/или в другую операционную среду требовал бы минимум затрат.

Непрерывно растущая популярность ОС UNIX объясняется тем, что, сама являясь мобильной операционной системой, ОС UNIX предоставляет своему пользователю и широкий набор мобильных прикладных программ, и средства для их создания. Как Вы, наверное, уже догадались, речь идет о языке программирования Си. Действительно, программа, написанная на этом языке и функционирующая под управлением ОС UNIX, будет функционировать на любой ЭВМ, работающей под управлением любой промышленной операционной системой, совместимой с ОС UNIX.

Необходимо отметить, что наиболее красноречивые доказательства в пользу принципа мобильности программного обеспечения приведены в широко известной книге [5], среди авторов которой, — Б. Керниган, один из основных разработчиков ОС UNIX. Хотя большинство принципов, положенных в основу ОС UNIX, могут показаться необычными пользователям, привыкшим к более традиционным операционным системам, тем не менее интерфейс пользователя, предоставляемый ОС UNIX, его краткость и выразительность вряд ли кого-нибудь удивят.

Основная доля исходных текстов ОС UNIX написана на языке Си, и лишь небольшая часть машинно-зависимых компонент ОС UNIX — с использованием языка ассемблера конкретной ЭВМ [5]. Это обстоятельство, а также то, что в основу системы управления памятью в ОС UNIX положен так называемый стековый принцип, с использованием которого построена и архитектура большинства популярных современных микропроцессоров, делает ОС UNIX самой мобильной операционной системой.

СТРУКТУРА ПРОГРАММНОГО ИНТЕРФЕЙСА ОС UNIX

С точки зрения структуры, ОС UNIX можно рассматривать как взаимосвязанную совокупность двух программных компонент: системной и прикладной[1]. Данная книга посвящена описанию интерфейса, связывающего указанные компоненты и рекомендации по его использованию при разработке пользовательских программ. Материал, приведенный в книге, рассчитан на то, что Вы хотя бы поверхностно знакомы с ОС UNIX (например, по одной из книг, перечисленных в конце этой главы). Еще лучше, если Вы одновременно получили и некоторые практические навыки использования этой системы.

Функционально программный интерфейс ОС UNIX может быть условно разделен на две подсистемы: *файловую систему и систему управления процессами*. Первая представляет собой совокупность специально организованных наборов данных, хранящихся на внешних устройствах ЭВМ, и программных средств, гарантирующих доступ к этим данным и их защи-

[1] Речь идет о так называемых ядре ОС UNIX и программном окружении ядра ОС UNIX. — *Прим. ред.*

ту; а вторая обеспечивает так называемое разделение времени, иначе говоря, возможность одновременного выполнения нескольких прикладных программ ЭВМ, имеющей один центральный процессор. Интерфейс между любой пользовательской программой и ядром ОС UNIX, названный нами выше программным интерфейсом ОС UNIX, реализуется с помощью так называемых *системных вызовов*. Когда пользовательская программа в процессе своего выполнения осуществляет системный вызов, последний отрабатывается ядром ОС UNIX и тем самым осуществляет доступ пользовательской программы к внутренним структурам ядра. Согласно терминологии, принятой в ОС UNIX, любая выполняющаяся программа называется *процессом*.

Каждый процесс в ОС UNIX можно рассматривать как *виртуальную машину*, выполняющую конкретную пользовательскую программу и предоставляющую ей целый набор услуг, в том числе услуги по осуществлению операций ввода-вывода и управлению процессами. Если выполняемый такой виртуальной машиной в данный момент код принадлежит программе пользователя, то имеет смысл говорить о *пользовательском режиме* ее работы; в противном случае говорят, что виртуальная машина переведена в *системный режим* работы. Характерной особенностью ОС UNIX является то, что процессу, функционирующему в системном режиме, соответствует реентерабельный код ядра ОС UNIX, а процессу, функционирующему в пользовательском режиме, соответствует исполняемый код пользовательской программы.

ИНТЕРФЕЙС ВВОДА-ВЫВОДА ОС UNIX

Интерфейс ввода-вывода ОС UNIX спроектирован таким образом, чтобы избавить пользователя от необходимости вникать в различия, существующие между внешними устройствами ЭВМ и методами доступа к ним. Действительно, любая операция ввода-вывода на любое внешнее устройство ЭВМ осуществляется таким же образом и теми же средствами, как если бы это была операция ввода-вывода информации в файл. Это означает, что использование привычных нам элементарных функций доступа к файлам, таких, как open, close, read, write и seek, правомочно и при осуществлении доступа к любому внешнему устройству ЭВМ. Такой подход позволяет добиться полной независимости текста прикладной программы от особенностей ввода-вывода на конкретное внешнее устройство ЭВМ, что, собственно, и дает право говорить о реальной мобильности пользовательских программ, функционирующих под управлением ОС UNIX.

Рассмотрим теперь в качестве примера следующую элементарную программу:

```
#include <stdio.h>
main(){
int c;
while((c=getchar())>=0)
      putchar(c);
}
```

Эта программа осуществляет циклический ввод из файла, называемого *стандартным вводом одного байта* информации, и вывод его в файл, называемый *стандартным выводом*. Она может быть использована для создания копии существующего текстового файла, для вывода содержимого текстового файла на терминал или же для создания нового файла. Представим теперь, что мы скомпилировали и скомпоновали эту программу, а результат поместили в файл с именем cat. Покажем, как можно использовать эту программу для осуществления вышеперечисленных операций без модификации ее исходного текста. Итак, создадим, прежде всего, новый файл, для чего введем с терминала:

```
cat > newfile
input
^d
```

Здесь мы переназначили стандартный вывод программы cat на файл с именем newfile, оставив в качестве ее стандартного ввода терминал. После этого мы ввели с терминала текст input, завершив его управляющим символом^ d[1], ввод которого (в некоторых случаях по соглашению таким символом может оказаться управляющий символ ^z) означает завершение операции ввода, эмулируя ситуацию "конец файла".

Далее создадим копию существующего файла, для чего введем с терминала:

```
cat <old> new
```

В этом случае и стандартный ввод, и стандартный вывод программы cat переназначены на файлы на диске, в результате чего программа cat будет осуществлять ввод из существующего файла с именем old и осуществлять вывод во вновь созданный файл с именем new.

И наконец, выведем на терминал с помощью программы cat содержимое существующего файла, например файла с именем cat.c[2], для чего введем с терминала

```
cat <cat.c
```

Здесь содержимое файла с именем cat.c будет выведено на стандартный вывод программы cat, в данном примере на терминал пользователя. Заметим, что стандартный ввод программы cat в этом случае переназначен на файл с именем cat.c. Аналогичным образом можно получить листинг программы cat на печатающем устройстве, для чего введем с терминала

```
cat <cat.c >/dev/lp
```

Здесь /dev/lp — это специальный файл, соответствующий печатающему устройству. Воспользуемся теперь программой cat для копирования со-

---

[1] При осуществлении ввода из файла на диске управляющий символ d автоматически генерируется ОС UNIX после ввода из файла последенего байта его содержимого.

[2] Авторы, по-видимому, считают очевидным, что для компиляции программы cat ее исходный текст должен быть помещен в файл с именем cat.c. — *Прим. ред.*

держимого всех файлов, хранящихся на двух магнитных лентах, для чего введем с терминала

```
cat </dev/mt0 >/dev/mt1
```

В результате информация, находящаяся на магнитной ленте, установленной на магнитофон с логическим номером 0, окажется скопированной на магнитную ленту, установленную на магнитофон с логическим номером 1. Так же легко создать копию содержимого магнитной ленты, установленной на магнитофон с логическим номером 0, в файле на диске, для чего достаточно ввести с терминала

```
cat </dev/mt0 >new.c
```

И в заключение создадим *канал* между программами cat и sort, для чего введем с терминала

```
cat <list | sort >list.sorted
```

Как видно, результат сортировки, выполняемой программой sort, помещен в файл с именем list.sorted. Вообще говоря, описывая подобную манипуляцию, говорят, что стандартный вывод программы cat *замкнут* на стандартный ввод программы sort[1].

УПРАВЛЕНИЕ ПРОЦЕССАМИ

Система управления процессами реализует такие элементарные функции, как порождение процесса, завершение его функционирования и обмен данными между двумя функционирующими процессами. Кроме того, она осуществляет динамическое распределение между ними оперативной памяти ЭВМ. Одна из замечательных особенностей ОС UNIX заключается в том, что среди реализуемых ее ядром ОС функций нет лишних или дублирующих друг друга. В отличие от большинства существующих интерактивных операционных систем разделения времени ядро ОС UNIX не реализует ни одной функции, специализированной для какого-либо конкретного приложения, например функции произвольного изменения приоритета пользовательского процесса, столь необходимой для решения задач реального времени. Вместо внесенных в ядро специализированных функций ОС UNIX имеет чрезвычайно мощную и одновременно гибкую систему управления процессами, дающую пользователю возможность разрабатывать такие специализированные программы на прикладном уровне.

Процедура раскрутки ОС UNIX предполагает автоматическое выполнение некоторой программы прикладного уровня, в рамках которой, вообще говоря, возможна переустановка некоторых параметров ОС

---

[1] Использовать здесь программу cat вовсе необязательно, вполне достаточно ввести с терминала

```
cat <list >list.sorted
```

UNIX[1]. По окончании процесса раскрутки, после перевода ОС UNIX в многопользовательский режим работы, для каждого терминала, с которого осуществлен вход в систему, порождается процесс, функционирующий в рамках программы shell. Все процессы, порожденные во время функционирования ОС UNIX в многопользовательском режиме, считаются равноправными с точки зрения управления.

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си

Поскольку ОС UNIX и все поставляемые в ее составе инструментальные средства написаны на языке программирования Си, нам представляется естественным использовать его и для разработки прикладных программ, особенно если при этом возникает необходимость использования каких-либо функций, реализуемых ядром ОС UNIX. Исходя из этого, мы считаем, что читатель, открывший нашу книгу, намерен активно использовать язык программирования Си и, более того, уже имеет определенный опыт работы с ним. Если же Вы собираетесь использовать для разработки своих прикладных программ какой-либо другой язык программирования, но так, чтобы при этом использовались и функции языка Си, то для реализации такого подхода Вам понадобится специально разработать целый набор так называемых интерфейсных модулей[2].

Делать подобный выбор неразумно по трем причинам: во-первых, резко упадет эффективность разработанной таким образом программы, во-вторых, разработка интерфейсных модулей потребует от Вас значительных усилий и, в-третьих, в описании языка Си Вы найдете все, что необходимо для того, чтобы сделать его идеальным средством для разработки прикладных программ. В заключение добавим, что выбор любого другого языка программирования проиллюстрирует Вам крайнюю немобильность Вашей программы.

СТРУКТУРА КНИГИ

Посвятим теперь несколько слов структуре предлагаемой Вам книги. Глава 1 — это введение. В гл. 2 дается краткий обзор основных особенностей программирования на языке Си под управлением ОС UNIX. От читателя требуется предварительное знакомство с этим языком и интерфейсом пользователя ОС UNIX. В главе достаточно подробно освещаются проблемы использования препроцессора языка Си, системных вызо-

---

[1] Авторы, по-видимому, имеют в виду файл с именем /etc/rc, который при переходе к многопользовательскому режиму передает процесс с идентификатором процесса 1 в рамках программы init интерпретатору команд shell. — *Прим. ред.*

[2] Например, если речь идет о языке программирования Фортран, то передача параметров подпрограммам осуществляется через регистры, в то время как компилятор с языка программирования Си осуществляет передачу параметров через стек. — *Прим. ред.*

вов ОС UNIX, а также рассматриваются вопросы системной поддержки пользовательских программ на этапе выполнения.

В гл. 3 Вы найдете описание программного интерфейса с файловой системой, которое будет продолжено в гл. 4 в рамках описания нижних уровней интерфейса системы ввода-вывода ОС UNIX. В гл. 5 говорится об использовании библиотек стандартных функций ввода-вывода ОС UNIX, что завершает обсуждение системы ввода-вывода. В гл. 6 содержится описание структуры процесса ОС UNIX, процедуры его порождения, функционирования и завершения. В конце ее Вы найдете также описание управления памятью в ОС UNIX и полезную информацию о некоторых внутренних структурах данных, используемых системой управления процессами. В гл. 7 завершается рассмотрение вопросов управления процессами; детально описываются средства коммуникации двух или более процессов, в том числе сигналов и каналов.

Эффективность программирования зависит, как известно, от стиля языка программирования; ОС UNIX представляет пользователю богатый набор средств для разработки мобильных и эффективных программ. Этим вопросам посвящена гл. 8, в которой описывается использование инструментальных средств ОС UNIX и, прежде всего, программы lint. Кроме того, в этой главе рассматриваются вопросы отладки подготовленных программ с использованием программы adb, а также применение профилирования программы с целью повышения ее эффективности.

И наконец, гл. 9 содержит подробное описание программ make и SCCS, служащих поддержанию дисциплины программирования при разработке сложных программных комплексов.

Каждая из перечисленных здесь глав заканчивается списком литературы, а некоторые содержат также задачи для читателя.

ЛИТЕРАТУРА

1. S. R. Bourne (1978), The UNIX Shell, Bell Sys. Tech. J., 57(6) pp 1971-1990, 1978.

2. S.R.Bourne (1982), The UNIX System, Addison Wesley.

3. S. C. Johnson & D. M. Ritchie (1978), Portability of C Programs and the UNIX System, Bell Sys. Tech. J., 57(6) pp 2021-2048, 1978.

4. B. W. Kernighan & P. J. Plauger (1976), Software Tools, Addison-Wesley.

5. B. W. Kernighan & D. M. Ritchie (1978), The C Programming Language, Prentice Hall Inc.
   [Перевод на русский язык см. в книге: Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку программирования Си: Пер. с англ. — М.: Финансы и статистика, 1985. — 279с.]

6. B. W. Kernighan (1978), UNIX for Beginners.

7. B.W. Kernighan & P.J. Plauger (1978), The Elements of Programming Style, Second Edition, Yourdon Inc.

8. B.W. Kernighan & Pike (1984), The UNIX Programming Environment, Prentice Hall.

9. G. W. R. Luderer, J. F. Maranzano & A B. A. Tague (1978), The UNIX Operating System as a Base for Applications Bell Sys. Tech. J., 57(6) pp 2201-2207, 1978.

10. D. M. Ritchie & K. Thompson (1974), The UNIX Time-Sharing System, CACM July 1974.

11. K. Thompson (1978), UNIX Implementation, Bell Sys. Tech. J., 57(6) pp 1931-1946, 1978.

Глава 2. ОС UNIX И ЯЗЫК ПРОГРАММИРОВАНИЯ Си

Как Вы, наверное, уже поняли, мы не будем учить Вас программированию на языке Си; более того, мы постоянно напоминаем Вам, что предполагается Ваше знакомство с этим языком и даже некоторый практический опыт написания программ на языке Си. Тем не менее, Вы сможете найти в книге несколько важных разделов, посвященных именно программированию на языке Си под управлением ОС UNIX и содержащих ряд ключевых положений, знание которых поможет Вам использовать язык СИ самым эффективным образом.

Итак, все программы на языке Си представляют собой совокупность функций, реализующих действия, которые должны быть выполнены программой. Как Вам известно, среди них обязательно должна присутствовать функция с именем main, с которой начинается выполнение программы. Ниже показана простейшая программа на языке Си, содержащая лишь описание единственной функции main:

```
main(){
}
```

Усложним теперь приведенную программу, добавив описания еще двух функций и поместив вызовы этих функций в тело функции main:

```
main(){
        a();
        b();
}

        a(){
        }

        b(){
        }
```

После того как программа или ее часть разработана и отлажена, представляется естественным поместить некоторые из отлаженных функций в отдельный файл[1]. В результате исходный текст разрабатываемой программы окажется разбитым на несколько самостоятельных фрагментов, каждый из которых помещен в отдельных файл. Пусть, например, файл с именем main.c содержит следующий текст:

```
main(){
      a();
}
```

а файл с именем a.c — текст

```
a(){
}
```

Описанный прием может оказаться чрезвычайно полезным при разработке большой программы, так как позволяет представить ее в виде системы небольших взаимосвязанных программных модулей, что, во-первых, делает ее более управляемой в смысле модификации исходного текста, а во-вторых, предоставляет пользователю великолепную возможность использовать отдельные модули в других программах, не копируя их. Для этого достаточно воспользоваться предоставляемой препроцессором языка Си возможностью включения содержимого одного или нескольких файлов в состав компилируемого файла. С другой строны, возможно более традиционное решение — создание библиотеки объектных файлоз. В последнем случае необходимые функции включаются в состав компилируемой программы на этапе компоновки последней; ОС UNIX предоставляет пользователю программу обслуживания таких библиотек.

Речь идет о программе ar, которая используется для создания и обслуживания библиотек объектных файлов. Для включения одного из этих файлов в состав компилируемой программы достаточно указать компоновщику имя библиотеки вместо имени объектного файла в командной строке. Например, вместо командной строки вида

```
$cc -o main main.o a.o b.o c.o
```

можно востользоваться командной строкой вида

```
$cc -o main main.o mylib.a
```

если файл mylib.a представляет собой библиотеку объектных файлов, в которую включены файлы a.o, b.o и c.o. Во втором случае компоновщик осуществит включение в состав компилируемой программы лишь тех объектных файлов из библиотеки с именем mylib.a, ссылки на которые

---

[1] Выносить в отдельный файл фрагменты текста неотлаженной программы неразумно, так как в результате этого теряется наглядность взаимосвязи этих фрагментов, что значительно усложняет отладку программы. — *Прим. ред.*

не удовлетворяются в самой компилируемой программе или в одном из включаемых в нее препроцессором файлов[1].

СТАНДАРТНЫЕ БИБЛИОТЕКИ ОС UNIX

Помимо создаваемых личных библиотек, пользователю ОС UNIX предоставляется большой набор стандартных библиотек объектных файлов. Файлы, являющиеся стандартными библиотеками, помещаются в одном из двух доступных всем пользователям каталогов, имеющих полные имена /lib и/usr/lib. Для того, чтобы воспользоваться одним из них, достаточно указать в командной строке флаг —1с параметром name следующим образом:

```
-l <name>
```

В результате этого компоновщик осуществит в каталоге с именем /lib (в случае неудачи в каталоге с именем /usr/lib) поиск библиотеки с именем libname.a, например, после ввода с терминала командной строки вида

```
$cc -o main main.o -lm
```

компоновщик осуществит поиск в каталоге с именем /lib библиотеки с именем libm.a, если же поиск завершится неудачей, то компоновщик повторит поиск, но уже в каталоге с именем /usr/lib. Затем компоновщик осуществит поиск в найденной библиотеке объектных файлов, ссылки на которые не удовлетворяются в самой компилируемой программе или в одном из включаемых в нее препроцессором файлов. При этом, если на этапе компоновки будут использоваться несколько библиотек, то порядок перечисления их имен в командной строке имеет немаловажное значение, так как поиск всех необходимых объектных файлов (ссылки на которые остались еще не удовлетворенными) будет последовательно осуществлен во всех перечисленных библиотеках именно в том порядке, который указан в командной строке. Ниже перечислены имена наиболее часто используемых стандартных библиотек объектных модулей (объектных библиотек):

/lib/libc.a	— библиотека функций языка Си
/usr/lib/libF77.a	— библиотека подпрограмм языка Фортран 77
/usr/lib/libI77.a	— —"—
/usr/.ib/libm.a	— библиотека математических функций
/usr/lib/libmp.a	— —"—
jusr/lib/libl.a	— библиотека поддержки программы Lex
/usr/lib/libln.a	— библиотека поддержки программы Yacc

[1] Авторы считают очевидным, что имя объектного файла, т. е. файла, содержащего объектный код, имеет суффикс .o (например, a.o или d.o), а имя файла, являющегося библиотекой, имеет суффикс .a (например, mylib.a). — *Прим. ред.*

/usr/lib/libplot.a — библиотека функций управления графопостроителем
/usr/lib/lib300.a — —"—
/usr/lib/lib300s.a — —"—
/usr/lib/lib4014.a — —"—
/usr/lib/lib450.a — —"—

/usr/lib/libcurses.a — библиотека функций управления экраном
/usr/lib/libtermcap.a — —"—
/usr/lib/libtermlib.a — —"—

Самой популярной обектной библиотекой, бесспорно, является библиотека с именем /lib/libc.a — библиотека стандартных функций языка Си. Это объясняется тем, что она содержит функции, необходимые для поддержки пользовательских программ на этапе выполнения и используется на этапе компоновки автоматически. Из этого, в частности, следует, что нет необходимости указывать в командной строке флаг —1с.

Объектные файлы, включенные в объектную библиотеку с именем /lib/libc.a, содержат объектный код всех системных функций, реализуемых ядром ОС UNIX. Итак, помимо функций, каждому включению которых в объектный код программы соответсвует появление в нем явной ссылки, библиотека с именем /lib/libc.a включает также все функции, используемые ядром ОС UNIX для реализации программного интерфейса ОС UNIX. Подробное описание этих функций можно найти в разд. 2 и 3 первого тома руководства "UNIX Programmers Manual".

СИСТЕМНЫЕ ВЫЗОВЫ

UNIX — это операционная система разделения времени, и ее главной задачей является поддержка одновременного выполнения нескольких пользовательских программ, находящихся на разных стадиях разработки. Тот факт, что, вызываемая на исполнение пользовательская программа может содержать ошибки программирования, указывает на опасность разрушения в процессе выполнения такой программы ядра ОС UNIX или пользовательского процесса, функционирующего в рамках другой пользовательской программы. Из этого следует, что предоставление пользовательскому процессу системных функций должно осуществляться самым безопасным и хорошо контролируемым способом. Таким "способом" в ОС UNIX является строго определенный и хорошо защищенный аппарат *системных вызовов.*

Синтаксически указание необходимости осуществления системного вызова очень похоже на вызов подпрограммы, однако необходимо хорошо понимать, что при осуществлении системного вызова исполняемый код, реализующий этот системный вызов, находится в ядре ОС UNIX, в то время как вызванная подпрограмма реализуется исполняемым кодом, находящимся в пользовательской программе, осуществившей этот вызов. При осуществлении системного вызова, как правило, использует-

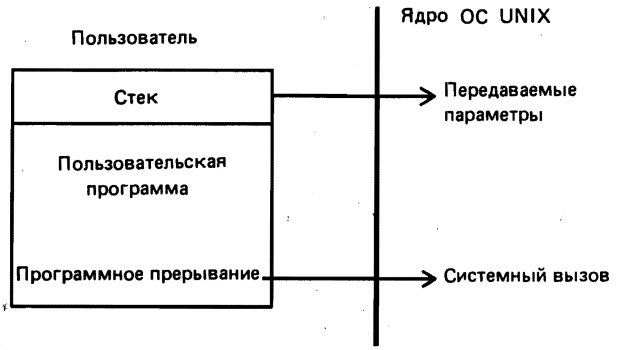

Рис. 2. 1. Схема осуществления системного вызова

ся механизм прерываний, реализуемый аппаратно. Более подробно эту процедуру можно описать так: при осуществлении системного вызова в стек пользовательского процесса заносятся соответствующие параметры (как и в случае вызова подпрограммы), после чего на процессор вызывается инструкция программного прерывания (в случае вызова подпрограммы на процессор вызывается инструкция перехода по стартовому адресу вызываемой подпрограммы). В результате отработки прерывания аппаратурой центрального процессора ЭВМ управление передается по адресу, хранящемуся в некоторой заранее определенной ячейке памяти ЭВМ, так называемом *векторе прерывания*, и тем самым начинается выполнение *подпрограммы обработки прерывания*, исполняемый код которой находится в ядре ОС UNIX. Подпрограмма обработки прерывания, прежде всего, извлекает из стека пользовательского процесса ранее помещенные туда параметры, а затем передает управление подпрограмме, реализующей системную функцию, соответствующую осуществленному системному вызову, исполняемый код этой подпрограммы также находится в ядре ОС UNIX. На рис. 2.1 представлено схематическое изображение последовательности перечисленных действий. После того как подпрограмма, реализующая указанную системную функцию, завершится, ядро ОС UNIX передаст управление в пользовательскую программу, осуществившую системный вызов, точно также, как если бы завершилась подпрограмма, вызванная на выполнение пользовательской программой.

В качестве примера рассмотрим следующий фрагмент программы на языке Си:

```
function(a,b);
```

Синтаксически приведенный фрагмент представляет собой вызов некоторой функции function. Если теперь предположить, что описание функции function приведено в самой пользовательской программе, то объектный код пользовательской программы, полученный в результате компиляции, очевидно, должен включать и объектный код функции function, а получаемый в процессе компиляции текст указанного фрагмента на язы-

ке ассемблера для микропроцессора Motorola 68000 будет иметь следующий вид:

```
movl    _b,-(sp)        |занести b в стек
movl    _a,-(sp)        |занести a в стек
jsr     _function
addql   #8,sp
```

Предположим теперь, что функция function осуществляет системный вызов; тогда получаемый в процессе компиляции текст указанного фрагмент на языке ассемблера Motorola 68000 примет следующий вид:

```
movl    _b,-(sp)        |занести b в стек
movl    _a,-(sp)        |занести a в стек
movw    #FUNC,d0        |занести номер сист. вызова
                        |в регистр d0
trap    #SYS            |программное прерывание
addql   #8,sp
```

Итак, в результате отработки программного прерывания управление передается ядру ОС UNIX, а в регистре d0 сохраняется номер осуществляемого системного вызова.

В этой книге мы рассмотрим 70 наиболее популярных системных вызовов ОС UNIX, которым в библиотеке объектных файлов с именем /lib/libc.a соответсвуют 70 подпрограмм, предоставляющих пользовательской программе возможность осуществить системный вызов. Ниже перечислены системные вызовы System V:

alatm	Посылает процессу сигнал побудки.
chdir	Изменяет текущий каталог.
chmod	Изменяет код защиты файла.
chown	Изменяет идентификаторы владельца и группы файла.
chroot	Изменяет каталог, устанавливаемый текущим, сразу после входа в систему.
close	Закрывает файл.
creat	Создает и открывает файл для записи.
dup, dup2	Создают копию пользовательского дискриптора файла.
execl, execv, execle, execve, execlp, execvp	Осуществляют загрузку и выполнение программы.
fchtl	Осуществляет управление режимом доступа к открытому файлу.
fork	Создает копию текущего процесса.
getpid, getpgrp getppid	Возвращают идентификаторы пользователя, группы, предка процесса.
gepuid, geteuid, getgid, getegid	Возвращают реальные или действующие идентификаторы пользователя или группы.
ioctl	Осуществляет управление режимом доступа к байториентированному специальному файлу.

kill	Посылает сигнал одному или нескольким процессам.
linK	Создает ссылку на существующий файл.
lseek	Перемещает указатель чтения-записи открытого файла.
mknod	Создает новый файл, каталог или специальный файл.
msgctl, msgget, msgop	Осуществляют передачу сообщений.
nice	Устанавливает приоритет текущему процессу.
open	Открывает существующий файл.
pause	Приостанавливает функционирование текущего процесса.
pipe	Осуществляет создание канала.
plock	Фиксирует в памяти текущий процесс.
profil	Осуществляет выдачу профиля времени выполнения.
ptrace	Осуществляет трассировку выполнения программы.
read	Осуществляет чтение из файла заданого числа байт.
semctl, semget, semop	Реализуют систему управления семафоров.
setgrp	Устанавливает идентификатор группы процесса.
setuid, setgid	Устанавливают идентификатор пользователя или группы.
shmctl, shmget, shmop	Осуществляют управление режимами работы средств разделения памяти.
sleep	Приостанавливает выполнение программы на заданный интервал времени.
stat, fstat	Осуществляет получение информации об описателе файла
stime	Осуществляет установку системного таймера.
sync	Осуществляет принудительное завершение всех операций ввода-вывода.
times	Осуществляет получение учетной информации об использовании процессорного времени.
ulimit	Осуществляет получение и установку текущих пользовательских ограничений.
umask	Осуществляет получение информации о значении битов кода защиты созданного файла.
unlink	Осуществляет удаление входа в каталог.
ustat	Осуществляет получение статических данных по использованию файловой системы.
utime	Осуществляет установку времени последнего обращения к файлу и последней модификации файла.
wait	Возвращает управление текущему процессу после завершения процесса-потомка.
write	Осуществляет запись в файл заданного числа байт.

Заметим, что помимо перечисленных в этой таблице подпрограмм, реализующих осуществление системного вызова, библиотека содержит объектные файлы стандартных подпрограмм. Ко вторым, например, от-

носятся стандартные подпрограммы осуществления ввода-вывода, вычисления различных математических функций и т. д.

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕК ОБЪЕКТНЫХ ФАЙЛОВ

Для того чтобы воспользоваться подпрограммой из библиотеки объектных файлов, вполне достаточно указать имя соответствующей подпрограммы в исходном тексте программы пользователя, специфицировав тем самым ссылку на эту подпрограмму. Действительно, обнаружив в объектном коде пользовательской программы ссылку на некоторый объектный файл, компоновщик автоматически осуществит поиск соответствующей библиотеки объектных файлов, а затем поиск указанного объектного файла в ней и включит его в объектный код пользовательской программы.

Но иногда этих действий оказывается недостаточно. Дело в том, что всем не объявленным в пользовательской программе функциям по умолчанию приписывается тип данных int и, следовательно, для хранения возвращаемого значения резервируется некоторое заранее известное число байт памяти, зависящее от типа ЭВМ[1]. Если возвращаемое конкретной функцией значение имеет тип данных, отличный от int, и для его хранения требуется другое количество байт памяти, то результат может быть непредсказуемым. Классическим примером подобной ошибки может быть попытка присвоения переменной типа int возвращаемого значения системного вызова lseek (позже мы рассмотрим этот случай более подробно). В действительности функция lseek имеет тип long int, и соответственно в случае, когда для хранения переменной типа int на ЭВМ отводится 16 бит памяти, результат осуществления этого системного вызова окажется неверным. В качестве примера рассмотрим ЭВМ типа PDP-11 или любую ЭВМ, построенную на базе микропроцессора Intel 8086. При компиляции синтаксически корректного фрагмента программы вида

```
int i = lseek(x, y, z);
```

на одной из таких ЭВМ будет сгенерирован объектный код, при выполнении которого будет получен неверный результат. Действительно, переменной 1 будет присвоено значение, соответствующее лишь младшим 16 битам возвращаемого значения системного вызова lseek[2]. Избежать подобных ошибок довольно просто: для этого достаточно в теле программы явно объявить используемую внешнюю функцию. Восполь-

---

[1] В зависимости от архитектуры конкретной ЭВМ для хранения данных типа int может использоваться различное число бит памяти. Например, на ЭВМ типа VAX или любой ЭВМ, построенной на базе микропроцессора Motorola 68000, для хранения данных типа int используется 32 бита (или 4 байта памяти), в то время как на ЭВМ PDP-11 или любой ЭВМ, построенной на базе микропроцессора Intel 8086, для хранения данных типа unt используется 16 бит (или 2 байта памяти).

[2] Ошибки, подобные описанной, обычно обнаруживаются с помощью программы lint (см. гл. 8).

зуемся этой рекомендацией и перепишем приведенный выше фрагмент программы следующим образом:

```
extern long lseek();
long l = lseek(x, y, z);
```

Теперь появление в теле программы оператора вида

```
int i = lseek(x, y, z);
```

будет обнаружено компилятором, который выведет на стандартный вывод сообщение об ошибке.

ПРЕПРОЦЕССОР ЯЗЫКА ПРОГРАММИРОВАНИЯ Си

Как правило, в программе на языке Си объявить некоторую внешнюю переменную очень просто, однако чем сложнее структура данных, соответствующая объявляемой переменной, тем больше хлопот доставляет ее объявление. В таких случаях полезно использовать заранее подготовленную конструкцию объявления необходимой переменной. Если, например, в пользовательской программе использован вызов некоторой стандартной подпрограммы из некоторой библиотеки объектных файлов, осуществляющей передачу в качестве параметра значения некоторой переменной, то логично было бы заранее подготовить текст объявления этой переменной, чтобы всякая пользовательская программа, осуществившая такой вызов, включала бы объявления переменной без особых затрат для программиста. Затем было бы логично подобные конструкции объединить в одном файле, чтобы всякая пользовательская программа, использующая данную стандартную библиотеку объектных файлов, могла бы осуществить включение объявления всех необходимых переменных из файла со стандартным именем.

Описанная выше проблема решается средствами препроцессора языка программирования Си на этапе компиляции пользовательской программы. Препроцессор языка Си позволяет включить содержимое некоторого заранее подготовленного текстового файла в состав исходного текста пользовательской программы на языке Си на этапе компиляции, но перед началом синтаксического разбора текста программы. Помимо прочих преимуществ использование этой возможности позволяет программисту избавиться от внесения в текст программы ошибок при многократном переписывании одних и тех же объявлений внешних переменных.

Другая проблема, с успехом решаемая средствами препроцессора языка Си, связана с использованием оператора typedef. Как мы уже упоминали, ОС UNIX в настоящее время функционирует на огромном числе самых различных по своей архитектуре ЭВМ, а это, в частности означает, что для хранения одних и тех же значений, в разных реализациях ОС UNIX может быть отведено, вообще говоря, разное число байт памяти. Пусть, например, переменная со значением, равным текущему номеру блока на диске, объявлена в пользовательской программе как переменная, имеющая тип данных daddr_t, а другая переменная, со значением, рав-

ным текущему времени, в результате использования системного вызова time() объявлена в пользовательской программе как переменная, имеющая тип данных time_t. Вам должно быть хорошо известно, что обе они имеют тип данных long integer, которому на ЭВМ PDP-11 соответствуют 4 байта памяти, а на ЭВМ VAX-11 — 8 байт памяти. Примененный здесь прием использования типа данных, определяемого пользователем, избавляет программиста от необходимости учитывать этот факт при разработке программы и одновременно повышает мобильность разрабатываемой прикладной программы. Этот прием легко реализовать средствами препроцессора языка Си, для чего необходимо поместить в одном файле следующие строки:

```
typedef long int daddr_t;
typedef long int dtime_t;
```

и в дальнейшем включать его при необходимости в состав пользовательской программы.

Итак, если в тексте пользовательской программы появится строка вида

```
#include <x.h>
```

или строка вида

```
#unclude "x.h"
```

то в результате обработки такой строки препроцессором языка Си вместо нее в исходный текст программы будет включено содержимое файла с именем x.h. При этом в первом случае поиск файла с именем x.h будет осуществляться препроцессором языка Си в каталоге с именем /usr/include, а во втором — сначала в текущем каталоге, а в случае неудачи — в каталоге с именем /usr/include. Тогда в качестве контекста x.h может быть использовано полное имя файла ОС UNIX любого вида, например в результате обработки препроцессором языка Си строки вида

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

вместо этой строки в исходный текст программы будет включено содержимое файла с именем /usr/include/sys/inode.h (разумеется, в случае удачного поиска). В соответствии с соглашениями, принятыми в ОС UNIX, существуют несколько стандарных каталогов, в которых поиск препроцессором языка Си осуществляется автоматически. Одним из таких каталогов является каталог с именем /usr/include. Добавим, что в каталоге с именем /usr/include/sys в соответствии с этими соглашениями находятся файлы, содержащие определения некоторых структур данных, используемых ядром ОС UNIX.

Проиллюстрируем сказанное примером, для чего рассмотрим следующий фрагмент программы на языке Си[1]:

---

[1] Системный вызов signal() и файл с именем signal.h более подробно будут рассмотрены в гл. 7.

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

В результате обработки препроцессором языка Си первой строки приведенного фрагмента программы вместо нее в исходный текст будет включено содержимое файла с именем /usr/include/signal.h. Этот текстовой файл содержит несколько строк вида

```
#define name value;
```

которые, как видно, представляют собой набор определений символьных замен. Если попытаться скомпилировать этот фрагмент программы, исключив из него первую строку, то в ответ будет получено сообщение компилятора об ошибке, заключающейся в неопределенности переменных SIGINT и SIG_IGN. Это произошло потому, что в составе файла с именем /usr/include/signal.h имеются строки вида

```
#define SIGINT  1
#define SIG_IGN 1
```

После обработки этих строк препроцессором языка Си вторая строка фрагмента программы примет вид

```
signal(1, 1);
```

Понятно, что попытка скомпилировать программу, содержащую описанный фрагмент текста, под управлением другой версии или реализации ОС UNIX может привести к необходимости скорректировать файл с именем /usr/include/signal.h в том случае, если в нем почему-либо отсутствуют определения указанных строковых констант.

УСЛОВНАЯ КОМПИЛЯЦИЯ

Использование средств препроцессора языка Си для включения содержимого заранее подготовленных файлов в состав исходного текста пользовательской программы, делает последнюю более наглядной и удобной для чтения. Кроме того, это дает возможность пользователю избежать появления в файлах, содержащих исходный текст его программы, противоречащих друг другу определений переменных.

Условная компиляция — еще одно интересное средство, предоставляемое пользователям ОС UNIX препроцессором языка Си. Строки исходного текста программы могут игнорироваться или компилироваться компилятором с языка Си в зависимости от выполнения некоторого условия. В качестве примера рассмотрим следующий простой фрагмент программы:

```
            if(getval() > 3){
#ifdef DEBUG
            printf("getval > 3\n");
#endif
            . . . . .
            }
```

Если строковая константа DEBUG к этому моменту была указана в одном из определений символьных замен, то оператор printf () будет скомпилирован в составе программы; в противном случае указанный оператор будет проигнорирован на этапе компиляции. При отладке разрабатываемой программы программист, как правило, дополняет программу вспомогательными, "отладочными" операторами, которые по окончании отладки программы обычно удаляются из исходного текста программы. Возможность условной компиляции некоторых операторов избавляет программиста от этой трудоемкой процедуры. Действительно, оператор printf () в нашем примере будет скомпилирован и в дальнейшем исполнен лишь в том случае, если одной из первых строк исходного текста программы является строка вида

```
#define DEBUG 1
```

или же вызов на выполнение компилятора с языка Си осуществляется с помощью командной строки вида

```
$cc -o prog _DDEBUG prog.c
```

Напомним, флаг –D команды cc позволяет указать определение символьной замены непосредственно в командной строке ОС UNIX. Например, указание в командной строке аргумента вида

```
-D <name> = <value>
```

в точности соответствует помещению в компилируемый файл строки вида

```
#define <name> <value>
```

Поэтому для того, чтобы исключить из процесса компиляции оператор printf () в нашем примере, достаточно опустить в командной строке флаг –D:

```
cc -o prog prog.c
```

Вообще использование приема условной компиляции весьма многообразно, например

```
#ifdef DEBUG
        . . . . .
#else
        . . . . .
#endif
```

или, обращая логические условия

```
#ifndef DEBUG
        . . . . .
#else
        . . . . .
#endif
```

В последнем варианте оператор printf () будет скомпилирован при отсутствии определения символьной замены.

С другой строны, само условие может быть сформулировано следующим образом:

```
#if N > 3
        . . . . .
#endif
```

или иначе:

```
#if ((N + 3) & 4) || (A)
```

В последнем случае условие считается выполненным, если выражения в скобках принимают истинные, т. е. ненулевые значения.

МАКРОПОДСТАНОВКИ

Помимо рассмотренного выше определения символьной замены препроцессор языка Си предоставляет пользователю средство определения макроподстановки. Приведем простой пример использования определения макроподстановки:

```
#define getchar() getc(stdin)
```

В результате обработки препроцессором языка Си такого определения макроподстановки, все контексты getchar (), содержащиеся в любой строке исходного текста программы, начиная со следующей строки, будут заменены на контексты getc(stdin). Фактически это означает, что на этапе выполнения такой программы вместо функции getchar () будет выполняться функция getc(stdin) (кстати, именно так и происходит в действительности). На самом деле стандартная библиотека ввода-вывода SIO не содержит подпрограммы, реализующей функцию getchar (). Вместо нее используется функция getc(stdin), а в файл с именем /usr/include/stdio.h, содержащий все необходимые для использования SIO описания и объявления, помещается приведенное выше определение макроподстановки[1].

Чрезвычайно полезным при разработке программ может оказаться следующее (обычно помещаемое в файл с именем /usr/include/assert.h) определение макроподстановки:

```
#ifndef NDEBUG
#define assert(ex)\
{\
        if(!(ex)){\
                fprintf(stderr,\
                "Assertion failed: file %s, line %d\n",
                __FILE__, \
                __LINE__);\
                exit(1);\
```

[1] Назначение файла с именем stdio.h будет подробно рассмотрено в гл. 5.

```
            }\
}
#else
#define assert(ex);
#endif
```

Если строковая константа NDEBUG была указана в определении символьной замены, то вместо полученного в результате обработки препроцессором контекста assert(expression) в исходном тексте программы появится контекст вида '; ', что синтаксически соответствует пустому оператору языка Си. Допустим, что строковая константа NDEBUG не была указана в определении символьной замены, тогда в результате обработки контекста assert(expression) препроцессор осуществит макроподстановку, определение которой приведено выше. Допустим далее, что выражение expression, указанное в качестве аргумента функции assert(), в результате вычисления принимает значение 0. Тогда выполнение программы будет прервано и на стандартный протокол будет выведено сообщение об ошибке. Мы уже указывали на полезность определенной таким образом функции assert. Представим себе, что одна из функций, описанных в теле разрабатываемой Вами программы, должна производить некоторые вычисления до тех пор, пока значение некоторой переменной не выйдет за пределы некоторого заранее известного интервала. Решить эту задачу легко с помощью функции assert(). Поместим для этого в исходном тексте программы строку

```
assert(LO < n && n < HI);
```

Выполним теперь вместо препроцессора языка Си операцию макроподстановки в соответствии с приведенным определением макроподстановки. В результате получим следующий фрагмент текста:

```
{
    if(!(LO < n && n < HI)){
            fprintf(stderr,
            "Assertion failed: file %s, line %d",
            "s.c",
        9);
            exit(1);
    }
}
```

Если теперь переменная n примет значение, меньшее значения переменной LO или большее значения переменной HI, то в результате на стандартный протокол[1] (которым по умолчанию является терминал пользователя) будет выведено сообщение вида

```
Assertion failed: file s.c, line 9
```

[1] Стандартным протоколом принято называть файл с пользовательским дескриптором файла 2. — *Прим. ред.*

Когда Вы убедитесь в правильности Ваших представлений относительно характера изменения значений всех интересующих Вас переменных и отладите разрабатываемую Вами программу, не удаляйте из исходного текста программы все вызовы функции assert (), достаточно воспользоваться в командой строке флагом — D так, как это было показано выше.

ПЕРЕДАЧА ПАРАМЕТРОВ ПРОГРАММЕ НА ЯЗЫКЕ Си

Если исполняемый файл, полученный в результате компиляции и компоновки программы, написанной на языке Си, вызывается на выполнение вводом с терминала его имени, то параметры этой командной строки могут быть переданы указанной программе в качестве аргументов функции main(argc, argv). При этом значение переменной argc равно числу параметров введенной командной строки, а аргумент argv представляет собой массив указателей на строки, являющиеся параметрами введенной командной строки[1]. Итак, рассмотрим фрагмент программы

```
main (argc, argv);
int argc;
char *argv[];
{
  . . . .
}
```

Если введенная с терминала командная строка имеет вид

```
prog p1, p2
```

то аргумент argc получит значение 3 (напоминаем, имя программы включается в число параметров), а массивы argv[0], argv [1] и argv [2] содержат соответственно строки "prog", "p1" и "p2". Приведенная ниже программа осуществляет вывод на стандартный вывод параметров командной строки, с помощью которой она была вызвана на выполнение. Фактически эта программа представляет собой простейшую версию команды ОС UNIX echo:

```
main(argc,argv)
int argc;
char **argv;
{
            int i;
            for(i = 1; i < argc; i++)
              printf("%s%c",argv[i], (i<argc-1)?' ':'\n');
}
```

В соответствии с принятыми в ОС UNIX соглашениями параметры командной строки, начинающиеся с символа — , являются флагами вызываемой на выполнение команды. Если следовать этим правилам при разработке программ, то, очевидно, разрабатываемые программы должны осуществлять разбор командной строки с целью распознавания флагов и

[1] Значения параметров заносятся в стек пользовательского процесса.

аргументов команды. Приводимый ниже фрагмент программы на языке Си успешно решает эту задачу:

```
main(argc,argv)
int argc;
char **argv;
{
            switch(argc){
            case 2:
                        dofile(argv[1]);
                        break;
            case 3:
                        if(argv[1][0] != '-')
                        error();
                        switch(argv[1][1]){
                                    case 'k':
                                      do_k(argv[2]);
                                      break;
                                    case 'p':
                                      do_p(argv[2]);
                                      break;
                                    default:
                unknown(arg[1][1]);
                        }
      default:
                                      error();
                                    }
                                    . . . . .
}
```

Более подробное описание проблемы передачи параметров программам, написанным на языке Си, Вы сможете найти в гл. 6 книги [5].

ЗАДАЧИ

1. Поместите исходный текст одной из разработанных Вами ранее программ в нескольких отдельных файлах и попробуйте получить исполняемый код после их раздельной компиляции. Поместите в отдельный файл объявления и определения переменных, общие для функций, описания которых помещены в различные файлы, и включите их в состав последних средствами препроцессора языка Си.

2. Разработайте программу на языке Си, которая использовала бы не самые популярные стандартные подпрограммы, и попробуйте на этапе компоновки исследовать смысл непонятных Вам флагов команды ld (при необходимости воспользуйтесь соответствующей страничкой из документа UNIX Programmer's Manual). Может оказаться, что Вам придется включить в исходный текст своей программы один из стандартно включаемых файлов.

3. Попробуйте осуществить условную компиляцию отдельных фрагментов программы, которую Вы использовали для решения первой задачи этой главы. Попытайтесь поместить в определение макроподстановки небольшие функции из этой программы.

4. Разработайте программу на языке Си, которая последовательно обрабатывала бы несколько текстовых файлов так, чтобы число этих файлов можно было бы

указать в командной строке. Постройте разбор командной строки таким образом, чтобы результат его не зависел от порядка ввода параметров (строка xx —a name —b была бы эквивалентна строке xx —b —a name).

ЛИТЕРАТУРА

1. S.R. Bourne (1978), An Introduction to the Shell, UNIX Programmers Manual Volume 2a.

2. B.W. Kernighan & D.M. Ritchie (1978), The C Programming Language, Prentice Hall Inc. (См. [5] в списке литературы к гл. 1).

3. Motorola Inc. (1980), MC68000, 16-bit Microprocessor User's Manual, second edition.

Глава 3. ФАЙЛЫ

Идеология ОС UNIX, созданная и реализованная Д. Ритчи и К. Томпсоном, предполагает, что пользователю будет предоставлен широкий набор тщательно выбранных, функционально дополняющих друг друга программных средств, лишенных ненужной сложности и специализации. Мощность и одновременная гибкость файловой системы ОС UNIX является, пожалуй, самым ярким доказательством разумности такого подхода. В этой главе мы рассмотрим структуру файла ОС UNIX , а гл. 4, 5 посвятим вопросам использования программного интерфейса файловой системы ОС UNIX.

Интерфейс между двумя пользовательскими программами (или между пользовательской программой и внешним устройством, а также между двумя процессами) в ОС UNIX реализуется в рамках единой структуры данных, называемой *файлом* ОС UNIX. Теоретически файл ОС UNIX представляет собой последовательность байт данных, завершающуюся символом логического конца файла. Физически такая последовательность байт может представлять собой, например, набор блоков диска или блоков магнитной ленты либо область оперативной памяти. Среди менее привычных представлений файла в ОС UNIX можно назвать несколько катушек перфоленты, пользовательский терминал или трафик локальной сети ЭВМ. Итак, использование файла как единой универсальной структуры данных, в рамках которой реализуется любая операция ввода-вывода, гарантирует пользователю, что если он умеет программировать операции ввода-вывода в файл, то сможет запрограммировать и операцию ввода-вывода куда угодно.

Все файлы ОС UNIX имеют имена, которые могут быть использованы пользовательскими программами как средство получения доступа к данным, содержащимся в соответствующих файлах. Файлы ОС UNIX являются составляющими некоторой структуры данных, называемой файловой системой ОС UNIX.

ТИПЫ ФАЙЛОВ ОС UNIX

Всякий файл ОС UNIX в соответствии с его типом может быть отнесен к одной из следующих четырех групп: обычные файлы, каталоги, специальные файлы и, наконец, каналы[1]. Каналы как средство связи двух пользовательских процессов будут подробно рассмотрены в гл. 5.

ОБЫЧНЫЕ ФАЙЛЫ ОС UNIX

Обычный файл представляет собой либо совокупность нескольких блоков, либо ни одного блока диска, входящих в состав файловой системы ОС UNIX. В указанных блоках диска может храниться произвольная информация, например объектный код Вашей программы или текст журнальной статьи.

Важная особенность обычных файлов заключается в том, что ОС UNIX не накладывает никаких ограничений на их внутреннюю структуру, которая определяется только прикладной программой, обрабатывающей указанные файлы. Например, компилятор с языка Си обрабатывает свои входные файлы так, как будто они содержат исходный текст программы на языке Си, и потому он (и только он) будет весьма "обескуражен", обнаружив вместо исходного текста объектный код. Таким образом, ОС UNIX никак не различает содержимого обычных файлов и использование суффиксов имен файлов, таких, как .c, .o, или .a, а отражает удобное, но искусственное и ни к чему не обязывающее соглашение, принятое в ОС UNIX.

Если Вы привыкли рассматривать файл как упорядоченную последовательность записей, такой подход покажется необычным, однако надо хорошо понимать, что создать файл, который имел бы необходимую внутреннюю структуру, очень просто средствами прикладной программы. Однородность внутренней структуры обычного файла как основной компоненты файловой системы ОС UNIX имеет чрезвычайно важное значение, так как является залогом функциональной гибкости последней.

