```
k_r_4.pas          [-M--]  0 L:[ 41+31  72/137] *(2432/5678b)=    32 0x20
const
  o7777=(1 shl 12)-1; (*восьмеричная константа = все 12 бит прав заданы *)
  o10  =8;            (*010  *)
  o100 =64;           (*0100 *)
  o1000=512;          (*01000*)
  symrights:array [0..7] of string=( (*базовые комбинации прав в символьной форме*)
       '---', (*0 = 000*)
       '--x', (*1 = 001*)
       '-w-', (*2 = 010*)
       '-wx', (*3 = 011*)
       'r--', (*4 = 100*)
       'r-x', (*5 = 101*)
       'rw-', (*6 = 110*)
       'rwx'  (*7 = 111*)
     );
  spec='tss';          (*массив специальных прав доступа*)
begin
  (*обрезаем старшие биты, не относящиеся к правам доступа (тип файла и т.п.)*)
  r:=r and o7777;(*восьмеричная константа 10000-1==1*8^4-1==1*(2^3)^4-1==2^12-1 *)
  (*выделяем числовые права для владельца, группы, остальных + специальные*)
  o:=r mod o10;
  s:=r div o1000;
  u:=(r div o100) mod o10;
  g:=(r mod o100) div o10;
  res:=symrights[u]+symrights[g]+symrights[o];(*формируем символьные права из базовых троек*)

  for i:=1 to 3 do   (*цикл проверки наличия чпециальных прав*)
    if s and (1 shl (i-1)) <> 0 then (*если право установлено*)
      if res[12-3*i]='x' then  (*если есть обычное право на выполнение*)
        res[12-3*i]:=spec[i]   (*заносим маленькую букву*)
      else
        res[12-3*i]:=upcase(spec[i]); (*иначе - большую*)

  getrights:=res; (*возвращаем результат - 9-символьное представление 12-битных прав*)
end;
1Помощь 2Запись 3Блок  4Замена 5Копия  6Перемес7Поиск  8Удалить9МенюMC 10Выход
```

Системное программирование в UNIX средствами Free Pascal

```
-rw-------    1          10 Tue Jul  6 20:23:35 2004 .octave_hist
drwxr-xr-x    5        4096 Tue Jul  6 20:28:50 2004 GNUstep
-rw-r--r--    1         243 Tue Jul  6 20:39:17 2004 gtkwin.c
-rw-r--r--    1         177 Tue Jul  6 20:38:36 2004 makefile
-rw-r--r--    1      132504 Tue Jul  6 20:39:33 2004 gtkwin.o
-rwxr-xr-x    1      101678 Tue Jul  6 20:40:30 2004 gtkwin
drwxr-xr-x    4        4096 Tue Jul  6 20:56:17 2004 Projects
-rw-rw-rw-    1       10764 Sun Jul 25 14:22:22 2004 maxima.1
-rw-r--r--    1        4642 Wed Jun 27 22:33:44 2001 lseek.2
drwxr-xr-x    5        4096 Sat Feb 12 00:15:48 2005 sysprfpc
-rw-r--r--    1        1108 Sat Aug 20 12:07:32 2005 .blackboxrc
[student@linux auto]$ ./k_r_4 .
drwxrwxrwx    3        7168 Thu Jan  1 03:00:00 1970 .
drwxr-xr-x    3           0 Sat Aug 20 20:12:47 2005 ..
-rwxrwxrwx    1        3024 Sat Aug 20 19:56:06 2005 pfinger.o
-rwxrwxrwx    1      128866 Sat Aug 20 19:56:06 2005 pfinger
-rwxrwxrwx    1       10548 Sat Aug 20 19:56:14 2005 serverex.o
-rwxrwxrwx    1      206714 Sat Aug 20 19:56:14 2005 serverex
-rwxrwxrwx    1        7716 Sat Aug 20 20:12:00 2005 k_r_4.o
-rwxrwxrwx    1      201035 Sat Aug 20 20:12:00 2005 k_r_4
-rwxrwxrwx    1      365056 Thu Sep  1 07:38:04 2005 tasks.doc
drwxrwxrwx    2         512 Thu Sep 15 12:42:32 2005 copya
-rwxrwxrwx    1         716 Thu Jul 13 12:10:32 2000 pfinger.pp
-rwxrwxrwx    1        8213 Thu Jul 13 12:10:32 2000 serverex.pp
-rwxrwxrwx    1        5772 Sat Aug 20 20:11:56 2005 k_r_4.pas
[student@linux auto]$
```

А.П. Полищук, С.А. Семериков

Системное программирование в UNIX средствами Free Pascal

Учебное пособие

Кривой Рог
Издательский отдел КГПУ
2005

Полищук А.П., Семериков С.А.

Системное программирование в UNIX средствами Free Pascal: Учебное пособие. – Кривой Рог: Издательский отдел КГПУ, 2005. – 418 с.

Учебное пособие, ориентированное на программирующего пользователя, посвящено методам системного программирования в UNIX-подобных операционных системах. Приведен лабораторный практикум по созданию приложений средствами компилятора Free Pascal. Изложение ориентировано на ОС Linux, однако все приведенные сведения могут быть применены в любой ОС, поддерживающей стандарты POSIX.

Для студентов высших учебных заведений, аспирантов, научных и инженерно-технических работников.

Рецензенты:

д. т. н., проф. **А.Н. Марюта**
(Днепропетровский национальный университет)
д. ф.-м. н., проф. **В.Н. Соловьёв**
(Криворожский экономический институт КНЭУ)

Подп. к печати 17.10.2005 Формат 80x84 1/16.
Бумага офсетная №1 Усл. печ. л. 24,31
Усл. кр.-от. 26 Уч.-изд. л. 24,82
Тираж 300 Зак. №03-1710

КГПУ, 50086, Кривой Рог-86, пр. Гагарина, 54

Криворожская городская типография
50050, Кривой Рог-50, пр. Металлургов, 28.

Содержание

Предисловие

О книге

В основу данной книги положено второе издание руководства программиста UNIX System Programming: A programmer's guide to software development by Keith Haviland, Dina Gray, Ben Salama. Очень удачное по структуре и подбору примеров, это руководство является одним из лучших учебников по системному программированию в UNIX, поэтому с самого начала мы посчитали уместным сохранить их, исправив и дополнив в соответствии с новыми возможностями Linux/BSD и компилятора Free Pascal.

На первом этапе нашей работы был создан модуль `stdio`, необходимый для совместимости со стандартной библиотекой языка Си. В модуль вошли множество структур данных, процедур и функций, не входящих в библиотечные модули Free Pascal, но существенно облегчающие жизнь программиста.

На втором этапе примеры из книги Кейт Хэвиленд, Даны Грей и Бена Саламы были переведены с Си на Паскаль. Это потребовало модификации значительной части текста книги, посвященной описанию используемых библиотечных функций и системных вызовов.

Наконец, книга была дополнена описанием структур данных, процедур и функций библиотечных модулей `linux`, `ipc` и `sockets`, специфичных для ОС Linux/BSD.

В результате проделанной работы была получена данная книга, в которой сохранилось часть исходного текста из книги Кейт Хэвиленд, Даны Грей и Бена Саламы. Разумеется, при необходимости эти фрагменты могут быть заменены на другие, но результатом этого будет всего лишь изложение известной справочной информации иными словами.

Назначение этой книги

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

Операционная система UNIX оправдала возлагавшиеся на нее надежды и теперь является ключевой деталью технологического пейзажа на рубеже XXI века. Не говоря уже о том, что UNIX всегда занимала сильные позиции в научном и техническом сообществах, в настоящее время существует множество крупномасштабных систем управления данными и обработки транзакций на платформе UNIX. Но, самое главное, ОС UNIX, безусловно, является ядром серверов магистральной сети Internet.

Основное внимание в книге уделяется программному интерфейсу между ядром UNIX (частью UNIX, которая делает ее операционной системой) и прикладным программным обеспечением, которое выполняется в среде UNIX. Этот интерфейс часто называется интерфейсом системных вызовов UNIX (хотя разница между такими вызовами и обычными библиотечными процедурами теперь уже не столь очевидна, как это было раньше). В дальнейшем мы увидим, что системные вызовы – это примитивы, на которых в конечном счете построены все программы UNIX – и поставляемые вместе с операционной системой, и разрабатываемые независимо. Целевая аудитория книги состоит из программистов, уже знакомых с UNIX, которые собираются разрабатывать программное обеспечение для ОС UNIX на языке Pascal. Эта книга в равной мере подойдет разработчикам системного программного обеспечения и прикладных или деловых приложений – фактически всем, кто серьезно интересуется разработкой программ для ОС UNIX.

Кроме системных вызовов мы также рассмотрим некоторые из наиболее важных библиотек подпрограмм, которые поставляются с системой UNIX. Эти библиотеки, конечно же, написаны с использованием системных вызовов и во многих случаях делают то же самое, но на более высоком уровне, или более удобны для использования программистами.

Надеемся, что при исследовании как системных вызовов, так и библиотеки подпрограмм вы получите представление о том, когда можно пользоваться существующими достижениями, не «изобретая велосипед», а также лучше поймете внутреннее устройство этой все еще прекрасной операционной системы.

Спецификация X/Open

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

Мы положили в основу текста и примеров документы (все датированные 1994 годом) System Interface Definitions (Описания системного интерфейса) и System Interfaces and Headers (Системные интерфейсы и заголовки) из 4-й версии второго выпуска X/Open, а также часть связанного с ними документа Networking Services (Сетевые сервисы). Для удобства мы будем применять сокращение XSI – от X/Open System Interfaces (Системные интерфейсы X/Open). При необходимости особенности различных реализации системы будут обсуждаться отдельно.

Здесь нужно сделать небольшое отступление. Консорциум X/Open первоначально объединил производителей аппаратного обеспечения, серьезно заинтересованных в открытых операционных системах и платформе UNIX, но со временем число его членов возросло. Одна из главных задач консорциума состояла 3 в выработке практического стандарта UNIX, и руководство по обеспечению мобильности программ, называемое XPG, послужило базовым документом для проекта нескольких основных производителей (включая Sun, IBM, Hewlett Packard, Novell и Open Software Foundation), обычно называемого Spec 1170 Initiative (1170 – это число охватываемых этим документом вызовов, заголовков, команд и утилит). Целью этого проекта было создание единой унифицированной спецификации системных сервисов UNIX, включая системные вызовы, которые являются основой этого документа. В результате получился удобный набор спецификации, объединивший многие конфликтующие направления в стандартизации UNIX, главную часть которых составляют вышеупомянутые документы. Другие представленные разработки охватывали основные команды UNIX и обработку вывода на экран.