Важной характеристикой операций ввода-вывода, осуществляемых под управлением ОС UNIX, является их буферизованность. Это означает, в частности, что пользователь может не интересоваться, какие конкретно блоки диска составляют файл, обрабатываемый его программой. Таким образом, прикладная программа может обрабатывать обычный файл как однородную последовательность байт данных, не учитывая его физичес-

Рис. 3.1. Логическая структура обычного файла ОС UNIX

[1] Существует еще один тип файлов — мультиплексные файлы. Он является экспериментальным типом файлов и имеется лишь в некоторых реализациях ОС UNIX. — *Прим. ред.*

кой структуры, определяемой файловой системой ОС UNIX. Логическая структура такого файла приведена на рис. 3.1.

КАТАЛОГИ ОС UNIX

Каталоги представляют собой файлы особого типа, отличающиеся от обычных прежде всего тем, что осуществить запись в них может только ядро ОС UNIX, в то время как доступ по чтению к каталогу может получить любой пользовательский процесс, имеющий соответствующие полномочия. Информация, содержащаяся в каталоге, отражает соответствие между файлами и их именами. Таким образом, можно говорить, что совокупность всех каталогов специфицирует структуру файловой системы в целом. Каждый элемент каталога (вход в каталог) состоит из двух полей: поля, содержащего имя файла, которое может состоять не более чем из 14 символов, и поля, содержащего указатель на "нечто". В терминологии ОС UNIX этот указатель называется ссылкой. Ссылка является указателем на "нечто", называемое описателем файла, где хранится вся информация о файле: дата его создания, имя владельца, код защиты файла и информация о положении на диске и т. д. Каждому файлу ОС UNIX соответствует только один описатель файла. В рамках каждой файловой системы описатели файлов пронумерованы последовательно, начиная с 1.

На рис. 3.2 приведен пример структуры каталога. Итак, каждый вход в каталог содержит в точности 16 байт информации: два из них содержат номер описателя файла, а остальные 14 байт — имя файла. Если имя файла содержит менее 14 символов, то оставшаяся свободной часть поля имени файла (входа в каталог) заполняется символами, имеющими восьмеричный код 0. В любом каталоге содержится, по крайней мере, два элемента, содержащие в поле имени файла имена . и . . . Элемент каталога (вход в каталог), содержащий в поле имени файла контекст . , в поле ссылки содержит ссылку на описатель файла, в котором хранится информация о самом каталоге. Это необходимо для того, чтобы любая программа могла осуществлять чтение информации из текущего каталога, не зная абсолютного полного имени файла. Элемент каталога, содержащий в поле имени файла контекст . . , в поле ссылки содержит ссылку на описатель файла, в котором хранится информация о родительском каталоге текущего каталога. Таким образом, вход в каталог, содержащий в поле имени файла имя . . , обеспечивает возможность переме-

Номер описателя файла	Имя файла
5287	
61	
2808	bin
2813	block. c
4169	copyfile. c
4168	copyfile. o
5187	files. n
4883	io. n
2805	io0. c
5191	julie. n
5184	move
3928	
4173	move. c
5102	move. o
4174	nohup. out

Рис. 3. 2. Пример каталога ОС UNIX

щения по дереву от текущего каталога на один уровень в сторону корня дерева[1].

Напомним, что файловая система ОС UNIX имеет иерархическую структуру, образующую дерево каталогов и файлов. Это означает, что каждому каталогу ОС UNIX, кроме корневого, соответствует в точности один каталог, называемый родительским, и несколько или ни одного каталога-потомка.

ПОЛНЫЕ ИМЕНА ФАЙЛОВ

Как мы упоминали, поле имени элемента каталога содержит 14 байт, т. е. имя файла может содержать не более 14 символов. Однако для спецификации файла как элемента иерархической структуры необходимо указать полный путь до него по дереву, начиная от корневого каталога, или, согласно терминологии, принятой в ОС UNIX, абсолютное полное имя файла. Добавим, что если путь по дереву указан не от корневого каталога, то его принято называть относительным полным именем файла. Полное имя файла является слоговым. Каждый слог в полном имени файла, кроме, быть может, последнего, представляет собой имя файла, являющегося каталогом. Слоги отделяются друг от друга символом /. Последний слог в полном имени файла задает имя файла, который может быть каталогом, обычным файлом или специальным файлом.

ССЫЛКИ НА ФАЙЛ

Номер описателя файла хранится в поле ссылки в элементе каталога, содержащем в поле имени файла то имя файла, по которому к нему обеспечивается доступ. При такой схеме взаимно однозначного соответствия между именем файла и самим файлом нет. Несколько каталогов могут содержать элементы, содержащие в поле имени файла различные имена файлов, а в поле ссылки номер одного и того же описателя файла. Элементы каталогов, обеспечивающие доступ к одному и тому же файлу, называются *ссылками на файл.* Наличие такого механизма обеспечивает возможность доступа к одному и тому же файлу по различным абсолютным полным именам, соответствующим этому файлу.

СПЕЦИАЛЬНЫЕ ФАЙЛЫ

Специальные файлы — это одна из самых неожиданных особенностей ОС UNIX. Каждому внешнему устройству, поддерживаемому ОС UNIX, ставится в соответствие некоторый файл, называемый *специальным файлом*. Итак, специальный файл ОС UNIX — это файл, имеющий некоторую специальную структуру, его нельзя использовать для хранения данных,

---

[1] Корень дерева (или корневой каталог) не имеет родительского каталога, поэтому его элементы, содержащие в поле имени имена файлов . и . . , в поле ссылки содержат ссылки на один и тот же описатель файла — описатель самого корневого каталога.

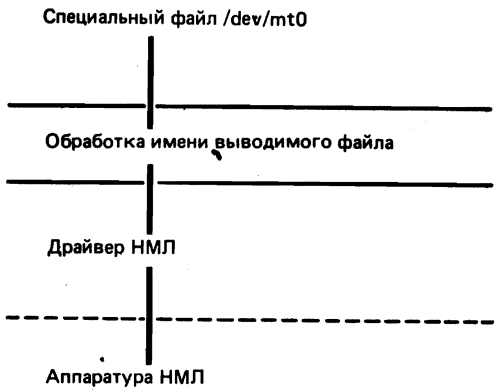

Рис. 3.3. Логическая схема осуществления вывода информации на НМЛ

как обычный файл или каталог. В то же время над специальным файлом можно производить те же операции, что и над любым другим: открывать, вводить и выводить информацию и т. д. Результат применения любой из этих операций зависит от того, какому конкретному физическому устройству соответствует обрабатываемый специальный файл, однако в любом случае будет осуществлена соответствующая операция ввода-вывода на внешнее устройство, которому соответствует выбранный специальный файл. Например, для того чтобы вывести информацию на магнитную ленту, установленную на накопитель с логическим номером 0, достаточно осуществить вывод в файл с именем /dev/mt0. Функциональная логическая схема такой операции ввода-вывода представлена на рис. 3.3. Добавим, что использование специального файла ОС UNIX предоставляет пользователю возможность получить непосредственный доступ к внешнему устройству, минуя программный интерфейс ОС UNIX, а это чревато различного рода осложнениями, вроде тупиковых ситуаций или разрушения файловой системы (например, при осуществлении операции вывода в специальный файл, соответствующий магнитному диску, на котором хранится файловая система ОС UNIX). Более подробно специальные файлы ОС UNIX и проблемы управления файловой системой ОС UNIX будут рассмотрены в следующей главе.

МНОГОТОМНЫЕ ФАЙЛОВЫЕ СИСТЕМЫ

Хотя корневой каталог файловой системы ОС UNIX располагается всегда на одном и том же магнитном диске, это вовсе не означает, что и вся файловая система должна находиться на этом же магнитном диске или пакете магнитных дисков. Иначе говоря, файловая система ОС UNIX может быть многотомной. Доступ к тому файловой системы, отличному от тома, на котором находится корневая файловая система, обеспечивается в результате *подключения* его к некоторому каталогу, уже существующему в корневой файловой системе. Фактически, на этапе раскрутки ОС UNIX осуществляется автоматическое подключение фиксированного тома файловой системы, который принято называть *корневой файловой системой*, так как его *корневой каталог* становится корневым каталогом всей файловой системы ОС UNIX. Построение многотомных файловых систем становится возможным благодаря единообразию иерархичес-

ких структур каждого тома, что позволяет дополнять дерево каталогов корневой файловой системы, подключая к одному из конечных ее узлов поддерево каталогов файловой системы подключаемого тома.

Право использования команды подключения тома файловой системы mount (ее полное имя — /etc/mount) обычно предоставляется только привилегированным пользователям ОС UNIX. Параметрами команды mount должны быть имя специального файла, соответствующего внешнему устройству (как правило, это накопитель на магнитных дисках), на которое установлен подключаемый том файловой системы, и имя каталога, к которому подключается том файловой системы. Например, в результате выполнения командной строки вида

```
/etc/mount /dev/mr0 /mr
```

том файловой системы, установленный на внешнее устройство, которому соответствует специальный файл с именем /dev/mr0, будет подключен к каталогу корневой файловой системы с полным именем /mr. Схематическое изображение процедуры подключения тома файловой системы можно увидеть на рис. 3. 4.

Итак, для осуществления доступа к корневому каталогу подключенного тома файловой системы достаточно осуществить доступ к каталогу с полным именем /mr. Это означает, что при осуществлении доступа к файлу с именем /mr/test фактически осуществляется доступ к физическому устройству, отличному от того, на котором установлена корневая файловая система. Действительно, файл с именем /mr/test находится на подключенном томе файловой системы и имеет в нем относительно полное имя /test.

При осуществлении доступа к специфицированному пользователем файлу ОС UNIX производит разбор по слогам его полного имени. Как Вы уже знаете, каждому слогу полного имени файла соответствует файл, для которого этот слог является именем. Итак, осуществляя разбор полного имени по слогам, ОС UNIX определяет, не является ли текущий файл каталогом, к которому был подключен том файловой системы ОС UNIX[1]. Если это так, то описатель следующего файла будет выбран уже с другого физического устройства, и тем самым поиск специфицированного пользователем файла будет продолжен на физическом устройстве, где установлен подключенный том файловой системы.

---

[1] Для решения этой задачи используется файл с именем /etc/mtab. Он обрабатывается командами mount и umount как таблица, каждая строка которой состоит из 64 символов; первые 32 из них представляют собой имя каталога, к которому подключен том файловой системы, а вторые 32 символа — это имя специального файла, соответствующего физическому устройству, на которое установлен подключаемый том файловой системы. Итак, команда mount заносит, а команда umount (команда отключения тома файловой системы) удаляет из нее соответствующую строку. — *Прим. ред.*

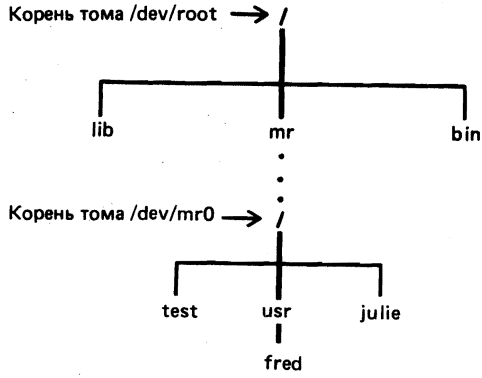

Рис. 3.4. Логическая схема подключения тома файловой системы ОС UNIX

Итак, пусть подключение тома файловой системы завершено, тогда иерархическая структура файлов этого тома включена как поддерево в общую иерархическую структуру файловой системы ОС UNIX, а сами файлы подключенного тома файловой системы доступны пользователю точно так же, как и файлы корневой файловой системы. Это означает, что пользователю не обязательно знать, на каком конкретно физическом устройстве хранится тот или иной файл ОС UNIX. Из этого правила есть, впрочем, одно исключение: все ссылки на файл должны находиться на томе файловой системы, содержащем этот файл. Если это ограничение нарушено, то в результате отключения тома файловой системы возникнут неудовлетворительные ссылки, что может привести к нарушению целостности файловой системы ОС UNIX. Отключить том файловой системы можно лишь после того, как будут закрыты все находящиеся на нем и открытые в процессе работы файлы. Попытка пользователя отключить том файловой системы, на котором находится открытый файл, приведет к появлению на пользовательском терминале сообщения[1]

```
mount device busy
```

Если на Вашем терминале появится такое сообщение, то, скорее всего, это означает, что Вы забыли сменить текущий каталог и "покинуть" отключаемый том файловой системы, и один из них, наверняка, является в данный момент текущим.

СТРУКТУРА ФАЙЛОВОЙ СИСТЕМЫ ОС UNIX

Как правило, файловая система ОС UNIX находится на магнитном диске, или пакете магнитных дисков, или другом устройстве произвольного доступа и физически представляет собой последовательность блоков диска. После создания на размеченном магнитном диске пустой файловой системы ОС UNIX множество блоков диска может быть условно разделено на четыре группы. К первой относится блок диска с номером 0; в. нем при необходимости хранится промежуточный загрузчик ОС UNIX. Ко второй — блок диска с номером 1; этот блок диска согласно терминологии ОС UNIX называется *суперблоком* файловой системы и содержит информацию о физической структуре файловой системы. К

[1] "Подключенное устройство занято". — *Прим. ред.*

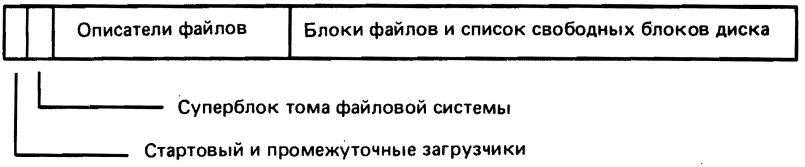

Рис. 3. 5. Логическая схема тома файловой системы ОС UNIX

третьей группе относится несколько последовательных блоков диска, начиная с блока диска с номером 2; блоки диска этой группы содержат некоторое заранее известное число описателей файлов. И наконец, к четвертой группе относятся все блоки диска, которые могут быть использованы для хранения информации.

На рис. 3. 5 схематически изображена описанная физическая структура файловой системы ОС UNIX.

ОПИСАТЕЛИ ФАЙЛОВ

Как уже отмечалось выше, каждый файл ОС UNIX может быть однозначно специфицирован некоторой структурой данных, называемой описателем файла; описатели файлов конкретной файловой системы находятся в последовательных блоках диска, начиная с блока диска с номером 2. Любой описатель файла представляет собой последовательность из 64 байт, т. е. каждый блок диска, если он имеет размер 512 байт, может содержать восемь описателей файла.

Опишем теперь следующую логическую структуру описателя файла (для чего рассмотрим объявление на языке Си переменной типа структура, находящееся в файле с именем /usr/include/sys/ino.h и используемое при генерации ОС UNIX) :

```
/*
 *  Структура описателя файла.
 */

struct dinode
{
  unsigned short  di_mode; /* режим доступа и тип файла    */
  short    di_nlink;       /* счетчик числа ссылок на файл */
  short    di_uid;         /* идентификатор владельца      */
  short    di_gid;         /* идентификатор группы         */
  off_t    di_size;        /* счетчик числа байт в файле   */
  char     di_addr[40];    /* указатели на блоки диска     */
  time_t   di_atime;       /* дата последнего доступа      */
  time_t   di_mtime;       /* дата последней модификации   */
  time_t   di_ctime;       /* дата создания                */
};

/*
 * из 40 байт массива di_addr использовано 39;
 * всего 13 указателей по три байта каждый.
 */
```

Как видите, описатель файла содержит всю необходимую информацию, характеризующую файл. Остается добавить, что занесение информации в описатель файла осуществляется ОС UNIX, а не пользователем. Ниже будут более или менее подробно описаны все поля структуры dinode, а в следующей главе—использование их при осуществлении операций ввода-вывода.

di_mode

Этому полю структуры dinode соответствует одно 16-разрядное слово, значения бит которого определяют тип файла (обычный файл, каталог, специальный файл или канал) и код защиты файла (ниже будет описано более подробно).

di_nlink

Это поле структуры dinode используется ОС UNIX для хранения значения счетчика числа ссылок на данный файл, т. е. числа элементов каталогов файловой системы ОС UNIX, содержащих в поле ссылки номер описателя данного файла. Понятно, что после установления в этом поле структуры dinode значения 0 файл, можно считать удаленным, а соответствующие ему описатель файла и блоки диска считаются свободными.

di_uid, di_gid

Эти поля структуры dinode специфицируют пользователя-владельца данного файла, а также группу, к которой он принадлежит.

di_size

Это поле структуры dinode используется ОС UNIX для хранения значения счетчика числа байт информации, хранящейся в данном файле, если он является обычным файлом, каталогом или каналом. Если же данный файл является специальным, то данное поле структуры dinode используется для хранения информации, специфицирующей физическое устройство, которому соответствует специальный файл.

di_addr

Это поле структуры dinode используется ОС UNIX для хранения указателей местоположения блоков диска, содержащих информацию, помещенную в данный файл. Всего в этом поле может храниться 13 указателей местоположения, каждому из которых отведено 24 бита. Первые 10 указателей местоположения указывают на местоположение первых 10 блоков файла. Если оказывается, что файл занимает более 10 блоков диска, то в 11-й указатель заносится информация о местоположении на диске *первичного блока косвенности,* который содержит до 128 указателей местоположения блоков файла (для хранения каждого такого указателя местоположения отводится уже по 32 бита). Далее, 12-й указатель местоположения указывает на *вторичный блок косвенности,* содержащий 128 указателей местоположения первичных блоков косвенности, а 13-й соответственно на *третичный блок косвенности,* содержащий 128

указателей местоположения вторичных блоков косвенности. Ниже приведено схематическое изображение описанной здесь структуры блоков файла и блоков косвенности:

```
di_addr->&lt;0-9&gt;--> 10 блоков файла
        <10 >--> 1-->128 блоков файла
        <11 >--> 1-->128-->128*128 блоков файла
        <12 >--> 1-->128-->128*128-->128*128*128 блоков файла
```

Используя подобную схему адресации блоков файла, можно теоретически создать файл, содержащий

(10 + 128 + 128 * 128 + 128 * 128 * 128) * 512

байт информации. При этом первые 5120 байт могут быть считаны из файла за одно обращение к накопителю на магнитном диске; для считывания байтов с порядковыми номерами из диапазона 5120 — 70655 требуются уже два обращения к накопителю на магнитном диске; для считывания из файла байт с порядковыми номерами из диапазона 70656 — 8459263 требуются уже три обращения к накопителю на магнитном диске; и наконец, для считывания из файла байт с порядковыми номерами из диапазона 8459264—1082201088 требуется уже четыре обращения к накопителю на магнитном диске. На практике использование кэша при осуществлении операций ввода-вывода на магнитный диск (подробно описанное в следующей главе книги) позволяет существенно повысить эффективность этих операций.

di_atime, di_mtime, di_ctime

Эти поля структуры dinode используются ОС UNIX для хранения информации о дате последнего доступа, дате последней модификации и дате создания данного файла соответственно. Эта информация активно используется целым рядом инструментальных средств ОС UNIX, например командой make, подробно описанной в гл. 9.

СУПЕРБЛОК

Опишем теперь логическую структуру суперблока файловой системы ОС UNIX, для чего рассмотрим объявление на языке Си переменной типа структура, находящееся в файле с именем /usr/include/sys/filsys.h и используемое при генерации ОС UNIX. Упрощенный вариант такого объявления переменной приведен ниже. Суперблок файловой системы ОС UNIX всегда находится в блоке диска с номером 1; он содержит чрезвычайно важную информацию, описывающую физическую структуру файловой системы в целом. Сюда включается информация о размере логического блока файловой системы, о диапазоне номеров блоков диска, в которых хранятся описатели файлов. Кроме того, здесь содержатся последовательность указателей на свободные описатели файлов и указатель на заголовок связанного списка свободных блоков диска. Итак, указанное объявление переменной имеет вид

```
/*
 * Структура суперблока
 */

struct filsys
{
  ushort   s_isize;   /* размер списка i-list в блоках   */
  daddr_t s_fsize:   /* размер тома файловой системы
                        в блоках                        */
  short   s_nfree;   /* счетчик числа свободных блоков  */
  daddr_t s_free[NICFREE];   /* список свободных блоков */
  short   s_ninode;  /* счетчик числа описателей файлов */
  ino_t   s_inode[NICINOD]; /* список описателей файлов */
  char    s_ronly; /* флаг доступности только по чтению */
  daddr_t s_tfree;   /*  общее  число  свободных блоков */
  ino_t   s_tinode;  /*  общее число свободных
                                    описателей файлов */
  char  s_fname[6];    /* метка тома файловой системы   */
  short s_clean;       /* флаг пустоты файловой системы */
};
```

СПИСОК СВОБОДНЫХ БЛОКОВ

Все блоки диска, составляющие файловую систему, но не используемые для хранения информации, связаны в единый список свободных блоков. Информация о списке свободных блоков хранится в массиве s_free — одном из полей структуры filsys. Массив s_free содержит s_nfree элементов (максимальная возможная размерность массива s_free задается системной константой NICFREE и обычно равна 150); элементы s_free [1] . . . , s_free [s_nfree—1] представляют собой указатели местоположения свободных блоков, а элемент s_free [0] — указатель местоположения заголовка списка свободных блоков. Ниже приводится логическая структура блока диска, являющегося элементом указанного списка свободных блоков, в виде объявления на языке Си переменной типа структура, которое хранится в файле с именем /usr/include/ /sys/fblk.h.

```
struct fblk
{
   short     df_nfree;
   daddr_t   df_free[NICFREE];
};
```

Итак, кадый блок диска, являющийся элементом списка свободных блоков, содержит информацию, имеющую вышеописанную логическую структуру. При этом содержимое полей df_free и df_nfree структуры fblk интерпретируется ОС UNIX аналогично содержимому полей s_free и s_nfree структуры filsys, которая, как известно, содержится в суперблоке файловой системы. В частности, значение элемента массива df_free [0] представляет собой указатель местоположения следующего элемента

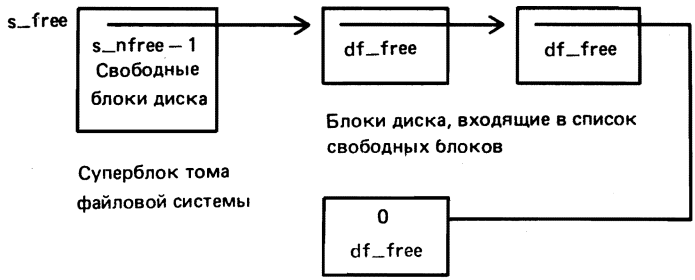

Рис. 3. 6. Логическая структура списка свободных блоков диска

списка свободных блоков. На рис. 3. 6 схематически изображен список свободных блоков.

СПИСОК СВОБОДНЫХ ОПИСАТЕЛЕЙ ФАЙЛА

Массив s_inode содержит информацию о свободных описателях файла и используется ОС UNIX аналогично массиву s_free (рис. 3. 7). Однако поскольку местоположение описателей файла на диске заранее известно и неизменно и, кроме того, каждый описатель файла сам содержит информацию о том, занят он или свободен, то нет нужды создавать списковую структуру. Массив s_inode содержит s_ninode элементов (максимальная возможная размерность массива s_inode задается системной константой NICINODE и обычно равна 100); все элементы массива s_inode представляют собой номера свободных описателей файла. Всякий раз, когда в процессе работы создается файл, ему ставится в соответствие описатель файла с номером, равным значению одного из элементов массива s_inode; если при этом оказывается, что все элементы массива имеют значения, равные 0, то ОС UNIX осуществляет просмотр суперблока файловой системы и заносит в массив s_inode новые значения. Понятно, что после того, как будет использован последний свободный описатель файла, создание нового файла станет невозможным.

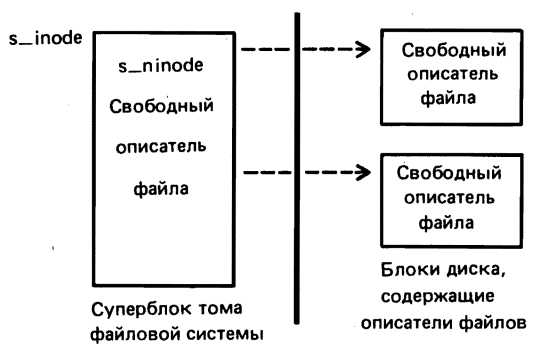

Рис. 3.7. Зарезервированные описатели файла

ЗАРЕЗЕРВИРОВАННЫЕ ОПИСАТЕЛИ ФАЙЛА

Описатели файла с номерами 1 и 2 используются ОС UNIX для специальных целей. Так, описатель файла с номером 1 используется ОС UNIX для спецификации испорченных блоков диска. Ссылка на созданный таким образом файл не заносится ни в один каталог и потому недоступен пользователю ОС UNIX. Описатель файла с номером 2 всегда соответствует корневому каталогу файловой системы, что гарантирует единообразие процедуры поиска файла в подключенном томе файловой системы.

ЗАЩИТА ФАЙЛОВ

Управление доступом к файлам ОС UNIX — одна из основных задач ее файловой системы. Поскольку файлы ОС UNIX могут содержать как данные, так и исполняемый объектный код, то задача управления файлами ОС UNIX — сложна и многообразна.

Каждый описатель файла содержит информацию о том, какого типа доступ разрешен к файлу, соответствующему этому описателю файла (эта информация хранится в поле di_mode). Кроме того, каждый файл ОС UNIX ассоциирован с некоторым пользователем и некоторой группой пользователей ОС UNIX.

Значения 9 младших бит поля di_mode определяют значения так называемого кода защиты файла, специфицируя разрешение на доступ к файлу по записи, доступ по чтению и доступ по выполнению для владельца файла, для членов группы владельца файла и для всех остальных пользователей ОС UNIX.

При попытке осуществить доступ к файлу файловая система сравнивает действующий идентификатор пользователя процесса с хранящимся в описателе файла идентификатором пользователя — владельца файла. Если идентификаторы пользователя совпадают, то пользователь получает (или не получает) необходимый доступ к файлу. Последнее зависит от того, разрешен или нет владельцу данного файла доступ к требуемому типу. Если же упомянутые идентификаторы пользователя не совпадают, то ОС UNIX аналогично сравнивает идентификаторы группы. И вновь, если эти идентификаторы совпадают, то пользователь получает (или не получает) доступ к файлу необходимого типа в зависимости от значений соответствующих бит кода защиты файла. Если идентификаторы группы не совпадают, то пользователь относится ОС UNIX к разряду "прочих пользователей" и получает или не получает доступ к файлу в зависимости от значений бит кода защиты файла, определяющих права прочих пользователей на осуществление того или иного типа доступа к файлу.

Приведем теперь более строгое определение типов доступа к файлам ОС UNIX. Итак, файловая система ОС UNIX различает три типа доступа к файлам: ***чтение из файла***, ***запись в файл*** и ***выполнение файла***.

ДОСТУП ПО ЧТЕНИЮ

Разрешение на доступ к файлу по чтению означает, что операция ввода данных из файла правомочна. Если файл, к которому разрешен доступ

по чтению, является каталогом, то наличие такого разрешения означает возможность осуществления операции вывода на стандартный вывод содержимого каталога.

ДОСТУП ПО ЗАПИСИ

Разрешение на доступ к файлу по записи означает, что операция вывода данных в файл правомочна, т. е. пользователь имеет право на изменение содержимого файла, в том числе на изменение его размера. Если файл, к которому разрешен доступ по записи, является каталогом, то наличие такого разрешения означает возможность создания в этом каталоге новых файлов и удаления из него уже существующих (например, с помощью системных вызовов create, mknod и unlink).

ДОСТУП ПО ВЫПОЛНЕНИЮ

Разрешение на доступ к файлу по выполнению означает, что правомочна операция копирования в оперативную память и передача управления содержимому объектного файла. Если файл, к которому разрешен доступ по выполнению, является текстовым и содержит программу на языке shell, то наличие такого разрешения означает правомочность использования его в качестве входного файла интерпретатора команд shell[1].

Как Вы уже знаете, каталог не может, в принципе, содержать информацию, которая могла бы быть интерпретирована как программа, поэтому разрешение доступа по выполнению к каталогу имеет особый смысл. Итак, если файл, к которому разрешен доступ по выполнению, является каталогом, то наличие такого разрешения означает наличие разрешения на доступ к файлам, содержащимся в этом каталоге, и установку каталога в качестве текущего каталога. Например, если код защиты некоторого каталога такой, что доступ по выполнению к этому каталогу разрешен, а доступ по чтению запрещен, то пользователь не сможет получить списка имен содержащихся в каталоге файлов. Тем не менее, если имена этих файлов ему откуда-нибудь известны, то он сможет осуществить доступ к самим файлам.

Два старших бита кода защиты файла разрешают или запрещают переустановку идентификаторов владельца и группы при вызове файла на выполнение. Как Вы знаете, каждый процесс имеет действующий и реальный идентификаторы пользователя и группы (всего их четыре). Установка бит, разрешающих переустановку идентификаторов пользователя и группы, приводит к тому, что при вызове файла на выполнение значения действующих идентификаторов пользователя и группы для процесса, в рамках которого будет выполняться содержащаяся в нем программа, будут установлены равными идентификатору владельца и идентификатору группы указанного файла.

Основное назначение бит разрешения переустановки идентификато-

---

[1] Для файлов, содержащих текст программы на языке shell, необходимо иметь разрешение на доступ по чтению.

ров пользователя и группы заключается в предоставлении пользователю, вызвавшему на выполнение некоторый файл, полномочий владельца файла на время его выполнения. Классическим примером такой ситуации может служить процедура смены пароля рядовым пользователем ОС UNIX. Информация о паролях хранится в файле с именем /etc/passwd в закодированном виде. Владельцем файла /etc/passewd является привилегированный пользователь. Доступ по записи к этому файлу разрешен только для его владельца, все прочие пользователи имеют разрешение на доступ к этому файлу только по чтению. Команда смены пароля passwd хранится в файле с именем /bin/passwd. Возможность выполнения команды passwd в режиме привилегированного пользователя обеспечивается тем, что владельцем файла с именем /bin/passwd является привилегированный пользователь и в описателе этого файла в коде защиты файла бит, разрешающий переустановку идентификатора пользователя, установлен в 1. Это приводит к тому, что в результате вызова команды passwd на выполнение действующим идентификатором пользователя того процесса, в рамках которого выполняется команда passwd, становится идентификатор владельца файла с именем bin/passwd, т. е. идентификатор привилегированного пользователя. Иначе говоря, на время выполнения команды passwd рядовой пользователь ОС UNIX становится как бы привилегированным пользователем. Аналогично может быть использован бит, разрешающий переустановку идентификатора группы владельца файла.

ТЕХНИКА ПРОГРАММИРОВАНИЯ

Рассмотрим теперь несколько приемов осуществления доступа к файлам файловой системы ОС UNIX, а для наглядности изложения изучим их на примере трех небольших программ на языке Си.

ДОСТУП К КАТАЛОГАМ

Очевидно, что для получения информации о логической структуре файловой системы ОС UNIX необходимо иметь разрешение на доступ по чтению к ее каталогам. Обычно функционирующая файловая система разрешает такой доступ всем пользователям ОС UNIX (доступ к каталогам по записи файловой системой ОС UNIX ограничивается), т. е. для решения такой задачи необязательно быть привилегированным пользователем ОС UNIX. Итак, получив доступ к каталогу по чтению, пользователь (или пользовательский процесс), в рамках которого этот доступ осуществляется, может считать из него имена содержащихся в нем файлов и номера соответствующих им описателей файла. Как Вы уже знаете, по номеру описателя файла можно получить доступ к самому описателю файла, который хранится в известном месте на магнитном диске. На практике необходимость получения доступа к описателю файла возникает очень часто, поэтому ОС UNIX предоставляет пользователям специальный системный вызов stat, с помощью которого такая задача решается

весьма просто. Подробно этот системный вызов будет описан в следующей главе.

ПРИМЕР 1. РЕКУРСИВНЫЙ СПУСК ПО ДЕРЕВУ КАТАЛОГОВ

Программа, исходный текст которой приведен ниже, обрабатывает один входной параметр, в качестве которого может быть использовано полное имя файла. Начиная с элемента иерархической файловой структуры, которому соответствует выбранный файл, программа осуществляет рекурсивный спуск по дереву файловой системы до ее конечных узлов, выводя на свой стандартный вывод имена всех узлов структуры, лежащих на получаемом таким образом пути по дереву файловой структуры.

Прежде чем привести текст программы на языке Си, сделаем ряд важных замечаний.

Системный вызов stat используется рассматриваемой программой для определения типа обрабатываемого файла. Если обрабатываемый файл является каталогом, программа осуществляет вызов на выполнение подпрограммы rec(), которая реализует очередную итерацию рекурсивного спуска.

Элементы каталога, содержащие номера описателей, равные 0, игнорируются, поскольку им не соответствуют никакие файлы.

При копировании в оперативную память очередного имени очередного файла программа осуществляет динамическое резервирование оперативной памяти с помощью системного вызова malloc. Такое решение представляется нам наиболее эффективным, поскольку необходимый объем оперативной памяти заранее не известен.

В ОС UNIX существует ограничение на число одновременно открытых программой файлов. Так как вызов на выполнение подпрограммы rec осуществляется рекурсивно, очень скоро может наступить момент, когда будет достигнуто упомянутое ограничение. Для решения этой проблемы был применен следующий прием: пусть обрабатываемый программой в данный момент каталог содержит еще один каталог файловой системы, для которого первый является родительским каталогом, тогда для его обработки необходимо осуществить рекурсивный вызов на выполнение подпрограммы rec (вызов второго уровня вложенности), что приведет к открытию второго каталога. В таком случае программа закрывает открытый ею родительский каталог, и только после этого осуществляет вызов самой себя на выполнение (рекурсивный вызов). Остается добавить, что после выполнения вызова второго уровня вложенности, подпрограмма rec вновь открывает родительский каталог.

Подпрограмма strlen — одна из стандартных подпрограмм, входящих в состав стандарной библиотеки ввода-вывода ОС UNIX (подробнее см. гл. 5). Возвращаемым значением функции strlen является целое число, равное числу байт, содержащихся в строке текста, переданной функции strlen в качестве параметра.

В описываемой нами программе используется также системный вызов

lseek, с помощью которого осуществляется считывание отдельных полей элементов каталога:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dir.h>
char *malloc();
/*
 * rec <dirname>
 */
main (argc, argv)
int argc;
char *argv[];
{
  if (argc != 2){
     fprintf(stderr,"usage: %s <pathname>\n",argv[0]);
     exit(1);
  }
  printf("%s\n", argv[1]);

  /*
   * Рекурсивный спуск по дереву каталогов и вывод
   * полного имени текущего каталога
   */
   rec(argv[1]);
}

/*
 * Эта функция выполняет основную работу.
 */
  rec(dirpath)
  char *dirpath;
  {
     struct stat stbuf;
     struct direct drbuf;
     int i, fd;
     char leaf[DIRSIZ+1], *newpath;
     off_t drsize;
     if( stat(dirpath, &stbuf) < 0 )
     return(-1);

 /* не является ли <dirpath> каталогом */
 if ((stbuf.st_mode & S_IFMT) != S_IFDIR )
 return(-1);

 /* открыть каталог */
 if((fd = open(dirpath, 0) < 0){
    fprintf("error in reading %s\n",
         dirpath);
    return(-1);
 }

 /* размер каталога в байтах */
 drsize = stbuf.st_size;
```

```
for( i=0;i<(drsize/sizeof(struct direct));i++){
    if((read(fd, &drbuf, sizeof(struct direct))
        != sizeof(struct direct)) {
        fprintf(stderr,"error in reading %s\n",
            dirpath);
        return(-1);
    }

if(drbuf.d_ino){
     /*
      * вывести полное имя
      */
     strncpy(leaf, drbuf.d_name,  DIRSIZ);
     newpath = malloc(strlen(dirpath) +
             strlen(leaf) + 1);

     if(strcmp(dirpath, "/") == 0)
          sprintf(newpath, "%s%s",
               dirpath, leaf);
     else
          sprintf(newpath, "%s%s",
               dirpath, leaf);

    printf("%s\n", newpath);

    /* определить тип файла */
    if(stat(newpath, &stbuf) < 0)
        continue;

    /* проверить, не является ли тот файл каталогом */
    if((stbuf.st_mode & S_IFMT) == S_IFDIR) {
        if(strcmp(leaf, ".") == 0 ||
            strcmp(leaf,"..") == 0)
            continue;

    /* закрыть текущий каталог */
    close(fd);
    /* выполнить рекурсивный вызов */
    rec(newpath);
    /* повторно открыть каталог */
    fd = open(dirpath, 0);
    lseek(fd, (long)((i + 1) *
         sizeof( struct direct)),0);
          }
          /* освободить занятую оперативную память */
          free(newpath);
       }
     }
     close(fd);
     return(0);
}
```

ПРИМЕР 2. ОПРЕДЕЛЕНИЕ АБСОЛЮТНОГО ПОЛНОГО ИМЕНИ ФАЙЛА

После того как пользователь осуществил вход в ОС UNIX, текущим станет так называемый основной каталог пользователя. Абсолютное полное имя основного каталога пользователя самому пользователю часто неизвестно. Тем не менее, знать это имя (владельцем которого являешься) бывает просто необходимо. Описанная здесь задача определения полного имени файла, находящегося в текущем каталоге пользователя, с успехом решается программой, текст которой на языке Си приведен ниже.

Алгоритм, положенный в основу этой программы, кратко можно описать следующим образом: прежде всего, относительное полное имя файла представляется в виде совокупности двух контекстов, второй из которых представляет собой последний слог этого имени, а первый — последовательность остальных слогов, порядок которых не нарушен. После этого в качестве текущего устанавливается каталог с именем, совпадающим с первым из вышеперечисленных контекстов. Известно, что элемент каталога, содержащий в поле имени контекст .., в поле ссылки содержит ссылку на родительский каталог, а элемент каталога, содержащий в поле имени контекст ., в поле ссылки содержит ссылку на текущий каталог. Программа осуществляет последовательный переход из очередного текущего каталога в родительский (который после этого станет текущим) до тех пор, пока текущим каталогом не станет корневой каталог. Установить последнее весьма просто — достаточно убедиться в том, что две вышеупомянутые ссылки совпадают, т. е. специфицируют один и тот же файл. На каждом шаге последовательного перехода из очередного текущего каталога в родительский, программа заполняет связанный список, элементы которого реализованы в виде структуры с шаблоном namseg. Как видно из описания этого шаблона, первое его поле представляет собой элемент каталога, а второе — указатель на структуру с тем же шаблоном namseg (следующий элемент связанного списка)[1]. Осуществив таким образом ряд итераций и установив в качестве текущего каталога корневой каталог, программа осуществляет обратную последовательность переходов по каталогам файловой системы, используя для этого только что составленный связанный список элементов каталога. В процессе обратного перемещения по дереву каталогов программа выводит на свой стандартный вывод имя файла, содержащееся в поле sd структуры namseg, являющейся текущим элементом связанного списка, отделяя при этом одно имя от другого символом /. После того как полученное таким обра-

---

[1] Подробнее: сначала с помощью системного вызова stat определяется номер описателя текущего каталога; затем осуществляется переход в родительский каталог, т. е. он устанавливается как текущий каталог; наконец, текущий каталог открывается как файл и в нем осуществляется поиск элемента каталога, содержащего в поле ссылки определенный выше номер описателя файла. Именно этот элемент каталога заносится в структуру namseg. Понятно, что имя, содержащееся в поле имени этого элемента каталога, является именем каталога, переход из которого был только что осуществлен. — *Прим. ред.*

зом имя будет дополнено еще одним слогом, являющимся именем файла, на стандартный вывод окажется выведенным абсолютное полное имя файла.

В заключение добавим, что в описываемой программе используются системные вызовы stat, malloc и chdir.

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
char *malloc();
char *strrchr();
/* char *strrchr(s,c);
 * char *s,c;
 * возвращает указатель на самый правый
 * символ c в строке s
 */

struct namseg {
     struct direct sd;
     struct namseg *sp;
};

main(argc, argv)
int argc;
char *argv[];
{  struct stat stbuf;
   struct direct drbuf;
   struct namseg *ls, *ln;
   char *lhs, *rhs;
   int fd, i;

   if(argc != 2){
     fprintf(stderr,"usage: %s <pathname>\n",argv[0]);
     exit(1);
   }
   rhs = strrchr(argv[1], '/');
   if(rhs){
      lhs = argv[1];
      *rhs = '';
      rhs = rhs+1;
   }
   else{
      lhs = '.';
      rhs = argv[1];
   }

   if(chdir(lhs) < 0 || stat(".", &stbuf) < 0){
      fprintf(stderr,"cannot read %s\n",lhs);
      exit(1);
   }
   ls = (struct namseg *)malloc(sizeof(struct namseg));
   ls->sd.d_ino = stbuf.st_ino;

   ls-> = sp = (struct namseg *)(0);
```

```
    do{
       if(chdir("..") < 0 || stat(".", &stbuf) < 0){
         fprintf(stderr,"cannot move up path\n");
         exit(1);
       }
       if((fd = open(".",0)) < 0))
           fprintf(stderr,
               "cannot open parent directory\n");
           exit(1);
       }
       for(i = 0; i < stbuf.st_size/sizeof(drbuf); i++){
          read(fd, &drbuf, sizeof(drbuf));
          if(drbuf.d_ino == ls->sd.d_ino){
             ls->sd = drbuf;
             break;
          }
       }
       close(fd);
       ln = (struct namseg *)malloc(sizeof(struct namseg));
       ln->sd.d_ino = stbuf.st_ino;
       ln->sp = ls;
       ls = ln;

       stat("..", &stbuf);
    }
    while(stbuf.st_ino != ls->sd.d_ino);

    for(ln = ls->sp; ; ln = ln->sp){
        char leaf[DIRSIZ+1];
        strncpy(leaf, ln->sd.d_name, DIRSIZ);
        leaf[DIRSIZ] = '\0';
        printf("/%s", leaf);
        if(ln->sp == (struct namseg *)(0) )
           break;
        }
        printf("/%s\n", rhs);
}
```

НЕПОСРЕДСТВЕННЫЙ ДОСТУП К ТОМУ ФАЙЛОВОЙ СИСТЕМЫ

Как мы уже упоминали, открыв специальный файл и осуществив операцию ввода-вывода, можно получить непосредственный доступ к физическому устройству, которому он соответствует. Если таким физическим устройством является накопитель на магнитном диске, то может быть осуществлен непосредственный доступ к тому файловой системы ОС UNIX, когда последний установлен на указанный накопитель на магнитном диске. Такой способ доступа к тому файловой системы может оказаться необходимым, если пользователю надо получить информацию об описателях файлов или о списке свободных блоков конкретного тома файловой системы ОС UNIX. Непосредственный доступ файловой системы используется и некоторыми служебными программами ОС UNIX, осуществляющими проверку и, быть может, восстановление целостности файловой системы ОС UNIX, например fsck, icheck.

ПРИМЕР 3. ОЦЕНКА СТЕПЕНИ СЕГМЕНТИРОВАННОСТИ ТОМА ФАЙЛОВОЙ СИСТЕМЫ

При создании пустой файловой системы список свободных блоков перечисляет свободные блоки диска в таком порядке, чтобы доступ к вновь создаваемым и потому использующим блоки из указанного списка файлам был наиболее эффективным. Критерием эффективности доступа и соответственно оптимальности порядка перечисления свободных блоков в списке при этом служит среднее время доступа к файлу. Уменьшение указанного среднего времени доступа достигается выбором такого взаимного расположения составляющих файл блоков диска на физической поверхности носителя информации, чтобы за один оборот магнитного диска можно было бы считать или записать возможно большее их число[1]. Однако оптимальным список свободных блоков остается недолго, так как блоки диска, ставшие свободными в результате удаления файла или уменьшения его размера, заносятся в список свободных блоков всегда перед его первым элементом. В результате эффективность доступа к вновь созданным файлам снижается, что и фиксируется повышением степени их сегментированности по сравнению с ее оптимальным значением.

Программа, текст которой на языке Си приведен ниже, была разработана для определения степени сегментированности обычного файла. Если степень сегментированности значительного числа файлов файловой системы ОС UNIX становится слишком высокой, эффективность файловой системы падает настолько низко, что возникает необходимость осуществить так называемое "уплотнение" тома файловой системы.

Итак, описываемая программа представляет собой совокупность четырех функций, каждая из которых может быть использована в дальнейшем для решения других задач. Прежде всего, функция whichdev в результате выполнения возвращает указатель на строку, содержащую полное имя специального файла, соответствующего физическому устройству, номер которого указан в качестве аргумента функции. Определить номер физического устройства можно с помощью системного вызова stat, использовав в качестве его первого аргумента полное имя обычного файла, степень сегментированности которого нас интересует.

```
#include <sys/param.h>
#include <sys/ino.h>
#include <sys/stat.h>
#include <sys/dir.h>
```

[1] Как правило, оптимальность выбранного взаимного расположения определяется одной константой — числом секторов диска, разделяющих два последовательных блока файла. Например, два таких блока файла на диске могут занимать два сектора диска подряд или последовательные сектора диска через один или через два сектора. Значение этой константы определяется с одной стороны характеристикой аппаратуры, а с другой — стратегией хранения, реализуемой файловой системой, под управлением которой работает ЭВМ. Именно эту константу имеют в виду, говоря об interleave factor. — *Прим. ред.*

```
char *malloc();
/*
 * Определяет, какой из блокоориентированных
 * специальных файлов соответствует номеру
 * устройства <dev>. Предполагается, что
 * все специальные файлы находятся в
 * каталоге с именем /dev
 */

char *whichdev(dev);

dev_t dev;
{
   struct stat   s;
   struct direct d;
   int i, fd;
   off_t dsize;
   char *devname;

   if(stat("/dev", &s) < 0){
     fprintf(stderr, "cannot stat /dev\n");
     exit(1);
   }
   dsize = s.st_size;
   if((fd = open("/dev", 0)) < 0){
       fprintf(stderr,"cannot read /dev\n");
       exit(1);
   }


   for(i = 0; i < dsize/sizeof(struct direct); i++){
       char leaf[DIRSIZ+1];
       leaf[DIRSIZ] = '\0';
       read(fd, &d, sizeof(struct direct));
       if(d.d_ino){
          strncpy(leaf, d.d_name, DIRSIZ);
          devname = malloc(sizeof("/dev") +
                  strlen(leaf) + 1);
          sprintf(devname, "/dev/%s", leaf);
          if(stat(devname, &s) < 0){
              fprintf(stderr,"cannot stat %s\n",
                  devname);
              exit(1);
          }
          if(((s.st_mode&S_IFMT) == S_IFBLK) &&
              (s.st_rdev == dev)){
              close(fd);
              return(devname);
          }
          free(devname);
       }
    }
    close(fd);
    return(0);
  }
```

Функция pmap в результате выполнения возвращает номер физического блока диска, соответствующего логическому блоку файла с номером ld. При этом предполагается, что fd — это пользовательский дескриптор специального блок-ориентированного файла, соответствующего физическому устройству, на которое установлен том файловой системы, содержащий интересующий Вас файл, а ip — указатель на структуру, имеющую шаблон dinode и содержащую описатель этого файла[1].

```
/* число непосредственно адресуемых блоков      */
#define DIR0 10
/* число первичных блоков косвенности           */
#define DIR1 (10+128)
/* число вторичных блоков косвенности           */
#define DIR2 (10L+128+128*128)
/* число третичных блоков косвенности           */
#define DIR3 (10L+128+128*128+128*128*128)
/* число указателей местоположения              */
#define NAPI 13
/* индекс косвенности                           */
#define NAPB 128
/* индекс указателя, являющегося указателем
   местоположения  первичных блоков косвенности */
#define IX1  10
/* индекс указателя, являющегося указателем
   местоположения  вторичных блоков косвенности */
#define IX2  11
/* индекс  указателя, являющегося указателем
   местоположения  третичных блоков косвенности */
#define IX3  12

int pmap(fd, ip,lb);
int fd, lb;
struct dinode *ip;
{
   long di_map[NAPI];
   long dd_map[NAPB];
   l3tol(di_map, ip->di_addr, NAPI);

   if( lb < DIR0 )
     return(di_map[lb]);

   if( lb < DIR1 ){
     lb -= DIR0;
     lseek(fd, di_map[IX1]*BSIZE, 0);
     read(fd, dd_map, BSIZE);
     return(dd_map[lb%NAPB]);
   }
```

[1] Пользовательский дескриптор файла — это получаемое пользователем в результате открытия файла число, которое впоследствии будет использоваться в операциях чтения, записи и позиционирования вместо имени файла. — *Прим. ред.*

```
    if( lb < DIR2 ){
      lb -= DIR1;
      lseek(fd, di_map[IX2]*BSIZE, 0);
      read(fd, dd_map, BSIZE);

      lseek(fd, dd_map[lb/NAPB]*BSIZE, 0);
      read(fd, dd_map, BSIZE);
      return(dd_map[lb%NAPB]);


    }

    if( lb < DIR3 ){
      lb -= DIR2;
      lseek(fd, di_map[IX3]*BSIZE, 0);
      read(fd, dd_map, BSIZE);

      lseek(fd, dd_map[lb/(NAPB*NAPB)]*BSIZE, 0);
      read(fd, dd_map, BSIZE);

      lseek(fd, dd_map[lb%(NAPB*NAPB)]*BSIZE, 0);
      read(fd, dd_map, BSIZE);
      return(dd_map[lb%NAPB]);
    }
}
```

Функция getfrag вычисляет значения степени сегментированности файла. Она имеет два аргумента: первый из них представляет собой указатель на строку символов и используется для задания имени специального блок-ориентированного файла, а второй – переменную типа ino_t, значением которой должен быть номер описателя файла. Значение степени сегментированности файла вычисляется по формуле вида

$$f = \frac{\sum_{j=2}^{N} p(j) - p(j-1)}{N},$$

где p(j) – номер физического блока диска, соответствующего логическому блоку файла.

Сделаем два замечания, не имеющие отношения к описываемой программе

1. Описатели файла нумеруются, начиная с номера 1. Программисты (в том числе и авторы этой книги) обычно забывают об этом, и часто при считывании из специального файла, соответствующего магнитному диску, описателей файлов безуспешно пытаются обнаружить описатель файла с номером 0. Напоминаем, что описатели файлов тома файловой системы находятся в последовательных блоках диска, начиная со второго физического блока диска.

2. Если рассматриваемый файл пуст, т. е. не содержит ни одного байта информации, то функция getfrag не должна производить никаких вычислений, а возвращаемым значением должно быть число 0.

```
double getfrag(dev, ino);
char *dev;
ino_t ino;
{
   struct dinode db;
   int fd, i, nb, ip, op;
   double ifrag;
   if(( fd = open(dev, 0)) < 0){
      fprintf(stderr, "cannot open %s\n", dev);
      exit(1);
   }
   lseek(fd, ((2+((ino-1)/INOPB))*BSIZE)
       +(sizeof(struct dinode)*((ino-1)%INOPB)),0);
   if(read(fd, &db, sizeof(struct dinode)) !=
       sizeof(struct dinode)){
       fprintf(stderr,"cannot read %s\n", dev);
       exit(1);
   }
   /* вычислть размер файла */
   nb = (db.di_size + BSIZE - 1)/BSIZE;
   /* игнорировать пустой файл */
   if(nb == 0)
      return(0.0);
   /* вычислить степень фрагментации непустого файла */
   for(op = pmap(fd, &db, 0),i = 1; i < nb; i++){
      ip = pmap(fd, &db, i);
      ifrag += (double)(ip - op);
      op = ip;
   }
   close(fd);
   return(ifrag/(double)nb);
}
```

И наконец, функция main осуществляет вызов на выполнение всех описанных выше функций и выводит на свой стандартный вывод действительное число, полученное в результате всех вычислений.

```
main(argc, argv)
int argc;
char *argv[];
{
   struct stat ss;
   char *dn;

   if(argc != 2|| stat(argv[1], &ss) < 0){
     fprintf(stderr, "usage: %s <pathname>\n", argv[0]);
     exit(1);
   }

   dn = whichdev(ss.st_dev);

   if(dn == (char *)(0)){
      printf("cannot stat special file \n");
      exit(1);
   }
```

```
else{
    printf("%8.3f\n", getfrag(dn. ss.st_ino));
}
```

Если Вы теперь, получив объектный код программы, вызовете ее на выполнение, задав в качестве входного параметра имя, например интерпретатора команд shell, то результат, скорее всего, будет близок к оптимальному значению для данного типа физического устройства. В противном случае Вы напрасно тратите время, пользуясь столь малоэффективной файловой системой.

Приведем несколько примеров степеней сегментированности наиболее часто используемых команд ОС UNIX нашей операционной системы:

```
/bin/sh    5.6
/bin/ls    4.5
/bin/cat  167.8
```

Как можно видеть из этого примера, степень сегментированности файла cat далека от оптимального значения.

ЗАДАЧИ

1. Разработайте программу на языке Си, подсчитывающую число свободных описателей файла и блоков диска неподключенного тома файловой системы.

2. Модифицируйте приведенную выше программу определения степени сегментированности обычного файла так, чтобы с ее помощью можно было определить степень сегментированности всей файловой системы ОС UNIX в целом, а также самого списка свободных блоков, рассматриваемого как файл, элементами которого являются элементы списка свободных блоков диска.

ЛИТЕРАТУРА

1. A.V. Aho, J.E. Hopcroft & J.D. Ullman (1983), Data Structures and Algorithms, Addison-Wesley.

2. T.J. Kowalski (1980), FSCK - The UNIX/TS File System Check Program, UNIX Programmers Manual (System III), Volume 2b.

3. E.I. Organick (1972), The Multics System - An Examination of Its Structure, M.I.T. Press.

4. K. Thompson (1978), UNIX Implementation, UNIX Programmers Manual (seventh edition), Volume 2b.

Глава 4. ПРОГРАММИРОВАНИЕ ОПЕРАЦИЙ ВВОДА-ВЫВОДА

В предыдущей главе мы более или менее подробно описали файлы и файловую систему ОС UNIX. Эту главу мы посвящаем вопросам программирования операций ввода-вывода.

Прежде всего, попытаемся расширить введенное понятие файла ОС UNIX как набора данных, структура которого однозначно описывается описателем файла. Особое внимание при этом будет уделено реализации системных вызовов ОС UNIX, применяемых для осуществления операций ввода-вывода, а также вопросам эффективности использования указанных системных вызовов при решении различных прикладных задач.

СТРУКТУРА СИСТЕМЫ УПРАВЛЕНИЯ ВВОДОМ-ВЫВОДОМ ОС UNIX

Основным назначением системы управления вводом-выводом ОС UNIX является создание интерфейса между программой и внешним устройством ЭВМ. Строго говоря, система управления вводом-выводом ОС UNIX осуществляет отображение файловой системы ОС UNIX на множество функций, реализуемых внешними устройствами ЭВМ. С функциональной точки зрения указанное отображение может быть представлено как суперпозиция четырех отображений, иначе говоря, система управления вводом-выводом ОС UNIX имеет четыре уровня управления. Каждый из них реализует вполне определенные функции, связь между двумя последова-

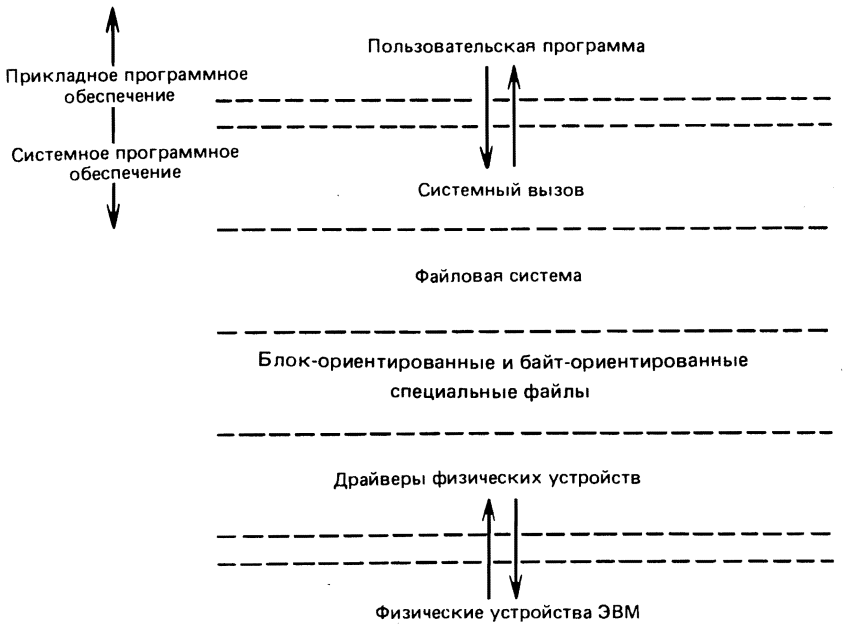

Рис. 4. 1. Структура системы управления вводом-выводом ОС UNIX

тельными уровнями управления осуществляется средствами программного интерфейса. На рис. 4.1 схематически изображена функциональная структура системы управления вводом-выводом ОС UNIX.

Детальное описание двух нижних уровней управления не входит в наши планы, и читатель не найдет его в этой книге, а иметь общее представление о наборе функций, реализуемых этими уровнями управления и обрабатываемых ими структурах данных, просто необходимо.

СИСТЕМНЫЕ ВЫЗОВЫ

Как Вы уже знаете, системные вызовы представляют собой единственное средство, реализующее интерфейс между пользовательской программой и ядром ОС UNIX. Поскольку всякая операция ввода-вывода для пользователя — это операция ввода-вывода в файл, то ему достаточно знать и понимать только те системные вызовы, которые реализуют указанные операции обмена с файловой системой ОС UNIX. Самыми популярными из таких системных вызовов являются вызовы open, close, read и write.

СИСТЕМНЫЕ ВЫЗОВЫ OPEN И CLOSE

Прежде чем получить доступ к файлу, его необходимо открыть. В программе на языке Си это можно осуществить, например, с помощью оператора

```
fd = open(pathname, oflag);
```

Системный вызов open возвращает небольшое положительное число, так называемый пользовательский дескриптор файла, который в дальнейшем можно использовать для спецификации открытого таким образом файла при осуществлении операций обмена с ним. Аргумент pathname — это указатель на строку символов, содержащую полное имя открываемого файла; аргумент oflag специфицирует режим открытия файла (открывается ли файл для чтения, записи и т. д. Если файл с указанным именем не существует или же данному пользователю не разрешен доступ требуемого им типа, то функция open () возвращает значение, равное –1. Очень важно помнить, что любой файл ОС UNIX может быть открыт одновременно несколькими пользователями при условии, что этот файл существует, а пользователи имеют право на соответствующий доступ к нему.

Системный вызов close осуществляет действия, прямо противоположные только что описанным: он уничтожает связь между пользовательским дескриптором файла, возвращенным системным вызовом open и файлом. Например,

```
close(fd);
```

Пользовательский дескриптор файла, который был значением переменной fd в приведенном выше примере, может быть вновь получен в качестве возвращаемого значения системного вызова open после очередного его использования и специфицировать очередной открытый файл.

СИСТЕМНЫЕ ВЫЗОВЫ READ И WRITE

Системные вызовы read и write реализуют большинство функций обмена и могут быть осуществлены с помощью следующих операторов языка Си:

```
nr = read(fd, buffer, count);
nw = write(fd, buffer, count);
```

В результате осуществления этих системных вызовов данные считываются из или записываются в файл, специфицированный пользовательским дескриптором файла fd. Понятно, что указанный пользовательский дескриптор файла fd был предварительно получен в качестве возвращаемого значения системного вызова open. Аргумент buffer в обоих случаях представляет собой указатель на символы, используемый для хранения данных, вводимых из файла или выводимых в файл, специфицированный пользовательским дескриптором файла fd. И наконец, последний аргумент — count представляет собой переменную типа int; ее значение специфицирует число байт информации, которое необходимо ввести из файла или вывести в файл. Оба вызова возвращают целое число. Если это число больше 0, то оно равно количеству реально введенных или выведенных байт информации; если возвращаемое значение оказывается равным 0, то это означает, что достигнут логический конец файла; если же возвращаемое значение представляет собой отрицательное число, значит, в процессе выполнения операции обмена была обнаружена ошибка.

Итак, все операции ввода-вывода реализуются ОС UNIX как операции ввода-вывода информации в файл средствами программного интерфейса с файловой системой ОС UNIX, предоставляемого системой управления вводом-выводом. Программный интерфейс с файловой системой ОС UNIX реализуется сравнительно небольшим набором системных вызовов. В результате осуществления одного из этих системных вызовов его имя и аргументы обрабатываются программными средствами верхнего уровня системы управления вводом-выводом, после чего использованному системному вызову ставится в соответствие последовательность системных вызовов, реализуемых программными средствами следующего уровня системы управления вводом-выводом. Именно этим вторым уровнем системы реализуется иерархическая структура файловой системы ОС UNIX, описанная в предыдущей главе.

УПРАВЛЕНИЕ ФАЙЛАМИ

Поскольку любая операция ввода-вывода осуществляется как операция ввода-вывода информации в файл, то логическая структура программного интерфейса, реализуемого системой управления вводом-выводом, не зависит ни от типа данных, ни от типа внешнего устройства ЭВМ.

При осуществлении операции ввода-вывода в файл, специфицированный пользовательским дескриптором файла, ОС UNIX ставит в соответствие использованному системному вызову последовательность про-

граммных запросов к аппаратуре ЭВМ с помощью целого ряда связанных наборов данных, структура которых поддерживается самой ОС UNIX, ее файловой системой и системой управления вводом-выводом. Основным из упомянутых наборов данных можно считать *таблицу описателей файлов*.

ТАБЛИЦА ОПИСАТЕЛЕЙ ФАЙЛОВ

Таблица описателей файлов представляет собой хранящуюся в оперативной памяти ЭВМ структуру данных, элементами которой являются копии описателей файлов, по одной на каждый файл ОС UNIX, к которому была осуществлена попытка доступа. При выполнении операции открытия файла в ОС UNIX сначала по полному имени файла определяется элемент каталога, где в поле имени файла содержится имя файла, для которого производится операция открытия файла. В найденном элементе каталога из поля ссылки извлекается порядковый номер описателя файла. Затем описатель файла с соответствующим номером копируется со съемного пакета дисков в оперативную память, в ее область, называемую таблицей описателей файлов (если до этого он там отсутствовал).

Отметим одно чрезвычайно важное обстоятельство: все изменения содержимого и размера файла, открытого пользователем, регистрируются файловой системой ОС UNIX в описателе файла, точнее, в его копии, хранящейся в таблице описателей файлов в оперативной памяти ЭВМ. В результате выполнения операции закрытия файла эта копия описателя файла, содержащая, таким образом, самую достоверную информацию о файле, будет перемещена из таблицы описателей файлов на съемный пакет диска.

ТАБЛИЦА ФАЙЛОВ

С таблицей описателей файлов тесно связана другая структура данных, называемая таблицей файлов. Каждому элементу таблицы описателей файлов обязательно соответствует один или несколько элементов таблицы файлов. Каждый элемент таблицы файлов содержит информацию о режиме открытия файла, специфицированном при открытии файла, а также информацию о положении указателя чтения-записи. Итак, при каждом открытии файла в таблице файлов появляется новый элемент.

Известно, что один и тот же файл ОС UNIX может быть открыт несколькими не связанными друг с другом процессами, при этом ему будет соответствовать один элемент таблицы описателей файлов и столько элементов таблицы файлов, сколько раз этот файл был открыт. Это последнее обстоятельство является известной гарантией целостности файловой системы ОС UNIX. Действительно, поскольку каждому открытому файлу соответствует только один элемент таблицы описателей файлов, то содержащаяся в нем информация о файле всегда соответствует действительным характеристикам файла и не зависит от того, какой конкретно процесс осуществляет операцию ввода-вывода информации в указанный файл.

Таким образом, мы показали, что информация, содержащаяся в таблице описателей файлов, доступна любому процессу, осуществившему открытие файлов ОС UNIX, открытых ранее другими процессами, не являющимися по отношению к данному ни процессами-потомками, ни процессами-предками.

Что касается таблицы файлов, то, как мы уже говорили, каждой операции открытия файла ставится в соответствие один новый элемент таблицы файлов. Однако из этого правила есть одно исключение: оно касается того случая, когда файл, открытый процессом-предком, открывается процессом-потомком, порожденным с помощью системного вызова fork. При возникновении такой ситуации операции открытия файла, осуществленной процессом-потомком, будет поставлен в соответствие тот из существующих элементов таблицы файлов (в том числе положение указателя чтения-записи), который в свое время был поставлен в соответствие операции открытия этого файла, осуществленной процессом-предком. Иначе говоря, процесс-потомок "наследует" файлы, открытые всеми его процессами-предками вместе с соответствующими указателями чтения-записи. В то же время, если два процесса, осуществившие операцию открытия одного и того же файла, не связаны "родственными отношениями", то каждый из них обрабатывает только тот указатель чтения-записи, который входит в состав элемента таблицы файлов, созданного в результате открытия файла именно этим процессом.

ТАБЛИЦА ОТКРЫТЫХ ФАЙЛОВ ПРОЦЕССА

Продолжая начатые рассуждения, опишем, наконец, третий набор данных, называемый таблицей открытых файлов процесса. Каждому процессу ОС UNIX сразу после его порождения ставится в соответствие таблица открытых файлов процесса. Если указанный процесс порождает, в свою очередь, новый процесс, например с помощью системного вызова fork, то процессу-потомку ставится в соответствие таблица открытых файлов процесса, которая в первый момент функционирования процесса-потомка представляет собой копию таблицы открытых файлов процесса-предка. Таким образом, оба этих процесса (и процесс-предок, и процесс-потомок) получают доступ одного типа к одним и тем же файлам, открытым в одном и том же режиме.

С помощью таблицы открытых файлов процесса становится возможным осуществить отображение множества пользовательских дескрипторов файла, открытых всеми процессами ОС UNIX, на множество элементов таблицы файлов. Это отображение является первым звеном в цепи последовательных преобразований пользовательского дескриптора файла ОС UNIX в связанный список составляющих этот файл блоков диска. Итак, каждый элемент таблицы открытых файлов процесса содержит указатель местоположения соответствующего элемента таблицы файлов. Если далее пользовательский дескриптор файла использовать для индексации элементов таблицы открытых файлов процесса, то логическую структуру алгоритма, с помощью которого осуществляется указанное

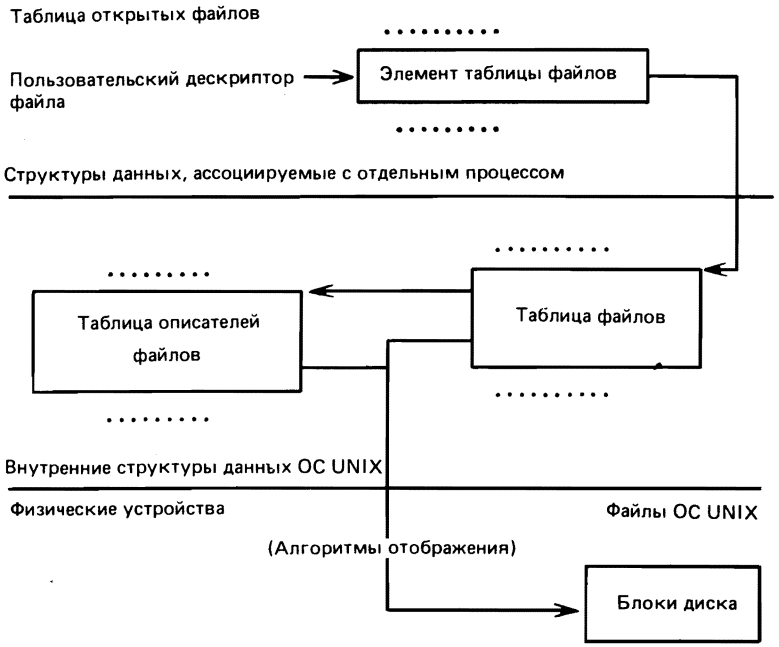

Рис. 4. 2. Логическая схема системы управления файлами

отображение, можно представить в виде логической схемы, изображенной на рис. 4. 2.

На рисунке хорошо видна взаимосвязь всех перечисленных наборов данных: таблицы описателей файлов, таблицы файлов и таблицы открытых файлов процесса. В результате осуществления пользовательским процессом системного вызова open последний получает доступ ко всем трем наборам данных; в результате осуществления системного вызова close указанный доступ прекращается.

ОПЕРАЦИЯ ВВОДА-ВЫВОДА, ЕЕ ЛОГИЧЕСКАЯ И ФУНКЦИОНАЛЬНАЯ СТРУКТУРА

После того как мы описали структуру и назначение трех перечисленных выше наборов данных, рассмотрим механизм их использования при осуществлении операций ввода-вывода с помощью системных вызовов read и write. Как вы уже знаете, все операции ввода-вывода в файл осуществляются с использованием таблицы описателей файла. Напомним: описатель файла содержит указатели местоположения блоков файла на диске, а таблица файлов содержит указатели чтения-записи всех открытых файлов; совокупность указанных данных позволяет однозначно отобразить синтаксические конструкции, специфицирующие системные вы-

зовы read и write на последовательность программных запросов к драйверам внешних устройств.

Если с помощью указанных системных вызовов осуществляется операция ввода-вывода информации в специальный файл, то в результате, очевидно, обработан описатель файла, соответствующий специфицированному специальному файлу; существенным в этом случае является то, что вместо указателей местоположения блоков файла на диске этот описатель файла будет содержать идентификатор класса и идентификатор устройства в классе физических устройств. Идентификатор класса физических устройств однозначно специфицирует драйвер устройства, который должен быть использован для осуществления операции ввода-вывода информации на физическое устройство, которому соответствует выбранный специальный файл.

БЛОКОВЫЕ И БАЙТОВЫЕ ОПЕРАЦИИ ВВОДА-ВЫВОДА

Операция ввода-вывода на физическое устройство может быть реализована двумя способами: в первом осуществляется байтовая (байт за байтом), а во втором — блоковая (блок за блоком) передача информации[1].

Подобное разделение обусловлено необходимостью оптимизировать операции обмена с физическими устройствами ЭВМ по времени их выполнения. Если физическое устройство ЭВМ имеет блочную структуру, то скорее всего эффективнее будет передача информации вторым способом; если это не так, то более эффективной окажется передача информации первым способом.

ИДЕНТИФИКАТОР КЛАССА ФИЗИЧЕСКИХ УСТРОЙСТВ И ИДЕНТИФИКАТОР УСТРОЙСТВА В КЛАССЕ

Каждому физическому устройству ЭВМ, поддерживаемому ОС UNIX, последняя ставит в соответствие уникальную совокупность двух чисел, называемых идентификатором класса физических устройств и идентификатором устройства в классе. Идентификатор класса физических устройств представляет собой неотрицательное целое число и идентифицирует драйвер устройства, т. е. программное средство ядра ОС UNIX, реализующее самый нижний уровень системы управления вводом-выводом и осуществляющее непосредственное управление физическим устройством ЭВМ.

Номер устройства в классе также представляет собой неотрицательное число, которое идентифицирует конкретное физическое устройство в классе устройств; как правило, это просто логический номер физического устройства. Иногда значение идентификатора устройства может содержать какую-либо другую полезную информацию, например если речь идет

---

[1] Блок (или логический блок) — последовательность байт, длина которой зависит от конкретной реализации ОС UNIX или аналогичной ей операционной системы. Например, в ОС UNIX версии 7, реализованной для ЭВМ PDP-11, логический блок содержит 512 байт информации, а в ОС UNIX BSD 2.9 для той же модели ЭВМ — уже 1024_{10} байта информации. — *Прим. ред.*

о накопителе на магнитной ленте, надо ли осуществить обратную перемотку ленты прежде чем начать выполнение специфицированной операции ввода-вывода.

МЕХАНИЗМ БЛОКОВЫХ ОПЕРАЦИЙ ВВОДА-ВЫВОДА

Блоковый способ передачи данных чрезвычайно эффективно может быть использован при осуществлении операций ввода-вывода информации на магнитный диск, так как физическая организация носителей информации, поддерживаемая аппаратурой ЭВМ, предполагает блоковую структуру хранящейся на них информации.

Логическая модель накопителя на магнитном диске может быть описана как конечное множество логических блоков, на котором введена операция выборки последовательности, составленной из произвольных логических блоков, перечисленных в произвольном порядке. Воспользуемся этим определением и выберем последовательность, составленную из всех возможных логических блоков, и перенумеруем их следующим образом: 0,1, . . . , N −1.

Говорят, что число N специфицирует емкость накопителя на магнитном диске, измеренную в логических блоках (чаще всего их несколько тысяч), каждый из которых содержит 512 байт информации. Функциональная поддержка определенной выше логической модели накопителя на магнитном диске реализуется программным средством ОС UNIX, называемым драйвером. Наиболее важной особенностью блоковых операций обмена с физическими устройствами ЭВМ является их буферизованность. Буферизованность блоковых операций ввода-вывода реализуется с помощью поддерживаемого ОС UNIX буферного пула (обычно в него входит 20 — 40 буферов), механизм управления которым аналогичен механизму управления кэшем. Буфер пула ассоциируется с драйверами физических устройств на время выполнения ими операций ввода-вывода по мере необходимости. Так, если выполняется операция ввода, то прежде чем произвести считывание очередного логического блока, ОС UNIX осуществляет проверку наличия его в буферах, составляющих буферный пул, и реальное считывание соответствующих блоков диска осуществляется лишь в случае отрицательного результата указанной проверки. Итак, если результат проверки оказался отрицательным, то с драйвером ассоциируется очередной, еще не использованный буфер; если такого буфера нет, то один из буферов, ассоциированных ранее с другими драйверами и не используемый в данный момент.

Использование подобного механизма буферизации операций ввода-вывода значительно повышает их эффективность за счет сокращения числа реальных обращений к физическим устройствам ЭВМ. Такой подход хорошо согласуется с интерпретацией информации (вводимой или выводимой на конкретное физическое устройство ЭВМ) как непрерывной, лишенной какой-либо внутренней структуры последовательности байт данных (иначе говоря потока данных). Однако при этом возникает и ряд проблем.

Самой серьезной из них является, пожалуй, проблема соответствия описателей файлов их копиям, хранящимся в оперативной памяти в таблице описателей файлов. Действительно, в результате выполнения операции ввода-вывода изменяется содрежимое только копии соответствующего описателя файла, являющейся элементом таблицы описателей файлов, сам же описатель файла, находящийся на магнитном диске, остается неизменным. Эта проблема разрешается с помощью системного вызова sync, осуществляющего принудительное обновление содержимого описателей файлов в соответствии с их копиями из таблицы описателей файлов, а также завершение всех операций ввода-вывода. Регулярное использование системного вызова sync позволяет сохранить целостность файловой системы ОС UNIX, но не решает проблемы в целом, так как в случае аварийного останова ЭВМ соответствие элементов таблицы описателей файлов и самих описателей файлов, находящихся на магнитном диске, нарушается, и тем самым нарушается целостность файловой системы[1].

Впрочем, ОС UNIX предоставляет пользователям ряд программных средств, позволяющих так или иначе восстановить целостность файловой системы, например программы icheck, dcheck, fsck.

Вторая проблема, возникающая при выполнении буферизованной операции, заключается в невозможности программного анализа ряда ошибок, происходящих при попытке осуществить запись информации на магнитный диск. Дело в том, что возвращаемое значение системного вызова write указывает на успешное или ошибочное завершение операции копирования всех блоков файла в буферный пул ОС UNIX. Таким образом, ошибки, возникающие на этапе копирования содержимого соответствующих буферов пула в блоки диска, остаются неизвестны программе, осуществившей указанный системный вызов. Информация об ошибках такого рода выводится ОС UNIX лишь на консольный терминал.

И наконец, еще одна проблема, связанная с буферизованностью блоковых операций обмена. Пусть некая пользовательская программа осуществляет блоковую операцию обмена. Это означает, что выводимые программой данные сначала копируются в буферный пул ОС UNIX, заполняя некоторые из ее буферов, а затем в составе указанных буферов (сначала из них формируются логические блоки) копируются в блоки диска. При этом реализация второго этапа выполняется под управлением только системы управления вводом-выводом ОС UNIX, в результате чего порядок заполнения буферов пула и порядок копирования их на физическое устройство ЭВМ в составе логических блоков могут не совпадать. Из этого следует, что если физическое устройство, на которое осуществляется вывод информации, является устройством произвольного доступа (например, накопитель на магнитном диске), то целостность находящейся на нем файловой системы может быть нарушена на время вы-

---

[1] Как правило, осуществление системного вызова sync выполняется ОС UNIX автоматически, например каждые 30 с, для этого достаточно один раз вызвать на [illegible]ение программу /etc/[illegible] [illegible]етствующими параметрами.

полнения операции вывода. Если же выбранное физическое устройство ЭВМ является устройством последовательного доступа (т. е. является физическим устройством неблоковой структуры, например накопитель на магнитной ленте), то для того чтобы порядок выведенных на магнитную ленту байт информации совпадал с порядком их генерации пользовательской программой, необходимо принять специальные меры, что и выполняется ОС UNIX.

МЕХАНИЗМ БАЙТОВЫХ ОПЕРАЦИЙ ВВОДА-ВЫВОДА

Если блоковый способ передачи данных оказывается эффективным при осуществлении операций обмена с физическим устройством блоковой структуры, то эффективность использования байтового способа передачи данных оказывается наибольшей при осуществлении операций обмена с физическими устройствами неблоковой структуры. Примером физического устройства неблоковой структуры может быть любое устройство последовательного доступа. Вместе с тем выполнение байтовой операции обмена с физическим устройством блоковой структуры также правомочно, и в некоторых случаях оказывается даже более эффективным (например, при считывании с магнитного диска большого числа последовательных блоков диска). Итак, все физические устройства ЭВМ, обмен с которыми может осуществляться с помощью байтовых операций обмена, условно могут быть разделены на три группы. К первой из них относятся те физические устройства последовательного доступа (назовем их устройствами байтового обмена), ввод или вывод информации на которые осуществляется только с помощью байтовых операций обмена, например; любые устройства связи двух ЭВМ или ЭВМ и терминала, устройство ввода с перфоленты и алфавитно-цифровое печатающее устройство; ко второй могут быть отнесены накопители на магнитных дисках; к третьей — накопители на магнитной ленте. Осуществление доступа к накопителям на магнитных дисках или магнитных лентах с помощью байтовых операций обмена дает пользователю возможность осуществлять считывание с магнитной ленты произвольного числа последовательных байт информации за один обмен с накопителем, а также копировать один магнитный диск на другой дорожку за дорожкой.

ТИПИЧНЫЕ УСТРОЙСТВА БАЙТОВОГО ОБМЕНА

Функциональная блок-схема байтового обмена с любым из устройств, которые выше мы назвали устройствами байтового обмена, приведена на рис. 4.3. Итак, драйвер, осуществляющий непосредственную байтовую передачу данных между пользовательской программой и физическим устройством, как это хорошо видно из рисунка, имеет двухуровневую функциональную структуру. При этом верхний уровень функционирует синхронно с пользовательской программой, осуществляющей байтовую операцию обмена, а нижний — синхронно с аппаратурой физического устройства, используя для этого систему аппаратных прерываний. Обмен данными между уровнями функциональной структуры драйвера осуществляется

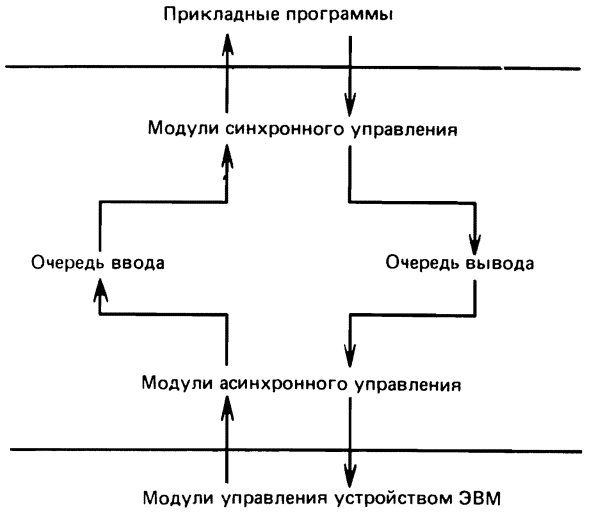

Рис. 4.3. Логическая схема реализации байт-ориентированного ввода-вывода

буферизованно, для чего используется такое хорошо известное программное средство, как очередь ввода-вывода, и в результате функционирование обоих уровней по отношению друг к другу осуществляется асинхронно.

С точки зрения структуры данных упомянутые очереди ввода-вывода представляют собой связанные списки байт, хранящиеся в буферном пуле. При этом оперативная память ЭВМ используется весьма эффективно, так как область буферного пула выделяется для хранения очередей ввода-вывода динамически. Например, типичный драйвер терминала создает по две очереди ввода-вывода для каждого функционирующего под управлением ОС UNIX терминала: одну — для буферизации ввода информации, другую — для буферизации вывода информации.

БАЙТОВЫЕ ОПЕРАЦИИ ВЫВОДА

При осуществлении пользовательской программой байтовой операции вывода данные, генерируемые программой, помещаются в очередь вывода драйвером соответствующего физического устройства той его программной компонентой, которая реализует верхний уровень функциональной структуры драйвера. После этого системный вызов write возвращает значение, специфицирующее результат завершения операции вывода. Непосредственный вывод этих данных на физическое устройство выполняется автоматически и асинхронно той программной компонентой драйвера, которая реализует нижний уровень его функциональной структуры.

БАЙТОВЫЕ ОПЕРАЦИИ ВВОДА

Данные, вводимые с физического устройства, асинхронно помещаются в очередь ввода драйвером соответствующего физического устройства, той его программной компонентой, которая реализует нижний уровень функциональной структуры драйвера. После этого данные копируются из очереди ввода в область оперативной памяти, специфицированную аргументами соответствующего системного вызова, который был осуществлен пользовательской программой для выполнения операции ввода. При этом из очереди вывода данные удаляются по мере их копирования в локальный буфер пользовательской программы той программной компонентой драйвера, которая реализует нижний уровень его функциональной структуры. Если в процессе выполнения операции ввода оказывается, что очередь ввода пуста, то пользовательская программа, выполняющая операцию ввода, может либо завершить операцию ввода, либо перейти в режим ожидания, пока не будут введены новые данные.

БАЙТОВЫЕ ОПЕРАЦИИ ВВОДА-ВЫВОДА НА УСТРОЙСТВА ПРЯМОГО ДОСТУПА

Рассматриваемые здесь байтовые операции обмена позволяют осуществить обмен данными между пользовательской программой и накопителем на магнитном диске и магнитной ленте с помощью прямого доступа к памяти, который реализуется аппаратурой ЭВМ. Необходимо заметить, что при осуществлении таких операций обмена вводимые и выводимые байты информации должны будут иметь на соответствующем физическом устройстве последовательные адреса[1]. В отличие от байтовых операций обмена при выполнении блоковых операций обмена очередной логический блок сначала копировался в буферный пул ОС UNIX и только потом на внешнее устройство или в локальный буфер пользовательской программы. На рис. 4.4 схематически изображена функциональная структура двух способов реализации операций обмена с физическим устройством прямого доступа. На практике любая байтовая операция обмена с физическими устройствами прямого доступа отображается ОС UNIX в последовательность блоковых операций обмена, но таких, что в процессе их выполнения данные перемещаются из локального буфера пользовательской программы непосредственно на физическое устройство (или наоборот), минуя буферный пул ОС UNIX. Необходимо отметить, что при такой реализации операции обмена снимаются все ограничения, накладываемые размером буферов буферного пула ОС UNIX на объем информации, вводимой или выводимой за одно обращение к физическому устройству.

И в заключении сделаем еще одно замечание. Существуют физические устройства ЭВМ, которые не могут быть отнесены ни к одной из перечисленных выше групп; например, существует драйвер, с помощью кото-

---

[1] "Последовательные" с точки зрения контроллера соответствующего физического устройства.

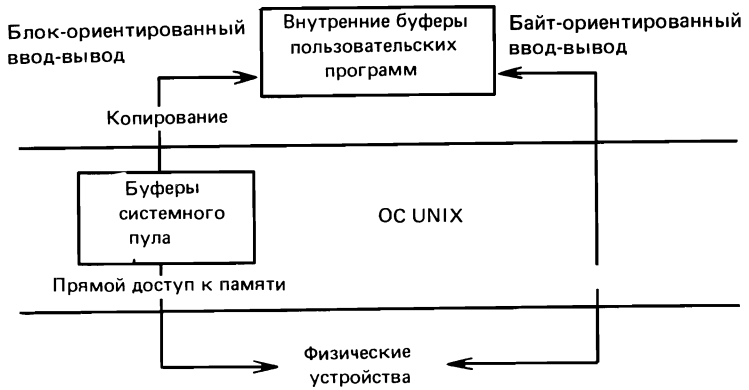

Рис. 4.4. Логическая схема реализации блок-ориентированного и байт-ориентированного ввода-вывода на физическое устройство, имеющее блоковую структуру

рого может быть получен доступ к той области оперативной памяти ЭВМ, в которой находится ядро ОС UNIX. Этот драйвер и соответственно это физическое устройство используются целым рядом инструментальных средств ОС UNIX (например, программой PS) для получения непосредственного доступа к наборам данных самого ядра ОС UNIX. Кроме описанных, существуют такие физические устройства, которые обрабатывают выведенную на них информацию чрезвычайно быстро, но являются устройствами последовательного доступа. Пример — фотонаборное устройство. Очевидно, что использование в этом случае байтовых операций вывода чрезвычайно неэффективно. В связи с этим, драйверы таких устройств осуществляют эмуляцию блокового способа передачи данных, используя для этого либо собственный локальный буфер, либо буфера пула ОС UNIX. На рис. 4.4 представлена логическая схема системы управления вводом-выводом.

ТЕХНИКА ПРОГРАММИРОВАНИЯ

Оставшуюся часть этой главы мы посвятим рассмотрению вопросов использования системных вызовов, реализующих операции ввода-вывода.

ОПЕРАЦИИ ВВОДА-ВЫВОДА НИЗКОГО УРОВНЯ

Одним из основных способов выполнения операций ввода-вывода в ОС UNIX является осуществление соответствующих системных вызовов. Как мы уже знаем, ОС UNIX предоставляет пользователям набор системных вызовов, реализующих операции ввода-вывода информации в файл. Хотя с точки зрения техники программирования непосредственное использование системных вызовов ОС UNIX не самый эффективный способ осуществления операций ввода-вывода, тем не менее, такой подход имеет свои, как нам кажется, существенные преимущества — нагляд-

ность процедуры выполнения операции ввода-вывода и высокая степень достоверности полученного результата[1].

СТАНДАРТНЫЙ ВВОД И СТАНДАРТНЫЙ ВЫВОД

Все мы привыкли к тому, что прежде чем выполнить операцию ввода-вывода информации в файл, последний необходимо открыть. В ОС UNIX это правило должно быть дополнено одним исключением — речь идет о файлах, называемых (согласно терминологии, принятой в ОС UNIX) стандартный ввод, стандартный вывод и стандартный протокол, автоматически открываемых для любого вновь порожденного процесса и имеющих пользовательские дескрипторы файлов 0, 1 и 2 соответственно. Добавим, что если не оговорено противное, то три этих дескриптора файлов ассоциированы с одним и тем же специальным файлом, соответствующим терминалу данного пользователя. Вы уже знаете, что переназначить стандартный ввод, стандартный вывод или стандартный протокол на любой другой файл весьма просто, вместе с тем разнесение этих трех потоков данных часто оказывается полезным[2].

Как мы уже говорили, обычно процессу нет необходимости открывать стандартный ввод, стандартный вывод и стандартный протокол. Дело в том, что все три файла были открыты его процессом-предком и впоследствии унаследованы процессом-потомком, ибо, как Вы помните, таблица открытых файлов процесса-потомка, только что порожденного с помощью системного вызова fork, представляет собой копию таблицы открытых файлов процесса-предка, осуществившего указанный системный вызов (если только в одном из процессов-предков не была осуществлена операция закрытия некоторых из этих трех файлов).

В чем же преимущество использования для операции обмена именно стандартного ввода, стандартного вывода и стандартного протокола? Во-первых, нет необходимости отыскивать и открывать специальный файл, соответствующий пользовательскому терминалу; во-вторых, чрезвычайно простой становится реализация диалога с пользователем (достаточно осуществить системные вызовы read и write для файлов с пользовательскими дескрипторами 0 и 1 соответственно); и в-третьих, переназ-

---

[1] Авторы апеллируют к следующему соображению: использование самого нижнего уровня программного интерфейса ввода-вывода, поддерживаемого ОС UNIX, приводит к тому, что пользователь вынужден самостоятельно, в рамках своей прикладной программы, реализовать все его промежуточные уровни (например, форматирование, обработка данных, получение пользовательского дескриптора файла и т. д.) вместо того, чтобы воспользоваться одной из стандартных функций, предоставляемых ОС UNIX (например, fprintf или fopen). Однако в этом случае реализация операции ввода-вывода доступна пользователю почти на всех этапах, вплоть до момента передачи данных драйверу соответствующего физического устройства, что и делает саму эту операцию более наглядной, а результат ее выполнения более достоверным. — *Прим. ред.*

[2] Операция вывода информации в стандартный протокол, как правило, осуществляется небуферизованно, что позволяет получать диагностические сообщения немедленно.

начить стандартный вывод, стандартный ввод или стандартный протокол с помощью интерпретатора команд shell очень просто, и если у вас возникает необходимость вывести результат выполнения Вашей программы не на стандартный вывод, а в некоторый файл, то ее легко реализовать, специфицировав это переназначение в командной строке, с помощью которой Вы вызываете свою программу на выполнение. Например, в результате выполнения интерпретатором команд shell командной строки

```
cat fred > jim
```

содержимое файла fred вместо того чтобы быть выведенным командой cat на свой стандартный вывод (т. е. на пользовательский терминал), будет скопировано в файл с именем jim, так как в результате обработки интерпретатором команд shell символа > стандартный вывод команды cat будет переназначен им на файл с именем jim.

Ниже приведен исходный текст программы на языке Си, осуществляющей копирование всего стандартного ввода на свой стандартный вывод, которую можно рассматривать как некую упрощенную версию команды cat:

```
/*
 * ourcat <input> output
 */
main()
{
char c;
while(read(0, &c, 1) > 0)
      write(1, &c, 1);
}
```

Итак, байты информации, вводимые один за другим из файла в пользовательским дескриптором файла 0, так же один за другим выводятся в файл с пользовательским дескриптором файла 1. При этом переменная C, имеющая тип данных char, используется в качестве локального буфера программы. Добавим, что оператор цикла while будет выполняться до тех пор, пока со стандартного ввода не будет введен специальный символ ^D, специфицирующий конец файла, или же не возникнет ошибка при обмене с физическим устройством. Если при этом стандартный ввод остался назначенным на пользовательский терминал, то эмулировать ситуацию "конец файла" можно, введя с терминала специальный символ EOT (как правило, символ ^D).

Воспользуемся теперь разработанной нами программой для копирования файла, для чего введем с терминала командную строку вида[1]

```
ourcat <file1 >file 2
```

---

[1] Авторы считают очевидным, что объектный код, полученный в результате компиляции и последующей компоновки программы, исходный текст которой рассматривается, помещен в файл с именем ourcat. — *Прим. ред.*

В результате интерпретации этой командной строки интерпретатором команд shell последний осуществляет переназначение стандартного ввода команды ourcat на файл file1, а ее стандартный вывод на файл file2. В рамках самой программы ourcat об этом переназначении ничего не известно; все операции обмена, выполняемые ею, лишь вводят и выводят соответствующую информацию в файлы, имеющие пользовательские дескрипторы файла 0 и 1 соответственно. Аналогичная ситуация возникает при использовании такого инструментального средства ОС UNIX, как канал. Если, например, с терминала введена командная строка вида

```
firstprog | ourcat
```

то в результате интерпретации ее интерпретатором команд shell стандартный вывод программы firstprog окажется замкнутым на стандартный ввод программы ourcat (подробнее каналы ОС UNIX будут рассмотрены в гл. 7 книги).

ОСУЩЕСТВЛЕНИЕ ВВОДА И ВЫВОДА ИНФОРМАЦИИ В ФАЙЛ

В рассмотренной нами выше программе на языке Си системные вызовы write и read были использованы для непосредственного копирования информации байт за байтом со стандартного ввода на стандартный вывод. В том случае, когда стандартный ввод и стандартный вывод переназначены на файл, хранящийся на магнитном диске, подобный метод копирования малоэффективен. Перепишем текст нашей программы, для того чтобы буферизовать процесс передачи данных и тем самым повысить эффективность программы копирования файлов:

```
/*
 *      Быстрый вариант программы ourcat
 */
main()
    copyfile(0,1);
}
#define BLOCK 512
/*
 * Эта программа копирует файл с пользовательским
 * дескриптором файла fd1  в файл с пользовательским
 * дескриптором файла fd2.
 */
copyfile(fd1, fd2)
int fd1, fd2;
{
    char buffer [BLOCK];
    int n;
    while((n=read(fd1, buffer, BLOCK)) > 0)
        write(fd2, buffer, n);
}
```

Как видно из исходного текста программы ourcat, новая ее версия выполняет копирование стандартного ввода на стандартный вывод блок за блоком. Таким образом, каждой паре осуществленных системных вы-

зовов read и write соответствует в новой версии программы ourcat один блок переданной информации вместо одного байта в ее старой версии. Если в процессе обмена возвращаемое значение системного вызова read оказывается равным 0, это означает, что достигнут конец файла и операция копирования может быть завершена. Предлагая Вашему вниманию исходный текст новой версии программы, мы выбрали размер буфера ввода-вывода равным 512 байт (имея в виду используемый в определении макроподстановки контекст BLOCK). При этом мы руководствовались тем, что обычно так выбирается размер логического блока в большинстве реализаций ОС UNIX[1]. И хотя фактически здесь также используются байтовые операции обмена, мы не видим оснований выбирать размер буфера больше или меньше 512 байт.

Попробуем теперь сравнить две версии программы копирования файлов ourcat по эффективности, для чего поставим им в соответствие некие численные характеристики, воспользовавшись для их получения приводимым ниже критерием. Предположим, что t, среднее время, необходимое для осуществления системного вызова read или write, может быть представлено в виде суммы двух составляющих, O и R_n, где O — некоторая постоянная величина, а R_n — переменная составляющая, пропорциональная числу байт, передаваемых с помощью одного системного вызова read или write. В таком случае время T_N, необходимое на копирование файла с помощью второй версии программы ourcat, может быть вычислено следующим образом:

$$T_N = 2S/(O + R_n)/N.$$

Понятно, что время, необходимое для копирования файла с помощью первой версии программы ourcat, т. е. байт за байтом (иначе говоря, при N = = 1) равно $T_1 = 2S(O+R)$. Вычислим теперь отношение E величин T_N и T_1:

$$E = T_N/T_1 = (O + R_N)/N/(O + R).$$

Пусть

$$A = O/R.$$

Тогда верно следующее утверждение:

$$E = (1/N).((A + N)/(A + 1)).$$

Если $O > R$ (значительно больше величины R), т. е. отношение A очень велико, то отношение E обратно пропорционально размеру буфера N. Если же величина O мала, т. е. отношение A очень мало, то отношение E мало зависит от размера буфера N. И, наконец, если величины O и R одинакового порядка, то величина отношения E растет по мере роста N, но не превышает 2. На основе этих рассуждений построим график (рис. 4. 5.). На рис. 4. 6 представлен график, отражающий реальные соотношения ве-

---

[1] На самом деле размер логического блока, используемого в конкретной реализации ОС UNIX, задается аналогичной макроподстановкой, содержащейся в файле /urs/include/param.h.

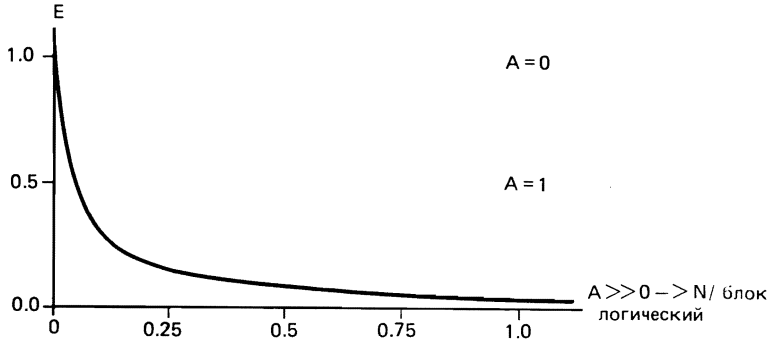

Рис. 4.5. График зависимости эффективности выполнения операций буферизованного ввода-вывода и размера используемого буфера

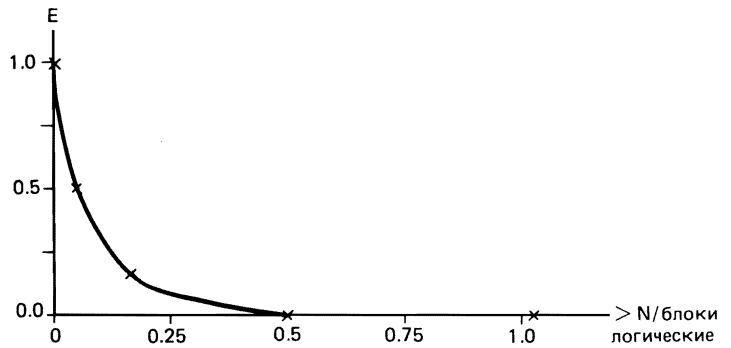

Рис. 4. 6. Результаты измерений для ОС System V

личин E и A, в частности, видно, что реальными являются значения 0 = 11 мс, R = 13 мкс, а A = 760.

В результате проведенного таким образом анализа Вам должно быть ясно, что вторая версия разработанной нами программы ourcat осуществляет копирование файла более эффективно, нежели первая. Обобщая этот вывод, заметим, что всякий буферизованный обмен всегда эффективнее обмена небуферизованного. Что же касается программных средств, реализующих механизм буферизованного обмена, то они могут быть помещены, например, в библиотечные подпрограммы, и тем самым скрыты от пользователя (подробнее эти программные средства будут рассмотрены в следующей главе).

СИСТЕМНЫЕ ВЫЗОВЫ open И create

Как мы уже говорили, файлы ОС UNIX, называемые стандартным вводом, стандартным выводом и стандартным протоколом и имеющие

пользовательские дескрипторы файла 0, 1 и 2 соответственно, всегда открыты для пользователя ОС UNIX. В противоположность этому, любой другой файл ОС UNIX должен быть явно открыт с помощью системного вызова open, если пользователь намерен получить к нему доступ. Напомним, что при осуществлении системного вызова open должны быть специфицированы по крайней мере два его аргумента. Первый представляет собой полное имя открываемого файла, а второй специфицирует режим открытия файла. Для того чтобы осуществить системный вызов open, достаточно в программе на языке Си использовать обращение к стандартной функции open, имеющее следующий формат:

```
open(pathname, oflag [,mode]);
```

При этом аргумент oflag представляет собой строку, содержащую последовательность символьных констант, разделенных символами | — знаком битовой операции *или*. Указанные символьные константы определены в файле с именем /usr/include/sys/fcntl.h с помощью символьной замены. Основными из символьных констант, составляющих строку oflag, можно считать константы O_RDONLY и O_WRONLY. Использование этих символьных констант в строке oflag специфицирует соответственно режимы открытия файла "только чтение" и "только запись" (понятно, что режимы открытия исключают друг друга и не могут быть специфицированы одновременно). Кроме указанных, может быть использована еще одна символьная константа — O_RDWR, которая специфицирует режим открытием файла "чтение-запись".