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

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

В заключение заметим, что:

– все стандарты являются очень обширными, охватывая альтернативные методы реализации и редко используемые, но все же важные функциональные возможности. Основное внимание в этой книге уделяется основам программирования в среде UNIX, а не полному описанию системы в соответствии

с базовыми стандартами;
– при необходимости создания программы, строго следующей стандартам, необходимо предусмотреть в ней установку (и проверку) соответствующих флагов, таких как _XOPEN_SOURCE или _POSIX_SOURCE.

Структура книги

Книга состоит из тринадцати глав.

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

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

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

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

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

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

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

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

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

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

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

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

И, наконец, глава 13 содержит задачи с решениями, обобщающими весь предыдущий материал.

Что вы должны знать

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

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

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

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

Упражнение 1. *Объясните назначение следующих команд UNIX:*

`ls cat rm cp mv mkdir fpc`

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

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

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

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

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

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

Упражнение 7. *Напишите программу, которая подсчитывает и выводит число вводимых слов, строк и символов при помощи функций*.

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

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

Соглашения

В книге приняты следующие выделения:

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

Глава 1. Основные понятия и терминология

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

1.1. Файл

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

```
$ vi my_test.pas
```

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

```
$ cat my_test.pas
```

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

```
$ fpc my_test.pas
```

которая вызовет компилятор языка Паскаль для создания программы `my_test` из исходного файла `my_test.pas`, если файл `my_test.pas` не содержит ошибок.

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

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

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

1.1.1. Каталоги и пути

Важное понятие, связанное с файлами, – *каталог* (directory). Каталоги представляют собой набор файлов, позволяя сформировать некоторую логическую структуру содержащейся в системе информации. Например, каждый пользователь обычно имеет начальный каталог, в котором он работает, а команды, системные библиотеки и программы администрирования обычно расположены в своих определенных каталогах. Кроме файлов, каталоги также могут содержать произвольное число подкаталогов, которые, в свою очередь, также могут содержать подкаталоги, и так далее. Фактически каталоги могут иметь любую

глубину вложенности. Таким образом, файлы UNIX организованы в иерархической древовидной структуре, в которой каждый узел, кроме конечных, соответствует каталогу. Вершиной этого дерева является один каталог, которая обычно называется *корневым каталогом* (root directory).

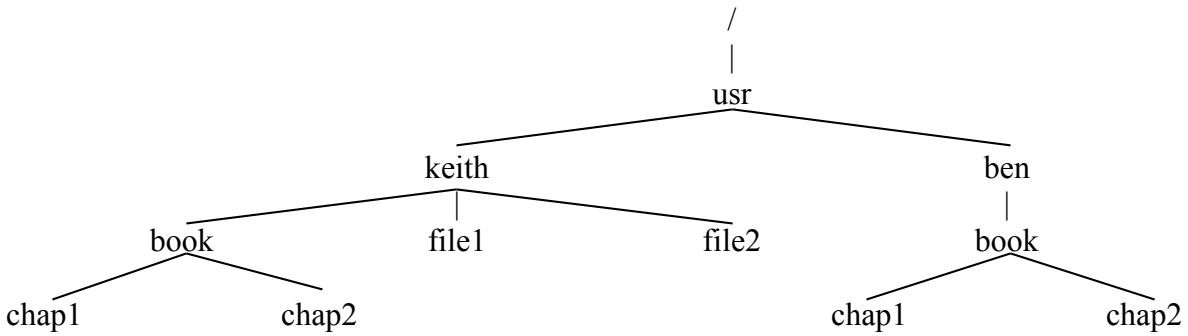


Рис. 1.1. Пример дерева каталогов

Подробное рассмотрение структуры каталогов системы UNIX содержится в главе 4. Тем не менее, поскольку в книге постоянно будет идти речь о файлах UNIX, стоит отметить, что полные их имена, которые называются *путями* (pathnames), отражают эту древовидную структуру. Каждый путь задает последовательность ведущих к файлу каталогов. Например, полное имя

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

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

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

```
chap1/intro.txt
```

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

```
intro.txt
```

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

1.1.2. Владелец файла и права доступа

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

1.1.3. Обобщение концепции файла

В UNIX концепция файлов расширена и охватывает не только *обычные файлы* (regular files), но и периферийные устройства, а также каналы межпроцессного взаимодействия. Это означает, что одни и те же примитивы могут использоваться для записи

и чтения из текстовых и двоичных файлов, терминалов, накопителей на магнитной ленте и даже оперативной памяти. Данная схема позволяет рассматривать программы как обобщенные инструменты, способные использовать любые типы устройств. Например,

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

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

1.2. Процесс

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

```
$ ls
```

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

1.2.1. Межпроцессное взаимодействие

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

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

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

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

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

1.3. Системные вызовы и библиотечные подпрограммы

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

Системные вызовы фактически дают разработчику программного обеспечения доступ к *ядру* (kernel). Ядро, с которым познакомились в предисловии, – это блок кода, который постоянно находится в памяти и занимается планированием системных процессов UNIX и управлением вводом/выводом. По существу ядро является частью UNIX, которое и делает ее операционной системой. Все управление, выделение ресурсов и контроль над пользовательскими процессами, а также весь доступ к файловой системе осуществляется через ядро.

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

```
read(fileptr, inputbuf);
```

или при помощи низкоуровневого системного вызова `fdread`:

```
nread := fdread(filedes, inputbuf, BUFSIZE);
```

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

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

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

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

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

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

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

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

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

Глава 2. Файл

2.1. Примитивы доступа к файлам в системе UNIX

2.1.1. Введение

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

Типичная программа UNIX вызывает для инициализации файла вызов `fdopen` (или `fdcreat`), а затем использует вызовы `fdread`, `fdwrite` или `fdseek` для работы с данными в файле. Если файл больше не нужен программе, она может вызвать `fdclose`, показывая, что работа с файлом завершена. Наконец, если пользователю больше не нужен файл, его можно удалить из системы при помощи вызова `unlink`.

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

Таблица 2.1. Примитивы UNIX

Имя	Функция
`fdopen`	Открывает файл для чтения или записи либо создает пустой файл
`fdcreat`	Создает пустой файл
`fdclose`	Закрывает открытый файл
`fdread`	Считывает информацию из файла
`fdwrite`	Записывает информацию в файл
`fdseek`	Перемещается в заданную позицию в файле
`unlink`	Удаляет файл
`fcntl`	Управляет связанными с файлом атрибутами

```
(* элементарный пример                    *)

uses linux;

var
  fd:integer;
  nread:longint;
  buf:array [0..1024-1] of char;
begin
  (* Открыть файл 'data' для чтения *)
  fd := fdopen ('data', Open_RDONLY);
  (* Прочитать данные *)
  nread := fdread (fd, buf, 1024);
  (* Закрыть файл *)
  fdclose (fd);
end.
```

Первый системный вызов программа примера делает в строке

```
fd := fdopen ('data', Open_RDONLY);
```

Это вызов функции `fdopen`, он открывает файл `data` в текущем каталоге. Второй аргумент функции, `Open_RDONLY`, является целочисленной константой, определенной в модуле `linux`. Это значение указывает на то, что файл должен быть открыт в режиме *только*

для чтения (read only). Другими словами, программа сможет только читать содержимое файла и не изменит файл, записав в него какие-либо данные.

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

После открытия файла программа делает системный вызов `fdread`:

```
nread := fdread (fd, buf, 1024);
```

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

Переменная `nread` имеет тип `longint`, определенный в модуле `linux`.

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

В конце примера файл закрывается:

```
fdclose(fd);
```

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

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

2.1.2. Системный вызов fdopen

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

Описание

```
uses linux;

Function  fdOpen(PathName:String;flags:longint):longint;
Function  fdOpen(PathName:Pchar;flags:longint):longint;
Function  fdOpen(PathName:String;flags,mode:longint):longint;
```

```
Function  fdOpen(PathName:Pchar;flags,mode:longint):longint;
```

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

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

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

```
keith/junk
```

или просто:

```
junk
```

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

Второй аргумент системного вызова fdopen, который в нашем описании называется flags, имеет целочисленный тип и определяет метод доступа. Параметр flags принимает одно из значений, заданных постоянными в модуле linux при помощи директивы const (fcntl является сокращением от *file control*, «управление файлом»). Так же, как и большинство стандартных модулей, файл linux.ppu может быть включен в программу при помощи директивы:

```
uses linux;
```

В модуле linux определены три постоянных, которые сейчас представляют для нас интерес:

Open_RDONLY Открыть файл только для чтения

Open_WRONLY Открыть файл только для записи

Open_RDWR Открыть файл для чтения и записи

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

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

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

```
uses linux;

const
  workfile = 'junk';    (* задать имя рабочего файла *)

var
  filedes:integer;
begin
```

```
  (* Открыть, используя постоянную Open_RDWR из модуля linux *)
  (* Файл открывается для чтения/записи *)
  filedes := fdopen (workfile, Open_RDWR);
  if filedes = -1 then
  begin
    writeln('Невозможно открыть файл ', workfile);
    halt(1);                  (* выход по ошибке *)
  end;
  writeln('Файл ', workfile, ' успешно открыт, дескриптор равен ', filedes);
  (*
   * Остальная программа
   *)
  halt(0);              (* нормальный выход *)
end.
```

Обратите внимание, что используется `halt` с параметром 1 в случае ошибки, и 0 – в случае удачного завершения. Это соответствует соглашениям UNIX и является правильной практикой программирования. Как будет показано в следующих главах, после завершения программы можно получить передаваемый вызову `halt` аргумент (*program's exit status* – код завершения программы).

Предостережение

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

```
filedes := fdopen(filename, 0);
```

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

```
filedes := fdopen (filename, Open_RDONLY);
```

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

2.1.3. Создание файла при помощи вызова fdopen

Вызов `fdopen` может использоваться для создания файла, например:

```
filedes := fdopen('/tmp/newfile', Open_WRONLY or Open_CREAT, octal(0644));
```

Здесь объединены флаги `Open_CREAT` и `Open_WRONLY`, задающие создание файла `/tmp/newfile` при помощи вызова `fdopen`. Если `/tmp/newfile` не существует, то будет создан файл нулевой длины с таким именем и открыт только для записи.

В этом примере вводится третий параметр `mode` вызова `fdopen`, который нужен только при создании файла. Не углубляясь в детали, заметим, что параметр `mode` содержит число, определяющее *права доступа* (access permissions) к файлу, указывающие, кто из пользователей системы может осуществлять чтение, запись или выполнение файла. В

[1] Согласно спецификации SUSV2 заголовочный файл `limits.h` должен определять константу OPEN_MAX, задающую это значение.