Итак, хотя мы оговорили, что строка oflag может содержать последовательность символьных констант, но составить эту последовательность из уже перечисленных символьных констант, очевидно, невозможно[1]. Если Вы используете для разработки своих программ на языке Си ОС System III или ОС System V, в Вашем распоряжении оказывается еще целый ряд символьных констант: O_NDELAY, O_APPEND, O_CREAT, O_TRUNC и O_EXCL. Использование этих символьных констант в строке oflag позволяет расширить понятие режима открытия файла. Так, если в строку oflag входит символьная константа O_NDELAY и открываемый только для чтения файл является каналом (см. гл. 7), то пользовательскому процессу, функционирующему в рамках такой программы, немедленно возвращается код завершения; при этом если файл, являющийся каналом, открывается только для записи и нет ни одного пользовательского

---

[1] Строго говоря, в ОС UNIX версии 7 используется непосредственная спецификация режимов открытия файлов; в этой версии отсутствует файл именем /usr/sys/include/sys/fcntl.h, и вместе с ним соответствующие символьные подстановки. Вместо них используются непосредственно числовые константы 0 — только чтение; 1 — только запись, 2 — чтение-запись. Если, например, в программе, выполняемой под управлением ОС System III или ОС System V, использован вызов стандартной функции вида open (name, O_RDONLY), то для получения того же результата при выполнении программы под управлением ОС UNIX версии 7 необходимо использовать вызов функции open (name, 0).

процесса, уже открывшего этот же файл для чтения (в режиме открытия файла "только чтение" или "только запись"), то возвращаемое соответствующим системным вызовом значение будет специфицировать аварийное завершение операции обмена. Сделаем еще одно замечание: если в описанной ситуации вместо канала был специфицирован файл, соответствующий пользовательскому терминалу, то код завершения будет возвращен пользовательскому процессу независимо от того, существует или нет реальная физическая связь.

Далее, если в состав строки o_flag входит символьная константа O_APPEND, то в результате открытия файла в таком режиме (режим "пополнения") указатель запись-чтение автоматически будет устанавливаться ОС UNIX на конец файла всякий раз, когда будет осуществлен системный вызов write для записи информации в этот файл; в противном случае, указатель чтения-записи после осуществления системного вызова open будет установлен на начало файла.

Рассмотрим теперь следующий режим открытия файла: если в состав строки oflag входит символьная константа O_CREAT, а открываемый с помощью соответствующего системного вызова файл не существует, то в результате специфицированный файл будет предварительно создан и ему будет поставлен в соответствие код защиты, специфицированный третьим аргументом системного вызова open, и уже после этого он будет открыт с соответствующим режимом открытия файла. Если же открываемый таким образом файл уже существует, то наличие в строке oflag символьной константы O_CREAT будет ОС UNIX проигнорировано.

Часто при разработке программ возникает необходимость создания временных файлов, имеющих, как правило, некие стандартные имена. Решить эту проблему можно с помощью следующего режима открытия файла. Пусть в состав строки oflag, являющейся аргументом системного вызова open, входит символьная константа O_TRUNC, тогда в результате открытия уже существующего файла его размер будет установлен равным 0.

И, наконец, еще один режим открытия файла. Если в состав строки oflag входят символьные константы O_CREAT и O_EXCLa открываемый файл существует, то пользовательскому процессу, осуществившему соответствующий системный вызов open, будет возвращено значение, специфицирующее аварийное завершение операции обмена. Использование такого режима открытия файла позволит пользователю избежать ошибочной перезаписи файла, а следовательно, и потери данных.

Рассмотрим теперь ряд примеров, иллюстрирующих использование системного вызова open. Прежде всего, попробуем создать временный файл:

```
#include >errno.h>
#include <fcntl.h>
extern int errno;
if((fd = open(scratch,O_RDWR|O_CREAT|O_EXCL,0600))< 0){
      if(errno == EEXIST)
         fprintf(stderr, "Scratch file already exists\n");
      exit(errno);
}
```

Далее откроем файл в режиме "пополнение":

```
#include <fcntl.h>
extern int errno;
if((fd = open(accounts, O_WRONLY|O_APPEND)) < 0)
exit(errno);
```

И, наконец, откроем канал в режиме "только запись" с проверкой существования этого канала, уже открытого другим пользовательским процессом для чтения:

```
#unclude <errno.h>
#include <fcntl.h>
extern int errno;
if((fd = open(pipe, O_WRONLY|O_NDELAY)) < 0){
        if(errno == ENXIO)
           fprintf("No process reading pipe\n");
        exit(errno);
}
```

Во всех перечисленных примерах мы пользовались тем фактом, что в случае аварийного завершения системного вызова open последний возвращает число —1, а внешняя переменная errno принимает значение, равное коду ошибки. Подробнее коды ошибок и способы их обработки описаны в гл. 7.

Кроме только что описанного способа, создать файл в ОС UNIX можно с помощью системного вызова creat. Если создаваемый с помощью системного вызова creat файл уже существует, то старое его содержимое будет потеряно, т. е. файл будет перезаписан. Надо сказать, что сохранение системного вызова creat в составе ОС System III и ОС System V вызвано требованием совместимости с более ранними версиями ОС UNIX, имеющими лишь три основных режима открытия файла, и вынужденными потому использовать для создания нового файла специальный системный вызов creat. Итак, системный вызов creat может быть осуществлен пользовательским процессом, функционирующим в рамках программы на языке Си, с помощью стандартной функции creat(), имеющей два аргумента. Первый из них представляет собой полное имя файла, а второй специфицирует код защиты создаваемого файла. Рассмотрим в качестве примера следующий фрагмент исходного текста программы:

```
int fd, access;
char *filename;
fd = creat(filename, access);
```

В случае успешного завершения системный вызов creat возвращает пользовательский дескриптор созданного файла. При аварийном завершении системного вызова возвращаемое значение будет равно —1. Далее, если при осуществлении системного вызова creat было специфицировано имя существующего файла, то содержимое файла будет разрушено, его размер установлен равным 0, а возвращаемое значение будет представлять собой пользовательский дескриптор файла (аналогично тому, как

это выполняется при использовании в составе строки oflag символьной константы O_TRUNC).

Воспользуемся теперь системными вызовами open и creat и модифицируем разработанную нами ранее на языке Си программу ourcat так, чтобы новая программа (назовем ее ourcp) могла быть использована для копирования содержимого одного файла в другой, создавая последний в случае необходимости. Наша новая программа будет обрабатывать два входных параметра: первый как имя файла, содержимое которого копируется, а второй как имя файла, в который осуществляется копирование. Как Вы знаете, параметры, передаваемые вызываемой на выполнение программе в командной строке, доступны ей как аргументы функции main. Рассмотрим теперь исходный текст программы ourcp на языке Си (функция copyfile() была описана нами в одном из предыдущих примеров).

```
/*
 *     ourcp.c
 */
#include <fcntl.h>
int fdin, fdout;
main(argc,argv)
char **argv;
int argc;
{
   if(argc < 3){
     printf("usage: ourcp oldfile newfile\n");
     exit(1);
   }
   if((fdin = open(argv[1], O_RDONLY)) < 0){
     printf("can't open %s\n",argv[1]);
     exit(1);
   }
   if((fdout = creat(argv[2], 0)) < 0){
     printf("can't creat %s\n",argv[2]);
     exit(1);
   }
   copyfile(fdin, fdout);
   exit(0);
}
```

Как видно из текста, программа ourcp осуществляет контроль корректности заданных в командной строке параметров программы; успешности завершения операции открытия входного файла; успешности завершения операции создания выходного файла. Если теперь одна из операций обмена завершится аварийно, то на терминал будет выведено соответствующее диагностическое сообщение и выполнение программы будет также аварийно завершено, о чем можно будет судить по возвращаемому значению программы ourcp. В соответствии с принятым с ОС UNIX соглашением успешное завершение вызванной на выполнение программы специфицируется только возвращаемым значением, равным 0. Добавим, что использованная здесь функция printf ( ) реализуется стандарт-

ной подпрограммой frintf из библиотеки SIO (стандартная библиотека ввода-вывода), которая используется для осуществления форматного вывода на стандартный вывод. Введите теперь с терминала команду

```
ourcp file1 file2
```

В случае ее успешного завершения содержимое файла с именем file1 будет скопировано в файл с именем file2, который при необходимости будет создан.

НАСЛЕДОВАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ДЕСКРИПТОРОВ ФАЙЛА

Мы уже рассказывали Вам о механизме наследования открытых файлов, о наборах данных, используемых системой управления ввода-вывода для осуществления доступа не совпадающих, но "родственных" процессов к одному и тому же файлу с помощью не совпадающих пользовательских дескрипторов файла.

Опишем теперь два системных вызова: dup и fcntl, использование которых позволяет создать копию пользовательского дескриптора файла и тем самым осуществить доступ к одному и тому же файлу из одного и того же пользовательского процесса с помощью двух различных пользовательских дескрипторов файла. Итак, системные вызовы dup и fcntl дают пользователю возможность осуществлять операцию обмена с одним файлом, используя для этого поочередно несколько различных пользовательских дескрипторов файла.

СИСТЕМНЫЙ ВЫЗОВ dup

Системный вызов dup обрабатывает свой единственный параметр как пользовательский дескриптор открытого файла и возвращает целое число, которое может быть использовано как еще один пользовательский дескриптор того же файла (его принято называть копией пользовательского дескриптора файла). Необходимо заметить, что с помощью копии пользовательского дескриптора файла к нему может быть осуществлен доступ того же типа и с использованием того же значения указателя записи-чтения, что и с помощью оригинального пользовательского дескриптора файла. Рассмотрим эту процедуру подробнее. Если Вы помните, каждому пользовательскому процессу соответствует таблица открытых файлов процесса, а каждому пользовательскому дескриптору файла, открытого этим процессом, соответствует один элемент указанной таблицы открытых файлов процесса. Далее, элемент таблицы открытых файлов процесса, соответствующий данному пользовательскому дескриптору файла, содержит указатель местоположения одного элемента таблицы файлов. И, наконец, указанный элемент таблицы файлов содержит информацию, однозначно специфицирующую номер описателя файла, значение указателя чтение-запись и режим открытия файла. Таким образом, для того чтобы создать вторую ссылку на файл с сохранением всех атрибутов доступа к нему и значением указателя запись-чтение, достаточно

создать в таблице открытых файлов процесса еще один элемент, являющийся точной копией элемента, порядковый номер которого был использован в качестве аргумента системного вызова dup. Вы, наверно, уже поняли, что порядковый номер созданного с помощью системного вызова dup элемента копии таблицы открытых файлов процесса и станет копией пользовательского дескриптора файла. Именно этот номер будет возвращен системным вызовом dup в результате его осуществления. Например:

```
new_fd = dup(old, fd);
```

Аварийное завершение системного вызова dup возможно в том случае, если указанный в качестве параметра системного вызова пользовательский дескриптор не соответствует открытому файлу или исчерпаны все файлы, которые могут быть одновременно открыты процессом (обычно их 20) .

Хорошим примером использования системного вызова dup может послужить способ реализации гарантии открытости для пользовательского процесса файлов с пользовательскими дескрипторами файлов 0, 1 и 2 и назначение на них соответственно стандартного ввода, стандартного вывода и стандартного протокола. Итак, рассмотрим следующий фрагмент исходного текста программы на языке Си:

```
stdin  = open("/dev/tty", O_RDONLY);  /* fd0 */
stdout = open("/dev/tty", O_WRONLY);  /* fd1 */
stderr = dup(1);
```

При внимательном изучении этого фрагмента у Вас могут возникнуть сомнения в том, что переменные stdin, stdout и stderr принимают значения, равные соответственно 0, 1 и 2. Действительно, если к моменту выполнения этого фрагмента программы пользовательским процессом, функционирующим в ее рамках, уже были открыты некоторые файлы, то значения переменных stdin, stdout и stderr могут оказаться, вообще говоря, произвольными. Чтобы избежать этих трудностей, модифицируем предлагаемый фрагмент программы следующим образом:

```
close(0); close(1); close(2);        /* игнорировать
                                      * возвращаемые значения */
stdin = open("/dev/tty", O_RDONLY); /* fd0 */
stdout = open("/dev/tty", O_WRONLY);/* fd1 */
stderr = dup(1);                     /* fd2 */
```

Итак, как видно из приведенного текста, прежде чем осуществить операцию открытия файлов с пользовательскими дескрипторами stdin, stdout и stderr, файлы с пользовательскими дескрипторами 0, 1 и 2 будут закрыты, в результате чего соответствующие элементы таблицы открытых файлов процесса окажутся свободными. Остается добавить, что системный вызов open, как, впрочем, и системный вызов dup, в результате своего выполнения возвращает порядковый номер первого от начала

свободного элемента таблицы открытых файлов процесса, уменьшенный на 1.

Нетрудно представить, насколько сложнее стала бы решенная нами только что задача, если бы принятые в ОС UNIX соглашения устанавливали в качестве стандартного ввода, стандартного вывода и стандартного протокола файлы с пользовательскими дескрипторами, например 18, 19 и 20 соответственно.

СИСТЕМНЫЙ ВЫЗОВ dup2

Операционная система UNIX версии 7 предоставляет пользователям системный вызов dup2. Он был создан с целью дать пользователю возможность самостоятельно назначать открываемым файлам конкретные пользовательские дескрипторы файлов. Системный вызов dup2 имеет два входных параметра, обрабатывая первый из них как оригинальный пользовательский дескриптор файла, а второй — как копию пользовательского дескриптора файла. Для того, чтобы осуществить системный вызов dup2, достаточно поместить в исходный текст программы на языке Си оператор вида

```
new_fd = dup2(old_fd, new_value);
```

В случае успешного завершения системного вызова dup2 последний вернет копию пользовательского дескриптора файла. Если окажется, что данным пользовательским процессом уже открыт файл с пользовательским дескриптором файла new_fd, то в результате выполнения системного вызова dup2 этот файл будет закрыт.

СИСТЕМНЫЙ ВЫЗОВ fcntl

Операционные системы System III и System V предоставляют своим пользователям другое, более общее решение задачи разделения пользовательских дескрипторов файлов. Речь идет о системном вызове fcntl. Для осуществления системного вызова fcntl необходимо поместить в текст программы на языке Си следующие строки:

```
#include <fcntl.h>
result = fcntl(fd, command, argument);
```

Аргумент command может принимать одно из пяти значений, являющихся символьными константами, и в зависимости от значения этого аргумента системный вызов fcntl выполняет те или иные действия. Сопоставим значения аргумента command и выполняемые системным вызовом fcntl действия:

F_DUPFD — Системный вызов fcntl возвращает первый свободный пользовательский дескриптор файла, значение которого не меньше значения аргумента argument. Этот пользовательский дескриптор файла должен быть копией пользовательского дескриптора файла, заданного аргументом fd.

F_SETFD — Системный вызов устанавливает ассоциированный с пользовательским дескриптором файла fd флаг close-on-exec в состояние, заданное младшим битом аргумента argument. Напомним, если флаг close-on-exec установлен в состояние "1", то файл, имеющий пользовательский дескриптор файла fd, будет закрыт немедленно при осуществлении пользовательским процессом системного вызова exec[1].

F_GETFD — Системный вызов fcntl возвращает значение флага close-on-exec, ассоциированного с пользовательским дескриптором файла fd.

F_GETFL — Целое число, являющееся возвращенным значением системного вызова fcntl, специфицирует режим открытия файла, имеющего пользовательский дескриптор файла fd. При этом 0 соответствует режиму доступа "только чтение"; 1 — "только запись"; 2 — "запись-чтение".

F_SETFL — Системный вызов fcntl переустанавливает режим открытия уже открытого файла, имеющего пользовательский дескриптор файла fd.

В качестве примера использования системного вызова fcntl рассмотрим, как с его помощью можно в среде ОС System V выполнить все те действия, которые в среде ОС UNIX версии 7 выполняются с помощью системного вызова dup2 (в ОС System V он отсутствует). Так, если аргумент command принимает значение F_DUPFD, то возвращаемое значение системного вызова fcntl специфицирует первый (не меньше значения переменной argument) пользовательский дескриптор файла. Таким образом, с помощью системного вызова fcntl можно эмулировать системный вызов dup2 следующим образом:

```
dup2(ofd, nfd)
int ofd, nfd;
{
   close(nfd);
   return(fcntl(ofd, F_DUPFD, nfd));
}
```

В гл. 6 мы увидим, как с помощью системных вызовов dup и fcntl интерпретатор команд shell реализует каналы и переназначение стандартного ввода и стандартного вывода на файл.

СОЗДАНИЕ СПЕЦИАЛЬНОГО ФАЙЛА И КАТАЛОГА

Системный вызов creat может быть использован любым пользователем для создания обычного файла. Однако создать с его помощью специальный файл или каталог невозможно, для этого необходимо воспользо-

[1] Флаг close-on-exec реализует программное средство управления механизмом разделения файлов, открытых процессом-предком, его процессами-потомками. — *Прим. ред.*

ваться системным вызовом mknod (имея полномочия привилегированного пользователя). Системный вызов mknod имеет три аргумента:

```
mknod(name, mode, addr);
char name;
```

Первый аргумент, name, обрабатывается им как указатель на строку, содержащую имя создаваемого специального файла; второй, mode, специфицирует код защиты создаваемого файла; третий, mknod, специфицирует идентификатор класса и идентификатор устройства в классе того физического устройства, которому должен соответствовать создаваемый специальный файл; при этом аргумент abbr должен быть задан равным 0, если с помощью системного вызова mknod создается обычный файл или каталог.

Приведем теперь два примера, в которых рассмотрим создание с помощью системного вызова mknod каталога с именем bin и специального файла с именем bdo.3 такого, что физическое устройство, которому он соответствует, может быть отнесено к классу устройств с идентификатором 4 и имеет в нем идентификатор устройства 3.

```
#include <sys/param.h>
#include <sys/inode.h>
/* создать каталог с именем bin, имеющий
 * код доступа rwxr-xr-x
 */
mknod("bin", IFDIR|0755, 0);
/* создать специальный файл с именем
 *  bd0.3, имеющий код доступа rw-r--r--
 */
mknod("bd0.3", IFBLK|0644, makedev(4,3));
```

Значение, возвращаемое системным вызовом mknod, равно 0, при успешном и -1 при аварийном завершении.

В приведенном нами примере контекст makedev задан определением макроподстановки, которое содержится в файле param.h. В результате обработки препроцессором языка Си выражения makedev (4,3) последний вместо него осуществит подстановку выражения ((4 $\ll$ 8) | (3)). Таким образом, в качестве своего третьего аргумента системный вызов mknod будет обрабатывать переменную типа int, значение которой получено в результате установки в 1 нулевого и первого битов ее старшего байта.

ОСУЩЕСТВЛЕНИЕ ПРОИЗВОЛЬНОГО ДОСТУПА К ФАЙЛУ

Сразу после открытия файла, соответствующий указатель запись-чтение устанавливается на первый байт его содержимого. Далее, по мере считывания из файла (или записи в файл) указатель запись-чтение перемещается по содержимому файла вперед байт за байтом. Очень часто такой подход оказывается неудобным. Действительно, представьте себе, что Вам необходимо ввести некую информацию из очень большого файла и

Вы знаете, что она находится во второй его половине. В такой ситуации необходимо воспользоваться системным вызовом lseek[1], с помощью которого указатель запись-чтение может быть установлен на любой байт, содержащийся в файле информации. Рассмотрим теперь выполнение системного вызова lseek, осуществить который можно, поместив в текст программы на языке Си строку вида

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

Итак, как видно из приведенного обращения к стандартной функции lseek, системный вызов lseek имеет три аргумента: первый, fd, является пользовательским дескриптором открытого файла; второй, offset, специфицирует относительное смещение указателя запись-чтение; третий, origin, может принимать одно из трех значений: 0, 1 или 2. Если значение аргумента origin равно 0, то аргумент offset обрабатывается как смещение относительно начала файла; если значение аргумента origin равно 1, то аргумент offset обрабатывается как смещение относительно текущего положения указателя запись-чтение; если же значение аргумента origin равно 2, то аргумент offset обрабатывается как смещение относительно конца файла. Заметим, что offset — это переменная, имеющая тип данных long int; это необходимо для осуществления доступа к очень большим файлам. Рассмотрим еще три примера осуществления системного вызова lseek:

```
lseek(fd, 0L, 2); /* установить указатель чтение-запись
                                        на конец файла */
lseek(fd, 0L, 0); /* установить указатель чтение-запись
                                        на начало файла */

lseek(fd, BSIZE*10L, 1); /* пропустить 10 блоков файла  */
```

В первых двух примерах указатель запись-чтение устанавливается соответственно на конец и на начало файла, а в третьем примере указатель запись-чтение перемещается на 10 логических блоков вперед относительно его текущего положения. Напомним, что символ L, помещенный непосредственно за константой, указывает компилятору с языка Си, что эта константа должна быть использована в формате long int.

Наличие в ОС UNIX системного вызова lseek дает возможность пользователю составлять программы, которые обрабатывали бы файлы ОС UNIX как массивы. Пример такой программы приведен ниже. В этом примере Вы найдете описания на языке Си функций aopen, aclose, agetb и asetb. Функция aopen используется для осуществления операции открытия файла; в процессе выполнения она осуществляет системный вызов malloc, с помощью которого резервируется оперативная память для создания буфера, а в случае его успешного завершения инициализирует

[1] В отличие от своего предшественника — системного вызова seek (поиск), с помощью которого указатель запись-чтение может быть перемещен от его текущей позиции не более чем на 65536 байт, системный вызов lseek (сокращение от long seek) такого ограничения не имеет.

структуру, имеющую шаблон f_addr и ассоциированную с пользовательским дескриптором файла fd. Для того чтобы использовать системную константу NOFILE, специфицирующую максимальное число файлов, которые могут быть одновременно открыты пользовательским процессом, достаточно выполнить операцию включения файла sys param.h. Как Вы уже знаете, этой системной константой фактически задается размер таблицы открытых файлов процесса и знание ее позволит эффективнее использовать оперативную память ЭВМ и тем самым избежать нарушения ограничений, накладываемых самой ОС UNIX. Итак, приведем исходный текст упомянутых функций:

```
#include <sys/param.h>
/* Эта программа обрабатывает содержимое файла
 * как последовательность байт.
 */
struct f_addr {
  long f_blkno;        /* обрабатываемый блок   */
  int  f_valid;        /* возвращаемое значение */
  char f_buffer[BSIZE];/* буфер обмена          */
  char f_flag;         /* флаг занятости буфера */
};
#define DIRTY 1        /* буфер занят           */
#define EOF   (-1)     /* конец файла           */
#define ERROR (-2)     /* ошибка ввода-вывода   */

struct f_addr *f_pntr[NOFILE];

aopen(name, flag);
char *name;
int flag;
{
   struct f_addr *fp;
   int fd = open(name, flag);
   if(fd >= 0){
       f_pntr[fd] = malloc(sizeof(struct f_addr));
       if(!f_pntr[fd])
          return(-1);
       fp = f_pntr[fd];
       fp->f_blkno = -1;
       fp->f_flag = fp->f_valid = 0;
   }
   return(fd);
}

aclose(fd)
int fd;
{
   struct f_addr *fp = f_pntr[fd];
   chkflush(fd);
   free(fp);
   f_pntr[fd] = 0;
   return(close(fd));
}
```

```
agetb(fd, address)
int fd;
long address;
{
   struct f_addr *fp = f_pntr[fd];
   int a_offset = address%BSIZE;
   long a_blkno = address/BSIZE;
   if(fp->f_blkno != a_blkno){
        chkflush(fd);
        fp->f_blkno = a_blkno;
        lseek(fd, fp->f_blkno*BSIZE, 0);
        fp->f_valid = read(fd, fp->f_buffer, BSIZE);
        lseek(fd, fp->f_blkno * BSIZE, 0);

        if(fp->f_valid <= 0)
                return(fp->f_valid-1);
        }
        if(a_offset < fp->f_valid)
                return(fp->f_buffer[a_offset]&0xFF);
        else
                return(EOF);
}

asetb(fd, address, value)
int fd, value;
long address;
{
     struct f_addr *fp = f_pntr[fd];
     int a_offset = address%BSIZE;
     long a_blkno = address/BSIZE;
     if(fp->f_blkno != a_blkno){
            chkflush(fd);
            fp->f_blkno = a_blkno;
            lseek(fd, fp->f_blkno*BSIZE, 0);
            fp->f_valid = read(fd, fp->fp_buffer, BSIZE);
            if(fp->f_valid <= 0)
                     return(fp->f_valid - 1);
      }
      fp->f_buffer[a_offset] = value&0xFF;
      fp->f_flag |= DIRTY;
      return(0);
}
chkflush(fd)
int fd;
{
     struct f_addr *fp = f_pntr[fd];
     if(fp->f_flag & DIRTY){
            lseek(fd, fp->f_blkno*BSIZE, 0);
            return(write(fd, fp->f_buffer, BSIZE));
     }
}
```

Использование функции aclose необходимо по двум причинам: во-первых, это гарантирует запись в файл обрабатываемого в данный момент логического блока в том случае, если содержимое логического блока из-

менилось; во-вторых, она осуществляет с помощью стандартной функции free освобождение зарезервированной ранее с помощью системного вызова mallos оперативной памяти ЭВМ. И в заключение сделаем еще несколько замечаний: приведенный в примере набор функций не предоставляет пользователю удобного средства для пополнения обрабатываемого файла; осуществление системного вызова lseek с целью перемещения указателя чтения-записи байт-ориентированного специального файла имеет смысл только для специальных файлов, соответствующих накопителю на магнитном диске и накопителю на магнитной ленте.

СОЗДАНИЕ И УДАЛЕНИЕ ЭЛЕМЕНТОВ КАТАЛОГА

В предыдущей главе, посвященной файлам ОС UNIX, мы рассмотрели структуру каталога, и Вы уже знаете, что каждый элемент каталога (или вход в каталог) состоит из двух полей — поля имени файла и поля ссылки. Вам также уже известно, что поле ссылки элемента каталога содержит ссылку на описатель соответствующего файла, которая реализована в виде номера этого описателя файла. Продолжая наши рассуждения, напомним, что вся информация о файле хранится в описателе файла, и только в нем, и следовательно, вполне корректна ситуация, при которой два разных элемента каталога содержат в поле ссылки номер одного и того же описателя файла, а в поле имени файла — разные имена файлов. Характеризуя описанную ситуацию, обычно говорят, что на данный файл имеется несколько ссылок.

Мы уже говорили, что описатель файла содержит информацию о числе ссылок на файл, которому он соответствует. Создать и удалить ссылку на существующий файл можно с помощью системного вызова link и системного вызова unlink соответственно. Создание двух или более ссылок на один и тот же файл, содержащихся в одном или более каталогах ОС UNIX, предоставляют пользователю возможность получать доступ к одному и тому же файлу по разным именам, являющихся в этом смысле синонимами (рис. 4.7).

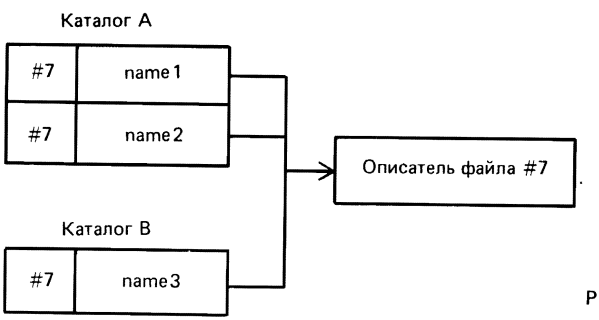

Рис. 4.7. Файлы синонимы

[1] В отличие от системного вызова creat в качестве входных параметров системных вызовов link и unlink могут быть использованы имена специальных файлов.

Единственное ограничение, возникающее при использовании системных вызовов link и unlink, может быть сформулировано следующим образом: и файл, и все ссылки на него должны находиться на одном и том же томе файловой системы.

Итак, с помощью системного вызова link можно создать ссылку на уже существующий файл и предоставить пользователю возможность доступа к этому файлу еще по одному имени[1]. Системный вызов link имеет два входных параметра, первый из которых он обрабатывает как имя существующего файла, а второй как вновь создаваемое имя-синоним.

```
link(oldname, newname);
char *oldname, *newname;
```

Оба системных вызова возвращают 0 в случае успешного завершения и — 1 в противном случае. Обычно все неудачи по созданию новых ссылок на файл бывают связаны с тем, что файл с именем newname уже существует или не существует файл с именем oldname.

Отметим одно интересное свойство описанного механизма осуществления доступа к файлу по его имени: все имена-синонимы одного файла совершенно равнозначны для файловой системы, реализующей упомянутый доступ к файлу. Представим себе, что некоторый каталог ОС UNIX содержит ссылку на некоторый файл, обеспечивающий доступ к этому файлу по некоторому имени. Создадим теперь с помощью системного вызова link в другом каталоге ОС UNIX еще одну ссылку на этот файл, обеспечивающую доступ к нему по другому имени, и удалим файл из первого каталога, специфицировав его тем именем, которое содержится в этом каталоге. В результате этих манипуляций доступ к файлу можно будет получить по его имени-синониму, а поскольку имя-синоним содержится в другом каталоге (в отличие от его первого оригинального имени), то у непосвященного пользователя возникает иллюзия, будто файл был скопирован из одного каталога в другой под новым именем, после чего первый файл был удален. Но мы с Вами знаем, что на самом деле ни один байт "копировавшейся" информации, хранившейся в оригинальном файле, не изменил в результате такого "копирования" своего положения на магнитном диске.

Перейдем теперь к рассмотрению системного вызова unlink, который выполняет действия, прямо противоположные действиям системного вызова link. С помощью системного вызова ulink можно удалить любую из существующих ссылок на файл. В результате его осуществления из каталога удаляется тот элемент, в поле имени которого находится контекст, указанный в качестве единственного системного вызова unlink, при этом счетчик числа ссылок, хранящийся в описателе соответствующего файла, уменьшается на 1.

```
unlink(pathname);
char *pathname;
```

Если в качестве аргумента системного вызова unlink использовано имя файла, на который в файловой системе ОС UNIX имеется единственная ссылка, то файл удаляется из файловой системы ОС UNIX, т. е. блоки этого файла считаются отныне свободными и включаются в список свободных блоков диска; из соответствующего каталога удаляется последняя ссылка на этот файл, а счетчику числа ссылок присваивается значение, равное 0; сам описатель файла считается свободным и может быть использован при создании нового файла.

Представим себе, однако, что один и тот же файл одновременно открыт двумя пользовательскими процессами, не связанными родственными отношениями. Пусть один из этих процессов удаляет из файловой системы ОС UNIX последнюю ссылку на указанный файл. В таких условиях попытка осуществить операцию ввода-вывода информации в этот файл, предпринятая вторым пользовательским процессом, приведет к аварийной ситуации. Чтобы избежать этого, все действия по удалению файла выполняются ОС UNIX в следующем порядке: сразу после осуществления пользовательским процессом системного вызова unlink выполняется единственное действие — уменьшается на 1 значение счетчика ссылок на данный файл; после того как все пользовательские процессы осуществят закрытие этого файла, из таблицы описателей файлов будет удален соответствующий элемент; затем ОС UNIX осуществит проверку значения счетчика числа ссылок на файл, соответствующий только что удаленному элементу таблицы описателей файла, и, если оно окажется нулевым, то завершит удаление файла, выполнив все действия по "освобождению" блоков диска, составляющих файл и содержавших соответствующий описатель файла.

Такое решение возникшей проблемы, позволяет, в частности, быть уверенным, что все временные файлы обязательно будут удалены, даже если обрабатывающая их программа завершилась аварийно. Проиллюстрируем последнее замечание с помощью следующего примера:

```
main()
{
     . . . . .
     fd = creat(temporary, mode);
     unlink(temporary);
     . . . . .
/*  выполнение операций записи и чтения      */
     . . . . .
     close(fd);   /* удаление временного файла */
}
```

Действительно, как видно из приведенного исходного текста прогаммы, только что созданный файл temporary удаляется из файловой системы ОС UNIX сразу после своего создания. Вместе с тем, будучи открытым, этот файл останется доступным пользовательскому процессу, функционирующему в рамках этой программы до тех пор, пока процесс не осуществит системного вызова close(fd). Известно, что при аварийном

завершении выполнения пользовательской программы ОС UNIX автоматически закрывает открытые в процессе выполнения файлы. Таким образом, даже если вызванная на выполнение программа, текст которой на языке Си приведен выше, завершится аварийно, в результате чего функция close (fd) не будет выполнена, файла с именем temporary в файловой системе ОС UNIX не окажется.

СИСТЕМНЫЕ ВЫЗОВЫ stat И fstat

Использование системных вызовов stat и fstat позволяет получить информацию о файле, не осуществляя явного доступа к нему. Характеристиками файла, чаще всего интересующими пользователя ОС UNIX, являются размер файла, дата его создания, тип файла, его код защиты и т. д. Системные вызовы stat и fstat имеют следующие форматы:

```
stat(name, structure);
char *name;
struct stat structure;

fstat(fd, structure);
int fd;
struct stat *structure;
```

Итак, системный вызов stat имеет два аргумента: первый из них обрабатывается как имя файла, а второй — как указатель на структуру с шаблоном stat; единственное различие вызова fstat и stat заключается в том, что он обрабатывает свой первый аргумент как пользовательский дескриптор уже открытого файла. В обоих случаях информация о файле, специфицированном первым аргументом описываемых системных вызовов, помещается в структуру, специфицированную их вторым аргументом. Объявление шаблона структуры stat содержится в файле с именем stat.h и имеет следующий вид:

```
/*
 * Structure of the result of stat
 */

struct stat
{
   dev_t     st_dev;
   ino_t     st_ino;
   ushort    st_mode;
   short     st_nlink;
   ushort    st_uid;
   ushort    st_gid;
   dev_t     st_rdev;
   off_t     st_size;
   time_t    st_atime;
   time_t    st_mtime;
   time_t    st_ctime;
};
#define  S_IFMT   0170000   /* тип файла                    */
#define  S_IFDIR  0040000   /* каталог                      */
```

```
#define  S_IFCHR  0020000   /*  байториентированный специальный файл        */
#define  S_IFBLK  0060000   /*  блокориентированный специальный файл        */
#define  S_IFREG  0100000   /*  обычный файл                 */
#define  S_IFIFO  0010000   /*  дисциплина FIFO              */
#define  S_ISUID    04000   /*  идентификатор владельца      */
#define  S_ISGID    02000   /*  идентификатор группы         */
#define  S_ISVTX    01000   /*  сохранить свопируемый текст  */
#define  S_IREAD    00400   /*  владельцу разрешено чтение   */
#define  S_WRITE    00200   /*  владельцу разрешена запись   */
#define  S_IEXEC    00100   /*  владельцу разрешено исполнение */
```

Глядя на приведенное здесь объявление шаблона stat, Вы можете заметить, что оно поразительно напоминает рассмотренное выше объявление шаблона dinode. В этом нет ничего удивительного, так как именно информация, хранящаяся в описателе файла, в основном и помещается системным вызовом stat (или fstat) в структуру, специфицированную его вторым входным параметром. Содержащиеся в этом же файле определения символьной замены дают возможность без особых затруднений проанализировать значения битов поля st_mode структуры с шаблоном stat. И, наконец, последнее замечание. Символьные константы, четыре первых символа которых совпадают с контекстом S_IF, могут быть использованы для определения типа, анализируемого с помощью системных вызовов stat или fstat файлов: обычный файл, каталог, специальный блок-ориентированный файл и т. д.

ПРОГРАММА move

Заканчивая главу, предлагаем Вам проанализировать исходный текст еще одной программы, в которой использованы все рассмотренные в этой главе системные вызовы. Обратимся к популярной команде ОС UNIX mv и рассмотрим ее упрощенный вариант, который назовем программой move. Итак, программа move (как и команда mv) осуществляет переименование файлов и также имеет два входных параметра, первый из которых обрабатывается как имя файла, а второй — как имя, присваиваемое этому файлу в результате его переименования. Если при этом второй входной параметр программы move специфицирует каталог ОС UNIX, то в результате ее выполнения файл, имя которого было использовано в качестве первого входного параметра этой программы, будет помещен в указанный каталог, а из своего исходного каталога будет удален (собственно имя файла будет при этом сохранено). Так же, как при использовании команды mv, с помощью программы move может быть переименован каталог, при этом переименованы будут и все содержащиеся в нем файлы (изменится первый слог их относительного полного имени). Напомним, что в последнем случае выполнение программы move (как, впрочем, и выполнение команды mv) будет корректным только при условии, что все поддерево файловой системы ОС UNIX, корнем которого является специфицированный каталог, находится на одном томе файловой системы ОС UNIX. Таким образом, предлагаемая программа на языке Си — весьма полезное программное средство.

Итак, приступим к рассмотрению входящих в состав программы move функций. Первая из них – функция filetype, используется для решения вопроса является ли обрабатываемый файл каталогом. Для этого функция foletype использует системный вызов stat. При удачном завершении функция filetype возвращает целое положительное число, специфицирующее тип файла, в противном случае она возвращает – 1.

```
#include <sys/types.h>
#include <sys/stat.h>
#define ECANSTAT  (-1)
#define BUFSIZ    256
filetype(filename)
char *filename;

{
   struct stat statbuf;
   if(stat(filename, &statbuf) < 0)
         return(ECANSTAT);
   else
         return(statbuf.st_mode & S_IFMT);
}
```

Функция mlink используется программой move для создания имени-синонима обрабатываемого файла и удаления из соответствующего каталога его оригинального имени. Она возвращает при своем удачном завершении число 1 и число 0 в противном случае. Заметим, что при выполнении функции mlink осуществляется системный вызов access, который мы до сих пор не описывали. Он имеет формат

```
access(name, mode);
```

Здесь аргумент name специфицирует имя файла, а аргумент mode – интересующий нас код защиты файла. При этом аргумент mode может принимать значение 1 – "доступ по выполнению", 2 – "доступ по записи", 4 – "доступ по чтению", либо суммы любых двух или всех трех выше перечисленных чисел, что соответствует одновременному требованию разрешения на доступ в любых двух или во всех трех режимах доступа к файлу. Если аргумент mode при осуществлении системного вызова access задан равным 0, то это означает, что нас интересует возможность осуществления доступа к файлу, содержащемуся в каталоге, имя которого специфицировано первым аргументом системного вызова. Если код защиты файла, имя которого специфицировано первым аргументом системного вызова access, такой, что пользовательский процесс, осуществивший этот системный вызов, имеет полномочия на доступ к файлу, тип которого специфицирован вторым аргументом системного вызова, то системный вызов access возвращает целое число 0, в противном случае он возвращает – 1.

```
mlink(f1, f2, type)
char *f1, *f2;
int type;
{
   char namebuf[BUFSIZ];
   switch(type){
   case S_IFDIR:
        if(access(f2, 0) == 0){
             printf("%s exists already\n", f2);
             exit(1);
        }
        if(link(f1, f2) < 0){
             printf("can't link to directory %s\n", f2);
             exit(1);
        }
        unlink(f1);
        return(1);
        break;
   default:
        switch(filetype(f2)){
        case S_IFDIR:
             strcpy(namebuf, f2);
             strcat(namebuf, "/");
             strcat(namebuf, f1);
             unlink(namebuf);
             if(link(f1, namebuf) < 0)
                      return(0);
             unlink(f1);
             return(1);
             break;
        default:
             unlink(f2);
             if(link(f1, f2) < 0)
                       return(0);
             unlink(f1);
             return(1);
             break;
        }
        break;
   }
}
```

Функция mcopy будет использована в процессе выполнения программы move в том случае, если выполнение системного вызова link или системного вызова unlink завершится аварийно. Функция mvopy осуществляет копирование файла, т. е. создает файл, содержимое которого представляет собой копию содержимого файла-оригинала, а затем из файловой системы файл-оригинал.

```
mcopy(f1, f2, type)
char *f1, *f2;
int type;
{
     int fd1, fd2;
     char namebuf[BUFSIZ];
```

```
        if(type == S_IFDIR || type == GORP)
                return(0);
        strcpy(namebuf, f2);
        fd1 = open(f1, O_RDONLY);
        if(filetype(f2) == S_IFDIR){
                strcat(namebuf, "/");
                strcat(namebuf, f1);
        }
        fd2 = creat(namebuf, 0);
        copyfile(fd1, fd2);
        close(fd1); close(fd2);
        unlink(f1);
        return(1);
}

main(argc, argv)
int argc;
char **argv;
{
   int type;
   if(argc != 3){
        printf("usage: mv source destination\n");
        exit(1);
   }
   switch(type = filetype(argv[1])){
   default:
           if(!mlink(argv[1], argv[2], type))
              if(!mcopy(argv[1], argv[2], type)){
                 printf("can't move %s\n",argv[1]);
                 exit(1);
              }
           break;
   case S_IFDIR:
           if(!mlink(argv[1], argv[2], S_IFDIR)){
                printf("no cross device links\n");
                exit(1);
           }
           break;
   case ECANSTAT:
           printf("can't stat %s\n", argv[1]);
           exit(1);
           break;
   }
   exit(0);
}
```

Заметим, что функции strcat и strcpy — стандартны и содержатся в одной из библиотек стандартных программ SIO. Стандартная функция strcpy осуществляет копирование одной строки символов в другую и имеет формат

```
strcpy(to, from)
char *to, *from;
```

При разработке программы move функция strcpy была использована вместе с функцией strcat для формирования относительного полного имени файла с помощью копирования конкатенации отдельных шагов. Итак, стандартная функция strcat имеет следующий формат:

```
strcat(head, tail)
char *head, *tail;
```

Обе стандартные функции обрабатывают как строку символов последовательность символов, последний из которых имеет восьмеричный код 0.

ЗАДАЧИ

1. Подготовьте и отладьте обе версии программы ourcat на доступной Вам ЭВМ, работающей под управлением ОС UNIX. После этого, варьируя размер буфера, попробуйте определить значения переменных O, R и A для Вашей ОС UNIX; чтобы получить метки времени, воспользуйтесь командой time.

2. Разработайте на языке Си небольшую программу, которая в результате своего выполнения создавала бы файл с именем name и помещала в него N логических блоков, не содержащих никакой информации, а размер логического блока был бы равен S байт. Пусть далее переменные name, N, S можно будет задавать в качестве трех входных параметров Вашей программы, а поскольку в этом случае числовые значения переменных N и S будут переданы функции main как строки цифр, то мы рекомендуем Вам воспользоваться для осуществления обратного преобразования стандартной функцией atoi, имеющей формат вызова

```
int atoi(string)
char *string;
```

3. Разработайте еще одну версию программы ourcp, которая в результате выполнения вместо "пересоздания" существующего файла осуществляла бы его перезапись.

4. Модифицируйте разработанную Вами в результате решения задачи 3 программу так, чтобы вместо перезаписи файла она в результате выполнения осуществляла его пополнение.

5. Еще раз модифицируйте разработанную Вами в результате решения задачи 3 программу. Пусть на этот раз в результате ее выполнения информация, содержащаяся в оригинальном файле, окажется помещенной в существующий уже файл-копию, начиная с логического блока, порядковый номер которого был специфицирован входным параметром Вашей программы. Что произойдет, если порядковый номер логического блока специфицированных входных параметров окажется больше общего числа логических блоков, содержащихся в уже существующем файле-копии.

6. Подумайте, имеет ли смысл организовать обмен информацией между двумя пользовательскими процессами реального времени, связанными родственными отношениями, путем использования имен файлов-синонимов? Убедитесь в верности своего вывода (для удобства рекомендуем Вам запустить программу на выполнение в асинхронном режиме).

7. Сравните эффективность команды ОС UNIX cp и разработанной нами программы ourcp. Чем, по-Вашему, могло быть вызвано различие между ними?

ЛИТЕРАТУРА

1. D.M. Ritchie and K. Thompson (1978), The UNIX Timesharing System, Bell Sys. Tech. J., 57(6) pp. 1905-1929.

2. K. Thompson (1978), UNIX Implementation, UNIX V7 Programming Manual, Volume. 2b.

3. D.M. Ritchie (1978), The UNIX I/O System. UNIX V7 Programming Manual, Volume. 2b.

4. B.W. Kernighan (1978), UNIX Programming - Second Edition. UNIX V7 Programming Manual, Volume. 2a.

5. B.W. Kernighan & D.M. Ritchie (1978), The C Programming Language, Chapter 7, Prentice Hall Inc. (См. [5] в списке литературы к гл. 1).

6. E.I. Organick (1972), The MULTICS System, MIT Press, Cambridge, Mass.

7. Western Electric Company, Inc. (1983), UNIX System V User's Manual.

Глава 5. БУФЕРИЗОВАННЫЕ ОПЕРАЦИИ ВВОДА-ВЫВОДА

Операционная система UNIX предоставляет пользователю набор системных вызовов, осуществление которых дает возможность пользовательскому процессу выполнить непосредственно байт за байтом ввод или вывод информации в файл. В предыдущей главе мы показали, что для повышения эффективности операций ввода-вывода их необходимо буферизовать, и даже продемонстрировали, как можно реализовать механизм буферизации в рамках самого пользовательского процесса. На самом деле выполнение буферизованных операций ввода-вывода настолько часто и общепринято, что имеет смысл реализовать указанный механизм буферизации один раз, например в рамках одной или нескольких стандартных подпрограмм. В ОС UNIX для этой цели служит библиотека SIO (стандартная библиотека ввода-вывода), которая, помимо функций, реализующих буферизованный ввод-вывод информации, содержит еще целый ряд полезных функций, реализующих, например, форматный ввод-вывод информации. Использованию библиотечных функций, входящих в состав библиотеки SIO, и посвящена эта глава.

УКАЗАТЕЛИ НА ФАЙЛ И ПОЛЬЗОВАТЕЛЬСКИЕ ДЕСКРИПТОРЫ ФАЙЛА

Любая библиотечная функция, входящая в состав библиотеки SIO, реализует операцию ввода-вывода информации с помощью все тех же системных вызовов. Например, библиотечные функции fread и fwrite фактически осуществляют системные вызовы read и write, и строка исходного текста программы на языке Си вида

```
read(fd, buffer, u);
```

эквивалентна строке вида

```
fread(buffer, n, 1, fp);
```

Существенное различие между этими двумя способами выполнения операции обмена заключается в том, что в первом случае (при осуществлении системного вызова read) файл, над которым выполняется операция обмена, специфицирован своим пользовательским дескриптором файла, а во втором — так называемым указателем на файл. Аргумент функции fread, имеющий в вышеприведенном примере значение 1, на самом деле специфицирует размер буфера, который может быть использован для выполнения буферизованных операций ввода-вывода. Например, в результате выполнения функции fread, вызванной с помощью строки исходного текста на языке Си вида

```
fread(buffer, 1, 512, fp);
```

будет осуществлен ввод информации их файла, специфицированного аргументом fp,. При этом операция ввода будет производиться над порцией информации размером 512 байт, местоположение которой в оперативной памяти специфицировано аргументом buffer. Рассмотрим теперь следующий фрагмент исходного текста программы:

```
fread(buffer, sizeof(long), 512, fp);
long *buffer;
```

Аргумент sizeof(long) представляет собой возвращаемое значение библиотечной функции sizeof, равное числу байт, составляющих переменную, имеющую тип данных long. В результате осуществления такого вызова библиотечной функции fread операция ввода будет производиться над порцией информации размером 512 X sizeof(long) байт (обычно 2048 байт информации) . Библиотечные функции, входящие в состав библиотеки SIO, выполняют буферизованные операции ввода-вывода, используя для организации буфера место, зарезервированное в теле пользовательской программы.

Указатель на файл представляет собой указатель на некую структуру FILE, в которой содержится информация об открытых для выполнения операции ввода-вывода файлах, в частности, значение указателя на буферы ввода (или вывода) и положение указателя чтения-записи. Обычно объявление самой структуры FILE скрыто от пользователя, и для того чтобы ею воспользоваться, необходимо включить в файл, содержащий исходный текст программы, файл с именем stdio.h, фрагмент которого приведен ниже.

```
#define BUFSIZ    512
#define _NFILE     20
```

```
extern  struct  _iobuf {
        int     _cnt;
        char    *_ptr;
        char    *_base;
        short   _flag;
        char    _file;
}   _iob[_NFILE];

#define NULL        0
#define FILE        struct _iobuf
#define EOF         (-1)
#define stdin       (&_iob[0])
#define stdout      (&_iob[1])
#define stderr      (&_iob[2])
#define getc(p)     (--(p)->_cnt>=0?*(p)->_ptr++&0377:\
                     _filbuf(p))
#define getchar()   getc(stdin)
#define putc(x,p)    (--(p)->_cnt>=0? ((int)( *(p)->_ptr++=\
                    (unsigned)(x))):_flsbuf((unsigned)(x),p))
#define putchar(x)  putc(x, stdout)
#define feof(p)     (((p)->_flag&_IOEOF)!=0)
#define ferror(p)   (((p)->_flag&_IOERR)!=0)
#define fileno(p)   ((p)->_file)

FILE    *fopen();
FILE    *fdopen();
FILE    *freopen();
long    ftell();
char    *fgets();
```

Помимо объявления структуры FILE, файл stdio.h содержит еще ряд объявлений структур, макроопределений и определений макроподстановок. Как мы уже говорили, пользовательские дескрипторы файлов 0, 1 и 2 всегда зарезервированы для файлов стандартного ввода, стандартного вывода, стандартного протокола, которым соответствуют указатели на файлы stdin, stdout и stderr. Как видно из приведенного выше фрагмента файла stdio.h, каждым пользовательским процессом может быть одновременно открыто _NFILE файлов (символьной константе _NFILE с помощью определения макроподстановки обычно ставится в соответствие целое число 20) . Это означает, что соответствующая таблица открытых файлов процесса может содержать не более _NFILE элементов.