вышеприведенном примере используется восьмеричное значение 0644. При этом пользователь, создавший файл, может выполнять чтение из файла и запись в него. Остальные пользователи будут иметь доступ только для чтения файла. В следующей главе показано, как вычисляется это значение. Для простоты оно будет использовано во всех примерах этой главы.

Следующая программа создает файл `newfile` в текущем каталоге:

```
uses linux;

const
  PERMS=0644;           (* права доступа при открытии с Open_CREAT *)
  filename='newfile';

var
  filedes:integer;

begin
  filedes := fdopen (filename, Open_RDWR or Open_CREAT, octal(PERMS));
  if filedes = -1 then
  begin
    writeln('Не могу создать ', filename);
    halt(1);                  (* выход по ошибке *)
  end;

  writeln('Файл ', filename, ' успешно создан (открыт для записи), дескриптор
равен ',filedes);

  (* Остальная программа *)

  halt(0);
end.
```

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

```
fd := fdopen('lock', Open_WRONLY or Open_CREAT or Open_EXCL, octal(0644));
```

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

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

```
fd := fdopen ('file', Open_WRONLY or Open_CREAT or Open_TRUNC, octal(0644));
```

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

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

Существует возможность установить размер файла не только в 0, но и в любое заданное количество байт. Это позволяет сделать функция `fdTruncate`.

Описание

```
uses linux;
```

```
Function fdTruncate (fd,size:longint): boolean;
```

`fdTruncate` устанавливает длину файла, заданного дескриптором `fd`, в `size` байт, где `size` должен быть меньше либо равен текущей длине файла `fd`. Функция возвращает `True`, если вызов был успешен, и `false` в случае ошибки.

2.1.4. Системный вызов fdcreat

Другой способ создания файла заключается в использовании системного вызова `fdcreat`. В действительности это исходный способ создания файла, но сейчас он в какой-то мере является излишним и предоставляет меньше возможностей, чем вызов `fdopen`. Мы включили его для полноты описания. Так же, как и вызов `fdopen`, он возвращает либо ненулевой дескриптор файла, либо -1 в случае ошибки. Если файл успешно создан, то возвращаемое значение является дескриптором этого файла, открытого для записи. Этот вызов не входит в модуль `linux`, поэтому для его использования в программе потребуется предварительное описание:

Описание

```
uses stdio;

function Fdcreat(PathName:Pchar;mode:longint):longint;cdecl;external 'c';
```

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

```
filedes := fdcreat('/tmp/newfile', octal(0644));
```

что эквивалентно вызову

```
filedes := fdopen('/tmp/newfile', Open_WRONLY or Open_CREAT or Open_TRUNC,
octal(0644));
```

Ключевое слово `cdecl` определяет для данной функции стиль вызова, характерный для языка C. Это необходимо для доступа к функциям в объектных файлах, сгенерированных компилятором языка C, таким, как функции стандартной библиотеки языка C (это указывается с помощью ключевого слова `external`, параметром которого является имя библиотеки – `'c'`).

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

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

2.1.5. Системный вызов fdclose

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

Описание

```
uses linux;

Function fdClose(fd:longint):boolean;
```

Системный вызов `fdclose` имеет всего один аргумент – дескриптор закрываемого

файла, обычно получаемый в результате предыдущего вызова fdopen или fdcreat. Следующий фрагмент программы поясняет простую связь между вызовами fdopen и fdclose:

```
filedes := fdopen('file', Open_RDONLY);
.
.
.
fdclose(filedes);
```

Системный вызов fdclose возвращает true в случае успешного завершения и false – в случае ошибки (которая может возникнуть, если целочисленный аргумент не является допустимым дескриптором файла).

При завершении работы программы все открытые файлы закрываются автоматически.

2.1.6. Системный вызов fdread

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

Заметим, что программисты на языке Паскаль часто любят использовать термины «символ» и «байт» как взаимозаменяемые. Байт является единицей памяти, необходимой для хранения символа, и на большинстве машин имеет длину восемь бит. Термин «символ» обычно описывает элемент из набора символов ASCII, который является комбинацией всего из семи бит. Поэтому обычно байт может содержать больше значений, чем число символов ASCII; такая ситуация возникает при работе с двоичными данными. Тип char языка C представляет более общее понятие байта, поэтому название данного типа является не совсем правильным.

Описание

```
uses linux;

Function fdRead(filedes:longint;var buffer;size:longint):longint;
```

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

```
var
  fd:integer;
  nread:longint;
  buffer:array [0..SOMEVALUE-1] of char;

(* Дескриптор файла fd получен в результате вызова fdopen *)
.
.
.
nread := fdread(fd, buffer, SOMEVALUE);
```

Как видно из примера, третьим параметром вызова fdread является положительное число (имеющее тип longint), задающее число байтов, которое требуется считать из файла.

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

Указатель чтения-записи

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

```
var
  fd:integer;
  n1,n2:longint;
  buf1, buf2 : array [0..511] of char;
.
.
.
fd := fdopen('foo', Open_RDONLY);
if fd = -1 then
  halt(-1);
n1 := fdread(fd, buf1, 512);
n2 := fdread(fd, buf2, 512);
```

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

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

Следующая программа `count` иллюстрирует некоторые из этих моментов:

```
(* Программа count подсчитывает число символов в файле *)

uses linux;

const
  BUFSIZE=512;

var
  filedes:integer;
  nread:longint;
  buffer:array [0..BUFSIZE-1] of byte;
  total:longint;
begin
  total := 0;

  (* Открыть файл 'anotherfile' только для чтения *)
  filedes := fdopen ('anotherfile', Open_RDONLY);
```

```
  if filedes=-1 then
  begin
    writeln('Ошибка при открытии файла anotherfile');
    halt(1);
  end;

  (* Повторять до конца файла, пока nread не будет равно 0 *)
  nread := fdread (filedes, buffer, BUFSIZE);
  while nread > 0 do
  begin
    inc(total,nread);         (* увеличить total на nread *)
    nread := fdread (filedes, buffer, BUFSIZE);
  end;

  writeln('Число символов в файле anotherfile: ', total);
  fdclose(filedes);
  halt(0);
end.
```

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

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

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

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

Следующая программа демонстрирует функции `fdread` и `fdtrucate`.

```
Uses linux;

Const Data : string[10] = '12345687890';

Var FD : Longint;
    l : longint;

begin
  FD:=fdOpen('test.dat',open_wronly or open_creat,octal(666));
  if fd>0 then
    begin
    { Fill file with data }
    for l:=1 to 10 do
      if fdWrite (FD,Data[1],10)<>10 then
```

```
        begin
        writeln ('Error when writing !');
        halt(1);
        end;
    fdClose(FD);
    FD:=fdOpen('test.dat',open_rdonly);
    { Read data again }
    If FD>0 then
      begin
      For l:=1 to 5 do
        if fdRead (FD,Data[1],10)<>10 then
          begin
          Writeln ('Error when Reading !');
          Halt(2);
          end;
      fdCLose(FD);
      { Truncating file at 60 bytes }
      { For truncating, file must be open or write }
      FD:=fdOpen('test.dat',open_wronly,octal(666));
      if FD>0 then
        begin
        if not fdTruncate(FD,60) then
           Writeln('Error when truncating !');
        fdClose (FD);
        end;
      end;
    end;
end.
```

2.1.7. Системный вызов fdwrite

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

Описание

```
uses linux;

Function fdWrite (fd:longint; var buf; size:longint):longint;
```

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

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

```
var
  fd:integer;
  w1, w2 : longint;
  header1: array [0..511] of char;
  header2: array [0..1023] of char;
```

```
.
.
.
fd := fdopen('newfile', Open_WRONLY or Open_CREAT or Open_EXCL, octal(0644));
if fd = -1 then
  exit;
w1 := fdwrite(fd, header1, 512);
w2 := fdwrite(fd, header2, 1024);
```

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

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

```
filedes := fdopen(filename, Open_WRONLY or Open_APPEND);
```

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

Программа, демонстрирующая `fdOpen`, `fdwrite` и `fdCLose`.

```
Uses linux;

Const Line : String[80] = 'This is easy writing !';

Var FD : Longint;

begin
  FD:=fdOpen ('Test.dat',Open_WrOnly or Open_Creat);
  if FD>0 then
    begin
    if length(Line)<>fdwrite (FD,Line[1],Length(Line)) then
      Writeln ('Error when writing to file !');
    fdClose(FD);
    end;
end.
```

2.1.8. Пример copyfile

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

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

Окончательное решение может выглядеть таким образом:

```
(* Программа copyfile: скопировать файл name1 в файл name2 *)

uses linux;

const
  BUFSIZE=512;          (* Размер считываемого блока *)
  PERM=0644;            (* Права доступа для нового файла в форме,
```

```
                        похожей на восьмеричную *)

(* Скопировать файл name1 в файл name2 *)
function copyfile(name1, name2: string):integer;
var
  infile, outfile: integer; (*дескрипторы файлов*)
  nread: longint;
  buffer: array [0..BUFSIZE-1] of byte; (*буфер для чтения/записи*)
begin

  infile := fdopen (name1, Open_RDONLY);
  if infile=-1 then
  begin
    copyfile:=-1; (*ошибка открытия первого файла*)
    exit;
  end;

  outfile := fdopen (name2, Open_WRONLY or Open_CREAT or Open_TRUNC,
octal(PERM));
  if outfile=-1 then
  begin
    fdclose(infile);
    copyfile:=-2; (*ошибка открытия второго файла*)
    exit;
  end;

  (* Чтение из файла name1 по BUFSIZE символов *)
  nread := fdread (infile, buffer, BUFSIZE);
  while nread > 0 do
  begin
    (* Записать buffer в выходной файл. *)
    if fdwrite (outfile, buffer, nread) < nread then
    begin
      fdclose (infile);
      fdclose (outfile);
      copyfile:=-3;          (* ошибка записи *)
      exit;
    end;
    nread := fdread (infile, buffer, BUFSIZE);
  end;

  (*закрываем прочитанные файлы*)
  fdclose (infile);
  fdclose (outfile);

  if (nread = -1) then
    copyfile := -4           (* ошибка при последнем чтении *)
  else
    copyfile := 0;              (* все порядке *)
end;

(* Программа для тестирования функции copyfile *)
var
  retcode:integer;
begin
  if paramcount<2 then
  begin
    writeln('Используйте: ',paramstr(0),' файл-источник файл-приемник');
```

```
    exit;
  end;
  retcode := copyfile(paramstr(1), paramstr(2));
  case retcode of
     0:      writeln('Файл   ',paramstr(1),'   успешно   скопирован   в   файл
',paramstr(2));
    -1:  writeln('Ошибка открытия файла ',paramstr(1),' для чтения');
    -2:  writeln('Ошибка открытия файла ',paramstr(2),' для записи');
    -3:  writeln('Ошибка записи в файл ',paramstr(2));
    -4:  writeln('Ошибка чтения из файла ',paramstr(1));
  end;
end.
```

Теперь функцию `copyfile` можно вызывать так:

```
retcode := copyfile('squarepeg', 'roundhole');
```

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

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

2.1.9. Эффективность вызовов fdread и fdwrite

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

```
(* Программа для тестирования функции copyfile *)
begin
  copyfile('test.in', 'test.out');
end.
```

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

Таблица 2.2. Результаты тестирования функции copyfile

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

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

Полученные результаты достаточно убедительны – чтение и запись по одному байту дает очень низкую производительность, тогда как увеличение размера буфера значительно повышает производительность. Наибольшая производительность достигается, если `BUFSIZE`

кратно размеру блока диска на диске, как видно из результатов, для значений BUFSIZE 512, 4096 и 8192 байта.