ОСНОВНЫЕ БИБЛИОТЕЧНЫЕ ФУНКЦИИ ВВОДА-ВЫВОДА

Для осуществления операции ввода-вывода информации в файл последний должен быть предварительно открыт. Открыть файл можно с помощью, например, библиотечной функции fopen; закрыть — с помощью библиотечной функции fclose. Библиотечная функция fopen имеет два аргумента, первый из которых она обрабатывает как полное имя открываемого файла, а второй как спецификацию режима открытия файла. Возвращаемое значение библиотечной функции fopen является указателем на файл, открытый в результате ее вызова. В качестве примера ис-

пользования функции fopen рассмотрим следующий фрагмент исходного текста программы на языке Си:

```
# include <stdio.h>
char *pathname, mode;
FILE *fp = fopen(pathname, mode);
```

Аргумент mode, обрабатываемый функцией fopen как строка символов, может принимать одно из следующих возможных значений: r, w, a, r+, w+ или a+. Первые два значения специфицируют соответственно режимы открытия файла "только чтение" и "только запись", которым в случае осуществления системного вызова соответствуют символьные константы O_RDONLY и O_WRONLY. Если в качестве аргумента mode функции fopen использована строка a или строка a+, то файл будет открыт в режиме "пополнения" файла, что соответствует осуществлению системного вызова open при условии, что режим открытия файла в этом случае специфицирован строкой O_WRONLY | O_CREAT | O_APPEND. Понятно, что в результате выполнения операции вывода в открытый таким образом файл выведенная в него информация будет помещена после последнего байта его прежнего содержимого, что исключит потерю этого прежнего содержимого файла. Если аргумент mode функции fopen, специфицирующий режим открытия файла, имеет значение r+, то файл будет открыт в режиме "чтение-запись" (что в случае использования системного вызова open соответствует строке O_RDWR). Далее, пусть аргумент mode функции fopen имеет значение w+, тогда файл будет открыт в режиме "чтение-запись". При этом, если открываемый таким образом файл уже существует, то его размер будет установлен равным 0 (т. е. его прежнее содержимое будет потеряно); иначе файл будет создан вновь. Когда осуществляется системный вызов open, этот режим открытия файла может быть специфицирован строкой O_RDWR | O_CREAT | O_APPEND. И, наконец, еще один режим открытия файла, специфицируемый значением аргумента mode функции fopen, равным a+. При осуществлении системного вызова open этот режим открытия файла может быть специфицирован строкой O_RDWR | O_CREAT | O_APPEND. В этом случае файл будет открыт в режиме "чтение-запись", и если открываемый файл уже существует, то выведенная в него информация будет помещена после последнего байта его прежнего содержимого; иначе файл будет создан вновь.

Перейдем теперь к рассмотрению библиотечной функции fclose. Она имеет один аргумент, который обрабатывает как указатель на файл. В процессе выполнения функция fclose осуществляет системный вызов close, с помощью которого, в свою очередь, осуществляется закрытие файла. Однако прежде чем осуществить системный вызов close, функция fclose производит принудительное завершение всех операций ввода-вывода информации в специфицированный файл. Ниже приведен формат вызова функции fclose:

```
FILE *fp;
fclose (fp);
```

В составе библиотеки SIO есть еще две библиотечные функции, с помощью которых можно выполнить операцию открытия файла: freopen и fdopen. Рассмотрим первую из них. Результат выполнения функции freopen аналогичен результату осуществления системного вызова dup. Функция freopen обеспечивает переназначение ввода из некоторого уже открытого файла (или вывода в некоторый уже открытый файл) на файл, специфицированный ее первым аргументом, который она обрабатывает как полное имя файла. Ниже приведен пример использования библиотечной функции freopen:

```
FILE   *stream;
char   *filename, *type;
FILE   *fp = freopen(filename, type, stream);
```

Чаще всего функция freopen используется для переназначения стандартного ввода, стандартного вывода или стандартного протокола на файл. В качестве своего значения функция freopen возвращает указатель на файл, для которого было осуществлено переназначение ввода или вывода.

Библиотечная функция fdopen позволяет указать для открываемого файла не имя файла, а его пользовательский дескриптор. Ниже приведен формат вызова библиотечной функции fdopen:

```
FILE *fp = fdopen(fildes, type);
```

Библиотечная функция fileno возвращает в качестве своего значения пользовательский дескриптор файла, соответствующего специфицированному ее аргументом указателю на файл. Ниже приведен формат вызова функции fdopen:

```
FILE *s;
fileno(s);
```

Заметим в заключение, что в действительности библиотечная функция fileno не существует как функция. Дело в том, что контекст fileno(p) указан в определении макроподстановки, находящейся в файле stdio.h,. В результате обработки препроцессором исходного текста программы на языке последний сначала осуществит включение в файл, содержащий исходный текст программы, содержимое файла с именем stdio.h, а затем выполнит макроподстановку. В результате этого контекст fileno(s) будет заменен значением поля _file того элемента массива структуры _iob, который соответствует специфицированному аргументу функции fileno файла.

ВЫПОЛНЕНИЕ ОПЕРАЦИЙ ВВОДА И ВЫВОДА

Библиотечные функции fread и fwrite могут быть использованы для выполнения операций ввода-вывода подобно тому, как используются в этих целях системные вызовы read и write соответственно. Ниже приведен пример использования функций fread и fwrite:

```
FILE *stream;
struct foo *ptr;
int n;
int nr = fread(ptr, sizeof(ptr), n, stream);
int nw = fwrite(ptr, sizeof(ptr), n, stream);
```

В результате выполнения функций fread и fwrite содержимое файла, специфицированного аргументом stream функции fread (который обрабатывается этой функцией как указатель на файл), и порции информации будут выведены из файла и помещены во внутренний буфер программы, местоположение которого в оперативной памяти специфицировано аргументом ptr, а размер специфицирован аргументом nitems. При этом размер каждой порции информации задается в виде числа объектов, тип которых совпадает с типом объектов, для указания на которые используется аргумент ptr. Формат вызова библиотечной функции fwrite полностью совпадает с форматом вызова функции fread, однако в результате ее выполнения информация будет выведена из специфицированного внутреннего буфера программы в специфицированный файл. Обе библиотечные функции возвращают в качестве своего значения число реально переданных порций информации в случае успешного завершения операции обмена и 0 в случае достижения конца файла или возникновения ошибки.

В качестве примера использования библиотечных функций fread и fwrite рассмотрим исходный текст простенькой программы на языке Си, назначение которой — копировать информацию со стандартного ввода на стандартный вывод:

```
char buffer[BLOCK];
main()
{
   int n;
   while((n = fread(buffer, sizeof(char), BLOCK, stdin)) > 0)
            fwrite(buffer, sizeof(char), n, stdout);
}
```

Из этого примера хорошо видно, что наличие дополнительного уровня буферизации, реализуемого функциями библиотеки SIO и потому открытого для пользователя, повышает эффективность программирования. В результате этого число необходимых операторов вызова функций fread и fwrite станет минимальным.

ПОЗИЦИОНИРОВАНИЕ В ФАЙЛЕ УКАЗАТЕЛЯ ЧТЕНИЕ-ЗАПИСИ

Как Вы помните, переместить указатель чтения-записи открытого файла можно осуществлением системного вызова lseek. В состав библиотеки SIO входит функция fseek, назначение которой совпадает с назначением системного вызова lseek. Ниже приведен формат вызова библиотечной функции fseek:

```
FILE *stream;
long offset;
```

```
int  ptrname;
int es = fseek(stream, offset, ptrname);
```

В результате выполнения библиотечной функции fseek указатель чтения-записи открытого файла, специфицированного аргументом stream (он обрабатывается функцией fseek как указатель на файл), будет перемещен описанным ниже способом. Значение смещения (в байтах) указателя чтения-записи специфицируется аргументом offset, а база смещения указателя чтения-записи с помощью аргумента ptr следующим образом: 0 — смещение от начала файла; 1 — смещение от текущего положения указателя чтения-записи; 2 — смещение от конца файла.

```
#include <stdio.h>
int getrec(buffer, n, ptr);
char *buffer;
long n;
FILE *ptr;
{
     int es;
     if(es = fseek(ptr, n, 0)) != 0)
            return(es);
     es = fread(buffer, n, 1, ptr);
     return(es);
}
```

Еще одна библиотечная функция ftell обрабатывает свой единственный аргумент как указатель на открываемый файл. Возвращаемое значение функции ftell представляет собой целое число, специфицирующее смещение (в байтах) текущего положения указателя чтения-записи относительно начала файла. Ниже приведен формат вызова функции ftell:

```
FILE *s;
long where ftell(s);
```

Если после выполнения функции ftell будет осуществлен вызов функции rewind

```
FILE *s;
rewind(s);
```

то результат последовательного осуществления этих двух вызовов библиотечных функций будет эквивалентен выполнению функции fseek, вызванной следующим образом:

```
fseek(s, 0L, 0);
```

В обоих случаях указатель чтение-записи будет установлен на начало файла.

В качестве примера использования описанных библиотечных функций рассмотрим исходный текст программы на языке Си, описанной в гл. 4, с помощью которой доступ к информации, хранящейся в файле, может быть осуществлен просто как к последовательности байт, например элементов массива типа char. Исходный текст этой новой версии программы

будет значительно проще и нагляднее, ибо весь механизм буферизации будет теперь реализован только в рамках библиотечных функций:

```
#include <stdio.h>
FILE *aopen(name, flag);
char *name;
int flag;
{
     switch(flag){
     case 0: /* только чтение */
             return(fopen(name, "r"));
     case 1: /* только запись ;/
             return(fopen(name, "w"));
     case 2: /* и чтение и запись */
             return(fopen(name, "r+"));
     default:/* ошибка */
             return(NULL);
     }
}
int aclose(fp)
{
     return(fclose(fp));
}
int agetb(fp, addr);
FILE *fp;
long addr;
{
         int es;
         if((es = fseek(fp, addr, 0)) != 0);
                 return(es);
         return(getc(fp));
}
int asetb(fp, addr, value)
FILE *fp;
long addr;
char value;
{
        int es;
        if((es = fseek(fp, addr, 0)) != 0);
                return(es);
        return(putc(value, fp);
}
```

Если Вы теперь попытаетесь сравнить эффективность этих двух версий программы, воспользовавшись для этого файлом, содержащим 100 X 512 байт информации, то обнаружите, что использование вместо системных вызовов библиотечных функций несколько повысило эффективность указанной программы.

УПРАВЛЕНИЕ МЕХАНИЗМОМ БУФЕРИЗАЦИИ

Как мы показали, всякое использование библиотечных функций fread и fwrite для выполнения операции ввода-вывода автоматически приводит к буферизации указанных операций. Механизм буферизации реали-

зован в рамках самих библиотечных функций и потому, на первый взгляд, неуправляем. На самом деле ОС UNIX предоставляет пользователю такую возможность: в состав библиотеки SIO входит функция setbuf, использование которой позволяет при желании специфицировать размер и местоположение внутренних буферов библиотечных функций fread и fwrite. Ниже приведен формат вызова библиотечной функции setbuf:

```
FILE *stream;
char *buf;
setbuf(stream, buf);
```

Опишем подробнее использование функции setbuf. Пусть файл, в который необходимо вывести (или ввести) информацию, был открыт с помощью библиотечной функции fopen, возвратившей указатель на файл stream. Если теперь осуществить вызов библиотечной функции setbuf, в качестве первого аргумента которой использован только что полученный указатель на файл, то все вызванные после этого функции fread и fwrite будут использовать для буферизации операций ввода-вывода информации в файл, специфицированный указатель на файл stream, область оперативной памяти, специфицированную вторым аргументом функции setbuf.

Обращаем Ваше внимание на важность указанной последовательности вызовов библиотечных функций fopen, setbuf и fread (или fwrite), ибо функции fread и fwrite при отсутствии специальных на то указаний (например, с помощью функции setbuf) сами резервируют себе с помощью системного вызова malloc область оперативной памяти, которая затем и используется для буферизации операций ввода-вывода. Если указатель, специфицированный с помощью аргумента buf, не указывает ни на какой файл (например, представляет собой символьную константу NULL), то операции ввода-вывода будут выполняться без буферизации.

Использование небуферизованных операций ввода-вывода может оказаться полезным при выводе в файл информации о возникновении аварийной ситуации. Дело в том, что при аварийном завершении пользовательского процесса все инициированные им операции ввода-вывода будут немедленно завершены, а все открытые им файлы немедленно закрыты, в результате чего сообщение об аварии может оказаться вообще не выведенным в файл (оно так и останется во внутреннем буфере функции fread или функции fwrite). Ниже приведен пример использования функции setbuf для отмены буферизации операций ввода-вывода информации в файл, специфицированный указателем на файл error:

```
main()
{
     . . . . .
     FILE *error;
     error = fopen("errfile", "w");
     setbuf(error, NULL);
     . . . . .
}
```

В состав библиотеки SIO входит также функция fflush, использование которой позволяет осуществить принудительное завершение операций ввода-вывода информации в файл, специфицированный ее единственным аргументом, обрабатываемым ею как указатель на файл. Вызов библиотечной функции fflush имеет формат

```
int fflush(sream);
```

В случае успешного завершения своего выполнения библиотечная функция fflush возвращает целое число 0; в противном случае она возвращает число, соответствующее символьной константе EOF, указанной в определении символьной замены, которое находится в файле stdio.h.

ОБРАБОТКА АВАРИЙНОЙ СИТУАЦИИ

Как Вы уже знаете, при аварийном завершении своего выполнения системные вызовы ОС UNIX возвращают целое число −1. Аналогично этому и библиотечные функции, входящие в состав библиотеки SIO, при аварийном завершении своего выполнения возвращают число, являющееся результатом выполнения символьной замены символьной константы NULL, определение которой находится в файле stdio.h (как правило, это целое число 0). Вместе с тем в состав библиотеки SIO входят три функции, назначение которых — анализ и управление аварийным завершением операций ввода-вывода[1]. Итак, пусть stream — указатель на файл, с которым осуществляется операция ввода-вывода информации. Опишем формат вызова трех упомянутых функций. Первая из них — функция, имеющая формат

```
feof(sream);
```

возвращает целое число, не равное 0, если в процессе выполнения последней операции ввода-вывода был достигнут конец файла и целое число 0 в противном случае. Следующая функция, имеющая формат вызова

```
ferror(stream);
```

возвращает целое число, не равное 0, если в процессе выполнения последней операции ввода-вывода возникла ошибка, и до тех пор, пока это состояние не будет сброшено с помощью библиотечной функции clearerr или пока файл, специфицированный указателем на файл stream, не будет закрыт, библиотечная функция ferror будет возвращать ненулевое значение. Итак, формат вызова библиотечной функции clearerr имеет вид

```
clearerr(stream);
```

---

[1] На самом деле в результате компиляции программы на языке Си, исходный текст которой содержит имена этих функций, препроцессор осуществит макроподстановку, определение которой содержится в файле strio.h. — *Прим. ред.*

ОПЕРАЦИИ ВВОДА-ВЫВОДА ОДНОГО СИМВОЛА

Одним из наиболее эффективных способов осуществления ввода или вывода в файл одного символа является использование библиотечных функций getc и putc соответственно. Приведем форматы их вызовов:

```
FILE *stream;
char c;
int nr = getc(stream);
int nw = putc(c, stream);
```

Эти функции осуществляют ввод или вывод одного байта информации в файл, специфицированный указателем на файл stream. При этом функция getc возвращает в качестве своего значения один введенный, а функция putc один выведенный в файл символ. Впрочем, понять механизм работы библиотечных функций getc и putc довольно просто, для этого достаточно изучить соответствующие определения макроподстановок, содержащиеся в файле stdio.h, ибо именно таким образом реализованы функции getc и putc.

Рассмотрим, например, опеределение макроподстановки, соответствующее функции getc. Легко заметить, что осуществление вызова библиотечной функции getc инициирует считывание очередного байта информации из файла (эту операцию выполняет функция _filbuf) лишь в том случае, если указанный буфер, специфицированный полем p→ptr сруктуры с шаблоном _iobuf, окажется пустым. Итак, если к моменту осуществления вызова функции getc внутренний буфер ввода, созданный ОС UNIX в результате открытия файла с помощью системного вызова open и соответствующий файлу, специфицированному указателем на файл stream, оказался пуст, то возвращаемым значением этой функции станет очередной байт информации из указанного файла; в противном случае им станет очередной байт информации из указанного внутреннего буфера ввода (значение счетчика числа байт информации, специфицированное полем p→ptr структуры с шаблоном _iobuf, будет в этом случае уменьшено на 1).

Перейдем теперь к рассмотрению определения макроподстановки, соответствующего функции putc. В этом случае все происходит аналогично, но в обратном порядке. Если к моменту осуществления вызова функции putc внутренний буфер вывода оказался полон, выводимый с ее помощью байт информации будет записан функцией _flsbuf непосредственно в файл; в противном случае он будет помещен во внутренний буфер вывода и значение счетчика числа байт информации увеличится на 1. Обращаем внимание читателя на то, что хотя функция getc осуществляет ввод из файла одного байта информации, ее возвращаемое значение имеет тип данных int и потому требует для своего хранения по крайней мере два байта оперативной памяти. Это обстоятельство может быть использовано для определения того, является возвращаемое значение очередным введенным байтом информации или свидетельствует об аварийном завершении операции ввода (например, достижении конца фай-

ла). Действительно, символьной константе EOF по определению символьной замены соответствует целое число −1, а получить такое возвращаемое значение функции getc в случае успешного завершения ее выполнения невозможно.

Ниже, в качестве примера использования функций getc и putc, приведен текст программы, осуществляющей копирование своего стандартного ввода на свой стандартный вывод:

```
main()
{
        int c;
        while((c = getc(stdin)) != EOF)
                putc(c, stdout);
}
```

В приведенном исходном тексте программы на языке Си переменная, используемая для хранения очередного копируемого байта информации, имеет тип данных int, что гарантирует успешность распознавания ситуации "достигнут конец файла". Вообще, задача ввода одного байта информации со стандартного ввода или соответственно вывода одного байта информации на стандартный вывод встречается настолько часто, что специально для ее решения в файл stdio.h были помещены еще два определения символьной замены вида

```
#define getchar()     getc(stdin)
#define putchar(x)    putc(x, stdout)
```

Напомним, что указатели на файлы stdin и srdout соответствуют файлам "стандартный ввод" и "стандартный вывод". Воспользуемся теперь функциями getchr и putchar и перепишем предыдущий пример следующим образом:

```
main()
{
        int c;
        while((c = getchar()) != EOF)
                putchar(c);
}
```

ОПЕРАЦИИ ВВОДА-ВЫВОДА СТРОКИ СИМВОЛОВ

Одной из наиболее популярных операций ввода-вывода является операция ввода-вывода строки символов. Напомним, что строкой символов называется конечная последовательность символов, имеющая специальный формат. Понятно, что, не имея специальных средств для осуществления подобных операций ввода-вывода, затруднительно выполнить даже такое ответственное действие, как вывод на экран пользовательского терминала диагностического сообщения.

В соответствии с принятыми в языке Си соглашениями формат строки символов может быть описан следующим образом: строкой символов

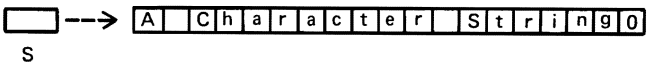

Рис. 5. 1. Логическая (байтовая) структура строки символов S

является конечная последовательность символов, завершаемая символом, имеющим код ASCII 0_8 (символ NULL). Из этого определения следует, что строки, содержащей хотя бы один символ NULL, не существует. На рис. 5. 1 проиллюстрировано приведенное определение строки символов. В состав библиотеки SIO входят две функции, getc и putc, с помощью которых можно ввести строку символов со стандартного ввода или вывести ее на стандартный вывод. Ниже приведены форматы вызовов обеих функций:

```
char *buffer;
char *gets(buffer);
int es = puts(buffer);
```

Итак, в результате осуществления вызова функции gets строка символов будет введена со стандартного ввода и помещена в область оперативной памяти, специфицированную ее аргументом, который функция getc обрабатывает как указатель на символы. Заметим, что в соответствии с принятыми в ОС UNIX соглашениями строкой символов считается последовательность символов, завершаемая символом, имеющим код ASCII 12_8 (символ NEWLINE) или символом логического конца файла, имеющим код ASCII 4_8 (символ EOT). После того как такая строка символов будет введена из файла с помощью функции getc, символ NEWLINE (или символ EOT) будет замещен символом NULL. Библиотечная функция putc осуществляет вывод строки символов, находящейся в области оперативной памяти, специфицированной указателем на символы buffer; при этом она замещает завершающий строку символ NULL символом NEWLINE.

Из приведенных описаний функций getc и putc можно сделать вывод, что основное из назначение — реализация диалога в интерактивных пользовательских программах. Проиллюстрируем сказанное одним небольшим примером: рассмотрим исходный текст программы на языке Си, которая в процессе всего выполнения осуществляет построчный ввод информации со стандартного ввода, конвертируя все имеющиеся в очередной введенной строке прописные буквы в строчные и выводя ее затем на стандартный вывод. Символьная константа TTYHOG, входящая в определение символьной замены, которое находится в файле sys/tty.h, специфицирует максимально возможный размер строки символов:

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

```
#include <sys/tty.h>
main()
{
     char *cp;
     char buffer[TTYHOG];
     while(gets(buffer) != NULL){
          for(cp = buffer; *cp; cp++)
               if(isupper(*cp))
                    *cp = tolower(*cp);

          puts(buffer);
     }
}
```

В этой программе использованы еще две функции: isupper и tolower, реализованные с помощью макроподстановок, находящихся в файле <type.h>. Функция isupper возвращает целое число, не равное 0, если ее аргумент — прописная буква, и 0 в противном случае. Функция tolower, аргумент которой является прописной буквой, возвращает один байт информации, содержащий соответствующую строчную букву. Для более детального изучения этого вопроса мы рекомендуем Вам обратиться к разд. type (3) [1].

ФУНКЦИИ fgets И fputs

Иногда перед программистом стоит задача разработать программу, которая осуществляла бы построчный обмен символьной информацией с файлом, не являющимся ни стандартным вводом, ни стандартным выводом. Для решения этой задачи могут быть использованы входящие в состав библиотеки SIO функции fgets и fputs. Форматы вызовов этих функций приведены ниже:

```
fgets(buffer< n, stream);
char *buffer;
int n;
FILE *stream;
fputs(buffer, stream);
char *buffer;
FILE *stream;
```

Библиотечная функция fgetc обеспечивает ввод строки символов из произвольного файла, открытого с помощью функции fopen. Ввод будет закончен либо после ввода n−1 символов, среди которых не было ни одного символа NEWLINE, либо после ввода символа NEWLINE. Строка символов, введенная с помощью этой библиотечной функции, будет дополнена символом, имеющим код ASCII 0_8. В качестве своего значения функция возвращает указатель на символы buffer, специфицирующий местоположение введенной строки символов в оперативной памяти. Библиотечная функция fputc обеспечивает вывод завершенной символом NULL строки символов в произвольный файл, открытый с помощью функции fopen, заменяя при этом символ NULL на символ NEWLINE.

Местоположение подлежащей выводу строки символов в оперативной памяти специфицируется с помощью указателя на символы s.

В качестве примера использования библиотечных функций fgets и fpurs рассмотрим приводимый ниже исходный текст программы на языке Си, представляющей собой простую версию команды ОС UNIX tr. В процессе своего выполнения рассматриваемая программа осуществляет построчное копирование своего стандартного ввода на свой стандартный вывод; при этом она выполняет контекстную замену всех символов вводимой строки, входящих также и в строку, специфицированную первым входным параметром программы, на соответствующий символ строки, специфицированной вторым входным параметром программы. Казалось бы естественным использовать в этой программе функции putc и getc вместо функций fputc и fgetc. Наш выбор объясняется тем, что использование функций fputc и fgetc позволяет осуществить ввод или вывод строки символов неограниченного размера.

Это обстоятельство может пригодиться, если нам захочется переназначить стандартный ввод или стандартный вывод программы на файл. Дело в том, что при осуществлении ввода строки из файла ОС UNIX вполне справедливо ожидать, что вводимая строка имеет размер, превышающий максимально возможный размер строки, специфицированный в исходном тексте программы символьный константой TTYHOG. Если строка символов вводится с пользовательского терминала, то вероятность такого события невелика. В то же время при нарушении ограничения на размер строки, введенной с терминала, драйвер терминала осуществляет циклическое заполнение буфера ввода последовательными фрагментами вводимой строки, при этом размер каждого такого фрагмента равен TTYHOG−1 символов. Таким образом, в результате выполнения такой операции обмена в буфере ввода окажутся только последний фрагмент введенной строки, размер которого не превосходит TTYHOG−1 символов:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/tty.h>
#define MX_ASCII 256
char mt[MX_ASCII];
mtinit()
{
register int i;
for(i = 0; i < MX_ASCII; i++)
          mt[i] = (char)i;
}
main(ac, av)
char **av;
int ac;
{
     register char *ti, *to;
     char buffer[TTYHOG+1];
     ti = av[1];
     to = av[2];
```

```
        mtinit();
        do{
            mt[(int)(*ti++)] = *to++;
        }
        while(*ti);
        to = mt;
        while(fgets(buffer, TTYHOG, stdin) != NULL){
              for(ti = buffer; *ti; ti++)
                    *ti = to[(int)*ti];
              fputs(buffer, stdout);
        }
}
```

Заметим, что использование указателей и регистровых переменных значительно повышает эффективность программы.

ОПЕРАЦИИ ФОРМАТНОГО ВВОДА-ВЫВОДА

Рассмотрим теперь еще несколько функций из библиотеки SIO, основное назначение которых заключается в выполнении операций форматного ввода и форматного вывода. Использование таких функций позволяет разрабатывать программы, способные генерировать данные в заданном формате, а также осуществлять элементарный синтаксический анализ вводимой символьной информации уже на этапе ее ввода. Набор библиотечных функций, реализующих форматный ввод и форматный вывод, может быть условно разбит на две группы. Первая включает библиотечные функции, с помощью которых может быть осуществлен форматный вывод информации: printf, fprintl и sprintf. Ко второй группе относятся библиотечные функции, с помощью которых может быть осуществлен форматный ввод информации. Все эти библиотечные функции обрабатывают переменное число своих аргументов. Пример вызова функции printf с различным числом аргументов приведен ниже:

```
printf("%d", i);
printf("%d %d", i, j);
```

В результате осуществления первого вызова функции printf на стандартный вывод будет выведено одно десятичное число, а в результате осуществления второго вызова — два. Число своих аргументов функция printf определяет в результате лексического разбора так называемой форматной строки, под управлением которой эта функция производит форматные преобразования и вывод на стандартных вывод своих аргументов. Если число аргументов функции printf не соответствует числу спецификаций преобразования и формата, содержащихся в форматной строке, то в результате либо на стандартный вывод будет выведена бессмыслица, либо часть аргументов будет проигнорирована.

ФУНКЦИИ ФОРМАТНОГО ВЫВОДА

В состав библиотеки SIO входят три функции, реализующие форматный вывод информации на стандартный вывод:

```
printf(format, arguments ...);
fprintf(stream, format, arguments ...);
sprintf(buffer, format, arguments ...);
```

Библиотечные функции printf и fprintf осуществляют форматирование и вывод своих аргументов; первая на стандартный вывод, а вторая в файл, специфицированный ее первым аргументом stream, обрабатываемым как указатель на файл. В отличие от двух первых функций, третья, sprintf, осуществляет форматирование и вывод своих аргументов не в файл, а в область оперативной памяти, специфицированную указателем на символы s. Аргумент format обрабатывается всеми тремя библиотечными функциями как указатель на символы и специфицирует так называемую форматную строку. Форматная строка может содержать объекты двух типов: просто символы и спецификации преобразования и формата. Объекты первого типа не подвергаются никаким преобразованиям и просто копируются, объекты второго типа используются соответствующей библиотечной функцией для проверки корректности задания аргументов этой функции и преобразования их требуемым образом. Спецификация преобразования и формата представляет собой последовательность символов, начинающуюся с символа %; напосредственно за которым в спецификации преобразования и формата следует так называемое поле формата. Например, вызов функции вида

```
printf("hello world\n");
```

не содержит ни одной спецификации преобразования и формата. В результате его осуществления на стандартный вывод будет выведена строка символов hello world, завершаемая символом NEWLINE. А вызов функции printf вида

```
printf("i = %d \n", i);
```

содержит одну спецификацию преобразования и формата вида %. В результате выполнения соответствующей операции преобразования и формата на стандартный вывод будет выведено десятичное число, являющееся значением аргумента i в нормальной форме с десятичной точкой:

```
I = 101;
```

СПЕЦИФИКАЦИИ ПРЕОБРАЗОВАНИЯ И ФОРМАТА

Мы уже говорили, что форматная строка представляет собой строку символов, в которую могут входить спецификации преобразования и формата. Спецификация преобразования и формата представляет собой последовательность символов, начинающуюся с символа %. Каждой содержащейся в форматной строке спецификации преобразования и формата должен соответствовать один подлежащий выводу аргумент функ-

ции. Приведем теперь список основных спецификаций преобразования и формата:

%C — один символ;
%s — строка символов, завершенная символом системы ASCII;
%d — десятичное число, занимающее два байта;
%o — восьмеричное число, занимающее два байта;
%x — шестнадцатеричное число, занимающее два байта;
%u — целое десятичное число, без знака занимающее два байта;
%f — десятичное число с плавающей точкой одинарной или двойной точности в нормальной форме;
%e — десятичное число с плавающей точкой одинарной или двойной точности в экспоненциальной форме;
%g — десятичное число с плавающей точкой одинарной или двойной точности в нормальной форме с подавлением незначащих нулей;
%% — символ %.

Внимательно рассмотрев перечисленные выше спецификации преобразования и формата, можно сделать вывод, что спецификация преобразования и формата представляет собой последовательность двух символов, первый из которых — символ %, а второй — буква. На самом деле спецификация преобразования и формата может включать и цифры. Например, спецификация преобразования и формата вида

```
P9%
```

указывает на необходимость вывода шести цифр целого десятичного числа в нормальной форме, выровненных по правому краю поля формата. Например, в результате осуществления вызова функции printf следующим образом:

```
printf("%6d", 446);  - - - > _ _ _ 446
```

на стандартный вывод будет выведено

```
446
```

Вместе с этим в состав спецификации преобразования и формата может быть включен и символ —, который в этом случае специфицирует операцию выравнивания по левому краю поля формата. Например, в результате осуществления вызова функции printf следующим образом:

```
printf("%-6d", 446);  - - - > 446 _ _ _
```

на стандартный вывод будет выведено

```
446000
```

Выравнивание по левому краю поля формата может оказаться полезным при необходимости осуществить вывод двоичного шаблона, составленного из фиксированного числа бит. Например,

```
printf("%06d", 0173); - - - > 000173
```

Спецификация преобразования и формата может также содержать и символ ., который в случае использования спецификации преобразования и формата f позволяет специфицировать отдельно поле формата всего числа с плавающей точкой и поле формата его мантиссы. Например, в результате осуществления вызова функции printf вида

```
printf("%8.3f", f); - - - > 6354.123
```

на стандартный вывод будет выведено

6354.123

И наконец, еще одно замечание: в состав преобразования и формата может входить и символ l, использование которого позволяет вывести значение переменных типа данных long. Например, в результате осуществления вызова функции printf вида

```
printf("%lo", 666666666L); - - - > 4757103300
```

на стандартный вывод будет выведено

4757103300

ФУНКЦИИ ФОРМАТНОГО ВВОДА

В состав библиотеки SIO входят три функции, реализующие операции форматного ввода информации:

```
scanf(format, pointers ...);
fscanf(stream, format, pointers ...);
sscanf(buffer, format, pointers ...);
```

Перечисленные библиотечные функции можно охарактеризовать как "зеркальное отражение" функций printf, fprintf и sprintf соответсвенно. Действительно, если библиотечные функции, реализующие форматный вывод, осуществляют последовательно операцию преобразования и форматирования, а затем вывода в файл, то библиотечные функции форматного ввода производят прямо противоположные действия: сначала осуществляют операцию ввода из файла, а затем операцию преобразования и форматирования. Рассмотрим в качестве примера последовательность действий, выполняемых библиотечной функцией в процессе ввода строки вида

```
9 green bottles
```

Воспользуемся для осуществления форматного ввода этой строки следующим вызовом функции scanf:

```
char *colour, *items;
int n;
scanf("%d %s %s", &n, colour, items);
```

В процессе выполнения вызванная таким образом функция scanf поместит в область оперативной памяти, специфицированную выражением &n, символ 9; в область оперативной памяти, специфицированную указателем на символы colour , последовательность символов, составляющих строку green ; , а в область оперативной памяти, специфицированную указателем на символы items , последовательность символов, составляющих строку bottles.

Библиотечные функции форматного ввода осуществляют ввод, преобразование и форматирование своих аргументов, как и функции форматного вывода, под управлением форматной строки. Свою форматную строку библиотечные функции форматного ввода обрабатывают аналогично соответствующим функциям форматного вывода. Остается добавить, что алгоритм обработки форматной строки библиотечными функциями форматного ввода содержит по сравнению с алгоритмом обработки форматной строки библиотечными функциями форматного вывода ряд усложняющих его дополнений. Перечислим их.

Если форматная строка содержит символ 'пробел', или символ TAB, или символ NEWLINE, то соответствующий ему введенный символ будет преобразован в символ 'пробел' и затем помещен в соответствующую область оперативной памяти.

Если форматная строка содержит вне спецификации преобразования и формата печатный символ, то соответствующий ему введенный символ должен в точности с ним совпадать.

Если спецификация преобразования и формата содержит символ *, следующий непосредственно за символом %, то при осуществлении ввода форматное поле, соответствующее этой спецификации преобразования и формата, будет исключено из дальнейшей обработки.

Проиллюстрируем последнее утверждение на примере, для чего рассмотрим следующий вызов библиотечной функции scanf:

```
scanf("%d%*s%d", &i, &j);
```

В результате осуществления ввода с терминала с помощью такого вызова функции scanf строки вида

```
6 times 7
```

переменной i будет присвоено значение 9, а переменной f будет присвоено значение 7.

Если спецификация преобразования и формата содержит заключенную в квадратные скобки строку символов, то при осуществлении операции ввода в соответствии с такой спецификацией преобразования и формата операция будет завершена, как только библиотечная функция форматного ввода введет символ, не совпадающий ни с одним из символов строки, заключенной в квадратные скобки. Если же первым символом строки, заключенной в квадратные скобки, является символ ^, то операция вво-

да, осуществляемая в соответствии с такой спецификацией преобразования и формата, будет завершена как только библиотечная функция форматного ввода введет символ, совпадающий с одим из символов, заключенным в квадратные скобки строки. Например, в результате ввода с помощью следующего вызова библиотечной функции scanf:

```
scanf("%2d%*f%s %*[abcd]%[^0123]", &i, s1, s2);
```

строки

```
6632.97NAMEabbdefdg123
```

следующим переменным будут присвоены соответствующие значения:

```
i  = 66;
s1 = "NAME";
s2 = "efdg";
```

ЛИТЕРАТУРА

1. **B.W. Kernighan (1978), UNIX Programming - Second Edition, UNIX V7, volume 2a.**

Глава 6. ПРОЦЕССЫ И ПРОГРАММЫ

Представьте себе, что Вам удалось сделать рентгеновский снимок вычислительной системы, функционирующей под управлением ОС UNIX, вместе со всеми ее аппаратными средствами. Что Вы на нем увидите? Прежде всего, стучащих по клавиатурам своих терминалов пользователей ОС UNIX, быстро вращающиеся накопители на магнитной ленте и подергивающиеся головки накопителя на магнитном диске. Невидимые простому глазу перемещаются по коммуникациям вычислительной системы перемещаются почтовые отправления, наполняются и очищаются буферы системного кэша. Мгновенное состояние любой из активных в данный момент подсистем ОС UNIX можно рассматривать как взаимодействие ее с другой такой подсистемой. Исходя из этого, функционирование ОС UNIX можно достаточно полно описать как взаимодействие взаимосвязанных динамических объектов, называемых в соответствии с принятой в ОС UNIX терминологией *процессами.*

Что же такое процессы, как они порождаются, как завершают свое функционирование и как взаимодействуют друг с другом? Ответ на эти и другие вопросы дает эта глава.

ПРОЦЕССЫ ОС UNIX

До сих пор, используя термины "процесс" и "программа", мы не делали различия между объектами, ими обозначенными. В действительности в операционной среде ОС UNIX программа и процесс — это два самостоятельных объекта.

Основной функцией ОС UNIX является, без сомнения, распределение вычислительных, информационных и других ресурсов ЭВМ между выполняющимися в данный момент пользовательскими программами, иначе говоря пользовательскими процессами. Таким образом, каждой пользовательской программе в ОС UNIX ставится в соответствие по крайней мере один пользовательский процесс.

Пользовательский процесс представляет собой совокупность выполняющейся пользовательской программы и всех обрабатываемых ею данных. Благодаря своей динамичности процесс реализует выполнение пользовательской программы, которая в данный момент выполняется в его рамках, а также управление ее взаимодействием с ОС UNIX. Проиллюстрируем это определение с помощью одной аналогии. Рассмотрим элементарную схему функционирования живой клетки: протоплазма клетки, основным назначением которой является обеспечение возможности развития и деления ядра клетки, по своей роли напоминает пользовательскую программу, в то время как ядро клетки, определяющее саму возможность развития и видоизменения последней, больше всего напоминает в нашей аналогии пользовательский процесс. Остается сделать лишь одно уточнение — помимо данных и исполняемого кода пользовательской программы "жизнеспособность" процесса обеспечивается целым рядом структур данных, генерируемых и поддерживаемых ОС UNIX (например, буферы системного кэша).

Итак, пользовательская программа представляет собой некую структуру данных, создаваемую пользователем с целью решения некоторой задачи; она состоит из некоторого числа байт информации, которые в определенном порядке вызываются на центральный процессор ЭВМ (ЦПУ) и исполняются им как инструкции (некоторые из них при этом обрабатываются как используемые указанными инструкциями данные).

Поскольку в ОС UNIX каждый подлежащий распределению ресурс связывается с процессом, а не с конкретной пользовательской программой, то решение задачи распределения ресурсов сводится к распределению ресурсов между процессами и поэтому позволяет с легкостью передавать ресурс (например, информационный) одной программы другой при условии, что обе они выполняются в рамках одного и того же процесса. Продолжая эти рассуждения, можно прийти к заключению, что логично и саму пользовательскую программу отнести к разряду распределяемых ОС UNIX ресурсов[1].

ПОРОЖДЕНИЕ ПРОЦЕССА

После того как ОС UNIX загружена, раскручена и переведена в многопользовательский режим функционирования, всякий новый процесс может функционировать только как процесс-потомок некоторого уже существующего процесса, называемого в этом случае процессом-предком.

---

[1] Передача этого ресурса реализуется в результате осуществления системного вызова fork.

Новый процесс (процесс-потомок) может быть порожден только с помощью системного вызова fork, а порожденный таким образом процесс представляет собой в первый момент точную копию своего процесса-предка. Последнее означает, что в обоих процессах открыты в одном и том же режиме одни и те же файлы и имеются совершенно одинаковый код и внутренние данные. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения системного вызова fork получает число 0, а процесс-предок — целое положительное число (идентификатор процесса-потомка, т. е. его уникальный номер). Последнее обстоятельство дает возможность определить, является данный процесс процессом-потомком или процессом-предком. Ниже приведен фрагмент программы на языке Си, иллюстрирующий сказанное:

```
int pid;
if(pid = fork() != 0){
     if(pid > 0){
               /*  процесс- предок  */
     } else {
               /* fork завершен аварийно */
     }
} else {
               /* процесс-потомок  */
}
```

ИДЕНТИФИКАТОРЫ ПРОЦЕССОВ

Каждый пользовательский процесс ОС UNIX имеет уникальный номер, называемый идентификатором процесса и однозначно его специфицирующий. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить необходимую информацию о функционирующих в данный момент процессах. Для того чтобы получить собственный идентификатор процесса, пользовательский процесс должен осуществить системный вызов getpid следующим образом:

```
int mypid;
mypid = getpid();
```

Получить идентификатор процесса-предка можно с помощью системного вызова geppid так:

```
int ppid;
ppid = getppid();
```

Итак, идентификатор процесса представляет собой целое число в диапазоне от 1 до 30 000, с помощью которого процесс однозначно идентифицируется ядром ОС UNIX. Часть ядра ОС UNIX, которая ответственна за генерацию идентификаторов процессов для вновь порождаемых процессов, гарантирует уникальность генерируемых идентификаторов процес-

сов, а это означает невозможность возникновения такой ситуации, при которой два или более процесса имеют одинаковые идентификаторы процесса.

ГРУППЫ ПРОЦЕССОВ

Вместе с идентификатором процесса каждому процессу ОС UNIX приписывается также идентификатор группы процессов. В группу процессов объединяются все процессы, являющиеся процессами-потомками одного и того же процесса, порожденного процессами с идентификатором процесса 1 для данного пользовательского терминала. Процесс с идентификатором процесса 1 порождается процессом с идентификатором процесса 0 на этапе раскрутки ОС UNIX и функционирует в соответствии с программой, находящейся в файле с именем /etc/init. Ни один процесс ОС UNIX не может изменить полученного им идентификатора группы процессов; идентификатор группы процессов процесса-потомка обычно совпадает (наследуется им) с идентификатором группы процессов его процесса-предка. Таким образом, любой процесс ОС UNIX оказывается членом некоторой группы процессов и тем самым ассоциируется с определенным пользовательским терминалом. Следствием такого подхода является возможность получения всеми процессами одной и той же группы процессов всех сигналов, посылаемых любому из ее членов вводом с терминала управляющего символа. Особенно важным это обстоятельство становится в том случае, когда терминал пользователя подключен к ЭВМ по телефонному каналу с помощью модема, так как любая неисправность телефонного канала может привести к невозможности дальнейшего управления группой процессов, ассоциированной с данным терминалом со стороны пользователя. Чтобы избежать подобной ситуации, в ОС UNIX принято соглашение, согласно которому в случае нарушения связи с некоторым терминалом ОС UNIX эмулирует ситуацию выдачи с этого терминала управляющего символа, инициирующего аварийное завершение процесса.

Как мы уже говорили, идентификатор группы процессов любого процесса-потомка, как правило, совпадает с идентификатором группы процессов его процесса-предка. Вместе с тем в рамках одной группы процессов может быть "организована" новая группа процессов, для чего необходимо использовать системный вызов setpgrp, имеющий формат

```
int newgroup;
newgroup = setpgrp();
```

В результате осуществления некоторым процессом системного вызова setpgrp все порожденные им процессы-потомки получат идентификатор группы процессов, равный идентификатору процесса, осуществившего указанный системный вызов (возвращаемым значением системного вызова setpgrp является идентификатор такого процесса). Если же процессу необходимо получить собственный идентификатор про-

цесса, то он должен осуществить системный вызов getpgrp, который имеет формат

```
int mygroup;
mygroup = getpgrp();
```

Более подробное описание групп процессов Вы найдете в следующей главе, посвященной вопросам коммуникации процессов.

ВЫПОЛНЕНИЕ ПОЛЬЗОВАТЕЛЬСКОЙ ПРОГРАММЫ

Как мы уже говорили, исполняемый код и данные, находящиеся в пользовательской программе, составляют основную часть всякого процесса. Если процесс, в рамках которого выполняется некоторая пользовательская программа, осуществляет системный вызов fork, порождая тем самым процесс-потомок, то последний будет функционировать в соответствии с той же программой, что и его процесс-предок до тех пор, пока процесс-потомок не сменит определяющую его функционирование программу с помощью системного вызова exec. В результате осуществления процессом системного вызова exec с этого момента его функционирование будет определяться программой, исполняемый код которой находится в файле, специфицированном своим полным именем, которое указано в качестве единственного аргумента системного вызова exec.

Итак, процесс-потомок сменил определяющую его функционирование программу, но сохранил унаследованные им от процесса-предка открытые файлы (кроме файлов-синонимов) и текущий каталог. Если при выполнении системного вызова exec возникла аварийная ситуация, связанная, например, с тем, что файл, имя которого было указано в качестве аргумента системного вызова exec, не существует или же содержащаяся в нем информация не является исполняемым кодом, то управление будет возвращено программе, которая выполнялась в рамках данного процесса до осуществления им системного вызова exec. Если же системный вызов exec был осуществлен успешно, процесс сменит программу, определяющую его функционирование, и управление будет передано новой программе, как если бы был выполнен вызов подпрограммы.

В качестве примера рассмотрим вызов на выполнение команды ls, для чего достаточно ввести с терминала

```
ls -l
```

В результате процесс, в рамках которого выполняется интерпретатор команд shell, осуществит системный вызов fork и тем самым породит новый процесс, который прежде всего осуществит системный вызов exec, указав в качестве его параметра имя файла, содержащего исполняемый код команды ls. Затем процесс-потомок обработает переданные ему ОС UNIX параметры введенной с терминала команды ls и выведет на терми-

нал содержимое текущего каталога в "длинном" формате. Надо заметить, что стандартный ввод, стандартный вывод и стандартный протокол интерпретатора команд shell всегда назначены на пользовательский терминал. Процесс, в рамках которого выполняется команда ls, наследует все открытые файлы своего процесса-предка, и,следовательно, стандартный ввод, стандартный вывод и стандартный протокол команды ls тоже назначены на пользовательский терминал. Таким образом, при необходимости вывести результат работы команды ls в некоторый файл, необходимо предпринять специальные действия — переназначить его стандартный вывод на файл. Для иллюстрации рассмотрим теперь следующий фрагмент программы на языке Си:

```
if((child_id = fork()) != 0){
     /* процесс-предок  */
     dead_id = wait(&return_status);
}else{
     /* выполнить команду ls  */
     execl("/bin/ls", "ls", "-l", 0);
     exit(1);   /* выполнение exec завершено аварийно */
}
```

Приведем еще один фрагмент программы на языке Си, который иллюстрирует один из способов переназначения стандартного вывода команды ls на файл:

```
if((child_id = fork()) != 0){
       /* процесс, в рамках которого выполняется shell */
       dead_id = wait(&return_status);
}else{
       /* переназначение стандартного вывода */
       fd = open(output_file);
       close(1);
       fcntl(fd, F_DUPFD, 1);
       close(fd);
       execl("/bin/ls", "ls", "-l", 0);
}
```

Использование стандартного ввода, стандартного вывода и стандартного протокола уже были описаны в гл. 4. Напомним только, что при последовательном осуществлении системных вызовов open, close и gcntl необходимо быть чрезвычайно осторожным и убедиться в том, что файл, в который Вам необходимо вывести результат работы команды ls, действительно имеет дескриптор файла 1.

ФОРМЫ СИСТЕМНОГО ВЫЗОВА exec

В ОС UNIX существует несколько форм системного вызова exec, отличающихся именами, использующихся при различных обстоятельствах, но в любом случае осуществляющих смену программы, определяющей функционирование данного процесса. Приведем форматы этих системных вызовов:

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;
execv(name, argv)
char *name, *argv[];
execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];
execve(name, argv, envp)
char *name, *argv[], *envp[];
```

Здесь аргумент name имеет тип данных указатель на символы и специфицирует имя файла, содержащего исполняемый код программ, а аргументы arg0, arg1, . . . , которые также имеют тип данных указатель на символы специфицируют аргументы этой программы, передаваемые ей при вызове ее на выполнение.

Системный вызов execl используется, как правило, в том случае, когда вызываемая на выполнение программа имеет фиксированное число аргументов. При этом аргумент arg0 специфицирует имя файла, содержащего вызываемую на выполнение программу, а в качестве последнего аргумента обязательно должен быть указан символ 0. Например, рассмотрим вызов на выполнение команды с помощью системного вызова execl:

```
execl("/bin/ls", "ls", "-l", 0);
```

Вы, наверное, обратили внимание на то, что в качестве аргумента name использовано абсолютное полное имя файла, содержащего команды ls, в противном случае поиск файла будет осуществляться только в текущем каталоге.

Системный вызов execv, наоборот, используется чаще всего в том случае, когда число аргументов вызываемой на выполнение программы заранее не известно. При этом аргумент name системного вызова специфицирует абсолютное полное имя файла, содержащего исполняемый код программы, а аргумент argv, имеющий тип данных массив указателей на символы, специфицирует весь список аргументов вызываемой на выполнение программы (последним элементом этого списка должна быть строка, содержащая только символ 0). В качестве примера рассмотрим вызов на выполнение команды ls с помощью системного вызова execv:

```
char *pv[] = {
            "cc",
            "-o",
            "fred",
            "fred.c",
            0
            };
main()
{
     . . . . .
     execv("/bin/cc", pv);
     . . . . .
}
```

КОНТЕКСТ ПРОЦЕССА

В гл. 2 мы уже говорили о том, что любая программа на языке Си должна содержать функцию main, формат объявления которой, как правило, имеет следующий вид:

```
main(argc, argv, envp)
int argc;
char *arghv[], *envp[];
```

В таком случае аргумент argc имеет тип данных int и специфицирует общее число параметров, передаваемых программе при вызове ее на выполнение, а аргумент argv имеет тип данных массив указателей на символы и специфицирует передаваемые параметры. При этом элемент argv[0] указанного массива содержит имя файла, содержащего вызываемую на выполнение программу. Поскольку строка argv[argc] в соответствии с принятыми в ОС UNIX соглашениями всегда содержит единственный символ — символ 0, то массив указателей argv, очевидно, может быть успешно и без всяких изменений использован в качестве второго аргумента системного вызова execv, осуществленного для вызова на выполнение программы, формат объявления функции main которой приведен выше.

Аргумент envp, называемый контекстом процесса, также представляет собой массив указателей на символы, последний элемент которого является строкой, содержащей только символ 0. Все остальные элементы массива указателей, специфицированного аргументом envp, представляют собой завершаемые символом 0 строки символов и имеют некоторый стандартный формат. Типичным примером элемента массива указателей envp является строка

```
HOME = /usr/spg/martin
```

Как хорошо видно из этого примера, элемент массива строк, специфицированного аргументом envp функции main, представляет собой последовательность имени внутренней переменной интерпретатора команд shell (переменной, имеющей для интерпретатора команд shell специальный смысл), символа = и строки символов, являющейся значением указанной внутренней переменной интерпретатора команд shell[1].

Напомним некоторые наиболее часто используемые переменные: HOME — внутренняя переменная интерпретатора команд shell, значением которой является абсолютное полное имя текущего каталога; PATH — внутренняя переменная интерпретатора команд shell, значением которой является список каталогов, в которых интерпретатор команд shell осуществляет поиск файла, содержащего вызванную на выполнение программу в том случае, если указанное в командной строке имя файла не содержит ни одного символа /; TERM — внутренняя переменная интерпретатора команд shell, значение которой специфицирует тип используе-

[1] Более подробное описание внутренних переменных интерпретатора команд shell можно найти в разделе shell(1) документа Unix Programmers Manual.

мого терминала (используется экранными редакторами текста и прикладными программами, осуществляющими управление экраном терминала). Текущие значения внутренних переменных интерпретатора команд shell автоматически используются системными вызовами execl и execv для генерации контекста порождаемого процесса.

Вместе с тем, пользователь имеет возможность повлиять на генерируемый контекст порождаемого процесса. Для этого вместо системных вызовов execl или execv необходимо воспользоваться системными вызовами execle или execve, которые отличаются от системных вызовов execl и execv наличием у них третьего аргумента envp (обрабатываемого ими как массив указателей на символы), содержащего значения внутренних переменных интерпретатора команд shell и имеющего тот же формат, что и аргумент encp функции msin. Таким образом, при смене программы, определяющей функционирование процесса, с помощью системного вызова execle или системного вызова execve может быть изменен и контекст процесса. Для иллюстрации сказанного рассмотрим фрагмент программы на языке Си, который может быть использован для вызова на выполнение интерпретатора команд shell так, чтобы его внутренние переменные (или, что то же самое, контекст процесса, в рамках которого он выполняется) имели бы следующие значения:

```
HOME  = /usr/spg/martin
SHELL = /bin/csh
PATH  =:/usr/spg/martin/bin:/usr/ucb:/bin:/usr/bin
TERM  = tvi950
USER  = martin
MAIL  = /usr/spool/mail/martin
```

Итак, соответствующий фрагмент программы на языке Си имеет вид

```
char *envp[] = {
     "HOME  = /usr/spg/martin",
     "SHELL = /bin/csh",
     "PATH  = :/usr/spg/martin/bin:/usr/ucb:/bin:/usr/bin",
     "TERM  = tvi950",
     "USER  = martin",
     "MAIL  = /usr/spool/mail/martin",
     0
     };
main(){
      . . . . .
      execle("/bin/sh", "sh", envp);
      . . . . .
}
```

Чтобы убедиться в том, что контекст процесса, в рамках которого выполняется интерпретатор команд shell, действительно стал таким, как мы хотели, рассмотрим значение глобальной системной переменной environ, имеющей тип данных массив указателей на символы и используемой ОС UNIX для хранения текущего значения контекста процесса. Если процесс, которому необходимо получить текущее значение его кон-

текста процесса, не имеет соответствующих полномочий, то можно воспользоваться библиотечной функцией getenv, вызов которой имеет следующий формат:

```
char *getenv(name)
char *name;
```

Здесь аргумент name имеет тип данных указатель на символы и представляет собой строку, содержащую имя внутренней переменной интерпретатора команд shell. В результате выполнения такого вызова функция getenv осуществит контекстный поиск в массиве строк, который специфицирован глобальной системной переменной environ, строки, содержащей слева от символа = контекст, специфицированный ее аргументом, т. е. строки вида

```
name = value
```

В случае удачного завершения контекстного поиска возвращаемым значением библиотечной функции getenv является указатель на символы, специфицирующий строку, которая содержит контекст, находящийся справа от символов = в строке, полученной в результате указанного контекстного поиска,в противном случае возвращается целое число 0. В нашем примере этот контекст обозначен value. Рассмотрим теперь в качестве примера две небольшие программы на языке Си. Обе эти программы получают текущие значения контекстов процессов, в рамках которых они выполняются, после чего первая выводит на стандартный вывод указанное значение контекста процесса в формате, используемом командой set интерпретатора команд shell, а вторая выводит на свой стандартный вывод только полученные значения внутренних переменных интерпретатора команд shell.

```
/*
 *   Эта программа выводит  на терминал контекст процесса
 */
extern char **environ;
main()
{
     char **ep = environ;
     while(*ep)
          printf("%s\n", *ep++);
}
/*
 *   Эта программа выводит на терминал
 *   внутренние переменные интерпретатора команд shell
 */
extern char *getenv();
main(argc, argv)
int argc;
char *argv[];
{
     printf("%s\n", getenv(argv[1]));
}
```

СПИСОК ПОИСКА

Рассмотрим еще два системных вызова — execlp и execvp. Единственное отличие этих системных вызовов от системных вызовов execl и execv соответственно заключается в том, что их использование избавляет пользователя от необходимости специфицировать вызываемую на выполнение программу абсолютным полным именем файла, содержащим исполняемый код этой программы. Дело в том, что оба этих системных вызова при выполнении используют внутреннюю переменную интерпретатора команд shell — PATH, значением которой, как известно, является список имен каталогов ОС UNIX, где интерпретатор команд shell осуществляет поиск файла, содержащего вызываемую на выполнение программу, иначе говоря, системные вызовы execlp и execvp в некотором смысле дублируют действия интерпретатора команд shell. Переменная PATH представляет собой строку, содержащую последовательность абсолютных полных имен каталогов ОС UNIX, отделенных друг от друга символом :. Сразу после входа пользователя в ОС UNIX эта внутренняя переменная интерпретатора команд shell, выполняющегося в рамках процесса, порожденного для пользовательского терминала, принимает начальное значение: /bin/:usr/bin и сохраняет его таким до тех пор, пока пользователь сам его не изменит.

Итак, в результате осуществления системного вызова execle или системного вызова execve будет выполнен последовательный поиск файла, содержащего вызываемую на выполнение программу в каталогах, имена которых перечислены в списке, являющемся значением внутренней переменной интерпретатора команд shell — PATH. Если найденный в результате этого файл окажется не объектным, а текстовым файлом, к которому, тем не менее, разрешен доступ по выполнению, то на выполнение будет вызван интерпретатор команд shell, содержащийся в файле /bin/sh, которому и будет передан в качестве входного параметра найденный файл. Таким образом, вызов на выполнение программы, находящейся в файле, местоположение которого в файловой системе не известно, пользователь может попытаться осуществить с помощью системного вызова execlp или системного вызова execvp, для чего достаточно поместить в исходном тексте программы на языке Си строку

```
execlp("prog", ..., 0);
```

ЗАВЕРШЕНИЕ ФУНКЦИОНИРОВАНИЯ ПРОЦЕССА

Если процесс в результате своего функционирования выполнил до конца программу, определяющую его функционирование, или же другой, имеющий соответствующие полномочия процесс прервал его функционирование, то ОС UNIX выполнит очистку некоторых областей оперативной памяти, например буферов ввода-вывода, исключит его из планирования оперативной памяти и времени и тем самым процесс прекратит свое существование.

Обычно причиной завершения функционирования процесса становится либо осуществление программой, выполняющейся в рамках этого процесса, системного вызова exit, либо выполнение оператора return, входящего в состав функции main этой программы. В некоторых случаях причиной может послужить получение процессом так называемого сигнала, посланного ему другим процессом (подробно механизм сигналов будет рассмотрен в следующей главе).

Итак, системный вызов exit имеет следующий формат:

```
exit(status);
```

В результате осуществления системного вызова exit все открытые процессом или унаследованные им файлы будут закрыты, а процессу-предку (в том случае, если он в свое время осуществил системный вызов wait) будет возвращено значение, хранящееся в младшем байте целой переменной status, специфицированной единственным аргументом системного вызова exit.

Если процесс-потомок завершил свое существование, то его процессу-предку посылается сигнал, указывающий последнему (если он еще функционирует) на то, что один из порожденных им процессов-потомков завершился. Печальна судьба процессов-предков ОС UNIX: основную часть своей жизни они проводят в ожидании смерти собственных детей и это скорбное известие получают в виде возвращаемого значения системного вызова wait.

СИСТЕМНЫЙ ВЫЗОВ wait

В результате осуществления процессом системного вызова wait функционирование процесса приостанавливается вплоть до момента завершения порожденного им процесса-потомка — процесс как бы "засыпает". По завершении процесса-потомка процесс-предок пробуждается и в качестве возвращаемого значения системного вызова wait получает идентификатор завершившегося процесса-потомка, что позволяет процессу-предку, имевшему более одного процесса-потомка, определить, какой же из его процессов-потомков завершился. Аргумент системного вызова wait представляет собой указатель на целую переменную status, которая после завершения выполнения этого системного вызова будет содержать в старшем байте код завершения процесса-потомка, установленный процессом-потомком в качестве аргумента системного вызова exit, а в младшем — индикатор причины завершения процесса-потомка, устанавливаемый ядром ОС UNIX. Значение, содержащееся в младшем байте переменной status, называют системным кодом завершения процесса, а значение, содержащееся в ее старшем байте, — пользовательским кодом завершения процесса. В соответствии с принятыми в ОС UNIX соглашениями нулевое значение системного кода завершения процесса указывает на его обычное завершение (например, в результате осуществления им системного вызова exit), а нулевое значение системного кода завершения процесса указывает на аварийное завершение процесса. Если же

процесс-потомок завершил свое функционирование вследствие получения сигнала, то пользовательский код завершения процесса будет иметь значение 0, а системный код — значение, равное номеру полученного процессом-потомком сигнала. Рассмотрим два примера использования системного вызова wait:

```
pid = wait(&status);
```

и

```
pid = wait(0);
```

Здесь pid — это переменная, имеющая тип данных int и используемая для хранения возвращаемого значения системного вызова wait, которое в обоих случаях будет положительным целым числом, равным идентификатору завершившегося процесса-потомка. Аргумент (&status) представляет собой адрес целой переменной status, содержащей пользовательский и системный коды завершения процесса-потомка. Заметим, что если процесс-предок, осуществивший системный вызов wait, не имел к этому моменту процессов-потомков, то "пробуждение" последует немедленно, а возвращаемое значение системного вызова wait окажется равным —1. Если же процесс предок имеет более одного процесса-потомка, то системный вызов wait должен быть осуществлен им столько раз, сколько процессов-потомков им порождено.

Для того чтобы подробнее рассмотреть использование системного вызова wait на практике, обратимся к проблеме вызова программы на выполнение в асинхронном режиме. Вызывая программу на выполнение в обычном (синхронном) режиме, пользователь вынужден ждать завершения ее выполнения, лишь после этого он получит возможность ввести со своего терминала следующую команду ОС UNIX. Такое правило объясняется тем, что при вызове программы на выполнение в синхронном режиме интерпретатор команд shell, породив новый процесс и передав ему в качестве входного параметра введенную командную строку, немедленно осуществляет системный вызов wait, а это означает, что следующий промптер[1] интерпретатора команд shell появится на экране терминала лишь после завершения процесса-потомка и в результате "пробуждения" процесса_предка, в рамках которого выполняется интерпретатор команд shell. Если теперь, вызывая на выполнение программу, дать указание интерпретатору команд shell, запрещающее ему осуществлять системный вызов wait (например, завершив для этого командную строку символом &), то последний, породив процесс-потомок и передав ему введенную командную строку, немедленно выведет на терминал свой промптер, показав тем самым свою готовность продолжать диалог. Такой режим вызова программы на выполнение называют асинхронным режимом. Например, результат выполнения следующего фрагмента программы на языке Си:

[1] Prompter — подсказчик. — *Прим. ред.*

```
if(for() == 0)
    exec(ls);
```

эквивалентен выполнению следующей команды интерпретатора команд shell:

```
ls &
```

ПРОЦЕССЫ ЗОМБИ

С использованием системного вызова wait связано возникновение одной довольно необычной ситуации — появлением так называемых зомби[1]. Представьте, что только что порожденный процесс-потомок завершил свое существование прежде, чем процесс-предок успел осуществить системный вызов wait. Реальность возникновения такой ситуации гарантируется тем, что вновь порожденный процесс-потомок имеет распределении ресурсов ЭВМ преимущество перед процессом-предком, и следовательно, будет вызван на процессор прежде него. Таким образом, если выполнение системного вызова exec, осуществленного процессом-потомком, закончилось аварийно, то весьма вероятно, что процесс-потомок завершится задолго до того, как процесс-предок будет вызван на процессор и осуществит системный вызов wait.

Чтобы описанная ситуация не стала тупиковой, в ОС UNIX приняты специальные меры, касающиеся процедуры завершения функционирования процессов. Как мы уже говорили, при завершении по тем или иным причинам некоторого процесса соответствующие ему исполняемый код и почти все данные разрушаются, а зарезервированная для их хранения оперативная память освобождается, сам же завершившийся процесс-потомок превращается в зомби и перемещается в страну забвения, где обитает до тех пор, пока его непосредственный процесс-предок не позаботится о нем. В распоряжении зомби остаются лишь те данные, которые могут понадобиться ему после осуществления непосредственным процессом-предком системного вызова wait. И наконец, после осуществления непосредственным процессом-предком системного вызова wait, зомби обретет покой и исчезнет. На рис. 6.1 приведено схематическое изображение описанной ситуации.

Если непосредственный процесс-предок по каким-либо причинам завершает свое функционирование раньше порожденных им процессов-потомков, оставив их "сиротами", то ситуация еще более усложняется. Для решения этой проблемы в ОС UNIX принято следующее соглашение: все непосредственные процессы-потомки завершившегося процесса-предка становятся непосредственными процессами-потомками некоторого процесса, который может быть назван *всеобщим предком*. Такое решение

[1] По поверьям западной Индии зомби — это телесная оболочка умершего человека, которая благодаря волшебной силе получила возможность двигаться и совершать различные действия. — *Прим. ред.*

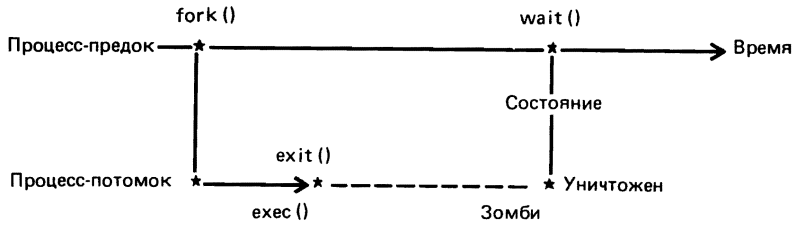

Рис. 6. 1. Процессы зомби

гарантирует, что для всякого процесса, кроме всеобщего предка, являющегося бессмертным, всегда существует процесс-предок, который позаботится о его бренных останках. Таким всеобщим предком является процесс с идентификатором процесса 1.

ЛОГИЧЕСКАЯ СТРУКТУРА ПРОЦЕССА ОС UNIX

Каждому процессу ОС UNIX соответствуют две отдельные структуры данных: первая называется программой и содержит исполняемый код программы, выполняющийся в рамках этого процесса, и совокупность обрабатываемых ею данных; вторая представляет собой совокупность данных, используемых ОС UNIX и обеспечивающих функционирование процесса в операционной среде. Если первая структура данных создается и модифицируется непосредственно пользователем, то доступ ко второй структуре пользователь может получить только косвенно, с помощью соответствующих системных вызовов.

ПРОГРАММА

Операционная система UNIX осуществляет динамическую загрузку пользовательских программ, предоставляя каждой из них некое логическое адресное пространство. Размер этого логического адресного пространства и способ его отображения на физическую оперативную память зависит от типа микропроцессора и архитектурных особенностей построенной на его базе ЭВМ, работающей под управлением конкретной версии ОС UNIX. Список современных (1984 г.) микропроцессоров, для которых существуют различные реализации ОС UNIX или подобных ей операционных систем, с каждым годом становится все длиннее. Он включает, в частности, такие широко известные микропроцессоры, как PDP-11, Motorola 68000, Z8000, Intel 8086, iapx186, iapx286, VAX и Natioval Semiconductor 32016. Тем не менее во всех этих случаях пользовательской программе предоставляется ограниченное и заранее известного размера логическое адресное пространство, иллюстрацию которого Вы найдете на рис. 6. 2.

Размер логического адресного пространства, предоставляемого пользовательским программам, определяется их потребностями в оперативной памяти и часто превосходит фактический объем оперативной памяти

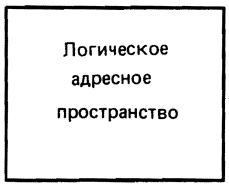

Рис. 6. 2. Структура логического адресного простанства программы

ЭВМ. В этом случае функционирование пользовательских программ гарантируется специальным механизмом управления памятью, который у большинства современных ЭВМ практически совпадает и имеет следующую логическую структуру: все логическое адресное пространство разбивается на отдельные сегменты; некоторые из сегментов считаются занятыми пользовательской программой и поэтому называются программными сегментами; другие считаются соответствующими несуществующей оперативной памяти ЭВМ, и попытки осуществления доступа к ним приводят к возникновению аварийной ситуации (более подробно этот вопрос будет рассмотрен в следующей главе). На рис. 6. 3 представлено схематическое изображение типичной структуры логического адресного пространства, два программных сегмента которого разделены сегментом, соответствующим несуществующей оперативной памяти ЭВМ.

Каждому программному сегменту приписывается ряд атрибутов, среди которых можно выделить атрибуты доступа. Значения атрибутов доступа определяют тип предоставляемого пользовательской программе доступа к программному сегменту. В частности, к одним сегментам пользовательская программа может осуществлять доступ только по чтению, а к другим — по чтению и по записи[1]. Как правило, программные сегменты пользовательской программы могут быть доступны ей и по чтению, и по записи. Аналогично этому программный сегмент, соответствующий стеку пользовательской программы, также имеет атрибуты доступа типа чтение-запись, так как он используется для осуществления передачи аргументов функций, хранения их локальных переменных и адресов возврата. Размер сегмента данных, как правило, не изменяется за время выполнения пользовательской программы, определяется на этапе компоновки программы и хранится в файле, содержащем исполняемый код программы.

После того как пользователь вызвал указанную программу на выполнение, введя со своего терминала соответствующую командную строку, информация об используемом ею сегменте данных обрабатывается ядром ОС UNIX, которое и реализует доступ к нему. В противоположность сегменту данных программный сегмент, соответствующий программно-

[1] Согласно принятой в ОС UNIX терминологии, сегментом данных называется фрагмент логического адресного пространства, не совпадающий со стеком ЭВМ и такой, что пользовательская программа может осуществлять к нему доступ по чтению и по записи. Сюда, в частности, может быть отнесен программный сегмент, содержащий исполняемый код программы, если последняя имеет полномочия на его модификацию.

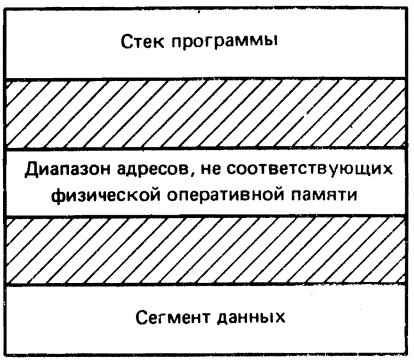

Рис. 6. 3. Логическая схема адресного пространства программы

му стеку, изменяется в процессе выполнения программы, увеличиваясь в размерах. Если при этом объем физической памяти ЭВМ невелик, то возможно возникновение ситуации, при которой рост программного стека приведет к разрушению сегмента данных программы, как это хорошо видно на рис. 6. 3.

СЕГМЕНТ ЧИСТОГО КОДА

Как мы уже говорили, обычно исполняемый код пользовательской программы и обрабатываемые ею данные относятся к одному программному сегменту, к которому разрешен доступ и по чтению, и по записи. Вместе с тем ОС UNIX предоставляет пользователям возможность разработки программ, использующих так называемый сегмент чистого кода. Исполняемый код такой программы относится к одному программному сегменту, называемому сегментом чистого кода; обрабатываемые ею данные — к другому, называемому сегментом данных. При этом к сегменту чистого кода возможен доступ только по чтению. Это последнее обстоятельство гарантирует возможность использования одного и того же сегмента чистого кода несколькими программами, функционирующими в рамках различных процессов.

Программный сегмент чистого кода, используемый несколькими программами, функционирующими в рамках различных процессов, принято называть разделяемым сегментом чистого кода. На рис. 6. 4 представлено схематическое изображение структуры логического адресного пространства с разделяемым сегментом чистого кода. Генерация содержимого разделяемого сегмента чистого кода осуществляется на этапе компоновки, и в это же время компоновщик устанавливает атрибуты доступа к нему. На большинстве современных ЭВМ[1] первый байт сегмента чисто-

[1] Архитектурные особенности некоторых микропроцессоров, например таких, как Intel 8086, обусловливают такое отображение логического адресного пространства на физическую оперативную память ЭВМ, при котором каждому программному сегменту ставится в соответствие отдельная область физической оперативной памяти ЭВМ.

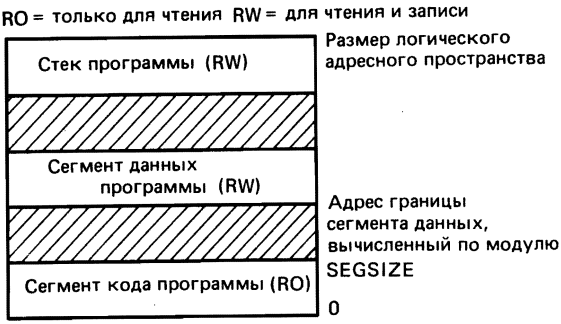

Рис. 6. 4. Логическая схема разделяемого адресного пространства

го кода имеет виртуальный адрес 0, а первый байт сегмента данных имеет виртуальный адрес, вычисляемый по следующей формуле:

(textsize + SEGSIZE - 1)/SEGSIZE

Здесь SEGSIZE — это минимальный размер фрагмента логического адресного пространства, к которому может быть осуществлен доступ только одного типа. Значение SIGSIZE определяется типом ЭВМ, работающей под управлением ОС UNIX, например, для ЭВМ PDP-11 оно равно 8 К байтам, а для ЭВМ VAX-11 — 512 байтам. Как легко видеть из приведенной формулы, виртуальный адрес первого байта сегмента данных несколько превосходит по значению виртуальный адрес последнего байта сегмента чистого кода.

Вообще разработка программ, построенных с сегментом чистого кода, вызвана не какой-либо функциональной необходимостью, а стремлением повысить эффективность разрабатываемой программы. Для доказательства последнего утверждения приведем несколько важных соображений:

использование сегмента чистого кода дает возможность избавиться от многократной откачки его на магнитный диск в процессе функционирования ОС UNIX, так как содержимое сегмента чистого кода не меняется; таким образом, процесс откачки и подкачки программных сегментов, неизбежный при использовании многозадачной многопользовательской вычислительной системы, становится вдвое эффективнее, ибо откачка программного сегмента может быть произведена лишь однажды, и всякий следующий раз будет осуществляться лишь подкачка программного сегмента. Это особенно важно при разработке интерактивных программ, программные сегменты которых обычно откачиваются в первую очередь, так как основное время своего функционирования интерактивные программы тратят на ожидание ввода с терминала;

использование сегмента чистого кода избавляет от необходимости копировать исполняемый код программы при порождении процессом, в рамках которого она выполняется, процесса-потомка; вместо этого логическое адресное пространство программы, выполняющейся в рамках процесса-потомка, отображается на ту же

область физической оперативной памяти ЭВМ, что и логическое адресное пространство программы, выполняющейся в рамках процесса-предка;

использование сегмента чистого кода при разработке программ, которые будут выполняться в рамках нескольких процессов (например, будут вызваны на выполнение несколькими пользователями), дают возможность сэкономить оперативную память ЭВМ, так как появляется возможность избежать копирования программного сегмента чистого кода, иначе говоря, вследствие разделения сегмента чистого кода. Примерами таких программ являются интерпретатор команд shell, редакторы текста и компиляторы с языков программирования.

ПОСТРОЕНИЕ СЕГМЕНТА ЧИСТОГО КОДА

Для того чтобы разрабатываемая программа использовала сегмент чистого кода, достаточно специфицировать в передаваемой компоновщику командной строке флаг —n. Например, в результате ввода с терминала командной строки вида

```
cc -o fred main.o a.o b.o
```

будет получена программа, использующая один неразделяемый программный сегмент, а в результате ввода командной строки

```
cc -o fred main.o a.o b.o
```

— программа, использующая отдельный программный сегмент для исполняемого кода и отдельный программный сегмент для данных. В последнем случае программный сегмент кода будет построен как сегмент чистого кода и может быть использован в качестве разделяемого сегмента чистого кода.

ФОРМАТ ИСПОЛНЯЕМОГО ФАЙЛА

Будет ли программа построена с сегментом чистого кода, определяется пользователем на этапе компоновки, а информация об этом хранится в первых 32 байтах исполняемого файла и имеет формат структуры, объявление которой находится в файле с именем a.out.h. На рис. 6.5 приведено объявление этой структуры. Вообще, информация, хранящаяся в исполняемом файле, может быть условно представлена в виде совокупности четырех последовательных секций, первую из которых мы только что описали. Вторая секция исполняемого файла представляет собой программный сегмент и содержит исполняемый код программы и обрабатываемые ею данные. Третья и четвертая секции соответственно содержат таблицу перемещения и таблицу символов программы. Понятно, что в результате загрузки программы в оперативную память фактически будет помещено содержимое лишь второй секции файла. Размеры всех четырех секций хранятся в первой секции исполняемого файла:

```
struct exec {
 int      a_magic; /* магическое число */
 unsigned a_text;  /* размер сегмента кода */
 unsigned a_data;  /* объем инициализированной части сегмента данных */
 unsigned a_bss;   /* объем неинициализированной части сегмента данных */
 unsigned a_syms;  /* размер таблицы символов */
 unsigned a_entry; /* точка входа */
 unsigned a_unused;/* объем свободной части программного сегмента */
 unsigned a_flag;  /* информация перемещения */
}
#define   A_MAGIC1    0407  /* запись и чтение */
#define   A_MAGIC2    0410  /* только чтение   */
```

В результате вызова на выполнение такого исполняемого файла, например с помощью осуществления системного вызова exec, в оперативной памяти ЭВМ будут созданы три программных сегмента: стека, кода (если программа построена с сегментом чистого кода) и данных. Значение поля a_magic структуры exec определяет, в частности, метод доступа к перечисленным программным сегментам. Например, значение, равное целому восьмеричному числу 407, специфицирует единый программный сегмент, к которому разрешен доступ и по чтению, и по записи, а сегмент данных следует непосредственно за сегментом кода. В то же время значение, равное восьмеричному числу 410, специфицирует сегмент чистого кода, к которому разрешен доступ только по чтению, и он может быть использован как разделяемый сегмент чистого кода. Виртуальный адрес первого байта сегмента данных в таком случае вычисляется по формуле, приведенной выше.

Более конкретную информацию о формате исполняемого файла ОС UNIX Вы найдете в разделе a.out (5) документа UPM.

УПРАВЛЕНИЕ ПАМЯТЬЮ

Как мы уже говорили, во время выполнения пользовательской программы ОС UNIX ставит ей в соответствие программные сегменты кода, данных и стека. Программный сегмент кода имеет фиксированный размер, который не может быть изменен, а размер программного сегмента стека автоматически увеличивается по мере необходимости. Размер программного сегмента данных, который первоначально специфицируется значениями полей a_data и a_bss, может быть изменен только с помощью системных вызовов brk и sbrk. Использование системного вызова brk позволяет специфицировать первый физический адрес области оперативной памяти ЭВМ, не используемой для сегмента данных. Формат системного вызова brk приведен ниже:

```
Normal (407)
```

Попытка осуществить доступ к адресу оперативной памяти, лежащему в диапазоне между физическим адресом, специфицированным значением аргумента addr и нижней границей программного сегмента стека, вызо-

вет возникновение аварийной ситуации. Приведем пример использования системного вызова brk:

```
char *brk(addr)
char *addr;
```

Использование системного вызова sbrk позволяет увеличить размер сегмента данных программы на число байт, специфицированное его аргументом; формат системного вызова sbrk приведен ниже:

```
#define round(x, s)         ((((int)(x)+s-1)/s)*s)
char *zone;
char *brk();
/*
 * Эта программа устанавливает указатель zone
 * на границу свободной области оперативной памяти
 */
zone = brk(round(end, SEGSIZE));
if((int)zone < 0){
   /* завершено аварийно */
      . . . . .
}
```

В случае удачного завершения оба системных вызова возвращают указатель на область оперативной памяти, которая может быть использована программой для хранения данных. Если же возвращемое значение любого из этих системных вызовов представляет собой целое число —1, то это означает, что расширение сегмента данных программы невозможно, например из-за отсутствия свободной оперативной памяти. В качестве примера, иллюстрирующего такое неудачное осуществление системного вызова sbrk, рассмотрим следующую программу на языке Си. Хотим предупредить Вас, что попытка вызвать на выполнение приведенную программу может привести к аварийному завершению функционирования ОС UNIX.

```
char *sbrk(incr)
unsigned incr;
```

ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ ВЫЗОВОВ brk И sbrk

Среди прикладных и инструментальных средств, функционирующих в среде ОС UNIX, существует немало таких, которые осуществляют динамическое распределение оперативной памяти, используемой для буферизации операций ввода-вывода. К ним, например, можно отнести компиляторы с различных языков программирования, редакторы текста и некоторые другие. Каждое из этих программных средств осуществляет резервирование некоторой области сегмента данных для реализации буферов ввода-вывода, однако непредсказуемость объема и характера обрабатываемой ими информации делает реальной необходимость динамического перераспределения оперативной памяти ЭВМ с целью увеличения

размера упомянутого программного сегмента. Для решения этой проблемы, впрочем, чаще всего используются не сами системные вызовы brk или sbrk, а использующие их стандартные подпрограммы. Две из них уже были описаны в гл. 3 — это функция malloc, имеющая формат вызова

```
char *sbrk();
main()
{
  char *tp;
  int top = 0;
  for(tp = sbrk(1024); tp != (char *) -1L; tp = sbrk(1024))
        top += 1024;
  printf("Maximum Memory is %ldkb\n", (top + 1023)/1024);
}
```

и функция free, формат вызова которой приведен ниже:

```
int *malloc(size);
unsigned size;
```

Первая из этих функций, как Вы помните, возвращает указатель на свободную область оперативной памяти, резервируя ее тем самым для использования пользовательской программой, вторая же освобождает зарезервированную таким образом область оперативной памяти, присоединяя ее к пулу свободной оперативной памяти ЭВМ. Использование организации свободной оперативной памяти в виде пула, аналогичного буферному пулу, позволяет динамически использовать одну и ту же область физической оперативной памяти для обеспечения функционирования нескольких процессов.

Добавим к сказанному, что использование стандартных подпрограмм избавляет пользователя от необходимости заботиться о тонкостях конкретных реализаций ОС UNIX, вызванных архитектурными особенностями используемой ЭВМ. Например, в случае использования ЭВМ PDP-11 для адресации байта могут быть использованы лишь четные значения адреса, и всякая попытка адресовать нечетный байт приведет к возникновению аварийной ситуации.

Еще две функции, realloc и calloc, являются в некотором смысле расширениями функции malloc. Ниже приведены форматы вызовов этих функций:

```
free(ptr)
char *ptr;
```

Библиотечная функция realloc обеспечивает в результате своего выполнения изменение размера области оперативной памяти, зарезервированной ранее с помощью библиотечной функции malloc.

```
char *realloc(ptr, size)
char *ptr;
unsigned size;
char *calloc(nelem, size)
unsigned nelem, size;
```

Библиотечная функция calloc возвращает в качестве своего значения указатель на свободную область оперативной памяти, размер которой достаточен для помещения в нее специфицированного аргументом nelem числа объектов, размер каждого из которых специфицирован аргументом elsize, или целое число 0, если свободная область оперативной памяти указанного размера отсутствует.

ЗАДАЧИ

1. Разработайте программу на языке Си, которая порождала бы процесс-потомок, и убедитесь в том, что все файлы, открытые процессом-предком, открыты и для процесса-потомка. Выясните, что произойдет, если процесс-потомок сменит определяющую его поведение программу. Убедитесь в том, что изменение процессом-потомком положения указателя записи-чтения некоторого открытого файла приведет к изменению его положения и для процесса-предка.

2. Попробуйте самостоятельно выяснить, как функционирует системный вызов fcntl.

3. Разработайте самостоятельно простую версию интерпретатора команд, такую, чтобы с ее помощью можно было вызывать на выполнение программу, специфицированную своим абсолютным полным именем как в синхронном, так и в асинхронном режимах.

4. Перепишите разработанную Вами версию интерпретатора команд, с тем чтобы вызываемую на выполнение программу можно было специфицировать ее относительным полным именем и интерпретатор команд автоматически обрабатывал текущее значение системной константы PATH.

5. Разработайте программу подсчета частоты вхождения слова в состав некоторого текстового файла для каждого хранящегося в этом файле слова (можете обратиться за помощью к гл. 4 книги).

ЛИТЕРАТУРА

1. B. W. Kernighan & D. M. Ritchie (1978). The C Programming Language, Prentice-Hall Software Series. (См. [5] в списке литературы к гл. 1).
2. Aho, Hopcroft & Ullman (1983). Data Structures and Algorithms, Addison-Wesley.
3. B. W. Kernighan & P. J. Plauger (1976), Software Tools, Addison-Wesley.

Глава 7. СРЕДСТВА И СПОСОБЫ КОММУНИКАЦИИ ПРОЦЕССОВ ОС UNIX

Функционирование ОС UNIX можно рассматривать как совместное функционирование нескольких взаимосвязанных процессов ОС UNIX. Такой подход делает очевидной роль средств и способов коммуникации процессов ОС UNIX. Представим себе, что нам необходимо разработать программу, которая решала бы сложную многофункциональную задачу. Нам кажется естественным реализовать ее в виде комплекса программ, каждая из которых решала бы некоторую относительно независимую

(функционально) часть указанной задачи; каждая такая программа выполнялась бы в рамках отдельного процесса ОС UNIX, а некое средство коммуникации связывало бы отдельные программы в единый комплекс. Весьма часто предложенное решение оказывается во многих отношениях выгоднее, нежели разработка одной большой и сложной программы. Среди хорошо известных и типичных примеров, где описанный подход более чем уместен, можно назвать задачу организации многопользовательского доступа к базе данных. Действительно, решение этой задачи требует обеспечения асинхронного выполнения нескольких программ, реализующих доступ к базе данных и, кроме того, достаточно корректного разрешения вопроса о разделении ресурсов ЭВМ.

Операционная система UNIX предоставляет своим пользователям два основных средства коммуникации процессов ОС UNIX — так называемые сигналы, представляющие собой средство синхронизации выполнения нескольких процессов, и каналы, реализующие собственно функцию передачи данных от одного процесса ОС UNIX другому.

СИГНАЛЫ

Сигнал — это программное средство, с помощью которого может быть прервано функционирование процесса ОС UNIX. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире. Как правило, получение некоторым процессом сигнала указывает ему на необходимость завершить свое функционирование, вместе с тем реакция процесса на принимаемый сигнал зависит от того, как сам процесс определил свое поведение в случае приема им данного сигнала. Например, получив сигнал, процесс может его просто проигнорировать или вызвать на выполнение некоторую программу (аналогичную программе обработки прерывания). В последнем случае, выполнив вызванную на выполнение программу, процесс сможет продолжить свое функционирование с той точки исходной программы, в которой процесс получил соответствующий сигнал.

Итак, процессы ОС UNIX могут принимать и обрабатывать сигналы; каждому сигналу ставятся в соответствие номер сигнала — целое положительное число из диапазона 1 — 19 и строковая константа, используемая для осмысленной идентификации сигнала. Поясним сказанное, для чего приведем фрагмент содержимого файла с именем <signal.h>, в котором хорошо видна взаимосвязь номера сигнала и идентифицирующей его мнемоники:

```
#define NSIG   20
/*
 * Сигналов может быть не более 32
 */
#define SIGHUP    1 /*разрыв связи                          */
#define SIGINT    2 /*прерывание                           */
#define SIGQUIT   3 /*аварийный выход                      */
#define SIGILL    4 /*неверная машинная инструкция         */
```

```
#define SIGTRAP   5 /*прерывание - ловушка                    */
#define SIGIOT    6 /*прерывание ввода-вывода                 */
#define SIGEMT    7 /*программное прерывание EMT              */
#define SIGFPE    8 /*авария при выполнении операции          *
                     *с плавающей точкой                      */
#define SIGKILL   9 /*уничтожение процесса                     */
#define SIGBUS   10 /*ошибка шины                             */
#define SIGSEGV  11 /*нарушение сегментации                   */
#define SIGSYS   12 /*ошибка выполнения системного вызова*/
#define SIGPIPE  13 /*запись в канал есть, чтения нет         */
#define SIGALRM  14 /*прерывание от таймера                   */
#define SIGTERM  15 /*программное прерывание                  */
#define SIGUSR1  16 /*определяется пользователем              */
#define SIGUSR2  17 /*определяется пользователем              */
#define SIGCLD   18 /*процесс-потомок завершился              */
#define SIGPWR   19 /*авария питания                          */

#define SIG_DFL (int(*)())0 /* все установки: "по умолчанию" */
#define SIG_IGN (int(*)())1 /* игнорировать этот сигнал      */
```

Как Вы, наверное, заметили, некоторые из перечисленных здесь сигналов (например, SIGALARM) соответствуют таким событиям, возможность возникновения которых не зависит от архитектурных особенностей используемой ЭВМ; существование других (например, SIGIOT, SIGEMT, SIGTRAP, и SIGFPE), наоборот, можно объяснить только тем, что самая первая версия ОС UNIX была разработана для ЭВМ PDP-11, так как события, которым они соответствуют, являются специфическими для этой ЭВМ. На самом деле последние перечисленные мнемоники — это анахронизмы, а фактическое соответствие этих мнемоник тем событиям, которым они реально соответствуют, определяется прихотью разработчиков очередной версии ОС UNIX и архитектурными особенностями ЭВМ, для которой осуществляется указанная разработка.

Перечислим события, которые могут стать причиной посылки сигнала:

введение пользователем управляющего символа с терминала всем процессам, входящим в группу процессов, ассоциированных с этим терминалом, будет послан один из следующих сигналов: SIGINT, SIGQUIT или SIGHUP;

возникновение аварийной ситуации при функционировании пользовательского процесса; всем процессам будет послан один их следующих сигналов: SIGILL,SIGTRAP,SIGFPE,SIGBUS,SIGSEGV, SYGSYS или SYGPIPE;

возникновение некоторого заранее описанного события; всем процессам будет послан сигнал SIGALARM;

возникновение непредусмотренного или не поддающегося идентификации события; всем процессам будет послан один из следующих сигналов: SIGTERM, SIGCLD или SIGPWR.

Возникновение любого события может сопровождаться посылкой всем или некоторым процессам любого сигнала, для чего процессу достаточно осуществить системный вызов kill (сигналы SIGUSR1, SIGUSR2 и

SIGKILL вообще могут быть посланы только таким образом). Системный вызов kill имеет следующий формат:

```
kill(pid, sig)
```

В результате осуществления такого системного вызова сигнал, специфицированный аргументом sig, будет послан всем тем процессам, которые являются родственниками процесса с идентификатором процесса pid. Если при этом значение аргумента pid не превосходит положительного целого числа 1, то специфицированный сигнал будет послан целой группе процессов.

Например, если аргумент pid имеет значение 0, то сигнал, специфицированный аргументом sig, будет послан всем процессам с идентификаторами группы, совпадающими с идентификатором группы процесса, который осуществил системный вызов kill, кроме процессов с идентификаторами процесса 0 и 1.

Далее, если аргумент pid имеет значение —1, то сигнал, специфицированный аргументом sig, будет послан всем процессам, которые имеют тот же идентификатор пользователя, что и процесс, осуществивший системный вызов kill; если же сигнал послан из процесса привилегированного пользователя, то он будет получен всеми процессами, за исключением процессов с идентификаторами процесса 0 и 1.

Если значение аргумента pid представляет собой отрицательное число, не равное —1, то сигнал, специфицированный аргументом sig, будет послан всем процессам, идентификаторы группы которых имеют значения, совпадающие с абсолютным значением аргумента pid.

При удачном завершении выполнения системного вызова kill последний возвращает целое число 0, в противном случае — целое число —1. Причиной аварийного завершения выполнения системного вызова kill может послужить, например, спецификация несуществующего в ОС UNIX сигнала или спецификация несуществующего процесса.

Вообще, если процесс, посылающий специфицированный аргументом sig сигнал, не является процессом привилегированного пользователя, то сигнал может быть получен только теми процессами, у которых действующие идентификаторы пользователя совпадают с действующим идентификатором пользователя процесса, осуществившего системный вызов kill.

Рассмотрим теперь ряд примеров. Приведем фрагмент исходного текста на языке Си

```
kill(0, SIGFPE);
```

в результате выполнения которого всем процессам одной группы будет послан сигнал о возникновении исключительной ситуации в случае выполнения арифметической операции. При выполнении фрагмента программы на языке Си

```
kill(37, SIGKILL);
```

процесс с идентификатором процесса 37 будет аварийно завершен. И, наконец, еще один пример; в результате выполнения фрагмента программы на языке Си

```
kill(getpid(), SIGALRM);
```

сигнал побудки будет послан самому процессу, осуществившему системный вызов kill.

СИСТЕМНЫЙ ВЫЗОВ signal

Использование системного вызова signal предоставляет процессу возможность самостоятельно определить свою реакцию на получение того или иного сигнала, изменив тем самым установку, которая была осуществлена ОС UNIX "по умолчанию". Системный вызов signal имеет следующий формат:

```
#include <signal.h>
int (*signal(sig, func))()
int sig;
int (*func)();
```

Как видно из приведенного фрагмента исходного текста программы на языке Си, реакцией процесса, осуществившего системный вызов signal с аргументом func(), на получение им сигнала sig будет теперь осуществление вызова функции func. Напомним, что приведенное в этом фрагменте программы объявление функции func специфицирует ее тип данных как указатель на функцию, возвращающую целое значение.

Существуют две, так сказать, стандартные реакции процесса на получение им сигнала — это игнорирование полученного сигнала и восстановление стандартной, установленной самой ОС UNIX реакции. Обе эти возможности могут быть легко реализованы при использовании вместо аргумента func системного вызова signal символьных констант SIG_IGN и SIG_DFL соответственно. Если выполнение системного вызова signal удачно завершено, он возвращает значение, полученное в результате последнего вызова функции, вызываемой процессом при получении им указанного сигнала (это может быть либо объявленная ранее пользователем функция, либо функция, которая используется для этих целей ОС UNIX "по умолчанию"), в противном случае он возвращает целое число −1. Например, в результате выполнения фрагмента программы на языке Си

```
signal(SIGINT, SIG_IGN);
```

получаемые процессом и прерывающие его функционирование сигналы SIGINT будут игнорироваться. Рассмотрим другой пример: в результате выполнения фрагмента программы на языке Си

```
signal(SIGINT< SIG_DFL);
```

будет восстановлена принятая в ОС UNIX "по умолчанию" реакция процесса на получение им сигнала SIGINT. А в результате выполнения фрагмента программы на языке Си

```
signal(SIGINT, cleanup);
```

получение процессом, осуществившим такой системный вызов signal, сигнала SIGINT приведет к вызову реализованной пользователем функции cleanup, которая, как мы уже говорили, должна иметь тип данных

"указатель на функцию, возвращающую целое". Обращаем Ваше внимание, что функция func должна быть описана (или объявлена) прежде, чем будет осуществлен системный вызов signal, так как в противном случае переменная cleaup, указанная в качестве его аргумента, будет обработана компилятором с языка Си как имеющая тип данных "целое". Для этого рекомендуем Вам придерживаться следующего формата системного вызова signal:

```
int (*cleanup)();
sognal(SIGINT, cleanup);
```

СИСТЕМНЫЙ ВЫЗОВ pause

Использование системного вызова pause позволяет приостановить функционирование осуществившего его процесса до получения им некоторого сигнала. Ниже приведен формат системного вызова pause:

```
pause();
```

Из приведенного формата видно, что системный вызов pause не имеет параметров и, следовательно, пользователь лишен возможности изменить реакцию процесса на получаемые им сигналы. Вместе с тем, все изменения реакции процесса на получаемые им сигналы, осуществленные ранее с помощью системного вызова signal, остаются в силе. Из последнего замечания следует, что возобновить функционирование процесса с помощью посылки ему сигнала, стандартная реакция процесса на который была ранее изменена на реакцию типа "игнорировать", невозможно.

ИСПОЛЬЗОВАНИЕ СИГНАЛОВ ОС UNIX

Круг задач, решаемых в ОС UNIX с помощью сигналов, чрезвычайно широк. Прежде всего, к ним относятся задачи, требующие взаимодействия двух или более процессов ОС UNIX. Остановимся более подробно на некоторых из них.