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

Функция fdFlush позволяет опустошить файловый буфер ядра UNIX для того, чтобы файл был действительно записан на диск.

Описание

```
uses linux;

Function fdFlush(fd:Longint):boolean;
```

2.1.10. Вызов fdseek и произвольный доступ

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

Описание

```
uses linux;

Function fdSeek(filedes, offset, SeekType:longint):longint;
```

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

SEEK_SET	Смещение offset вычисляется от начала файла, обычно имеет значение = 0
SEEK_CUR	Смещение offset вычисляется от текущего положения в файле, обычное значение = 1
SEEK_END	Смещение offset вычисляется от конца файла, обычное значение = 2

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

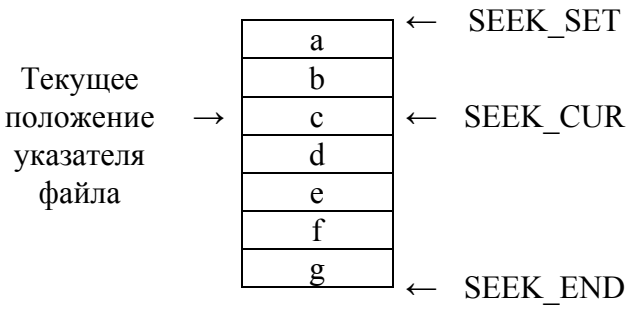


Рис. 2.1. Символьные значения флага SeekType

Пример использования вызова fdseek:

```
var
  newpos:longint;
.
```

```
.
.
newpos := fdseek(fd, -16, SEEK_END);
```
который задает положение указателя в 16 байтах от конца файла.

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

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

В качестве простого примера мы можем создать фрагмент программы, который будет дописывать данные в конец существующего файла, открывая файл, перемещаясь на его конец при помощи вызова `fdseek` и начиная запись:
```
filedes := fdopen(filename, Open_RDWR);
fdseek(filedes, 0, SEEK_END);
fdwrite(filedes, outbuf, OBSIZE);
```
Здесь параметр направления поиска для вызова `fdseek` установлен равным `SEEK_END` для перемещения в конец файла. Так как перемещаться дальше нам не нужно, то смещение задано равным нулю.

Вызов `fdseek` также может использоваться для получения размера файла, так как он возвращает новое положение в файле.
```
var
  filesize:longint;
  filedes:integer;
.
.
.
filesize := fdseek(filedes, 0, SEEK_END);
```
Упражнение 2.8. *Напишите функцию, которая использует вызов* `fdseek` *для получения размера открытого файла, не изменяя при этом значения указателя чтения-записи.*

2.1.11. Пример: гостиница

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

Следующая функция `getoccupier` вычисляет по заданному целому номеру комнаты положение первого байта фамилии жильца, затем перемещается в эту позицию и считывает данные. Она возвращает либо указатель на строку с фамилией жильца, либо нулевой указатель в случае ошибки (мы будем использовать для этого значение `nil`). Обратите

внимание, что мы присвоили переменной дескриптора файла `infile` исходное значение –1. Благодаря этому мы можем гарантировать, что файл будет открыт всего один раз.

```
(* Функция getoccupier - получить фамилию из файла residents *)

uses linux;

const
  NAMELENGTH=41;

var
  namebuf:array [0..NAMELENGTH-1] of char;     (* Буфер для фамилии *)
const
  infile:integer=-1;          (* Для хранения дескриптора файла *)

function getoccupier(roomno:integer):pchar;
var
  offset, nread:longint;
begin
  (* Убедиться, что файл открывается впервые *)
  if infile = -1 then
  begin
    infile := fdopen ('residents', Open_RDWR);
    if infile = -1 then
    begin
      getoccupier := nil;          (* Невозможно открыть файл *)
      exit;
    end;
  end;

  offset := (roomno - 1) * NAMELENGTH;

  (* Найти поле комнаты и считать фамилию жильца *)
  if fdseek (infile, offset, SEEK_SET) = -1 then
  begin
    getoccupier := nil;
    exit;
  end;

  nread := fdread (infile, namebuf, NAMELENGTH);
  if nread <= 0 then
  begin
    getoccupier := nil;
    exit;
  end;

  (* Создать строку, заменив символ перевода строки на '\0' *)
  namebuf[nread - 1] := #0;
  getoccupier := namebuf;
end;
```

Если предположить, что в гостинице 10 комнат, то следующая программа будет последовательно вызывать функцию `getoccupier` для просмотра файла и выводить каждую найденную фамилию при помощи процедуры `writeln` из стандартного модуля `system`:

```
(* Программа listoc выводит все фамилии жильцов *)

const
  NROOMS=10;
```

```
var
  j:integer;
  p:pchar;
begin

  for j := 1 to NROOMS do
  begin
    p := getoccupier (j);
    if p<>nil then
      writeln('Комната ', j:2, ', ', p)
    else
      writeln('Ошибка для комнаты ', j);
  end;
end.
```

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

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

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

```
i := StrToInt(str);
```

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

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

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

2.1.12. Дописывание данных в конец файла

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

```
(* Поиск конца файла *)
fdseek(filedes, 0, SEEK_END);
fdwrite(filedes, appbuf, BUFSIZE);
```

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

Можно использовать флаг `Open_APPEND` следующим образом:

```
filedes := fdopen('yetanother', Open_WRONLY or Open_APPEND);
```

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

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

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

r – *открыть файл только для чтения;*
w – *открыть файл только для записи;*
rw – *открыть файл для чтения и записи;*
a – *открыть файл для дописывания.*
процедура fileopen *должна возвращать дескриптор файла или код ошибки -1.*

2.1.13. Удаление файла

Существует один метод удаления файла из системы – при помощи вызова unlink.

Описание

```
uses linux;

Function UnLink(Var Path): Boolean;
```

Вызов имеет единственный аргумент – строку с именем удаляемого файла, например:

```
unlink('/tmp/usedfile');
```

Вызов возвращает true в случае успешного завершения и false – в случае ошибки.

2.1.14. Системный вызов fcntl

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

Описание

```
uses linux;

(* Примечание: тип последнего параметра может меняться *)

Function Fcntl(filedes:longint;Cmd:Integer):integer;
Function Fcntl(var filedes:Text;Cmd:Integer):integer;
Procedure Fcntl(Fd:text;Cmd:Integer;Arg:longint);
Procedure Fcntl(Fd:longint;Cmd:longint;Arg:Longint);
```

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

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

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

```
(*
 * Функция filestatus описывает текущий статус файла
 *)

uses linux;

function filestatus(filedes:integer):integer;
var
  arg1:integer;
begin
  arg1 := fcntl (filedes, F_GETFL);
  if arg1 = -1 then
```

```
  begin
    writeln('Ошибка чтения статуса файла');
    filestatus := -1;
    exit;
  end;

  write('Дескриптор файла ', filedes, ': ');

  (*
   * Сравнить аргумент с флагами открытия файла.
   *)
  case (arg1 and Open_ACCMODE) of
    Open_WRONLY:
      write('Только для записи');
    Open_RDWR:
      write('Для чтения-записи');
    Open_RDONLY:
      write('Только для чтения');
    else
      write('Режим не существует');
  end;

  if (arg1 and Open_APPEND)<>0 then
    write (' - установлен флаг append');
  writeln;
  filestatus := 0;
end;
```

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

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

```
if (fcntl(filedes, F_SETFL, Open_APPEND) = -1) then
  writeln('Ошибка вызова fcntl');
```

2.2. Стандартный ввод, стандартный вывод и стандартный вывод диагностики

2.2.1. Основные понятия

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

По умолчанию вызов `fdread` для стандартного ввода приведет к чтению данных с клавиатуры. Аналогично запись в стандартный вывод или стандартный вывод диагностики приведет по умолчанию к выводу сообщения на экран терминала. Это первый пример

использования примитивов доступа к файлам для ввода/вывода на устройства, отличные от обычных файлов.

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

```
$ prog_name < infile
```

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

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

```
$ prog_name > outfile
```

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

```
$ prog_1 | prog_2
```

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

2.2.2. Программа io

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

```
(* Программа io копирует стандартный ввод *)
(* в стандартный вывод *)
uses linux;

const
  SIZE=512;
var
  nread:longint;
  buf:array [0..SIZE-1] of byte;
begin
  nread := fdread (0, buf, SIZE);
  while nread > 0 do
  begin
    fdwrite (1, buf, nread);
    nread := fdread (0, buf, SIZE);
  end;
  halt(0);
end.
```

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

```
$ fpc io.pas
```

Если теперь запустить программу `io` на выполнение, просто набрав имя файла

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

```
$ io              Пользователь печатает io и нажимает Return
Это строка 1      Пользователь печатает строку и нажимает Return
Это строка 1      Программа io выводит строку на дисплей
.
.
.
```

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

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

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

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

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

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

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

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

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

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

Упражнение 2.16. *Иногда данные в файле могут медленно накапливаться в течение продолжительного промежутка времени. Напишите версию программы* `io` *с именем* `watch`*,*

которая будет выполнять чтение из стандартного ввода до тех пор, пока не встретится символ конца файла, выводя данные на стандартный вывод. После достижения конца файла программа `watch` *должна сделать паузу на пять секунд, а затем снова начать чтение стандартного ввода, чтобы проверить, не поступили ли новые данные, не открывая при этом файл заново и не изменяя положение указателя чтения-записи. Для прекращения работы процесса на заданное время вымажете использовать стандартную библиотечную процедуру* `delay`*, которая имеет единственный аргумент – целое число, задающее продолжительность ожидания в миллисекундах. Например, вызов*

```
sleep(5000);
```

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

2.2.3. Использование стандартного вывода диагностики

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

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

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

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

```
var
  msg:array [0..5] of char='boob'#$a;
.
.
fdwrite(2, msg, 5);
```

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

2.3. Стандартная библиотека ввода/вывода: взгляд в будущее

Системные вызовы доступа к файлам лежат в основе всего ввода и вывода программ UNIX. Тем не менее эти вызовы действительно примитивны и работают с данными только как с простыми последовательностями байтов, оставляя все остальное на усмотрение программиста. Соображения эффективности также ложатся на плечи разработчика.

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

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

```
uses stdio;
```

```
var
  stream:PFILE;

begin
  stream := fopen ('junk', 'r');
  if stream = nil then
  begin
    printf('Невозможно открыть файл junk'#10,[]);
    halt(1);
  end;
end.
```

Первая строка примера

```
uses stdio;
```

подключает модуль библиотеки ввода/вывода `stdio`, описанной в приложении. Этот файл, кроме всего прочего, содержит определение `TFILE`, `PFILE` и объявления для таких функций, как `fopen`.

Настоящее содержание этого примера заключается в операторе:

```
stream := fopen ('junk', 'r');
if stream = nil then
begin
.
.
end;
```

Здесь `junk` – это имя файла, а строка `'r'` означает, что файл открывается только для чтения. Строка `'w'` может использоваться для усечения файла до нулевой длины или создания файла и открытия его на запись. В случае успеха функция `fopen` проинициализирует структуру `TFILE` и вернет ее адрес в переменной `stream`. Указатель `stream` может быть передан другим процедурам из библиотеки. Важно понимать, что где-то внутри тела функции `fopen` осуществляется вызов нашего старого знакомого `fdopen`. И, естественно, где-то внутри структуры `TFILE` находится дескриптор файла, привязывающий структуру к файлу. Существенно то, что процедуры стандартного ввода/вывода написаны на основе примитивов системных вызовов. Основная функция библиотеки состоит в создании более удобного интерфейса и автоматической буферизации.

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

Описание

```
uses stdio;

(* Считать символ из __stream *)
function getc(__stream:pfile):integer;

(* Поместить символ в __stream *)
function putc(__c:integer;__stream:pfile):integer;
```

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

```
var
  c:integer;
  istream, ostream:PFILE;

(* Открыть файл istream для чтения и файл ostream для записи *)
.
.
.
c := getc (istream);
while c<>-1 do
```

```
begin
  putc (c, ostream);
  c := getc (istream);
end;
```

Значение -1 возвращается функцией getc при достижении конца файла, поэтому тип возвращаемого функцией getc значения определен как integer.

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

Такой подход весьма удобен, так как он освобождает программиста от беспокойства по поводу эффективности работы программы. Это также означает, что данные записываются большими блоками, и запись в файл будет производиться с задержкой (для терминалов сделаны специальные оговорки). Поэтому весьма неблагоразумно использовать для одного и того же файла и стандартные процедуры ввода/вывода, и системные вызовы, такие как fdread, fdwrite или fdseek. Это может привести к хаосу, если не представлять четко, что при этом происходит. С другой стороны, вполне допустимо смешивать системные вызовы и процедуры стандартного ввода/вывода для разных файлов.

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

```
printf('Целое число %d'#10, [ival]);
```

Кстати, функция printf неявно осуществляет запись в стандартный вывод.

Вывод сообщений об ошибках при помощи функции writeln

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

```
uses stdio;     (* Для определения stderr *)
.
.
writeln (stderr, 'Ошибка номер ', linuxerror);
```

Отличие между использованием writeln и вызовом printf заключается в параметре stderr, являющемся указателем на текстовый файл, автоматически связанный с потоком вывода стандартной диагностики.

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

```
(* Функция notfound - вывести сообщение об ошибке и выйти *)

uses linux;

function notfound(progname, filename: string):integer;
begin
  writeln(stderr, progname, ': файл ', filename, ' не найден');
```

```
  halt(1);
end;
```

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

2.4. Системные вызовы и переменная linuxerror

Из вышеизложенного материала видно, что все описанные до сих пор системные вызовы файлового ввода/вывода могут завершиться неудачей. В этом случае возвращаемое значение всегда равно -1. Чтобы помочь программисту получить информацию о причине ошибки, система UNIX предоставляет глобальную целочисленную переменную, содержащую код ошибки. Значение кода ошибки связано с сообщением об ошибке, таким как *no permission* (нет доступа) или *invalid argument* (недопустимый аргумент). Полный список кодов и описаний ошибок приведен в Приложении 1. Текущее значение кода ошибки соответствует типу последней ошибки, произошедшей во время системного вызова.

Переменная, содержащая код ошибки, имеет имя `linuxerror` (сокращение от *linux error number* – номер ошибки в Linux). Программист может использовать переменную `linuxerror` в программе на языке Паскаль, подключив модуль `linux`.

Следующая программа использует вызов `fdopen`, и в случае его неудачного завершения использует функцию `writeln` для вывода значения переменной `linuxerror`:

```
(* Программа err1.pas - открывает файл с обработкой ошибок *)
uses linux;

var
  fd:integer;
begin

  fd := fdopen ('nonesuch', Open_RDONLY);
  if fd=-1 then
    writeln(stderr, 'Ошибка ', linuxerror);
end.
```

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

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

2.4.7. Подпрограмма perror

Кроме `linuxerror` UNIX обеспечивает библиотечную процедуру (не системный вызов) `perror`. Для большинства традиционных команд UNIX использование `perror` является стандартным способом вывода сообщений об ошибках. Она имеет единственный аргумент строчного типа и при вызове отправляет на стандартный вывод диагностики сообщение, состоящее из переданного ей строчного аргумента, двоеточия и дополнительного описания ошибки, соответствующей текущему значению переменной `linuxerror`. Важно, что сообщение об ошибке отправляется на стандартный вывод диагностики, а не в стандартный вывод.

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

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

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

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

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

Глава 3. Работа с файлами

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

3.1. Файлы в многопользовательской среде

3.1.1. Пользователи и права доступа

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

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

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

указывает, что пользователь `keith` имеет uid 35.

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

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

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

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

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

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

Действующие идентификаторы пользователей и групп

Необходимо сделать одно уточнение: создание файла определяется связанным с процессом *действующим идентификатором пользователя euid* (effective user-id). Хотя процесс может быть запущен одним пользователем (скажем, keith), при определенных

обстоятельствах он может получить права доступа другого пользователя (например, dina). Вскоре будет показано, как это можно осуществить. Идентификатор пользователя, запустившего процесс, называется *истинным идентификатором пользователя* (real user-id, сокращенно ruid) этого процесса. Разумеется, в большинстве случаев действующий и истинный идентификаторы пользователя совпадают.

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

3.1.2. Права доступа и режимы файлов

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

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

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

– чтение из файла;
– запись в файл;
– запуск файла на выполнение. В этом случае файл обычно является программой или последовательностью команд оболочки.

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

Система хранит связанные с файлом права доступа в битовой маске, называемой *кодом доступа к файлу* (file mode). Хотя модуль `linux` и определяет символьные имена для битов прав доступа, большинство программистов все еще предпочитает использовать восьмеричные постоянные, приведенные в табл. 3.1 – при этом символьные имена являются относительно недавним и весьма неудобным нововведением. Следует обратить внимание, что в языке С восьмеричные постоянные всегда начинаются с нуля, иначе компилятор будет расценивать их как десятичные. В Паскале же для записи восьмеричных чисел удобно использовать функцию `octal`, параметром которой является восьмеричное число, записанное десятичными числами, а результатом – значение в десятеричной системе:

Описание

```
uses linux;

Function Octal(l:longint):longint;
```

Таблица 3.1. Восьмеричные значения для прав доступа к файлам

Восьмеричное значение	Символьное обозначение	Значение
0400	STAT_IRUSR	Владелец имеет доступ для чтения
0200	STAT_IWUSR	Владелец имеет доступ для записи
0100	STAT_IXUSR	Владелец может выполнять файл
0040	STAT_IRGRP	Группа имеет доступ для чтения
0020	STAT_IWGRP	Группа имеет доступ для записи
0010	STAT_IXGRP	Группа может выполнять файл
0004	STAT_IROTH	Другие пользователи имеют доступ для чтения
0002	STAT_IWOTH	Другие пользователи имеют доступ для записи
0001	STAT_IXOTH	Другие пользователи могут выполнять файл

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

```
STAT_IRUSR or STAT_IRGRP or STAT_IROTH
```

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

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

Поэтому чаще встречается значение кода доступа:

```
0700 + 050 + 05 = 0755
```

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

Легко понять, почему программисты UNIX предпочитают использовать восьмеричные постоянные, а не имена констант из модуля `linux`, когда просто значение `0755` представляется выражением:

```
STAT_IRUSR or STAT_IWUSR or STAT_IXUSR or STAT_IRGRP or STAT_IXGRP or
STAT_IROTH or STAT_IXOTH
```

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

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

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

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

3.1.3. Дополнительные права доступа для исполняемых файлов

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

`04000 STAT_ISUID` Задать user-id при выполнении

`02000 STAT_ISGID` Задать group-id при выполнении

`01000 STAT_ISVTX` Сохранить сегмент кода (бит фиксации)

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

Подобный механизм может использоваться для управления доступом к критическим данным. Конфиденциальная информация может быть защищена от публичного доступа или

изменения при помощи стандартных прав доступа на чтение/запись/выполнение. Владелец файла создает программу, которая будет предоставлять ограниченный доступ к файлу. Затем для файла программы устанавливается флаг доступа `STAT_ISUID`, что позволяет другим пользователям получать ограниченный доступ к файлу только при помощи данной программы. Очевидно, программа должна быть написана аккуратно во избежание случайного и умышленного нарушения защиты.[1]

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

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

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

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

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

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

3.1.4. Маска создания файла и системный вызов umask

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

```
filedes := fdopen('datafile', Open_CREAT, ocatl(0644));
```

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

Основная идея просматривается четко: если в маске создания файлов задан какой-либо бит доступа, то при создании файлов он всегда остается выключенным. Биты в маске могут быть установлены при помощи тех же восьмеричных постоянных, которые были описаны ранее для кода доступа к файлам, хотя при этом могут использоваться только

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

основные права доступа на чтение, запись и выполнение. Экзотические права доступа, такие как STAT_ISUID, не имеют смысла для маски создания файла.

Таким образом, с точки зрения программиста, оператор

```
filedes := fdopen(pathname, Open_CREAT, mode);
```

эквивалентен оператору

```
filedes := fdopen(pathname, Open_CREAT, (not mask) and mode);
```

где переменная mask содержит текущее значение маски создания файла, not – это оператор побитового отрицания, а and – оператор побитового И.

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

```
filedes := fdopen(pathname, Open_CREAT, octal(0644));
```

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

Маску создания файла можно изменить при помощи системного вызова umask.