УПРАВЛЕНИЕ НЕПРЕДУСМОТРЕННЫМИ СОБЫТИЯМИ

Если разработанная Вами программа в процесса своего выполнения не осуществляет ввода с терминала, то получение процессом, в рамках которого она выполняется, сигналов SIGINT или SIGHUP является для нее событием непредусмотренным. Использование такого программного средства, как сигналы, позволяет в рамках самого процесса проанализировать возникшее событие и в соответствии с заранее подготовленным алгоритмом обработать его. Известно, что при разработке пользовательских программ часто приходится прибегать к следующему приему: разрабатываемая программа в процессе своего функционирования создает один или более так называемых временных файлов, которые после завершения выполнения такой программы удаляются из файловой системы ОС UNIX. Понятно, что при аварийном завершении выполнения такой программы указанные временные файлы могут остаться неудаленными. Причиной аварийного завершения выполнения программы может

стать, например, получение сигнала SIGINT процессом, в рамках которого она выполняется. Посылка же процессу указанного сигнала может быть вызвана вводом с терминала специального символа INTERRUPT[1]. Итак, приведем исходный текст программы на языке Си, в которой описанная выше проблема решается с помощью системного вызова signal:

```
#include <signal.h>
main()
{
     int cleanup();
     if(signal(SIGINT, SIG_IGN) != SIG_IGN)
          signal(SIGINT, cleanup);
     . . . . .
     creat(tempfile, 0);
     . . . . .
     unlink(tempfile);
     exit(0);
}
/*
 * Выход по получении прерывания от терминала
 */
cleanup()
{
     unlink(tempfile);
     fprintf(stderr, "Terminated Abnormally\n");
     exit(1);
}
```

Итак, в результате ввода с терминала специального символа INTERRUPT всем процессам, ассоциированным с данным пользовательским терминалом, будет послан сигнал SIGINT. Однако получение процессом, в рамках которого выполняется эта программа, сигнала SIGINT приведет к тому, что последний сначала вызовет на выполнение функцию cleanup, которая осуществит удаление созданных временных файлов, а затем выведет на терминал соответствующее информационное сообщение и завершит свое функционирование. Добавим, что когда пользователь вызывает программу на выполнение в асинхронном режиме, интерпретатор команд shell выполняет ряд аналогичных действий, проиллюстрировать которые можно с помощью фрагмента программы на языке Си

```
if(fork() == 0)
     signal(SIGHUP, SIG_IGN);

     signal(SIGINT, SIG_IGN);
     exec(cmd);
}
```

[1] Специальный символ INTERRUPT в зависимости от версии ОС UNIX может быть введен с терминала нажатием на его клавиатуре клавиши DEL (на отечественных терминалах ей, как правило, соответствует клавиша ЗБ) или одновременным нажатием клавиши CTRL (на отечественных терминалах ей, как правило, соответствует клавиша УС) и клавиши С. — *Прим. ред.*

Сделаем одно важное замечание: всякий раз, когда процесс получает один из неигнорируемых им сигналов (исключение составляют сигналы SIGILL и SIGTRAP), ОС UNIX автоматически устанавливает его реакцию на всякое следующее получение этого сигнала такой, какой она была установлена "по умолчанию". Из сказанного следует, что при необходимости многократной обработки процессом одного и того же сигнала процесс должен всякий раз вновь осуществлять системный вызов signal, чтобы вновь установить необходимую реакцию на получение этого сигнала. Рассмотрим в качестве примера решения этой проблемы прием, применяемый обычно при разработке редакторов текста и приведем следующий фрагмент исходного текста на языке Си:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf cjb;
int abandon();
main()  {
     . . . . .
     setjmp(cjb);  /* сохранить стек процесса */
     signal(SIGINT, abandon);
          /* основной цикл */
     . . . . .
}
abandon()
{
     longjmp(cjb, 1);
     /* переход в начало главного цикла */
}
```

Здесь setjmp и longjmp — библиотечные функции, используемые для осуществления нелокальных безусловных переходов в программах на языке Си. Функционально осуществление вызовов этих функций эквивалентно выполнению операции установления метки и операции перехода на метку соответственно. Обращаем Ваше внимание на то, что системный вызов осуществляется внутри основного цикла непосредственно после выполнения возврата в программу из подпрограммы обработки полученного сигнала SIGINT. Приведем форматы вызовов библиотечных функций setjmp и longjmp:

```
#include <setjmp.h>

int setjmp(stack_frame)
jmp_buf stack_frame;

int longjmp(stack_frame, return)
jmp_buf stack_frame;
```

Всякий раз, когда библиотечная функция setjmp вызывается на выполнение с аргументом, который должен иметь тип данных jmp_buf (тип данных jmp_buf специфицирован в файле setjmp.h), она сохраняет часть содержимого стека процесса, описывающую состояние вызвавшей ее функции, в области оперативной памяти, специфицированной значени-

ем своего аргумента: при удачном завершении выполнения библиотечная функция setjmp возвращает целое число 0. Если теперь процесс получит сигнал SIGINT, то выполнение основного цикла будет автоматически прервано, а управление будет передано функции abandon, которая вызовет на выполнение библиотечную функцию longjmp с аргументом, также имеющим тип данных jmp_buf и значение, совпдающее со значением аргумента функции setjmp. В результате этого содержимое стека процесса будет восстановлено таким, каким оно было до момента прерывания выполнения основного цикла, и прерванное выполнение основного цикла будет продолжено с его начала, при этом возвращаемое значение библиотечной функции setjmp станет равным значению переменной return_val, описание которой содержится в файле setjmp.h. Поясним это несколько туманное утверждение на примере, для чего приведем фрагмент исходного текста программы на языке Си:

```
#include <setjmp.h>
jmp_buf stack;
if(setjmp(stack)){
     /* код, выполняемый после возврата из прерывания */
}else{
     /* код,  выполняемый после вызова функции setjmp */
}
```

Тип данных jmp_buf зависит от типа используемой ЭВМ и должен быть таким, чтобы структура stack_frame могла быть использована для хранения регистров центрального процессора ЭВМ. Сохранение содержимого стека процесса означает по сути сохранение состояния программы на момент прерывания ее выполнения. В то же время необходимо понимать, что после осуществления возврата из функции обработки полученного процессом сигнала с помощью библиотечной функции longjmp, переменные, значения которых были изменены после вызова функции setjmp, но перед получением процессом сигнала, сохранят свои текущие значения, в то время как управление будет возвращено в начало основного цикла. Это противоречие может сделать невозможным дальнейшее выполнение основного цикла программы. И еще одно важное замечание: функция, осуществляющая вызов библиотечной функции setjmp, не должна завершить своего выполнения до тех пор, пока существует вероятность получения процессом сигнала SIGINT. Если же это произойдет и она вернет управление в вызвавшую ее функцию, то фрагмент стека процесса, использовавшийся для хранения состояния функции, осуществившей вызов библиотечной функции setjmp, будет считаться свободным для дальнейшего использования и текущим станет аналогичный фрагмент стека, используемый для хранения состояния функции, которая осуществила вызов упомянутой завершившейся функции. Понятно, что выполнение в таких условиях библиотечной функции longjmp приведет к разрушению стека процесса и аварийному завершению функционирования этого процесса.

СОХРАНЕНИЕ ЗНАЧЕНИЯ ПЕРЕМЕННЫХ

Мы уже говорили, что в результате осуществления возврата с помощью библиотечной функции longjmp из функции, вызванной процессом для обработки полученного им сигнала, значения используемых в этот момент переменных не будут восстановлены, и это может привести к невозможности дальнейшего выполнения основного цикла программы. Для решения этой проблемы обычно используют следующий прием: выделяется некоторый фрагмент основного цикла, и прерывание выполнения этого фрагмента программы считается запрещенным. Если процесс получит обрабатываемый им сигнал в момент выполнения указанного фрагмента программы, то вызываемой при этом функции обработки полученного сигнала будет сообщено о том, что прерывание выполнения основного цикла запрещено, и функция обработки сохранит информацию о получении процессом сигнала SIGINT, а затем немедленно вернет управление в вызвавшую ее функцию с помощью оператора RETURN, т. е. в то место основного цикла, в котором произошло прерывание его выполнения. После того как выполнение фрагмента программы, прерывание которого запрещено, закончится, будет вызвана функция, которая, проанализировав сохраненную информацию о получении или неполучении процессом сигнала SIGINT, выполнит соответствующие действия. Ниже приведен фрагмент исходного текста программы на языке Си, который (кроме прочего) содержит описания двух функций: ecs и lcs. Первая из них вызывается перед, а вторая после выполнения фрагмента основного цикла, прерывание выполнения которого запрещено:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf cs_stack;
int in_cs, sig_recd;
main() {
     . . . . .
     if(setjmp(cs_stack)){
            /* обработка прерывания */
     }else{
            /* основная программа */
     }
     . . . . .
     ecs();
          /* фрагмент, который нельзя прерывать */
     lcs();
}
sig_vec() {
          /* запоминание полученных сигналов */
     if(in_cs){
       signal(SIGINT< SIG_IGN);
       sig_recd = 1;
       return;
     }else{
       signal(SIGINT, sig_vec);
       longjmp(cs_stack, 1);
     }
```

```
}
ecs(){
     sig_recd = 0;
     in_cs = 1;
}
lcs(){
     in_cs = 0;
     if(sig_recd){
        sig_recd = 0;
        signal(SIGINT, sig_vec);
        longjmp(cs_stack, 1);
     }
}
```

ПРЕРЫВАНИЯ ВЫПОЛНЕНИЯ СИСТЕМНОГО ВЫЗОВА

Посмотрим что случится, если непредусмотренное событие произойдет в момент осуществления процессом системного вызова. Как правило, прерывание выполнения системного вызова в результате получения пользовательским процессом сигнала невозможно, это определяется самой природой системного вызова — функционирование ядра ОС UNIX не может быть прервано. Исключение составляют ряд системных вызовов, связанных с выполнением операций ввода-вывода, таких, как read, write, open и close, а также системные вызовы pause и wait. Возвращаемым значением системного вызова, выполнение которого было прервано, всегда является целое число —1, а значение глобальной системной переменной errno устанавливается равным EINTR. Проиллюстрируем сказанное фрагментом исходного текста программы на языке Си:

```
#include <errno.h>
extern int errno;
do{
     result = read(fd, buf, n);
}while(result < 0 && errno == EINTR);
```

Для получения доступа к глобальной переменной errno необходимо объявить ее внешней переменной так, как это сделано в приведенном фрагменте программы на языке Си; значение символьной константы EINTR задается с помощью определения символьной замены входящего в состав файла errno.h. Следствием такого подхода будет то, что прерывание выполнения операции ввода из файла с дескриптором файла fd не окажется неожиданным для процесса, в рамках которого выполняется данная программа и который в этом случае повторит операцию ввода.

СИНХРОНИЗАЦИЯ ФУНКЦИОНИРОВАНИЯ ПРОЦЕССОВ

При синхронизации функционирования двух или более процессов часто возникает задача прерывания выполнения процессом некоторой программы, если в течение некоторого интервала времени не произойдет то или иное событие. Для решения этой задачи, как правило, используется системный вызов alarm, который информирует ядро ОС UNIX о необходимости послать процессу, его осуществившему, сигнал побудки SIGA-

LARM через указанное с помощью его аргумента число секунд. Если значение аргумента системного вызова alarm равно целому числу 0, то специфицированная ранее посылка процессу сигнала побудки будет отменена. Значение, возвращаемое системным вызовом alarm, представляет собой число секунд, заданное при предыдущем осуществлении этого системного вызова. Системный вызов alarm имеет формат

```
alarm(secs)
unsigned secs;
```

В качестве примера, иллюстрирующего использование системного вызова alarm, рассмотрим приводимый ниже исходный текст программы на языке Си. В процессе выполнения этой программы осуществляется операция копирования ее стандартного ввода на ее стандартный вывод, при этом на копирование одного байта отводится время, не превышающее 5 с; если в течение 5 с ввод байта не будет успешно завершен, то попытка считается неудачной. После осуществления трех неудачных попыток вся операция копирования стандартного ввода на стандартный вывод считается неудачной и завершается аварийно.

```
#define  TO_TIME  5
#define  MX_TRYS  3   /* максимум 3 попытки */
#include <signal.h>
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/tty.h>
char buffer[TTYHOG];
extern int errno;

timeout()
{
     signal(SIGALRM, timeout);
}
main()
{
     int n, trys = 0;
     signal(SIGALRM, timeout);
     while(trys < MX_TRYS){
          alarm(TO_TIME);
          /* установка тайм-аута */
          n = read(0, buffer, TTYHOG);
          /* ожидание ввода */
          alarm(0);
          /* конец тайм-аута */
          /* проверка ввода */
          if(n < 0 && errno == EINTR){
             fprintf(stderr, "timed out\n");
             trys++;
             continue;
          }

          /* аварийное завершение */
          if( n < 0){
```

```
                fprintf(stderr, "read error\n");
                exit(1);
            }

            /* достигнут конец файла */
            if(n == 0){
                fprintf(stderr, "EOF\n");
                exit(0);
            }

            /* выполнение копирования */
            write(1, buffer, n);
            trys = 0;
        }
    }
```

КАНАЛЫ ОС UNIX

Всякому, кто имеет хотя бы небольшой практический опыт использования ОС UNIX, должно быть хорошо знакомо такое программное средство, как каналы. Рядовой пользователь ОС UNIX обращается к этому программному средству с тем, чтобы осуществить так называемое замыкание стандартного вывода одной команды ОС UNIX на стандартный ввод другой ее команды. Например, ввод с терминала строки

```
ls | wc -l
```

приведет к тому, что стандартный вывод команды ls окажется замкнутым на стандартный ввод команды wc, вызванной на выполнение с флагом —1, или, иначе говоря, для двух этих команд ОС UNIX будет создан канал. В нашем конкретном примере результатом выполнения команды ls будет последовательность символьных строк, которая будет передана на обработку команде wc, а результатом выполнения команды wc будет положительное целое число, равное количеству строк указанной последовательности.

Аналогичного результата, впрочем, можно добиться с помощью прямого переназначения стандартного ввода одной команды и стандартного вывода другой команды следующим, например, образом:

```
ls > tmp && wc -l < tmp && rm tmp
```

Однако между этими на первый взгляд эквивалентными решениями одной задачи существует весьма существенное различие: во втором случае команды ls и wc будут вызваны на выполнение друг за другом, и, следовательно, общее время выполнения всей операции окажется равным сумме времен выполнения команд ls и wc, в то время как в первом случае прежде всего будет создан канал, а затем будут вызваны на выполнение две команды одновременно, в результате чего общее время выполнения всей операции окажется равным времени работы наиболее медленной из команд. И наконец, с практической точки зрения ввести соответствующую командную строку в первом случае значительно менее хлопотно, нежели во втором.

РЕАЛИЗАЦИЯ КАНАЛОВ ОС UNIX

Каналы ОС UNIX реализованы в виде снабженных специальным механизмом управления буферов ввода-вывода, и для получения доступа к ним может быть использован тот же аппарат, что и при осуществлении доступа к файлам ОС UNIX. Это последнее обстоятельство особенно приятно, так как избавляет пользователя от необходимости модифицировать разработанные им программы для обеспечения возможности построения из них конвейера.

Каждому каналу ОС UNIX ставится в соответствие один описатель файла, и все дальнейшее управление каналом со стороны ОС UNIX мало чем отличается от управления любым файлом. Назовем лишь два отличия канала от файла. Во-первых, каналу ОС UNIX может, вообще говоря, не соответствовать ни один элемент ни одного каталога ОС UNIX, и в этом случае время его существования определяется временем функционирования процесса, выполнившего операцию создания канала; во-вторых, размер канала ограничен числом блоков диска, указатели местоположения которых на диске содержатся в самом описателе файла (использование блоков косвенности в случае создания канала невозможно), чаще всего размер канала бывает ограничен 5120 байтами.

Вообще говоря, ограничение размера канала является залогом высокой эффективности операций обмена с ним. Объяснением этому факту может служить следующее обстоятельство: при осуществлении операций обмена с каналом ОС UNIX отводит для их реализации системные буферы ввода-вывода, а ввиду небольшого размера канала он почти целиком умещается внутри этих буферов; таким образом, фактический обмен информацией между двумя процессами, для которых был создан канал, осуществляется без использования магнитного диска. В заключение добавим, что если процесс, осуществляющий вывод информации в канал, делает это быстрее, чем процесс, осуществляющий ввод информации из канала, то функционирование первого процесса приостанавливается; такой подход обеспечивает надежную синхронизацию взаимного обмена информацией двух пользовательских процессов, использующих для этого канал.

Различные версии ОС UNIX поддерживают, вообще говоря, различные типы каналов; остановимся для начала на простейшем из них — неименованных каналах, единственном типе каналов, поддерживаемом ОС UNIX версии 7. Создать неименованный канал можно с помощью системного вызова pipe, формат которого приведен ниже:

```
int fd[2];
pipe(fd);
```

Здесь fd — идентификатор целого массива, содержащего два элемента. Системный вызов pipe возвращает в случае удачного завершения своего выполнения два дескриптора файла, первый из которых хранится в первом элементе упомянутого массива fd[0] и должен в дальнейшем использоваться для осуществления операций ввода из канала, а второй —

во втором элементе массива fd[1] и должен в дальнейшем использоваться для осуществления вывода в канал. Полученные в результате осуществления системного вызова pipe дескрипторы файлов могут быть использованы точно так же, как и любые другие дескрипторы файлов ОС UNIX, например, они могут быть использованы в качестве аргументов системных вызовов dup и fcntl. Рассмотрим, например, фрагмент исходного текста программы на языке Си, в результате выполнения которого стандартный вывод программы будет переназначен на созданный канал:

```
int fd[2];
pipe(fd);
close(1);
fcntl(fd[1], F_DUPFD, 1);
close(fd[1]);
```

Примерно таким образом решается интерпретатором команд shell проблема реализации возможности переназначения стандартного вывода одной команды ОС UNIX на стандартный ввод другой.

Далее, если пользовательский процесс создал канал и после этого осуществил системный вызов fork, породил тем самым процесс-потомок, последний, как известно, унаследует оба дескриптора файла, соответствующих созданному каналу. В результате автоматически возникнет возможность установления коммуникаций между процессом-предком и процессом-потомком. Ниже приведен фрагмент исходного текста программы на языке Си, иллюстрирующий сказанное:

```
pipe(fd);
if(fork()){
     /* процесс-предок */
     close(fd[0]);
     . . . . .
}else{
     /* процесс-потомок */
     close(fd[1]);
     . . . . .
]
```

Понятно, что в такой ситуации закрытие процессом-предком файла с дескриптором, используемым для выполнения операций вывода информации в канал, означает автоматическое закрытие файла с дескриптором, использовавшимся процессом-потомком для выполнения операций ввода информации из канала. Приведем фрагмент исходного текста программы на языке Си, который может быть использован при создании канала для команд ls и wc:

```
if((pid = fork())){
    /* процесс-предок */
    dead = wait(&status);
}else{
    pipe(fd)    /* создание канала */
    if(fork()){
```

```
        close(1);
        fcntl(fd[1], F_DUPFD, 1);
        close(fd[1]);
        close(fd[0]);
        execl("/bin/ls", "ls", 0);
     }else{
    close(0);
    fcntl(fd[0], F_DUOFD, 0);
    close(fd[0]);
    close(fd[1]);
    exec("/bin/wc", "wc", 0);
    }
}
```

Рассмотрим исходный текст еще одной программы, назначением которой является копирование файлов. Сравните ее с программой копирования файлов, рассмотренной в гл. 4.

```
#define BLOCK  512
char buffer[BLOCK];
main(argc, argv)
int argc;
char *argv[];
{
   int i = atoi(argv[1]);  /* число блоков для передачи */
   int pipe_fd[2];         /* дескриптор файла канала   */
   pipe(pipe_fd);          /* создать канал             */
   while(i--){
        write(pipe_fd[1], buffer, BLOCK);

         read(pipe_fd[0], buffer, BLOCK);
         }
}
```

ИСПОЛЬЗОВАНИЕ КАНАЛОВ И СИГНАЛОВ ДЛЯ ОСУЩЕСТВЛЕНИЯ СВЯЗИ МЕЖДУ ПРОЦЕССАМИ ОС UNIX

Установление связи между двумя функционирующими процессами – одна из типичных задач, с которыми приходится сталкиваться пользователю ОС UNIX, и, как правило, для решения этой задачи используются такие программные средства, как каналы и сигналы. При этом на каналы возлагается роль среды передачи информации, а на сигналы – роль системы управления процессом передачи информации. Такой подход сразу же делает актуальным вопрос о разработке протокола обмена. Разработанный протокол обмена может быть реализован с помощью сигналов, которые один из связанных процессов может посылать другому с целью спецификации возникновения того или иного события. Чаще всего для этого используются такие поддерживаемые ОС SYSTEM V сигналы, как SIGUSR1, SIGUSR2 и SIGTERM. Хорошим примером использования всех перечисленных выше программных средств и приемов может служить задача, для решения которой один пользовательский процесс должен осуществлять ввод информации от нескольких источников информации одновре-

менно. Решение такой задачи с помощью только системного вызова read не может быть успешным, так как в этом случае возможно зацикливание процесса на вводе информации из одного источника информации. Действительно, выполнение системного вызова read может быть завершено лишь после ввода им очередной порции информации или специального символа EOT, символизирующего возникновение ситуации "конец файла".

Решить возникшую проблему можно с помощью сигналов, для чего достаточно посылать процессу, принимающему информацию, сигнал только тогда, когда процесс, передающий информацию, готов к выводу ее в канал. Обращаем Ваше внимание на то, что описанный механизм управления функционированием процесса очень напоминает механизм аппаратных прерываний, используемый большинством современных ЭВМ для управления выполнением операций обмена с периферийными устройствами ЭВМ.

Обратимся теперь к примерам и рассмотрим исходные тексты двух программ на языке Си, первую из которых было бы логично назвать "ведущей", а вторую "ведомой", и рассмотрим функциональную схему коммуникаций процессов, в рамках которых эти программы будут выполняться. Опишем теперь решение задачи одновременного ввода информации с нескольких пользовательских терминалов. Итак, после вызова на выполнение программы, названной нами "ведущей", процесс, в рамках которого она выполняется, породит для каждого терминала, ввод информации с которого предполагается осуществлять, новый процесс, поведение которого будет определять программа, названная нами "ведомой". После этого процесс, в рамках которого выполняется "ведущая" программа, перейдет в состояние ожидания получения сигнала, для чего "ведущая" программа предпишет ему выполнение бесконечного цикла. Каждый порожденный этим процессом процесс-потомок, в рамках которого выполняется "ведомая" программа, постоянно находится в состоянии осуществления ввода информации с того терминала, для которого он был порожден процессом-предком в соответствии с логикой, заложенной в "ведущей" программе. После завершения процессом-потомком ввода очередной порции информации или специального символа EOT, процесс-потомок пошлет процессу-предку сигнал SIGTERM, в результате чего процесс-предок осуществит ввод из канала очередной порции информации, введенной пользователем с очередного терминала.

```
/*
 * Ведомая программа
 */
#include <signal.h>
#include <sys/types.h>
#include <sys/tty.h>

main(argc, argv)
int argc;
char *argv[];
{
```

```
    int n;
    char buffer[TTYHOG];
    for(;;){
            n = read(0, buffer, TTYHOG);
            if(n == 0){
                 /* достигнут конец файла */
                 kill(atoi(argv[1]), SIGFPE);
                 close(1);
                 exit(0);
            }
            kill(atoi(argv[1]), SIGTERM);
            write(1, buffer, n);
    }
}
```

Советуем Вам тщательно изучить приведенный выше исходный текст "ведомой" программы, так как, несмотря на его кажущуюся простоту, использованный при его разработке прием является основополагающим для большинства многопользовательских программных средств. В заключение приведем исходные тексты на языке Си "ведущей" программы, а также ряда вспомогательных функций:

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/tty.h>
int  fd[2];
char pids[TTYHOG];
int slave_count;
/*
 * Программа обработки сигнала SIGTERM
 */
service(){
   int n;
   char buffer[TTYHOG];
   signal(SIGTERM, service);
   n = read(fd[0], buffer, TTYHOG);
   write(1, buffer, n);
}
/*
 * Программа обработки сигнала SIGFPE
 */
slavedrop()
{
    signal(SIGFPE, slavedrop);
    if(--slave_coubr == 0)
         exit(0);
}
/*
 *     Ведущая программа;
 * исходный текст хранится в файле master.c;
 * вызов на выполнение имеет вид:
 *              master /dev/ttyY /dev/ttyX ...
 */
main(argc, argv)
```

```
int argc;
char *argv[];
{
   int i;
   signal(SIGTERM, service);
   signal(SIGFPE, slavedrop);
   sprintf(pids, "%06d", getpid());
   pipe(fd);
   slave_count = ac - 1;
   for(i=1; i < ac; i++)
       slave(argv[i]);
   close(fd[1]);
   for(;;)
       pause();
}
/*
 * Программа инициации ведомой программы
 */
slave(tty)
char *tty;
{
   if(for() == 0){
      int tfd;
      /* Переназначение стандартного ввода на терминал */
      close(0);
      tfd = open(tty, 0);
      if(tfd != 0){
         fprintf(stderr, "bad tty %s\n", tty);
             exit(1);
      }
      /* Переназначение стандартного вывода на канал */
      close(1);
      dup(fd[1]);
      close(fd[0]);
      close(fd[1]);
      /* Вызов ведомой программы */
      execl("./slave", "slave", pids, 0);
      fprintf(stderr, "Can't exec slave on %s\n", tty);
      exit(1);
   }
}
```

ИМЕНОВАННЫЕ КАНАЛЫ

Этот тип каналов не поддерживается ОС UNIX версии 7. Однако роль именованных каналов в разработке программных средств коммуникации процессов ОС UNIX чрезвычайно велика. Это объясняется тем, что круг задач, решаемых с использованием неименованных каналов, в основном ограничивается созданием коммуникаций между процессами-родственниками, чего, очевидно, зачастую оказывается недостаточно. Отличительной особенностью именованных каналов ОС UNIX является то, что каждому именованному каналу соответствует один элемент некоторого каталога ОС UNIX, что делает возможным осуществление ссылки к нему по имени файла, хранящемуся в поле имени соответствующего ему

элемента каталога. Описываемый тип каналов поддерживается ОС SYSTEM III и ОС SYSTEM V. Использование именованных каналов позволяет реализовать коммуникации между процессами, не связанными родственными отношениями.

СОЗДАНИЕ ИМЕНОВАННЫХ КАНАЛОВ

Именованный канал ОС UNIX может быть создан с помощью системного вызова mknod и будет существовать до тех пор, пока не будет явно уничтожен с помощью системного вызова unlink. Как Вы помните, второй аргумент системного вызова mknod специфицирует тип открываемого файла и режим его открытия. При использовании системного вызова mknod для создания именованного канала тип открываемого файла должен быть специфицирован символьной константой IFIFO, например:

```
mknod(name, IFIFO|ACCESS, 0);
```

Первый аргумент системного вызова mknod специфицирует имя создаваемого именованного канала, а режим его открытия специфицирован в нашем случае символьной константой ACCESS. Например, для создания именованного канала с именем fifo, к которому разрешен доступ по записи и по чтению всем активным процессам, можно осуществить системный вызов mknod следующим образом:

```
mknod("fifo", IFIFO|0666, 0);
```

Именованные каналы также могут быть созданы с помощью команды ОС UNIX /etc/mknod, имеющей следующий формат:

```
$ /etc/mknod pipename p
```

Сказанное означает, что для создания именованного канала с именем fifo, к которому разрешен доступ по записи и по чтению всем активным процессам, достаточно ввести с терминала две следующие командные строки:

```
$ /etc/mknod fifo p
$ chmod 666 fifo
```

ИСПОЛЬЗОВАНИЕ ИМЕНОВАННЫХ КАНАЛОВ

Как мы уже говорили, именованные каналы могут быть доступны одновременно нескольким не связанным родственными отношениями процессам, и это обстоятельство делает возможным создание с их помощью сложных программных комплексов, способных функционировать в среде вычислительных сетей. Мы ограничимся рассмотрением одного довольно простого примера, иллюстрирующего использование именованных каналов для реализации многопользовательского доступа к базе данных. Основная проблема, с которой неизбежно сталкиваются разработчики подобных программных средств, заключается в реализации управления доступом нескольких пользователей к одному и тому же фай-

лу. Для решения этой проблемы предлагается осуществлять косвенный доступ к файлу, содержащему базу данных, путем вывода контекста запроса в именованный канал; далее, процесс, реализующий управление доступом к указанному файлу, осуществит ввод контекста запроса из именованного канала и обработает его. Введя из именованного канала контекст очередного запроса, процесс, реализующий управление доступом к базе данных, пошлет соответствующему пользовательскому процессу сигнал и целью подтверждения получения его запроса на доступ к базе данных.

Итак, перейдем к рассмотрению исходных текстов программ на языке Си, реализующих описанный выше механизм. Ниже приведен исходный текст программы на языке Си, которая определяет поведение процесса, реализующего управление доступом к базе данных:

```
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include "packet.h"

int datapipe, ctrlpipe, datafile, got_sig;
char *dataname = "/usr/lib/tmac/tmac.s";
handler(){
   signal(SIGUSR1, handler);
   got_sig++;
}
main(){
   struct packet pk;
   struct packet sendpk;
   ctrlpipe = open(CNAME, O_RDONLY|O_NDELAY);
   datafile = open(dataname, 0);
   sendpk.pk_code = SENDPID;
   handler();
   for(;;){
          int n;
          while((n = read(ctrlpipe, &pk, sizeof(pk)))){
                process(&pk, &sendpk);
          }
   }
}

process(pkp, spkp)
struct packet *pkp, spkp;
{
       char pbuf[PBUFSIZE];
       /* запись в канал fifo будет осуществлена
        * только в том случае, если он уже открыт для чтения
        */
        datapipe = open(DNAME, O_WRONLY|ONDELAY);
        switch(pkp->pk_code){
        case CONNECT:
              write(datapipe, spkp, sizeof(struct packet));

        break;
case RQ_READ:
```

```
        lseek(datafile, pkp->pk_blk*PBUFSIZE, 0);
        read(datafile, pbuf, PBUFSIZE);
        write(datapipe, pbuf, PBUFSIZE);
        break;
    default:
        fprintf(stderr, "unknown packet code\n");
        exit(1);
    }
    got_sig = 0;
    kill(pkp->pk_pid, SIGUSR1);
    while(!got_sig);
    close(datapipe);
```

Теперь приведем исходный текст функции connect, используемой для установления связи некоторого пользовательского процесса с процессом, реализующим управление доступом к базе данных:

```
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include "packet.h"
int datapipe, ctrlpipe;
extern int got_sig;
connect(){
    struct packet pk;
    datapipe = open(DNAME, O_RDONLY|O_NDELAY);
    ctrlpipe = open(CNAME, O_WRONLY|O_NDELAY);
    if(datapipe < 0 || ctrlpipe < 0){
        fprintf(stderr, "cannot open pipes\n");
        exit(1);
    }
    pk.pk_pid = getpid();
    pk.pk_code = CONNECT;
    got_sig = 0;
    write(ctrlpipe, &pk, sizeof(pk));
    while(!got_sig);
    read(datapipe, &pk, sizeof(pk));
    kill(pk.pk_pid, SIGUSR1);
    return(pk.pk_pid);
}
```

Функция request, исходный текст которой на языке Си приведен ниже, используется для считывания одного блока файла, содержащего указанную базу данных:

```
#include <stdio.h>
#include <signal.h>
#include "packet.h"
extern int ctrlpipe, datapipe, got_sig;
request(ptr, blk, spid)
char *ptr;
int blk;
int spid;
{
```

```
    struct packet pk;
    pk.pk_pid = getpid();
    pk.pk_blk = blk;
    pk.pk_code = RQ_READ;
    got_sig = 0;
    write(ctrlpipe, &pk, sizeof(pk));
    while(!got_sig);
    read(datapipe, ptr, PBUFSIZE);
    kill(spid, SIGUSR1);
}
```

Ниже приведено содержимое файла, включаемого на этапе трансляции, представляющее собой (кроме всего прочего) объявление структуры packet, используемой при формировании контекста запроса к базе данных:

```
/* packet.h */
struct packet {
   int pk_pid;  /* идентификатор процесса */
   int pk_blk;  /* номер блока файла      */
   int pk_code; /* код запроса            */
};
 /* коды запросов */
#define RQ_READ  0  /* запрос на чтение   */
#define CONNECT  1  /* запрос на обслуживание */
#define SENDPID  2  /* ответ на запрос CONNECT */
 /* имена каналов */
#define DNAME   "datapipe"
#define CNAME   "ctrlpipe"
 /* размер блока файла */
#define PBUFSIZE 512
```

И наконец, в заключение приведем исходный текст программы на языке Си, которая может быть использована для иллюстрации работы всего комплекса программ в целом:

```
/* Программа, транслирующая запросы
 *пользователя к базе данных */
#include <stdio.h>
#include <signal.h>
#include "packet.h"
int datapipe, ctrlpipe;
handler(){
          /* перехват сигналов */
          signal(SIGUSR1, handler);
          got_sig++;
}
main(argc, argv)
int argc;
char *argv[];
{
     int spid;    /* идентификатор процесса-сервера */
     int blk;
     char buffer[PBUFSIZE];
     blk = atoi(argv[1]);
```

```
        handler();   /* сигнал перехвачен */
        spid = connect();
        for(;;){
                request(buffer, blk, spid);
                fwrite(buffer, PBUFSIZE, 1, stdout);
        }
}
```

ЗАДАЧИ

1. В приведенном выше примере использования неименованных каналов процесс-предок ожидает завершения функционирования всех своих процессов-потомков, после чего и сам завершает функционирование. Разработайте такую версию программы, чтобы завершение функционирования любого из процессов-потомков вызывало со стороны процесса-предка порождение нового процесса-потомка, который выполнял бы те же функции, что и его завершивший функционирование "брат".

2. В приведенном выше примере использования именованных каналов реализована лишь операция чтения информации из базы данных. Модифицируйте исходные тексты приведенных в примере программ так, чтобы с их помощью можно было осуществлять и запись в базу данных. Подумайте, насколько усложнится эта задача при условии допустимости записей переменной длины.

3. В приведенном выше примере использования именованных каналов остался нерешенным вопрос управления завершением функционирования процесса, реализующего управление доступом к базе данных. Восполните этот пробел и разработайте новый запрос к базе данных, в результате выполнения которого процесс, реализующий управление доступом к базе данных, завершал бы свое функционирование, а всем пользовательским программам, пытающимся осуществить этот доступ, посылалось бы сообщение о том, что база данных недоступна.

ЛИТЕРАТУРА

1. B.W. Kernighan, D.M. Ritchie (1978), UNIX Programming, UNIX Programmers Manual, seventh edition, vol 2a.

2. A.M. Lister (1983), Fundamentals of Operating Systems, Third Edition, Macmillan.

Глава 8. АНАЛИЗ И ОТЛАДКА ПРОГРАММ НА ЯЗЫКЕ Си

Вряд ли кто-нибудь рискнет утверждать, что может разработать программу на языке Си, которая сразу же заработает и поэтому не нуждается в средствах отладки программ. Для решения этой проблемы ОС UNIX предоставляет своим пользователям широкий набор инструментальных программных средств, использование которых позволяет выполнить предварительный анализ эффективности и мобильности разработанной программы, а также осуществить ее отладку. Сюда входят верификатор lint, отладчик adb и ряд инструментальных средств профилирования.

Эту главу мы посвятим рассмотрению перечисленных инструментальных средств. Итак, программа lint, называемая обычно верификатором, представляет собой инструментальное средство, позволяющее пользова-

телю отыскивать в исходном тексте своей программы на языке Си такие его фрагменты, которые, являясь синтаксически корректными, могут вызывать определенные сомнения с точки зрения их мобильности и эффективности. Программа adb, называемая обычно отладчиком объектного кода, представляет собой инструментальное средство, использование которого позволяет осуществлять трассировку отлаживаемой программы на языке Си и определять место и причину возникновения аварийной ситуации при вызове программы на выполнение. И наконец, несколько слов об инструментальных средствах профилирования программ. Использование этих инструментальных средств позволяет получать профиль времени выполнения отдельных фрагментов пользовательской программы, что немаловажно для повышения эффективности разрабатываемого программного обеспечения.

Как мы уже говорили, эта глава будет посвящена описанию практического использования перечисленных выше инструментальных средств, поэтому без описания процесса разработки некоей гипотетической программы на языке Си нам не обойтись. Итак, рассмотрим программу на языке Си, исходный текст которой приведен ниже: эта программа представляет собой фильтр, с помощью которого может быть осуществлено преобразование содержимого некоторого файла ОС UNIX в последовательность строк символов; каждая такая очередная строка представляет информацию, содержащуюся в очередном байте входного файла, в шестнадцатеричной системе счисления.

Исходный текст рассматриваемой программы хранится в трех файлах: main.c, содержащем описание функции main, которая осуществляет разбор введенной командной строки, hexd.c, содержащем описание функции hexd, осуществляющей вывод шестнадцатеричного представления введенной информации на свой стандартный вывод, и hexout.c, содержащем описание функции hexout, которая вызывается функцией hexd. Приведенный ниже исходный текст программы содержит целый ряд ошибок, вызывающих возникновение аварийной ситуации при попытке вызвать скомпилированную программу на выполнение. Мы предполагаем отыскать и исправить эти ошибки с помощью перечисленных инструментальных средств и тем самым продемонстрировать Вам типичное использование их на практике. После того как все обнаруженные ошибки будут исправлены, мы осуществим профилирование программы hexd и выявим таким образом те фрагменты программы, реализация которых была осуществлена наименее эффективно.

```
/*
 *      программа main.c содержит ошибки
 */
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{
```

```
    long length = 0;
    int i;
    FILE *fp;
    while(i < argc && argv[i][0] == '-'){
      if((i+1) >= argc){
          fprintf(stderr,
                  "specify length as -l n\n");
          exit(1);
      }
      switch(argv[i][1]){
      case'l':
            length = atoi(&argv[i++]);
            break;
      default:
            fprintf(stderr, "unknown option\n");
            exit(1);
      }
      i++;
    }
    if(i >= argc){
         /* имя файла не задано - используется стандартный ввод */
         hexd(length, stdin);
    }else{
         if((fp = fopen(argv[i], "r")) == 0){
            fprintf(stderr,
                    "cannot open %s\n", argv[i]);
            exit(1);
         }
         hexd(length, fp);
    }
    exit(0);
}
```

```
/*
 *   hexd.c
 */
#include <stdio.h>
int address = 0;
hexd(fp, size)
FILE *fp;
long size;
{
   int ch, length = 0;
   /* насильно завершить вывод */
   hexout(-1);
   address = 0;

    while(length++ < size || size == 0){
          if((ch = getc(fp)) == EOF){
               break;
          }
          hexout(ch);
    }
    /* насильно завершить вывод */
```

```
    hexout(-1);
}
```

```
/*
 *   hexout.c
 */
extern int address;    /* передается hexd() */
int count = 0;         /* число байт в строке */
#define MAXCOLS 16     /* максимальная длина строки */
hexout(ch)
int ch;
{
     if(ch != -1){
        if(count++ == 0){
             printf("%06x\t", address);
        }
       printf("%02x ", ch);
       address++;
     }
     if(ch == -1 || count >= MAXCOLS){
       printf("\n");
       count = 0;
     }
}
```

Для поддержки процесса отладки указанной программы воспользуемся таким известным инструментальным средством, как команда make, и приведем содержимое соответствующего файла описаний:

```
hexd :   main.o hexd.o hexout.o
         cc -o hexd main.o hexd.o hexout.o
```

Сама команда make, а также структура файла описаний будут рассмотрены в следующей главе.

КОМАНДА lint

Появление в составе ОС UNIX такого инструментального средства, каким является программа lint, продиктовано, по-видимому, одним простым соображением, которое, несмотря на его кажущуюся простоту, можно назвать философской основой построения всех инструментальных средств ОС UNIX и которое звучит следующим образом: каждое инструментальное средство должно решать лишь свою узкую задачу, но зато решать ее очень хорошо. Следуя этому принципу, разработчики компилятора с языка Си добились того, чтобы это инструментальное средство быстро и безошибочно компилировало программы, написанные на языке Си, но не заботилось бы об эффективности примененной техники программирования. Задача по оценке корректности и эффективности примененных пользователем приемов программирования может быть решена с помощью другого инструментального средства — программы lint.

Существует, впрочем чисто практическое объяснение описанного разделения синтаксического и семантического контроля исходного текста

пользовательской программы на языке Си. Известно, что первые версии ОС UNIX были разработаны для ЭВМ класса PDP-11, объем оперативной памяти которых весьма невелик, а потому объединение в рамках одного программного средства двух столь сложных функций представляло значительную трудность.

Итак, вся совокупность решаемых инструментальным средством lint задач может быть условно разбита на три большие группы: проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений; поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки; общая оценка мобильности пользовательской программы.

По своей функциональной структуре программа lint чрезвычайно напоминает компилятор с языка Си, но вместо объектного кода, как это делает компилятор с языка Си, программа lint генерирует комментарии, описывающие результаты семантического разбора исходного текста рассматриваемой пользовательской программы. При этом так же, как и компилятор с языка Си, программа lint использует препроцессор языка Си и, следовательно, обрабатывает все использованные в составе рассматриваемой пользовательской программы определения символьной замены и макроподстановок, а также осуществляет включение специфицированных файлов. Вместе с тем, в отличие от компилятора с языка Си, программа lint осуществляет комплексный синтаксический контроль использования всех описанных в программе функций, иначе говоря, сопоставление и проверку соответствия описания функции формату ее вызова.

Если же объявленная в программе функция является внешней, т. е. имеет тип extern, то для проверки корректности ее использования программе lint, очевидно, окажется недостаточно лишь исходного текста самой пользовательской программы, и в таком случае она сделает попытку извлечь ее из заранее подготовленного пользователем текстового файла, называемого файлом объявлений программы lint и содержащего описания всех параметров внешних функций, а также их возвращаемых значений.

Обычно каждой стандартной библиотеке объектных файлов ставится в соответствие файл объявлений программы lint. Таким образом, при обработке с помощью программы lint некоторой пользовательской программы все используемые ею стандартные функции, объявленные как внешние, анализируются последней в соответствии с содержимым специфицированного файла объявлений программы lint. Заметим, что если не указано противное, то программа lint обрабатывает файл объявлений, соответствующий стандартной библиотеке объектных файлов, хранящейся в файле с именем libc.a. Специфицировать другой файл объявлений программы lint можно с помощью флага —lm, который она обрабатывает аналогично тому, как обрабатывает одноименный флаг команда ld. Файлы объявлений программы lint обычно находятся в каталоге /usr/

lib и имеют имена, начинающиеся с контекста "llib". Перечислим некоторые из них:

```
llib-lc    - файл объявлений, соответствующий
             стандартной библиотеке libc.a
llib-lm    - файл объявлений, соответствующий
             библиотеке математических функций libm.a
llib-port  - файл объявлений, соответствующий
             набору мобильных функций, должен быть
             подключен при компиляции с флагом -p
```

ВЫЗОВ ФУНКЦИИ И ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Осуществляя синтаксический анализ программы на языке Си, компилятор с языка Си, как правило, исключает из рассмотрения вопрос соответствия типа данных возвращаемого значения вызванной функции типу данных переменной, используемой для хранения указанного возвращаемого значения. Так, в результате осуществления на этапе компиляции синтаксического анализа программы на языке Си, исходный текст которой приведен ниже, компилятором с языка Си будет получен результат, свидетельствующий о ее синтаксической корректности:

```
main(){
      int x;
      x = val();
}
int val(){
       return;
}
```

Кроме того, компилятор с языка Си, как правило, не контролирует соответствие типов данных формальных параметров описанных функций типам данных использованных фактических параметров этих функций и даже числа формальных параметров функции числу ее фактических параметров. Так, в результате осуществления на этапе компиляции синтаксического анализа программы на языке Си, исходный текст которой приведен ниже, компилятором с языка Си будет получен результат, свидетельствующий о ее синтаксической корректности:

```
main(){
      int x;
      x = val();
}
int val(y)
long *y;
{
       return;
}
```

Итак, две приведенные выше программы на языке Си корректны с точки зрения синтаксиса языка Си, но содержат ряд семантических ошибок.

Для выявления указанных семантических ошибок необходимо воспользоваться программой lint. Пусть исходный текст программы на языке Си, приведенной во втором примере, хранится в файле с именем retval.c. В таком случае в результате ввода с терминала командной строки вида

```
$ lint retval.c
```

на терминал будет выведено следующее[1]:

```
retval.c:
retval.c(3): warning: x set but not used in function main
retval.c(6): warning: argument y unused in function val.
val: variable # of args.     retval.c(7) :: retval.c(3)
val value is used, but none returned
```

Как видно из приведенного примера, программа lint с успехом обнаружит все семантические ошибки, скрытые в исходном тексте обрабатываемой ею программы.

ПОТЕНЦИАЛЬНЫЕ ПРИЧИНЫ ВОЗНИКНОВЕНИЯ АВАРИЙНОЙ СИТУАЦИИ

Продолжим описание возможностей программы lint. Помимо осуществления семантического анализа программ на языке Си, примеры которого были рассмотрены выше, программа lint может быть использована для выявления потенциальных причин возникновения аварийных ситуаций на этапе выполнения программы. Для проведения такого дополнительного анализа корректности рассматриваемой программы на языке Си необходимо вызвать на выполнение программу lint с флагом —h. В результате такого вызова программы lint последняя осуществит ряд дополнительных эвристических проверок корректности исходного текста программы на языке Си, что приведет к выявлению таких, например, ситуаций, как использование переменных, значение которых осталось неопределенным, или присвоение переменной, описанной как переменная, имеющая тип данных unsigned, значения, являющегося отрицательным числом. Кроме того, с помощью программы lint можно обнаружить ряд конструкций, которые могут быть обработаны компилятором с языка Си не так, как рассчитывает пользователь, например:

```
if(x&0177 == 0)
```

или

```
*ptr++;
```

Так, во втором примере операция инкрементирования будет осуществлена над указателем, а не над значением, на которое он указывает, и на это обстоятельство программа lint обратит внимание пользователя, применившего в своей программе на языке Си подобную конструкцию.

---

[1] "внимание: переменная x вычисляется, но не обрабатывается функцией main; внимание: аргумент y не обрабатывается функцией val; несоответствие числа аргументов; возвращаемое значение функции val обрабатывается, но не вычисляется". — *Прим. ред.*

ВОПРОСЫ МОБИЛЬНОСТИ

Среди основных достоинств ОС UNIX, более других обусловивших столь быстрый рост ее популярности, обычно называют мобильность функционирующих под управлением ОС UNIX пользовательских программ. Однако это вовсе не означает, что пользователь ОС UNIX навсегда избавился от необходимости самостоятельно заботиться о мобильности разрабатываемых им программ. Из этого следует, что задача анализа и обеспечения мобильности разрабатываемого программного обеспечения по-прежнему актуальна. Решить указанную задачу можно с помощью программы lint, для чего необходимо вызвать ее на выполнение с флагом — p. Обратимся к примерам. Известно, что в зависимости от архитектурных особенностей ЭВМ для хранения переменных, имеющих тип данных char, может использоваться как семь, так и восемь бит одного байта, и это обстоятельство должно быть обязательно учтено при разработке мобильной программы. Если же исходный текст такой программы на языке Си был обработан программой lint, то все аналогичные описанной спорные конструкции будут выявлены программой lint и соответствующим образом прокомментированы.

Кроме архитектурных особенностей ЭВМ на мобильность пользовательской программы могут оказывать влияние и особенности реализации той или иной версии компилятора с языка Си. Рассмотрим, например, принятый в конкретной реализации порядок обработки функцией своих аргументов:

```
process(a[i++], a[i++], a[i++]);
```

Понятно, что передача функции process трех последовательных элементов массива a будет осуществлена корректно только в том случае, если в результате выполнения объектного кода, полученного в результате компиляции и соответствующего функции process, ее аргументы будут обработаны в порядке слева направо.

ПРОГРАММА hexd

Подводя итог этому короткому описанию возможностей программы lint, вернемся к программе, с которой мы начали это рассмотрение, — к программе hexd. Обработаем исходный текст нашей программы с помощью программы lint:

```
$ lint -h main.c hexd.c hexout.c
main.c:
main.c(14): warning: i may be used before set
hexd.c:
hexout.c:
atoi, arg. 1 used inconsistenly llib-lc(152):: main.c(21)
hexd, arg. 1 used inconsistenly hexd.c(9)   :: main.c(32)
hexd, arg. 2 used inconsistenly hexd.c(9)   :: main.c(32)
hexd, arg. 2 used inconsistenly hexd.c(9)   :: main.c(39)
hexd, arg. 1 used inconsistenly hexd.c(9)   :: main.c(39)
```

В результате этого программой lint будут выявлены следующие три проблемы: во-первых, переменная i была обработана прежде, чем проинициализирована, во-вторых, формат вызова функции atoi не соответствует ее описанию, поскольку в качестве аргумента фактически специфицирован указатель на него:

```
atoi(&argv[++i]);
```

Для того чтобы исправить эту ошибку, достаточно переписать указанный вызов функции atoi следующим образом:

```
atoi(argv[++i]);
```

И наконец, в-третьих, аргументы функции hexd специфицированы неверно, так как аргумент, имеющий тип данных "указатель на файл", является вторым, в то время как в соответствии с описанием функции он должен быть первым аргументом. Исправим и эту ошибку:

```
/*
 *      исправленный вариант программы main.c
 */
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{
   long length = 0;
   int i;
   FILE *fp;
/*>>*/ i = 1;
   while(i < argc && argv[i][0] == '-'){
     if((i+1) >= argc){
         fprintf(stderr,
                 "specify length as -l n\n");
         exit(1);
     }
     switch(argv[i][1]){
     case'l':
/*>>*/       length = atoi(argv[i++]);
            break;
     default:
            fprintf(stderr, "unknown option\n");
            exit(1);
     }
     i++;
  }
  if(i >= argc){
       /* имя файла не задано - используется стандартный ввод */
/*>>*/ hexd(stdin, length);
  }else{
       if((fp = fopen(argv[i], "r")) == 0){
          fprintf(stderr,
                  "cannot open %s\n", argv[i]);
          exit(1);
       }
       hexd(length, fp);
```

```
    }
    exit(0);
}
```

ОТЛАДЧИК ОБЪЕКТНОГО КОДА

Операционная система UNIX предоставляет своим пользователям еще одно инструментальное средство, называемое отладчиком объектного кода и представляющее собой программу, объектный код которой хранится в файле с именем adb и которую в связи с этим чаще называют отладчиком adb. Основное назначение отладчика adb заключается в осуществлении анализа "посмертного" дампа памяти, автоматически осуществляемого ОС UNIX при возникновении аварийных ситуаций в процессе ее функционирования. Кроме того, отладчик adb может быть использован при отладке пользовательской программы, для чего ее объектный код должен быть выполнен под управлением отладчика adb. В процессе работы отладчик adb использует таблицу символов, хранящуюся в том же файле, что и объектный код отлаживаемой программы. Необходимо заметить, что в процессе функционирования отладчик adb предоставляет пользователю возможность выводить на экран терминала содержимое специфицированного адреса оперативной памяти ЭВМ, при этом формат выводимой информации таков, что для его понимания требуется знание языка ассемблера. В частности, для понимания материала, к изложению которого мы сейчас приступаем, читателю понадобится знание основ программирования на языке ассемблера для ЭВМ, построенных на базе микропроцессора MC68000.

АНАЛИЗ ДАМПА ПАМЯТИ

Если при выполнении пользовательской программы возникает аварийная ситуация (например, в результате выполнения ошибочного фрагмента ее объектного кода управление передается в область оперативной памяти, не входящую в состав сегмента кода выполняемой программы), то ОС UNIX автоматически осуществляет аварийное завершение функционирования процесса, в рамках которого выполняется эта программа, для чего ему посылается соответствующий сигнал; после этого ОС UNIX осуществляет вывод содержимого сегментов кода и данных программы, выполнение которой было прервано, в формате образа памяти в файл с именем core, содержащийся в текущем каталоге. Таким образом, содержимое файла core представляет собой пресловутый "посмертный" дамп оперативной памяти. Анализ содержимого файла с именем core может оказаться полезным при необходимости выяснить причину возникновения аварийной ситуации, так как позволяет получить значения всех регистров процессора, использованных процессом, функционирование которого было прервано, а также содержимое стека процесса на момент возникновения указанной аварийной ситуации. Осуществить такой анализ содержимого файла core проще всего с помощью отладчика adb.

В качестве примера рассмотрим программу, исходный текст которой на языке Си содержит описание рекурсивной функции f. Причиной возникновения аварийной ситуации станет переполнение стека процесса, в рамках которого выполнялась рассматриваемая пользовательская программа. В результате этого функционирование процесса будет завершено путем посылки ему сигнала SIGSEGV, и в текущем каталоге будет создан файл с именем core, содержащий сегменты кода и данных процесса в формате образа памяти. Исходный текст этой программы на языке Си имеет вид

```
int count = 0;
main(){
        f(1);
}
f(i)
int i;
{
        count++;
        f(i+1);
}
```

Пусть файл, содержащий приведенный исходный текст, имеет имя calls.c. В таком случае для осуществления компиляции и последующего вызова на выполнение рассматриваемой программы необходимо ввести с терминала последовательность команд ОС UNIX вида