Описание

```
uses linux;

Function Umask(Mask:Integer):Integer;
```

Например:

```
var
  oldmask:integer;
.
.
.
oldmask := umask(octal(022));
```

Значение octal(022) запрещает присваивание файлу прав доступа на запись всем, кроме владельца файла. После вызова в переменную oldmask будет помещено предыдущее значение маски.

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

```
uses linux,stdio;

function specialcreat(pathname:string;mode:longint):integer;
var
  oldu,filedes:integer;
begin
  (* Установить маску создания файла равной нулю *)
  oldu:=umask(0);
  if oldu = -1 then
  begin
    perror('Ошибка сохранения старой маски');
    specialcreat:=-1;
    exit;
  end;
  (* Создать файл *)
  filedes:=fdopen(pathname, Open_WRONLY or Open_CREAT or Open_EXCL, mode);
  if (filedes = -1) then
```

```
    perror ('Ошибка открытия файла');
  (* Восстановить прежний режим доступа к файлу *)
  if (umask (oldu) = -1) then
    perror ('Ошибка восстановления старой маски');
  (* Вернуть дескриптор файла *)
  specialcreat:=filedes;
end;
```

3.1.5. Вызов fdopen и права доступа к файлу

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

Если для создания файла используется расширенная форма вызова `fdopen`, то использование флагов `Open_CREAT`, `Open_TRUNC` и `Open_EXCL` позволяет по-разному работать с существующими файлами. Примеры использования вызова `fdopen` с заданием прав доступа к файлу:

```
filedes := fdopen(pathname, Open_WRONLY or Open_CREAT or Open_TRUNC,
octal(0600));
```

и:

```
filedes := fdopen(pathname, Open_WRONLY or Open_CREAT or Open_EXCL,
octal(0600));
```

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

Упражнение 3.5.

А. Предположим, что действующий идентификатор пользовать euid процесса равен 100, а его действующий идентификатор группы egid равен 200. Владельцем файла `testfile` *являет пользователь с идентификатором 101, а идентификатор группы файла gid равен 200. Для каждого возможного режима доступа (только для чтения, только для записи, для записи-чтения) определите, будет ли успешным вызов* `open`*, если файл* `testfile` *имеет следующие права доступа:*

```
rwxr-xrwx   r-xrwxr-x   rwx--x---   rwsrw-r--
--s--s--x   ---rwx---   ---r-x--x
```

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

3.1.6. Определение доступности файла при помощи вызова access

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

Описание

```
uses linux;

Function Access(PathName:Pathstr; AMode:integer):Boolean;
```

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

R_OK – имеет ли вызывающий процесс доступ на чтение;

W_OK – имеет ли вызывающий процесс доступ на запись;

X_OK – может ли вызывающий процесс выполнить файл.

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

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

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

```
(* Пример использования вызова access *)
uses linux,stdio;

const
  filename = 'afile';

begin
  if not access (filename, R_OK) then
  begin
    writeln(stderr, 'Пользователь не имеет доступа на чтение к файлу ',
filename);
    halt(1);
  end;
  writeln(filename, ' доступен для чтения, продолжаем');
  (* Остальная программа *)
end.
```

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

3.1.7. Изменение прав доступа при помощи вызова chmod

Описание

```
uses linux;

Function Chmod(PathName:Pathstr; NewMode:Longint):Boolean;
```

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

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

Пример использования вызова chmod:

```
if not chmod(pathname, octal(0644)) then
  perror('Ошибка вызова chmod');
```

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

`lsoct`, *которую вы разработали в упражнении 3.3.*

3.1.8. Изменение владельца при помощи вызова chown

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

Описание

```
uses linux;

Function Chown(PathName:Pathstr; Owner_id,Group_id:Longint):Boolean;
```

Например:

```
Uses linux;

Var UID,GID : Longint;
    F : Text;

begin

  Writeln ('This will only work if you are root.');
  Write ('Enter a UID : ');readln(UID);
  Write ('Enter a GID : ');readln(GID);
  Assign (f,'test.txt');
  Rewrite (f);
  Writeln (f,'The owner of this file should become : ');
  Writeln (f,'UID : ',UID);
  Writeln (f,'GID : ',GID);
  Close (F);
  if not Chown ('test.txt',UID,GID) then
    if LinuxError=Sys_EPERM then
      Writeln ('You are not root !')
    else
      Writeln ('Chmod failed with exit code : ',LinuxError)
  else
    Writeln ('Changed owner successfully !');
end.
```

Вызов имеет три аргумента: `pathname`, указывающий имя файла, `owner_id`, задающий нового владельца, и `group_id`, задающий новую группу. Возвращаемое значение `retval` равно `true` в случае успеха и `false` – в случае ошибки.

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

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

3.2. Файлы с несколькими именами

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

файл, – весьма полезно.

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

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

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

3.2.1. Системный вызов link

Описание

```
uses linux;

Function Link(original_path, New_Path:pathstr):Boolean;
```

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

Системный вызов `link` возвращает значение `true` в случае успешного завершения и `false` – в случае ошибки. В последнем случае новая ссылка на файл не будет создана.

Например, оператор

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

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

3.2.2. Системный вызов unlink

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

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

удалит файл `/tmp/scratch`.

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

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

```
(* Программа move - переименование файла *)
uses linux,stdio;

const
  usage = 'Применение: move файл1 файл2';
```

```
(*
 * Программа использует аргументы командной строки,
 * передаваемые обычным способом.
 *)
begin
  if (paramcount <> 2) then
  begin
    writeln(stderr, usage);
    halt(1);
  end;
  if not link(paramstr(1), paramstr(2)) then
  begin
    perror('Ошибка в вызове link');
    halt(1);
  end;
  if not unlink (argv[1]) then
  begin
    perror('Ошибка в вызове unlink');
    unlink(paramstr(2));
    halt(1);
  end;
  writeln('Успешное завершение');
  halt(0);
end.
```

До сих пор не было упомянуто взаимодействие вызова `unlink` и прав доступа к файлу, связанных с аргументом, задающим имя файла. Это объясняется тем, что права просто не влияют на вызов `unlink`. Вместо этого успешное или неуспешное завершение вызова `unlink` определяется правами доступа к содержащему файл *каталогу*. Эту тема будет рассмотрена в главе 4.

3.2.3. Системный вызов frename

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

Описание

```
uses linux;

Function Frename(oldpathname, newpathname:Pchar):Boolean;
Function Frename(oldpathname, newpathname:String):Boolean;
```

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

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

3.2.4. Символьные ссылки

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

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

Для создания символьной ссылки используется системный вызов `symlink`:

Описание

```
uses linux;

Function SymLink(realname, symname:pathstr):Boolean;
```

После завершения вызова `symlink` создается файл `symname`, указывающий на файл `realname`. Если возникает ошибка, например, если файл с именем `symname` уже существует, то вызов `symlink` возвращает значение `false`. В случае успеха вызов возвращает истинное значение.

Пример использования SymLink:

```
Uses linux;

Var F : Text;
    S : String;

begin
  Assign (F,'test.txt');
  Rewrite (F);
  Writeln (F,'This is written to test.txt');
  Close(f);
  { new.txt and test.txt are now the same file }
  if not SymLink ('test.txt','new.txt') then
    writeln ('Error when symlinking !');
  { Removing test.txt still leaves new.txt
    Pointing now to a non-existent file ! }
  If not Unlink ('test.txt') then
    Writeln ('Error when unlinking !');
  Assign (f,'new.txt');
  { This should fail, since the symbolic link
    points to a non-existent file! }
  {$i-}
  Reset (F);
  {$i+}
  If IOResult=0 then
    Writeln ('This shouldn''t happen');
 { Now remove new.txt also }
 If not Unlink ('new.txt') then
   Writeln ('Error when unlinking !');
end.
```

Если файл символьной ссылки открывается при помощи `fdopen`, то системный вызов `fdopen` корректно прослеживает путь к файлу `realname`. Если необходимо считать данные из самого файла `symname`, то нужно использовать системный вызов `readlink`.

Описание

```
uses linux;

Function ReadLink(sympath, buffer:pchar; bufsize:longint):longint;
Function ReadLink(name:pathstr):pathstr;
```

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

буфере или -1 – в случае ошибки.

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

3.2.5. Еще об именах файлов

Для выделения из имени файла его частей можно воспользоваться функциями:

Описание

```
uses linux;

Function BaseName(Const Path;Const Suf:Pathstr):Pathstr;
Function DirName(Const Path:Pathstr):Pathstr;
Procedure FSplit(const Path:PathStr; var Dir:DirStr; Var Name:NameStr;
                 Var Ext:ExtStr);
```

`BaseName` выделяет из полного пути Path имя файла, обрезая окончание `Suf`, если оно существует. Для каталогов завершающий слэш всегда убирается (за исключением корневого каталога).

`DirName` возвращает часть пути, соответствующую имени каталога. Это будет часть параметра `Path` до завершающего слэша, или ничего в его отсутствие.

`FSplit` разбивает полное имя файла на 3 части: путь `Path`, имя `Name` и расширение `ext`. Расширением считаются все символы, следующие за последней точкой.

Например:

```
uses Linux;

var
  Path,Name,Ext : string;

begin
  FSplit(ParamStr(1),Path,Name,Ext);
  WriteLn('Split ',ParamStr(1),' in:');
  WriteLn('Path     : ',Path);
  WriteLn('Name     : ',Name);
  WriteLn('Extension: ',Ext);
end.
```

3.3. Получение информации о файле: вызов fstat

До сих пор были лишь рассмотрены вопросы, как можно установить или изменить основные связанные с файлами свойства. Системный вызов `fstat` позволяет процессу определить значения этих свойств в существующем файле.

Описание

```
uses linux;

Function FStat(Path:Pathstr;Var Info:tstat):Boolean;
Function FStat(Fd:longint;Var Info:tstat):Boolean;
Function FStat(var F:Text;Var Info:tstat):Boolean;
Function FStat(var F:File;Var Info:tstat):Boolean;
Function LStat(Path:Pathstr; Var Info:tstat):Boolean;
```

Системный вызов `fstat` имеет два аргумента: первый из них – `path`, как обычно, указывает на полное имя файла. Второй аргумент `info` является ссылкой на структуру `tstat (stat)`. Эта структура после успешного вызова будет содержать связанную с файлом информацию. Вместо имени файла может также использоваться его дескриптор или

файловая переменная.

```
.
.
.
var
  s:tstat;
  filedes:integer;
  retval:boolean;

filedes := fdopen('/tmp/dina', Open_RDWR);

(* Структура s может быть заполнена при помощи вызова ... *)
retval := fstat('/tmp/dina', s);

(* ... или *)
retval := fstat(filedes, @s);
```

Определение структуры `tstat` находится в модуле `linux` и включает следующие элементы:

```
stat=record
  dev      :  word;
  pad1     :  word;
  ino      :  longint;
  mode     :  word;
  nlink    :  word;
  uid      :  word;
  gid      :  word;
  rdev     :  word;
  pad2     :  word;
  size     :  longint;
  blksze   :  Longint;
  blocks   :  Longint;
  atime    :  Longint;
  unused1  :  longint;
  mtime    :  Longint;
  unused2  :  longint;
  ctime    :  Longint;
  unused3  :  longint;
  unused4  :  longint;
  unused5  :  longint;
end;
```

Системный вызов `lstat` получает информацию о символьной ссылке. Например:

```
uses linux;

var f : text;
    i : byte;
    info : stat;

begin
  { Make a file }
  assign (f,'test.fil');
  rewrite (f);
  for i:=1 to 10 do writeln (f,'Testline # ',i);
  close (f);
  { Do the call on made file. }
  if not fstat ('test.fil',info) then
     begin
     writeln('Fstat failed. Errno : ',linuxerror);
```

```
    halt (1);
    end;
  writeln;
  writeln ('Result of fstat on file ''test.fil''.');
  writeln ('Inode   : ',info.ino);
  writeln ('Mode    : ',info.mode);
  writeln ('nlink   : ',info.nlink);
  writeln ('uid     : ',info.uid);
  writeln ('gid     : ',info.gid);
  writeln ('rdev    : ',info.rdev);
  writeln ('Size    : ',info.size);
  writeln ('Blksize : ',info.blksze);
  writeln ('Blocks  : ',info.blocks);
  writeln ('atime   : ',info.atime);
  writeln ('mtime   : ',info.mtime);
  writeln ('ctime   : ',info.ctime);

  If not SymLink ('test.fil','test.lnk') then
    writeln ('Link failed ! Errno :',linuxerror);

  if not lstat ('test.lnk',info) then
    begin
    writeln('LStat failed. Errno : ',linuxerror);
    halt (1);
    end;
  writeln;
  writeln ('Result of fstat on file ''test.lnk''.');
  writeln ('Inode   : ',info.ino);
  writeln ('Mode    : ',info.mode);
  writeln ('nlink   : ',info.nlink);
  writeln ('uid     : ',info.uid);
  writeln ('gid     : ',info.gid);
  writeln ('rdev    : ',info.rdev);
  writeln ('Size    : ',info.size);
  writeln ('Blksize : ',info.blksze);
  writeln ('Blocks  : ',info.blocks);
  writeln ('atime   : ',info.atime);
  writeln ('mtime   : ',info.mtime);
  writeln ('ctime   : ',info.ctime);
  { Remove file and link }
  erase (f);
  unlink ('test.lnk');
end.
```

Элементы структуры `stat` имеют следующие значения:

- `dev, ino`

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

- `mode`

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

- `nlink`

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

- `uid, gid`

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

- `rdev`

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

- `size`

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

- `atime`

  Содержит время последнего чтения из файла (хотя первоначальные вызов `fdcreat` и `fdopen` устанавливают это значение).

- `mtime`

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

- `ctime`

  Содержит время последнего изменения информации, возвращаемой в структуре `stat`. Это время изменяется системными вызовами `link` (меняется элемент `nlink`), `chmod` (меняется `mode`) и `fdwrite` (меняется `mtime` и, возможно, `size`).

- `blksize`

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

- `blocks`

  Содержит число физических блоков, занимаемых определенным файлом.

Системный вызов `utime` позволяет установить время доступа и модификации файла. Структура `utimbuf` содержит два поля, `actime` и `modtime`, оба типа `Longint`. Они должны быть заполнены значениями времени в секундах с 1.1.1970 г. относительно последнего времени доступа и последнего времени модификации.

Описание

```
uses linux;

Function Utime(path:pathstr; utim:utimbuf):Boolean;
```

Например:

```
Uses linux;

Var utim : utimbuf;
    year,month,day,hour,minute,second : Word;

begin
  { Set access and modification time of executable source }
  GetTime (hour,minute,second);
  GetDate (year,month,day);
  utim.actime:=LocalToEpoch(year,month,day,hour,minute,second);
  utim.modtime:=utim.actime;
  if not Utime('ex25.pp',utim) then
    writeln ('Call to UTime failed !')
  else
```

```
    begin
    Write ('Set access and modification times to : ');
    Write (Hour:2,':',minute:2,':',second,', ');
    Writeln (Day:2,'/',month:2,'/',year:4);
    end;
end.
```

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

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

```
(* Процедура filedata выводит данные о файле *)
uses linux;

(*
 * Массив octarray используется для определения
 * установки битов прав доступа.
 *)
const
  octarray:array[0..8] of integer= (
              0400, 0200, 0100,
              0040, 0020, 0010,
              0004, 0002, 0001);

(*
 * Мнемонические коды для прав доступа к файлу,
 * длиной 10 символов, включая нулевой символ в конце строки.
 *)
const
  perms:pchar = 'rwxrwxrwx';

function filedata(pathname:string):integer;
var
  statbuf:tstat;
  descrip:array [0..9] of char;
  j:integer;
begin
  if not fstat (pathname, statbuf) then
  begin
    writeln('Ошибка вызова stat для ', pathname);
    filedata:=-1;
    exit;
  end;

  (* Преобразовать права доступа в удобочитаемую форму *)
  for j:=0 to 8 do
  begin
      (*
       * Проверить, установлены ли права доступа
       * при помощи побитового И
       *)
      if (statbuf.mode and octal(octarray[j]))<>0 then
        descrip[j] := perms[j]
      else
        descrip[j] := '-';
```

```
  end;
  descrip[9] := #0;          (* задать строку *)
  (* Вывести информацию о файле *)
  writeln(#10'Файл ', pathname, ':');
  writeln('Размер ',statbuf.size,' байт');
  writeln('User-id ',statbuf.uid,', Group-id ',statbuf.gid,#10);
  writeln('Права доступа: ', descrip);
  filedata:=0;
end;
```

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

```
(* Программа lookout сообщает об изменении файла *)

uses linux, stdio;

const
  MFILE=10;

var
  sb:tstat;
  j:integer;
  last_time:array [1..MFILE] of longint;

procedure sleep(t:longint);cdecl;external 'c';

procedure cmp(name:string;last:longint);
begin
  (*
   * Проверять время изменения файла,
   * если можно считать данные о файле.
   *)
  if not fstat(name,sb) or (sb.mtime <> last) then
  begin
    writeln('lookout: файл ',name,' изменился');
    halt(0);
   end;
end;


begin
  if (paramcount < 1) then
  begin
    writeln('Применение: lookout имя_файла ...');
    halt(1);
  end;
  if (paramcount > MFILE) then
  begin
    writeln('lookout: слишком много имен файлов');
    halt (1);
  end;
  (* Инициализация *)
  for j:=1 to paramcount do
  begin
```

[1] Фоновый процесс при попытке вывода на консоль остановится, получив сигнал SIGTTOU (см. главу 6). Следует придумать иной способ оповещения об изменениях файлов.

```
    if not fstat(paramstr(j), sb) then
    begin
      writeln ('lookout: ошибка вызова stat для ', paramstr(j));
      halt(1);
    end;
    last_time[j]:=sb.mtime;
  end;
  (* Повторять до тех пор, пока файл не изменится *)
  while true do
  begin
    for j:=1 to paramcount do
      cmp(paramstr(j), last_time[j]);
      (*
       * Остановиться на 60 секунд.
       * Функция 'sleep' стандартная
       * библиотечная процедура UNIX.
       *)
    sleep (60);
  end;
end.
```

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

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

3.3.1. Подробнее о вызове chmod

Системный вызов `fstat` расширяет использование вызова `chmod`, поскольку позволяет предварительно узнать значение кода доступа к файлу, что дает возможность изменять отдельные биты, а не менять весь код доступа целиком.

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

```
(* Программа addx разрешает доступ на выполнение файла *)
uses linux,stdio;

const XPERM=0100;       (* Право на выполнение для владельца *)

var
  k:integer;
  statbuf:tstat;

begin
  (* Выполнить для всех файлов в списке аргументов *)
  for k := 1 to paramcount do
  begin
    (* Получить текущий код доступа к файлу *)
    if not fstat(paramstr(k), statbuf) then
    begin
      writeln('addx: ошибка вызова stat для ',paramstr(k));
      continue;
```

```
    end;
      (*
         Попытаться разрешить доступ на выполнение
         при помощи оператора побитового ИЛИ
       *)
    statbuf.mode := statbuf.mode or octal(XPERM);
    if not chmod (paramstr(k), statbuf.mode) then
      writeln('addx: ошибка изменения прав доступа для файла ', paramstr(k));
    end; (* Конец цикла *)
  halt(0);
end.
```

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

```
statbuf.mode := statbuf.mode or octal(XPERM);
```

Для ясности использована более короткая форма. Можно было бы также использовать вместо `XPERM` предусмотренную в системе постоянную `STAT_IXUSR`.

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

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

Глава 4. Каталоги, файловые системы и специальные файлы

4.1. Введение

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

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

4.2. Каталоги с точки зрения пользователя

Даже случайный пользователь системы UNIX будет иметь некоторое представление о том, как выглядит структура каталогов системы. Тем не менее для полноты изложения кратко опишем обычное расположение файлов с точки зрения пользователя.

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

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

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

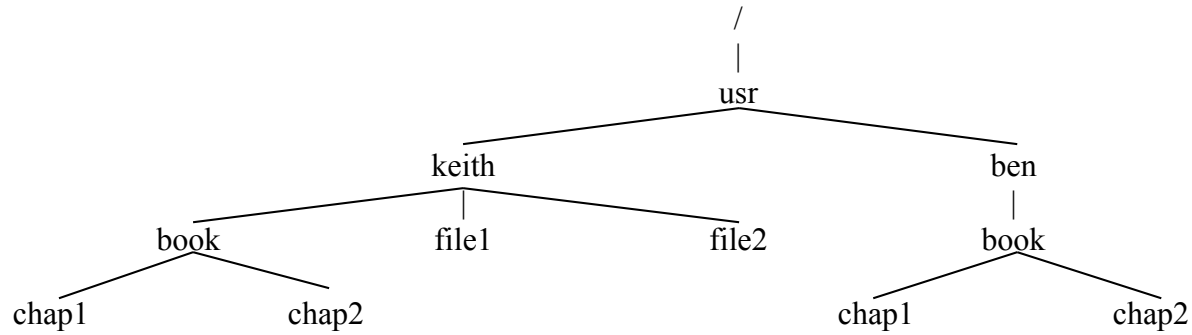


Рис. 4.1. Пример дерева каталогов

На вершине этого дерева, так же как и на вершине любого дерева каталогов UNIX, находится единственный каталог, который называется *корневым каталогом* (root directory) и имеет очень короткое имя `/`. Все узлы дерева, кроме конечных, например узлы `keith` или `ben`, всегда являются каталогами. Конечные узлы, например узлы `file1` или `file2`, являются файлами или пустыми каталогами. В настоящее время в большинстве систем UNIX имена каталогов могут содержать до 255 символов, но, так же как и в случае с файлами, для обеспечения совместимости со старыми версиями системы их длина не должна превышать 14 символов.