```
$ cc -o calls calls.c
$ calls
Memory fault - core dumped
$
```

Попытаемся теперь с помощью отладчика adb выяснить, в результате чего возникла аварийная ситуация. Для этого прежде всего исследуем текущее содержимое стека процесса, в рамках которого выполнялась указанная программа. Таким образом, нам удастся понять, в процессе выполнения какой функции возникла аварийная ситуация, а также получить значения аргументов этой функции в момент ее вызова. Отладчик adb имеет два входных параметра, первый из которых он обрабатывает как имя исполняемого файла, а второй — как имя файла, содержащего дамп памяти (по умолчанию это файл с именем core). Заметим, что отладчик adb не имеет своего промптера несмотря на то, что является диалоговым программным средством. Итак, вызовем на выполнение отладчик adb:

```
$ adb calls core
,10 $c
_f+26:      f     (0x5542)
_f+26:      f     (0x5541)
_f+26:      f     (0x5540)
_f+26:      f     (0x553F)
_f+26:      f     (0x553E)
_f+26:      f     (0x553D)
```

```
_f+26:      f      (0x553C)
_f+26:      f      (0x553B)
_f+26:      f      (0x553A)
_f+26:      f      (0x5539)
```

Первая строка приведенного выше примера представляет собой командную строку, с помощью которой был вызван на выполнение отладчик adb, а каждая из последующих строк была выведена на терминал самим отладчиком adb и содержит имя функции, в процессе выполнения которой возникла аварийная ситуация, адрес точки вызова указанной функции и, наконец, текущие значения ее аргументов. В нашем случае значение аргумента функции совпадает с номером шага рекурсии.

Являясь диалоговым инструментальным средством, отладчик adb может обрабатывать ряд команд, которые вводятся пользователем с терминала. Любая команда adb представляет собой контекст, который можно представить в виде совокупности четырех полей: первое содержит адрес оперативной памяти, второе — значение счетчика повторений, третье — имя команды adb и, наконец, четвертое поле содержит модификатор команды. При этом первое поле команды отладчика adb отделяется от второго ее поля с помощью символа ,. Поле адреса может отсутствовать в формате команды отладчика adb. В таком случае будет использован текущий адрес оперативной памяти.

В приведенном выше примере поле адреса отсутствует, в поле счетчика повторений содержится целое положительное число 10, а в поле имени команды — контекст $c. В результате выполнения введенной команды отладчика adb десять раз будет выполнена команда трассировки стека процесса. Большинство наиболее часто используемых команд отладчика adb осуществляет в результате своего выполнения вывод на терминал адресов и содержимого рассматриваемых областей оперативной памяти, при этом использование модификатора команды отладчика adb позволяет управлять форматом выводимой информации, определяя, например, систему счисления, в которой будет представлена выводимая информация, или же ее общий объем в байтах. Так, использование в качестве модификатора команды отладчика adb символа d или символа x специфицирует вывод в десятичном или шестнадцатеричном представлении соответственно двух байт информации, а использование в качестве модификатора команды отладчика adb символа D или символа X специфицирует вывод в десятичном или шестнадцатеричном представлении соответственно четырех байт информации. Команда = отладчика adb используется для получения текущего адреса. Например, в результате ввода в нашем случае команды отладчика adb вида

```
f = x
        0x800070
```

на терминал будет выведен адрес точки вызова функции f в шестнадцатеричном представлении. Команда ? отладчика adb используется для получения информации, хранящейся в исполняемом файле по адресу, специ-

фицированному содержимым поля адреса; команда / отладчика adb отличается от команды ? отладчика adb только тем, что в результате ее использования будет получена информация, хранящаяся по соответствующему адресу в файле core. Заметим, что обычно содержимое сегмента данных выводится с помощью команды / отладчика adb из файла core, а содержимое сегмента кода выводится на терминал с помощью команды ? отладчика adb из исполняемого файла. Итак, приведем первые три инструкции языка ассемблера, соответствующие первым машинным инструкциям, входящим в состав рассматриваемой программы,

```
main,3?i
_main:      link    a6, #0x0
            tstb    -136,(a7)
            moveml  #<>,-(a7)
```

после чего выведем на терминал содержимое описанной в программе переменной count в десятичном представлении. Для этого достаточно ввести с терминала

```
count/D
          21825
```

КОНТРОЛЬНЫЕ ТОЧКИ

Вернемся вновь к рассмотренной ранее программе на языке Си, которую мы назвали hexd. После того как все синтаксические и семантические ошибки, обнаруженные с помощью компилятора с языка Си и верификатора lint, были удалены из текста программы, можно попытаться вызвать ее на выполнение следующим, например, образом:

```
$ make
      cc -c main.c
      cc -o hexd main.o hexd.o hexout.o
$ hexd -l 10 hexd
000000 02 06 00 14 00 00 17 06 00 00 02 e2 00 00 04 0c
000010
<interrupt>
```

Как видно из приведенного примера, программа hexd не смогла обработать флаг —1, использование которого должно было ограничить число обрабатываемых ею байт входного файла, а кроме того, завершение выполнения программы вызвано возникновением аварийной ситуации. Воспользуемся теперь отладчиком adb и выясним причину возникновения указанной аварийной ситуации, для чего необходимо использовать команду b отладчика adb:

```
:b
```

Основным назначением команды b отладчика adb является установка контрольных точек отладчика adb в объектном коде обрабатываемой им пользовательской программы. Итак, предположим, что программа hexd

вызвана на выполнение под управлением отладчика adb так, как это было показано выше на примере программы calls. Установим контрольную точку отладчика adb на байт, имеющий смещение 8 байт относительно адреса точки входа функции hexd. Заметим, что выбор величины смещения сильно зависит от архитектурных особенностей используемой ЭВМ (в нашем случае смещение 8 байт соответствует ЭВМ, построенной на базе микропроцессора MC68000). В результате получим возможность осуществить трассировку заполнения стека процесса, в рамках которого выполняется данная программа. Начать выполнение под управлением отладчика adb отлаживаемой программы можно с помощью команды r отладчика adb:

```
:r <args>
```

Прежде всего убедимся в том, что флаг —1 был правильно обработан программой hexd, а функции hexd было верно передано специфицированное этим флагом число байт:

```
$ adb hexd
hexd+8:b
:r -l 10 main.c
hexd:running
breakpoint   _hexd+8:    clrl   -8.(a6)
,1 $c
_main+308:   _hexd (0xE4, 0)
```

Поясним теперь, что же представляет собой установленная нами выше контрольная точка отладчика adb и как она им обрабатывается. Если процесс, в рамках которого выполняется отладчик adb, вызванный на выполнение для отладки объектного файла hexd, осуществляет выборку содержимого адреса оперативной памяти, на которой была установлена контрольная точка отладчика adb, то последний прекращает обработку объектного кода программы hexd и выводит на терминал соответствующее сообщение. После этого пользователь может с помощью команд отладчика adb получить необходимую ему информацию о содержимом стека процесса (в результате многократного повторения этого приема и будет получена трассировка заполнения стека процесса). Итак, как видно из приведенного примера, значение, переданное функции hexd в качестве ее аргумента, не соответствует значению, которое было специфицировано с помощью флага —1 программы. В таком случае проверим правильность преобразования типов данных, выполняемого с помощью функции atoi, вызов которой осуществляет функция main. Для этого установим еще одну контрольную точку adb на байт, имеющий смещение 8 байт относительно точки входа функции atoi, и повторим всю процедуру еще раз:

```
atoi+8:b
:r -l 10 main.c
hexd:running
breakpoint      _atoi+8     moveml  #<d6,d7,a5>,-(a7)
,1 $c
_main+142       _atoi (0x7fffbf)
```

Как известно, аргумент стандартной функции atoi должен иметь тип данных "указатель на символы" и должен специфицировать строку цифр, подлежащую преобразованию к целому типу данных. Убедимся в том, что строка цифр, обрабатываемая функцией atoi в качестве входного параметра, отвечает выбранному нами алгоритму, для чего воспользуемся командой / отладчика adb:

```
0x7fffbf/s
0x7FFFBF:     -1
```

Итак, в результате ошибки в программе аргумент функции atoi получает неверное значение. Попытаемся теперь найти допущенную ошибку. Достаточно беглого взгляда на исходный текст программы, чтобы увидеть, что функция atoi вызывается функцией main слеующим образом:

```
atoi(argv[i++]);
```

Мы надеемся, что после тщательного рассмотрения исходного текста функции main Вы убедитесь в том, что ошибка заключается в использовании постфиксной формы операции инкрементирования, а для устранения ошибки необходимо использовать ее префиксную форму:

```
atoi(argv[i++]);
```

Приведем соответствующим образом скорректированный текст функции main:

```
/*
 *      исправленный вариант программы main.c
 */
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{
   long length = 0;
   int i;
   FILE *fp;
   i = 1;
   while(i < argc && argv[i][0] == '-'){

      if((i+1) >= argc){
          fprintf(stderr,
                  "specify length as -l n\n");
          exit(1);
      }
      switch(argv[i][1]){
      case'l':
/*>>*/      length = atoi(argv[++i]);
          break;
      default:
          fprintf(stderr, "unknown option\n");
          exit(1);
      }
```

```
        i++;
    }
    if(i >= argc){
        /* имя файла не задано - используется стандартный ввод */
        hexd(stdin, length);
    }else{
        if((fp = fopen(argv[i], "r")) == 0){
            fprintf(stderr,
                    "cannot open %s\n", argv[i]);
            exit(1);
        }
        hexd(length, fp);
    }
    exit(0);
}
```

Если мы теперь заново скомпилируем программу hexd и вновь вызовем ее на выполнение, то получим следующий результат:

```
$ make
    cc -c main.c
    cc -o hexd main.o hexd.o hexout.o
$ hexd -l 256 main.c
000000 2f 2a 0a 20 2a 20 63 6f 72 72 65 63 74 20 6d 61
000010 69 6e 28 29 20 66 6f 72 20 68 65 78 64 20 70 72
000020 6f 67 72 61 6d 2e 0a 20 2a 2f 0a 23 69 6e 63 6c
000030 75 64 65 20 3c 73 74 64 69 6f 2e 68 3e 0a 0a 6d
000040 61 69 6e 28 61 63 2c 20 61 76 29 0a 69 6e 74 20
000050 61 63 3b 0a 63 68 61 72 20 2a 2a 61 76 3b 0a 7b
000060 0a 09 6c 6f 6e 67 20 6c 65 6e 67 74 68 20 3d 20
000070 30 3b 0a 09 69 6e 74 20 69 3b 0a 09 46 49 4c 45
000080 20 2a 20 66 70 3b 0a 0a 09 69 20 3d 20 31 3b 0a
000090 09 77 68 69 6c 65 28 69 3c 61 63 20 26 26 20 61
0000a0 76 5b 69 5d 5b 30 5d 20 3d 3d 20 27 2d 27 29 20
0000b0 7b 0a 09 09 60 66 20 28 28 69 2b 31 29 20 3e 3d
0000c0 20 61 63 29 20 7b 0a 09 09 09 66 70 72 69 6e 74
0000d0 66 28 73 74 64 65 72 72 2c 20 22 73 70 65 63 69
0000e0 66 79 20 6c 65 6e 67 74 68 20 61 73 20 2d 6c 20
0000f0 6e 5c 6e 22 29 3b 0a 09 09 09 65 78 69 74 28 31
```

ПРОФИЛИРОВАНИЕ ПРОГРАММЫ

После того как мы с Вами добились корректного выполнения программы hexd, попытаемся оценить эффективность этой программы и заодно попробуем добиться ее повышения. Понятно, что для такой небольшой задачи, которую решает рассматриваемая нами программа hexd, повышение ее эффективности не столь уж существенно и вряд ли может оказаться очень заметным, однако мы воспользуемся ею для демонстрации принципиальной возможности повышения эффективности программного обеспечения. Итак, оценим полное, пользовательское и системное

время выполнения программы hexd, для чего введем с терминала следующую командную строку[1]:

```
$ time hexd /etc/passwd >/dev/null
      19.1 real         6.8 user        0.6 sys
```

Здесь real специфицирует полное время функционирования пользовательского процесса, в рамках которого выполняется программа hexd; user специфицирует время, затраченное на выполнение объектного кода, лежащего в сегменте кода указанного пользовательского процесса; и наконец, sys специфицирует время, затраченное ОС UNIX на обеспечение функционирования пользовательского процесса, в рамках которого выполняется программа hexd (сюда, в частности, входит время, затраченное на осуществление системных вызовов). Понятно, что для повышения эффективности программы hexd необходимо добиться уменьшения пользовательского времени выполнения программы.

Пойдем дальше и попытаемся определить соотношения времени выполнения отдельных функций нашей программы, иначе говоря, осуществить то, что принято называть "профилированием пользовательской программы". Итак, скомпилируем еще раз нашу программу, вызвав для этого на выполнение компилятор с языка Си с флагом —p. В результате ОС UNIX будет через фиксированные промежутки времени прерывать выполнения программы hexd, чтобы определить, какая функция в данный момент выполняется. Кроме того, каждый раз, как та или иная функция программы начинает или завершает свое выполнение, ОС UNIX также будет фиксировать времена возникновения этих событий. Таким образом, в результате профилирования программы hexd будет получена информация о том, сколько раз была вызвана каждая отдельная функция этой программы и сколько времени было затрачено на осуществление каждого такого вызова этой функции. Полученная информация будет сформатирована в таблицу и помещена в файл с именем mon.out. Для анализа полученной информации может быть использована программа prof.

Сделаем теперь одно важное замечание: для полного профилирования программы необходимо использовать флаг —p только на этапе компиляции программы, но и на этапе ее компоновки, так как это позволит включить в процесс профилирования также и те библиотечные функции, которые, как известно, будут подключены только на этапе компоновки.

[1] Файл с именем /dev/null представляет собой специальный файл, доступ к которому по записи и по чтению разрешен всем пользователям ОС UNIX и который может быть использован для вывода информации "в никуда". Так, в нашем случае стандартный вывод программы hexd переназначен на файл /dev/null, и следовательно, выводимая в процессе ее выполнения информация не попадет на экран терминала и вместе с тем никакой файл не будет создан. Если попытаться осуществить ввод из файла /dev/null, то первый же введенный символ окажется символом EOF, что свидетельствует о достижении конца файла. Заметим, что в нашем примере команда time осуществляет вывод не на стандартный вывод, а на стандартный протокол, и поэтому результат ее выполнения окажется выведенным на терминал.

Итак, для полного профилирования программы hexd необходимо воспользоваться последовательностью команд ОС UNIX вида

```
$ rm -f hexd.o hexout.o main.o
$ make CFLAGS = -p
        cc -p -c main.c
        cc -p -c hexd.c
        cc -p -c hexout.c
        cc -p -o hexd main.o hexd.o hexout.o
$ hexd /etc/passwd >/dev/null
$ prof hexd >prof.out
```

Ниже приведено содержимое файла prof.out:

```
%time   cumsecs   #call   ms/call   name

73.6      10.53   11842      0.89   _printf
10.6      12.05                     mcount
 7.8      13.17   10527      0.11   _hexout
 5.2      13.92       1    750.25   _hexd
 1.6      14.15                     _monstartup
 0.6      14.24       3     27.79   _read
 0.2      14.27       1     33.34   _open
 0.1      14.29       1     16.67   _fopen
 0.1      14.30       1     16.67   _isatty
 0.1      14.32       3      5.56   _sbrk
 0.0      14.32       3      0.00   _filbuf
 0.0      14.32       5      0.00   _flsbuf
 0.0      14.32       2      0.00   _fstat
 0.0      14.32       1      0.00   _gtty
 0.0      14.32       1      0.00   _ioctl
 0.0      14.32       1      0.00   _main
 0.0      14.32       1      0.00   _malloc
 0.0      14.32       1      0.00   _profil
 0.0      14.32       4      0.00   _write
```

Как показывают результаты профилирования, основное время было затрачено на выполнение функции printf. Вызов функции printf осуществляет функция hexout для вывода на стандартный вывод номера байта в начале каждой выводимой программой hexd строки, а также для вывода на стандартный вывод символа NEWLINE в конце каждой выводимой программой hexd строки. Кроме того, вызов библиотечной функции printf осуществляется для выполнения форматного преобразования и вывода на стандартный вывод каждого байта введенной из входного файла информации, что, конечно же, более существенно. Ниже приведен исходный текст новой версии программы hexd, в которой форматное преобразование выполняется самой функцией hexout, а вывод информации на стандартный вывод осуществляется с помощью библиотечной функции putchar:

```
/*
 *   hexout.c
 */
```

```
extern int address;    /* передается hexd() */
int count = 0;         /* число байт в строке */
#define MAXCOLS 16     /* максимальная длина строки */

char *hx = "0123456789abcdef";

hexout(ch)
int ch;
{
     if(ch != -1){
        if(count++ == 0){
             printf("%06x\t", address);
        }
       putchar(hx[(ch >> 4) & 017]);

        putchar(hx[ch & 017]);
        putchar(' ');
        address++;
     }
     if(ch == -1 || count >= MAXCOLS){
        printf("\n");
        count = 0;
     }
}
```

Выполним теперь профилирование этой новой версии:

```
$ make CFLAGS = -p;
       cc -p -c hexout.c
       cc -p -o hexd main.o hexd.o hexout.o
$ hexd /etc/passwd >/dev/null
$ prof hexd

 %time   cumsecs   #call   ms/call   name

 27.6      2.17   10527      0.21   _hexout
 27.2      4.30   31575      0.07   _putchar
 18.7      5.77                     mcount
 12.8      6.77    1317      0.76   _printf
  9.6      7.52       1    750.25   _hexd
  2.8      7.74                     _monstartup
  0.8      7.80       3     23.23   _read
  0.4      7.84       1     33.34   _open
  0.2      7.85       2      8.34   _fstat
  0.0      7.85       3      0.00   _filbuf
  0.0      7.85       5      0.00   _flsbuf
  0.0      7.85       1      0.00   _fopen
  0.0      7.85       1      0.00   _gtty
  0.0      7.85       1      0.00   _ioctl
  0.0      7.85       1      0.00   _isatty
  0.0      7.85       1      0.00   _main
  0.0      7.85       1      0.00   _malloc
  0.0      7.85       3      0.00   _profil
  0.0      7.85       4      0.00   _sbrk
  0.0      7.85       4      0.00   _write
```

Как видим, время, затраченное на выполнение библиотечной функции printf, сильно сократилось. Скомпилируем теперь новую версию программы hexd без использования флага —p и вновь оценим полное, пользовательское и системное время выполнения программы, для чего введем с терминала последовательность командных строк ОС UNIX вида

```
$ rm hexd.o hexout.o main.o
$ make
        cc -c main.c
        cc -c hexd.c
        cc -c hexout.c
        cc -o hexd main.o hexd.o hexout.o
$ time hexd /etc/passwd >/dev/null
         8.9 real         3.1 user         0.3 sys
```

Итак, в результате наших стараний пользовательское время выполнения программы hexd уменьшилось более чем в два раза.

ЛИТЕРАТУРА

1. S.C Johnson (1978), Lint, a C Program Checker, Unix Programmers Manual, seventh edition, volume 2a.

2. J.F. Maranzano & S.R. Bourne (1978), A Tutorial Introduction to ADB, Unix Programmers Manual, seventh edition, vol 2a.

Глава 9. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПОДДЕРЖКИ ПРОГРАММНЫХ СИСТЕМ

В предыдущей главе мы описали инструментальные средства adb и lint, использование которых облегчает задачу отладки разработанных пользователем программ. Перейдем теперь к рассмотрению проблемы поддержки программных систем. В развитии любой программной системы наступает в конце концов такой момент, когда ее структура становится слишком сложной для того, чтобы можно было удержать в голове логическую схему этой структуры, и пользователь начинает ощущать острую потребность в средствах автоматизации процедуры поддержки разрабатываемой им программной системы. Решению этой задачи и служат предоставляемые пользователям ОС UNIX инструментальные средства make и SCCS.

ПОСТРОЕНИЕ ПРОГРАММНЫХ СИСТЕМ

Сквозь все главы нашей книги красной нитью проходит идея представления большой пользовательской программы в виде связанной совокупности сравнительно небольших программных модулей, иначе говоря, в виде программной системы. Были описаны некоторые приемы, позволяющие реализовать эту идею, и при этом мы особо подчеркивали, что предлагаемое расчленение может и должно быть осуществлено на этапе подготовки исходного текста программной системы.

Пусть, например, пользователь занят разработкой экранного редактора текста. Функциональная структура такой программы может быть легко представлена в виде совокупности взаимосвязанных функциональных элементов. Возникает мысль, что было бы, наверное, удобно реализовать каждый такой функциональный элемент программы отдельно, независимо от других ее функциональных элементов.

Продолжим наши рассуждения и предположим, что для решения поставленной перед ним задачи пользователь создал каталог с именем editor, в котором для хранения каждого программного модуля, реализующего тот или иной функциональный элемент редактора текста, будет создан отдельный каталог. Если же при реализации того или иного функционального элемента вновь окажется удобным представление его в виде совокупности отдельных функциональных элементов, то для хранения исходного текста программы, реализующей такой функциональный элемент, вновь будет создан отдельный каталог. Таким образом, исходный текст разрабатываемого экранного редактора текста, представленного в виде программной системы, окажется распределенным по поддереву каталогов ОС UNIX, имеющему корень в каталоге с именем editor (рис. 9. 1).

Главной проблемой, с которой столкнется в такой ситуации пользователь, станет проблема поддержки разработанной им программной системы: каким образом осуществлять компиляцию и последующую совместную компоновку отдельных программных модулей? Традиционным решением этой проблемы является использование так называемых командных файлов, т. е. текстовых файлов, содержащих последовательность имен команд ОС UNIX с соответствующими флагами. Предполагается, что упомянутая последовательность команд ОС UNIX построена таким образом, что, будучи выполнена интерпретатором команд shell, в результате передачи ему на выполнение такого командного файла, приведет к получению исполняемого файла, содержащего объектный код разрабатываемой пользователем программы. Ниже приведен пример командного файла, который может быть использован для получения объектного кода гипотетического экранного редактора текста и перед вы-

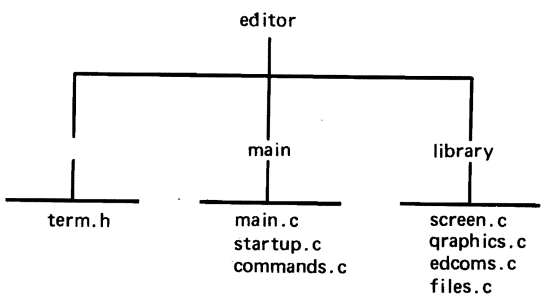

Рис. 9. 1. Логическая схема поддерева файловой системы ОС UNIX содержащего исходные тексты экранного редактора editor

полнением которого необходимо установить в качестве текущего каталога каталог с именем main:

```
cc -c -O main.c
cc -c -O commands.c
cc -c -O startup.c
cd ../library
cc -c -O screen.o
cc -c -O edcom.c
cc -c -O files.c
cc -c -O graphics.c
cd ../main
cc -o editor main.o commands.o startup.o ../library/screen.o\
../library/graphics.o ../library/edcom.o ../library/files.o
```

Использование описанного традиционного подхода несет с собой одно очень существенное неудобство: всякий раз, когда пользователь модифицирует исходный текст одного из модулей программной системы, он встает перед необходимостью повторной компиляции всех модулей, в том числе и тех, исходный текст которых не подвергался модификации. Понятно, что это ведет к непроизводительным затратам машинного времени и снижению эффективности программирования.

Для решения описанной проблемы в составе ОС UNIX имеется инструментальное средство, называемое командой make. В процессе своего выполнения команда make обрабатывает файл, который принято называть файлом описаний и который отдаленно напоминает упомянутый выше командный файл. Обычно файл описаний имеет имя Makefile, и в этом случае он может быть не специфицирован в командной строке вызова на выполнение команды make. Ниже приведено содержимое двух файлов описаний команды make, которые могут быть использованы вместо описанного выше командного файла; первый из них должен храниться в каталоге с именем main, а второй — в каталоге с именем library:

```
#
# main/makefile
#
LDR = ../library
CFLAGS = $(LDR)/screen.o $(LDR)/graphics.o $(LDR)/edcom.o\
         $(LDR)/files.o
editor: main.o commands.o startup.o ../h/term.h
        (cd ../library; make CFLAGS = $(CFLAGS))
        cc -o editor main.o commands.o startup.o $(LIBF)
```

```
#
# library/makefile
#
library: screen.o graphics.o  files.o edcom.o
screen.o  graphics.o files.o edcom.o: ../h/term.h
```

Итак, для получения исполняемого файла, содержащего объектный код нашего гипотетического экранного редактора текста, достаточно, установив в качестве текущего каталога каталог с именем main, ввести с терминала

```
make
```

после чего будет выполнена последовательность команд ОС UNIX вида

```
$ make
cc -c -O main.c
cc -c -O commands.c
cc -c -O startup.c
(cd ../library; make CFLAGS = -O)
cc -c -O screen.c
cc -c -O graphics.c
cc -c -O edcom.c
cc -c -O files.c
cc -o editor main.o commands.o startup.o ../library/screen.o\
../library/graphics.o ../library/edcom.o ../library/files.o
```

Однако если попытаться повторить вызов на выполнение команды make при условии, что ни один из файлов, содержащих исходные тексты модулей, составляющих программную систему, не подвергался модификации, то никакие действия команды make предприняты не будут, а вместо этого на терминал будет выведено соответствующее сообщение:

```
$ make
'editor' is up to date.
```

Перейдем теперь к описанию логики работы команды make. Упоминавшийся нами файл описаний содержит информацию о взаимосвязи файлов, составляющих программную систему, и используется командой make для выработки последовательности действий, которые необходимо выполнить для получения конечного результата, каковым является объектный код разрабатываемой программы. Помимо файла описаний команда make использует в процессе своего функционирования набор встроенных правил, среди которых мы выделим следующее: повторной компиляции должны быть подвергнуты лишь те исходные файлы, дата последней модификации которых оказалась старше даты создания соответствующего им объектного файла. Понятно, что в эту группу входят также все те исходные файлы, для которых соответствующие объектные файлы не существуют вовсе. При этом принцип, по которому устанавливается соответствие между названными исходными и объектными файлами, должен быть сформулирован в файле описаний команды make. Теперь Вы должны понять, почему в результате первого вызова на выполнение команды make в нашем примере будет получен объектный файл экранного редактора текста (объектные файлы соответствующих модулей не существовали), а в результате второго не будет сделано ничего (даты последней модификации всех файлов не изменились, и, следовательно, дата создания объектного файла старше даты создания соответствующего исходного файла).

Предположим теперь, что возникла необходимость модифицировать исходный текст модуля программной системы, исходный текст которого хранится в файле commands.c. Осуществив необходимую модификацию исходного текста, вновь вызовем на выполнение команду make, в результате чего будет выполнена последовательность команд ОС UNIX вида

```
$ make
cc -O -c commands.c
(cd ../library; make CFLAGS = -O)
cc -o editor main.o commands.o startup.o ../library/screen.o\
../library/graphics.o ../library/edcom.o ../library/files.o
```

Глядя на приведенную последовательность команд, легко понять, что команда make осуществила те минимальные действия, без которых невозможно было бы получить новую версию экранного редактора текста.

Как мы уже говорили, файл описаний содержит перечень взаимозависимостей отдельных модулей программной системы и соответственно файлов, содержащих исходный текст и объектный код этих модулей. Необходимо заметить, что сюда могут быть отнесены и все те файлы, которые включаются в состав соответствующих исходных файлов препроцессором в результате обработки им директивы #include. Так, в рассматриваемом примере файл описаний может (и должен) содержать строку вида

```
screen.o graphics.o files.o edcom.o: ../h/term.h
```

Информация, заключенная в этой строке файла описаний, специфицирует зависимость перечисленных объектных файлов от включаемого файла с полным именем ../h/term.h, а это означает, что всякая модификация его содержимого вызовет повторную компиляцию соответствующих исходных файлов. Ниже приведена последовательность команд ОС UNIX, которая будет в этом случае выполнена:

```
$ make
(cd ../library; make CFLAGS = -O)
cc -O -c screen.c
cc -O -c graphics.c
cc -O -c edcom.c
cc -O editor main.o commands.o startup.o ../library/screen.o\
../library/graphics.o ../library/edcom.o ../library/files.o
```

ФАЙЛ ОПИСАНИЙ КОМАНДЫ make

Как мы уже говорили, файл описаний представляет собой текстовый файл, содержащий последовательность строк. Указанные строки составляют набор спецификаций взаимозависимостей, используемых командой make при ее функционировании. Каждая такая спецификация взаимозависимостей имеет определенную логическую структуру, в соответствии с которой она может быть представлена в виде совокупности нескольких отдельных компонент:

имя целевого файла;

последовательность имен файлов, от которых зависит целевой файл;

последовательность команд ОС UNIX, которая должна быть выполнена, если дата последней модификации хотя бы одного из файлов, от которых зависит целевой файл, старше даты модификации целевого файла.

Ниже приведен формат спецификации взаимозависимостей:

```
target1[target2...]:[:][dependent1...][;commands][#..]
[(tab)commands][#]
```

Здесь символ # специфицирует начало комментария; это означает, что содержимое строки, начиная с символа # и до ее конца, не будет обрабатываться командой make. Непосредственно после имени целевого файла строка файла описаний должна содержать один или два последовательных символа :. В первом случае соответствующая последовательность команд ОС UNIX должна содержаться в одной строке файла описаний, а во втором — она может содержаться в нескольких последовательных строках файла описаний, следующих непосредственно за строкой, содержащей имя целевого файла. Например:

```
editor::    startup.c
        $(CC) -S $(CFLAGS) startup.c
        sed -e script <startup.c | as -o startup.o
editor::    commands.o main.o
        (cd ../library;make CFLAGS = -$(CFLAGS))
        cc -o editor main.o commands.o startup.o $(LIBF)
```

Такие сложные спецификации взаимозависимостей на практике используются сравнительно редко, поэтому мы не будем их подробно рассматривать, а тех, у кого они вызвали интерес, отошлем к документу UPM.

Если количество файлов, от которых зависит целевой файл, столь велико, что последовательность их имен не умещается на одной строке Вашего терминала, можно воспользоваться символом \ и тем самым указать команде make на то, что следующая строка является продолжением текущей, например

```
target:  d1 \
         d2 \
         d3 \
         commands
```

Добавим, что среди имен файлов, от которых зависит целевой файл, могут быть, в свою очередь, указаны имена целевых файлов.

МАКРООПРЕДЕЛЕНИЯ КОМАНДЫ make

Макроопределения команды make представляют собой средство удобной записи спецификаций взаимозависимостей и обрабатываются последней аналогично тому, как интерпретатор команд shell обрабатывает свои внутренние переменные. Ниже приведен формат макроопределения команды make:

```
<name> = <value> . . . . . <newline>
```

Как видно из формата, макроопределение представляет собой строку, содержащую символ =. Часть строки слева от символа = называется именем макроса команды make. Приведем пример макроопределения:

```
FRED = a b c d
```

Здесь макроопределение ставит в соответствие макросу FRED строку символов

```
a b c d
```

В результате обработки этого макроопределения команда make будет обрабатывать всякий контекст вида

```
$ (FRED)
```

так, как если бы то был контекст вида

```
a b c d
```

Использование круглых скобок при спецификации имени макроса команды make необходимо в том случае, если оно состоит более чем из одного символа. Использование макроопределений дает возможность добиться большей наглядности содержимого файла описаний, что зачастую бывает немаловажно.

ВСТРОЕННЫЕ ПРАВИЛА

Рассмотрим следующую спецификацию взаимозависимостей:

```
editor: main.o commands.o startup.o
```

Приведенная спецификация взаимозависимостей не содержит перечня команд ОС UNIX, с помощью которых указанные в ней объектные файлы могут быть получены из соответствующих исходных файлов. И тем не менее содержащейся в ней информации достаточно для того, чтобы команда make выполнила все действия, необходимые для получения исполняемого файла с именем editor. Дело в том, что в процессе функционирования команда make использует набор встроенных правил, который, в частности, содержит правило автоматического установления взаимозависимостей между файлами по суффиксам их имен. Это правило и содержит всю ту последовательность команд ОС UNIX, с помощью которых из исходного файла можно получить соответствующий объектный файл. Так, если исходный файл содержит программу на языке Си и соответственно имя исходного файла имеет суффикс .c, то указанное правило имеет вид

```
CC = cc
.c.o:
    $(CC) -c $(CFLAGS) $@ $*.c
```

Здесь имя целевого файла вида .c.o. указывает на то, что префиксы имен исходного и объектного файлов должны совпадать, а суффиксы имеют соответственно вид .c и .o. Кроме того, здесь были использованы внутренние макросы команды make. Перечислим имена внутренних макро-

сов команды make: *, @, <, и ?. При этом использование контекста $* равносильно спецификации одного лишь префикса имени целевого файла; использование контекста $@ равносильно спецификации полного имени целевого файла; использование контекста $< равносильно спецификации списка имен целевых файлов, от которых зависит данный целевой файл; наконец, использование контекста $? равносильно спецификации списка имен файлов, от которых зависит целевой файл, и дата последней модификации имен файлов старше даты последней модификации целевого файла.

Необходимо заметить, что пользователь может автоматически изменить используемое встроенное правило, для чего оно должно быть описано явно, т. е. должно содержаться в файле описаний. В случае только что рассмотренного нами правила автоматического установления взаимозависимостей файлов в соответствии с суффиксами их имен такое явное описание встроенного правила команды make может иметь вид

```
.SUFFIXES: .c .o .f .s
```

ЯВНОЕ ОПРЕДЕЛЕНИЕ ВСТРОЕННОГО ПРАВИЛА

Рассмотрим следующую простую командную строку ОС UNIX:

```
cc -o prog prog.c
```

Эта командная строка регулярно используется для выполнения компиляции и компоновки исходных файлов. Взаимозависимость специфицированных в ней файлов очевидна и не требует использования столь сложного встроенного правила команды make, которое было описано выше. Это, в частности, означает, что можно скорректировать указанное встроенное правило, поместив для этого в файл описаний следующие строки:

```
.c:
  cc -o $@ $*.c
```

Если теперь ввести с терминала

```
make prog
```

то исходный файл с именем prog.c будет скомпилирован, а затем скомпонован; при этом соответствующий объектный файл получит имя prog.o, а соответствующий исполняемый файл — имя prog.

ФЛАГИ КОМАНДЫ MAKE

Описание принципов работы и возможностей команды make — весьма сложная задача, особенно для большой и разветвленной программной системы, и мы не испытываем большого желания непременно решить ее в рамках этой главы. Вместе с тем не упомянуть о двух чрезвычайно полезных флагах команды просто невозможно.

Итак, рассмотрим использование флага — n команды make. Указание его в составе командной строки, с помощью которой была вызвана на выполнение команда make, приводит к тому, что вместо того, чтобы выполнить специфицированные в файле описаний последовательности команд ОС UNIX, команда make осуществит лишь вывод на терминал соответствующих командных строк. Таким образом, решение задачи отладки подготовленного пользователем файла описаний становится тривиально простым. Например, если ввести с терминала:

```
make -n
```

то спустя всего 20 с можно получить убедительное подтверждение правильности или ошибочности подготовленного файла описаний.

Рассмотрим теперь использование еще одного флага команды make — флага — к. Обычно, если в процессе обработки командой make файла описаний возникает аварийная ситуация, связанная с выполнением специфицированных последовательностей команд ОС UNIX, то выполнение команды make немедленно завершается. Если же в командной строке, с помощью которой была вызвана на выполнение команда make, указан флаг — к, то обработка ею файла описаний будет продолжена до тех пор, пока это окажется возможным. Таким образом, использование флага — к команды make позволит Вам получить максимум информации о содержащихся в модулях Вашей программной системы ошибках за один прогон команды make, сэкономив тем самым Ваше и машинное время.

Проиллюстрируем использование флага — к на следующем примере. Пусть программа, исходный текст которой находится в файле commands.c, содержит синтаксические ошибки. В таком случае использование команды make без указания флага — k даст следующий результат:

```
$ make
cc -O -c main.c
cc -O -c commands.c
\#11"commands.c", line1: warning: old-fashioned initialization: use =
\#10"commands.c", line 1: syntax error
\#11"commands.c", line1: warning: old-fashioned initialization: use =
\#10*** Error code 1

Stop.
```

в то время как использование флага — k позволит выполнить почти всю специфицированную последовательность команд ОС UNIX:

```
$ make -k
cc -O -c main.c
cc -O -c commands.c
\#11"commands.c", line1: warning: old-fashioned initialization: use =
\#10"commands.c", line 1: syntax error
\#11"commands.c", line1: warning: old-fashioned initialization: use =
\#10*** Error code 1
cc -O -c startup.c
```

```
(cd ../library; make CFLAGS = -O)
cc -O -c screen.c
cc -O -c files.c
cc -O -c edcom.c
"editor" not remade because of errors
```

УПРАВЛЕНИЕ БОЛЬШИМИ ПРОГРАММНЫМИ ПРОЕКТАМИ

Только что рассмотренное инструментальное средство make может быть с успехом использовано для управления построением и поддержкой программных систем, представляющих собой сложную взаимосвязанную совокупность отдельных программных модулей, каждый из которых представлен файлом, содержащим исходный текст программного модуля. Существуют, однако, ситуации, когда использование инструментального средства оказывается недостаточным. Речь идет о больших программных проектах, в процессе создания которых составляющие его программные компоненты подвергаются многократным преобразованиям и могут быть использованы одновременно в нескольких различных промежуточных формах. Примером такого программного проекта может служить программная система, в процессе создания и развития которой ее отдельные компоненты, имеющие вид исходных файлов, модифицируются одновременно несколькими программистами, в результате чего формируются различные версии каждого из этих файлов и конечного программного продукта в целом.

Мы рассмотрим сейчас инструментальное средство, называемое системой управления программным проектом (SCCS — Source Code Control System), с помощью которого может быть решена описанная выше проблема. Итак, инструментальное средство SCCS представляет собой пакет программ, который может функционировать под управлением ОС System III и ОС System V, файловая система которых поддерживает такой атрибут, как версия файла. Поясним сказанное. В процессе модификации исходного файла, находящегося под управлением инструментального средства SCCS, изменение его содержимого фиксируется инструментальным средством SCCS всякий раз, когда пользователь осуществляет запись файла на магнитный диск. Таким образом, текущая версия исходного файла представляет собой совокупность его текущего содержимого и информации и последовательных модификациях этого содержимого, начиная с его первой версии.

Упомянутая информация о последовательных модификациях исходного файла, очевидно, может быть представлена в виде набора последовательных изменений его содержимого, каждому элементу которого присваивается уникальный идентификационный номер — номер версии. Например, первая версия исходного файла может иметь идентификационный номер 1.1, а две последующие — идентификационные номера 1.2 и 1.3 соответственно. Подобный подход дает пользователю возможность хранить в файле с одним и тем же именем несколько версий его содержимого. Вместе с тем при осуществлении ссылки на этот исходный

файл пользователь может специфицировать различные идентификационные номера версий и таким образом обработать последовательно несколько различных вариантов исходного файла. Заметим, что при отсутствии спецификации идентификационного номера версии инструментальное средство SCCS автоматически использует последнюю версию файла.

ЦИКЛ ПРЕОБРАЗОВАНИЙ ПРОГРАММНОГО ПРОЕКТА

На рис. 9. 2 схематически изображен функциональный цикл преобразований программного проекта, реализуемого под управлением инструментального средства SCCS. Из этого рисунка хорошо видно, какие состояния проходит программный проект на пути своего преобразования и какими командами инструментального средства SCCS эти преобразования реализуются. Итак, первым состоянием программного проекта, очевидно, является первая версия исходного файла, полученная в результате передачи некоторого файла, содержащего исходный текст некоторой программы, под управление инструментального средства SCCS. Начиная с этого момента все преобразования такого исходного файла осуществляются только под управлением инструментального средства SCCS.

ПЕРЕДАЧА ИСХОДНОГО ФАЙЛА ПОД УПРАВЛЕНИЕ ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА SCCS

Любой исходный файл, независимо от того, содержит он исходный текст некоторой программы или текст журнальной статьи, может быть передан под управление инструментального средства SCCS, для чего необходимо воспользоваться командой admin и ввести с терминала, например, следующее:

```
admin -ifiles s.file
```

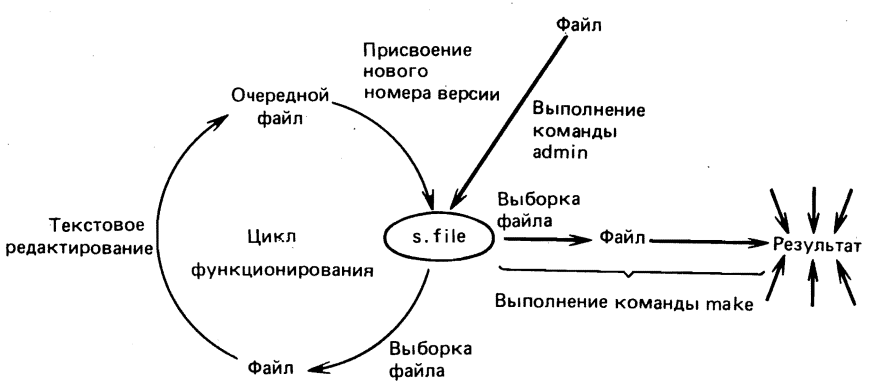

Рис. 9. 2. Схема функционирования пакета SCCS

В соответствии с принятым в ОС UNIX соглашением имя исходного файла, находящегося под управлением инструментального средства SCCS, имеет префикс s и суффикс, совпадающий с именем, которое имел исходный файл до передачи его под управление инструментального средства SCCS. Специфицированный в приведенной выше командной строке флаг — i указывает инструментальному средству SCCS на тот факт, что файл с именем file впервые передается под управление инструментального средства SCCS. Обычно после ввода такой командной строки на терминале появляется сообщение

```
No id keywords (cm7)
```

Появление этого сообщения означает, что передаваемый под управление инструментального средства SCCS исходный файл не содержит ключевых слов SCCS (позже мы объясним, что это такое), и Вы можете проигнорировать его.

После этого оригинальный исходный файл может быть удален из файловой системы ОС UNIX, так как для восстановления его содержимого (разумеется, с учетом всех осуществленных модификаций) достаточно ввести с терминала следующую строку:

```
get  s.file
```

В результате в текущем каталоге появится исходный файл ОС UNIX, не находящийся под управлением инструментального средства SCCS, доступ к которому разрешен только по чтению. В то же время ввод командной строки вида

```
get  -p  s.file
```

приведет к выводу содержимого исходного файла на стандартный вывод.

МОДИФИКАЦИЯ ФАЙЛА, НАХОДЯЩЕГОСЯ ПОД УПРАВЛЕНИЕМ ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА SCCS

Для того чтобы осуществить модификацию (например, с помощью редактора текста) содержимого исходного файла, находящегося под управлением инструментального средства SCCS, необходимо ввести с терминала следующую командную строку:

```
get  -e  s.file
```

В результате этого в текущем каталоге появится файл, доступ к которому разрешен и по чтению, и по записи, представляющий собой копию оригинального исходного файла, а также будет создан рабочий файл инструментального средства SCCS. После того как редактирование исходного файла завершено, необходимо сообщить об этом инструментальному средству SCCS с помощью следующей командной строки:

```
delta  s.file
```

после чего на экране появится предложение ввести комментарий:

```
comments?
```

Теперь пользователь может при желании ввести текст комментария, поясняющего проделанную модификацию, например:

```
comments? added support for more users
```

После этого инструментальное средство SCCS создаст новую версию исходного файла, находящегося под управлением инструментального средства SCCS (в нашем примере это версия 1.2), и цикл преобразований исходного файла будет завершен.

Приведенное описание цикла преобразований исходного файла, находящегся под управлением инструментального средства SCCS, более чем кратко, однако в большинстве случаев этого вполне достаточно, а желающие познакомиться с ним более подробно могут обратиться к документу UPM.

ОБЗОР ПОСЛЕДОВАТЕЛЬНЫХ МОДИФИКАЦИЙ ИСХОДНОГО ФАЙЛА

Как мы показали выше, с помощью команды delta можно указать инструментальному средству SCCS на необходимость зафиксировать осуществленные модификации исходного файла. В такой ситуации было бы естественным потребовать от инструментального средства SCCS вывести на терминал всю последовательность осуществленных преобразований содержимого исходного файла, иначе говоря, "историю развития" файла. Такая возможность существует, для ее реализации необходимо ввести следующую командную строку:

```
prs  s.chlib.c
```

В результате на терминал будет выведено следующее:

```
s.chlib.c:

D 1.4 84/08/28 12:27:57 beta 4 3        00010/00000/01011
MRs:
COMMENTS:
changes and fixes for real interface

D 1.3 84/08/28 16:04:42 beta 3 2        00003/00001/01008
MRs:
COMMENTS:
stopped eof char being put on clist in rxin - geraint

D 1.2 84/08/03 14:43:27 beta 2 1        00017/00003/00992
MRs:
COMMENTS:
added changes for delayed opens in modem control - geraint

D 1.1 84/07/31 14:15:17 beta 1 0        00995/00000/00000
MRs:
COMMENTS:
date and time created 84/07/31 14:15:17 by beta
```

КОМАНДА make И ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО SCCS

Внимательный читатель наверняка уже понял, насколько полезным может стать для него инструментальное средство SCCS, однако у него может возникнуть идея использовать совместно инструментальное средство SCCS и команду make. И это действительно возможно, так как версия команды make, входящая в состав ОС System III и ОС System V, располагает соответствующими встроенными правилами, позволяющими ей автоматически обрабатывать команды инструментального средства SCCS.

Для иллюстрации сказанного вернемся к рассмотренному выше примеру разработки экранного редактора текста. Пусть, например, файл editor был передан нами под управление инструментального средства SCCS, в таком случае вызов на выполнение команды make и обработка последней соответствующего файла описаний приведут к выполнению последовательности команд ОС UNIX вида

```
get -p s.main.c > main.c
cc -O -c main.c
rm -f main.c
get -p s.commands.c > commands.c
cc -O -c main.c
rm -f commands.c
get -p s.startup.c > startup.c
cc -O -c startup.c
rm -f startup.c
(cd ../library; make  CFLAGS = -O)
get -p s.screen.c > screen.c
cc -O -c screen.c
rm -f screen.c
get -p s.graphics.c > graphics.c
cc -O -c graphics.c
rm -f graphics.c
get -p s.files.c > files.c
cc -O -c files.c
rm -f files.c
get -p s.edcom.c > edcom.c
cc -O -c edcom.c
rm -f edcom.c
cc -o editor main.o commands.o startup.o ../library/screen.o \
```

КЛЮЧЕВЫЕ СЛОВА SCCS

Вы, наверное, помните выведенное на терминал инструментальным средством SCCS сообщение об отсутствии в исходном файле ключевых слов SCCS, после того как Вы ввели с терминала команду admin:

```
No id keywords (cm7)
```

Поясним значение этого сообщения. Оно говорит о том, что инструментальное средство SCCS не обнаружило в составе обрабатываемого исходного файла текстовых строк специальной структуры, которые оно ис-

пользует для хранения текущей версии файла и которые понадобятся ему при выполнении команды get.

Если переданный под управление инструментального средства SCCS исходный файл содержит текст программы на языке Си, то в его состав может входить целый ряд специальных символов, которые после обработки инструментальным средством SCCS будут заменены контекстом, содержащим такую полезную информацию, как, например, текущая дата, имя программного модуля, номер его версии и т. д. Единственным ограничением при этом является требование, чтобы указанные специальные символы были использованы в описаниях статических переменных, например:

```
static char SCCSid[] = "%W%"
```

После обработки такой строки инструментальным средством SCCS она примет вид

```
static char SCCSid[] = "@(#)fred.c
```

Использование этого приема позволяет сохранить информацию о текущей версии исходного файла даже в составе объектного файла, и это тем более полезно, что инструментальное средство SCCS способно отыскать в содержимом объектного файла указанный контекст и таким образом идентифицировать версию обрабатываемого файла. Для этого необходимо воспользоваться командой what, например:

```
$ what iop_nv
```

В результате на терминал будет выведено следующее:

```
iop_nv
        blklib.c    1.2
        chlib.c     1.4
        lbuf.c      1.1
        warn.c      1.1
        gcpc.c      1.3
        cio.c       1.2
        intr.c      1.1
        reg.c       1.3
        mesg.c      1.3
        init.c      1.1
        debugsch.c  1.1
        docvt.c     1.1
```

Описанная возможность чрезвычайно легка в реализации и может быть с успехом использована при построении сложной программной системы из объектных файлов.

ЗАДАЧИ

1. Создайте файл описаний команды make для одного из разрабатываемых Вами исходных файлов и попрактикуйтесь в использовании команды make, описанном в этой главе.

2. Исходный файл, содержащий текст, подготовленный для форматирования с помощью форматера nroff, должен иметь имя с суффиксом .n; разработайте пос-

ледовательность правил, с помощью которых команда make смогла бы осуществить преобразование его в выходной файл форматера, содержащий сформатированный текст и имеющий имя с суффиксом .x.

3. Поместите исходный файл, использованный Вами для решения задачи 1, под управление инструментального средства SCCS и убедитесь в том, что использование последнего действительно удобно.

ЛИТЕРАТУРА

1. S.I. Feldman (1978), Make - A Program for Maintaining Computer Programs, UNIX V7, Volume 2a.

2. E.A. Bradford (1980), An Augmented Version of Make, UNIX SIII Volume 2b.

3. S.C. Johnson (1978), Lint - a C Program Checker, UNIX V7 Volume 2a.

4. L.E. Bonani (1980), SCCS/PWB User's Guide, UNIX SIII Volume 2b.

ОГЛАВЛЕНИЕ