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

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

Текущий рабочий каталог

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

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

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

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

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

```
$ cat book/chap1
```

эквивалентна команде

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

а команда

```
$ cat file1
```
эквивалентна команде
```
$ cat usr/keith/file1
```

4.3. Реализация каталогов

На самом деле каталоги UNIX – не более чем файлы. Во многих аспектах система обращается с ними точно так же, как и с обычными файлами. Они имеют владельца, группу, размер и связанные с ними права доступа. Многие из системных вызовов для работы с файлами, которые были рассмотрены в предыдущих главах, могут использоваться и для работы с каталогами, хотя так делать и не рекомендуется. Например, каталоги можно открывать на чтение при помощи системного вызова `open`, и возвращенный этим вызовом дескриптор файла может использоваться для последующих вызовов `fdread`, `fdseek`, `fstat` и `fdclose`.

Тем не менее между каталогами и обычными файлами существуют некоторые важные различия, налагаемые системой. Каталоги не могут быть созданы при помощи системных вызовов `fdcreat` или `fdopen`. Системный вызов `fdopen` также не будет работать с каталогом, если установлен любой из флагов `Open_WRONLY` или `Open_RDWR` (только для записи или чтение/запись). При этом вызов вернет ошибку и запишет в переменную `linuxerror` код ошибки `Sys_EISDIR`. Эти ограничения делают невозможным изменение каталога при помощи системного вызова `fdwrite`. Фактически из-за особой природы каталогов для работы с ними гораздо лучше использован выделенное семейство системных вызовов, которое будет далее изучено.

Структура каталогов состоит из набора элементов каталогов, по одному элементу для каждого содержащегося в них файла или подкаталога. Каждый элемент каталога состоит, по крайней мере, из одного положительного числа, *номера индексного дескриптора* (inode number), и символьного поля, содержащего имя файла. Когда имена файлов были длиной не более 14 символов, элементы каталога имели фиксированную длину и большинство систем UNIX использовали один и тот же метод их реализации (исключение составлял Berkeley UNIX). Тем не менее после введения длинных имен файлов элементы каталога стали иметь различную длину, и реализация каталогов стала зависеть от файловой системы. Поэтому при разработке программ не следует полагаться на формат каталога, и для того, чтобы сделать их действительно переносимыми, необходимо использовать для работы с каталогами системные вызовы из спецификации XSI.

Часть каталога, содержащая три файла, может выглядеть примерно так, как показано на рис. 4.2. (Информация, необходимая для управления свободным пространством в файле каталога, исключена.) Этот каталог содержит имена трех файлов `fred`, `bookmark` и `abc`, которые могут быть и подкаталогами. Номера индексных дескрипторов для этих файлов равны соответственно 120, 207 и 235. На рис. 4.2 представлена логическая структура каталога; в действительности же каталог представляет собой непрерывный поток байтов.

120	f	r	e	d	\0				
207	b	o	o	k	m	a	r	k	\0
235	a	b	c	\0					

Рис. 4.2. Часть каталога

Номер индексного дескриптора однозначно идентифицирует файл (на самом деле номера индексных дескрипторов уникальны только в пределах одной файловой системы, но подробнее об этом будет рассказано ниже). Номер индексного дескриптора используется операционной системой для поиска в дисковой структуре данных *структуры индексного дескриптора* (inode structure), содержащей всю информацию, необходимую для обращения файлом: его размер, идентификатор владельца и группы, права доступа, время последнего доступа, последнего изменения и адреса блоков на диске, в которых хранятся данные файла. Большая часть информации, получаемой системным вызовом `fstat`, описанными в

предыдущей главе, фактически получается напрямую из структуры индексного дескриптора. Более подробно структура индексного дескриптора будет рассмотрена в разделе 4.5.

Важно понимать, что представление каталога является только логической картиной. Просмотр содержимого каталога при помощи команды `cat` может завершиться выводом «мусора» на экран терминала. Более удобно исследовать каталог при помощи команды восьмеричного вывода `od` с параметром `-c`. Например, для вывода содержимого текущей директории следует набрать в командной строке:

```
$ od -c .
```

Символ «точка» (`.`) в этой команде является стандартным способом задания текущего рабочего каталога.

4.3.1. Снова о системных вызовах link и unlink

В предыдущей главе было описано, как можно использовать системный вызов `link` для создания различных имен для одного и того же физического файла, поэтому уже должно быть понятно, как работает этот вызов. Каждая ссылка просто представляет собой еще одну позицию в каталоге с тем же самым номером индексного дескриптора, что и исходный, но с новым именем.

Если в каталоге с рис. 4.2 создать ссылку на файл `abc` с именем `xyz` при помощи следующего вызова

```
link('abc', 'xyz');
```

то рассматриваемый участок каталога будет выглядеть примерно так, как показано на рис. 4.3. При удалении ссылки при помощи системного вызова `unlink` соответствующие байты, содержащие имя файла, освобождаются для дальнейшего использования. Если имя файла представляет последнюю ссылку на этот файл, то вся связанная с ним структура индексных дескрипторов стирается. Связанные с файлом блоки на диске, которые содержали данные файла, добавляются к поддерживаемому системой списку свободных блоков и становятся пригодными для дальнейшего использования. В большинстве систем UNIX восстановление удаленных файлов невозможно.

120	f	r	e	d	\0				
207	b	o	o	k	m	a	r	k	\0
235	a	b	c	\0					
235	x	y	z	\0					

Рис. 4.3. Пример каталога с новым файлом

4.3.2. Точка и двойная точка

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

```
$ cat ./fred
```

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

```
$ ls .
```

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

```
$ cd ..
```

позволяет пользователю переместиться на один уровень вверх по дереву каталогов.

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

Можно более ясно это представить себе, рассмотрев участок дерева каталогов,

приведенный на рис. 4.4.

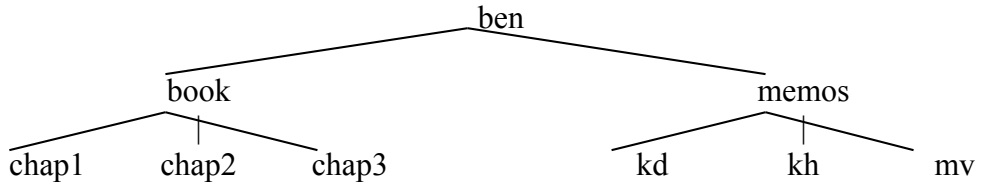


Рис. 4.4. Часть дерева каталогов

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

4.3.3. Права доступа к каталогам

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

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

Тем не менее, хотя эти права доступа представлены так же, как и у обычных файлов, интерпретируются они по-другому:

- право доступа к каталогу на чтение показывает, что соответствующий класс пользователей может выводить список содержащихся в каталоге файлов и подкаталогов. Однако это не означает, что пользователи могут читать содержащуюся в файлах информацию – это определяется правами доступа к отдельным файлам;
- право доступа к каталогу на запись позволяет пользователю создавать в каталоге новые файлы и удалять существующие. И снова это не дает пользователю права изменять содержимое существующих файлов, если это не разрешено правами доступа к отдельным файлам. Вместе с тем при этом можно удалить существующий файл и создать новый с тем же самым именем, и это, по сути, означает то же самое, что и изменение содержимого исходного файла;
- право доступа к каталогу на выполнение, называемое также *правом выполнения*, или *правом прохождения*, *поиска* (search permission), позволяет пользователю перейти в каталог при помощи команды `cd` или системного вызова `chdir` в программе (который будет рассмотрен позже). Кроме этого, чтобы иметь возможность открыть файл или выполнить программу, пользователь должен иметь право доступа на выполнение для всех ведущих к файлу каталогов, входящих в абсолютный путь файла;
- бит фиксации, `STAT_ISVTX`, позволяет установить дополнительную защиту файлов, находящихся в каталоге. Из такого каталога пользователь может удалить только те файлы, которыми он владеет или на которые он имеет явное право доступа на запись, даже при наличии права на запись в каталог. Примером является каталог `/tmp`.

Каталог ben

123	.	\0				
247	.	.	\0			
260	b	o	o	k	\0	
401	m	e	m	o	s	\0

Каталог book

260	.	\0

123	.	.	\0			
566	c	h	a	p	1	\0
567	c	h	a	p	2	\0
590	c	h	a	p	3	\0

Каталог memos

401	.	\0	
123	.	.	\0
800	k	h	\0
810	k	d	\0
077	m	w	\0

Рис. 4.5. Каталоги ben, book и memos

На уровне командного интерпретатора связанные с каталогами права доступа можно вывести при помощи команды `ls` с параметром `-l`. Подкаталоги будут обозначаться буквой `d` в первой позиции, например:

```
$ ls -l
total 168
-rw-r-----    1 ben other 39846 Oct 12 21:21 dir_t
drwxr-x---    2 ben other 32    Oct 12 22:02 expenses
-rw-r-----    1 ben other 46245 Oct 13 10:34 new
-rw-r-----    1 ben other 3789  Sep 2  18:40 pwd_text
-rw-r-----    1 ben other 1310  Sep 13 10:38 test.c
```

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

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

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

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

4.4. Использование каталогов при программировании

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

```
PDirent  =  ^Dirent;
Dirent  =  Record
    ino,                    (* Номер индексного дескриптора *)
    off      :  longint;
    reclen   :  word;
    name     :  string[255]  (* Имя файла *)
end;
```

Спецификация XSI не определяет размер `name`, но гарантирует, что число байтов, предшествующих нулевому символу, будет меньше, чем число, хранящееся в переменной `_PC_NAME_MAX`, определенной в заголовочном файле `<unistd.h>`. Обратите внимание, что нулевое значение переменной `ino` обозначает пустую запись в каталоге.

4.4.1. Создание и удаление каталогов

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

Описание

```
uses stdio;

function mkdir(pathname:pchar;mode:integer):integer;
```

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

```
var
  retval:integer;
retval := mkdir('/tmp/dir1', octal(0777));
```

Как обычно, системный вызов `mkdir` возвращает нулевое значение в случае успеха, и -1 – в случае неудачи. Обратите внимание, что `mkdir` также помещает две ссылки (`.` и `..`) в создаваемый новый каталог. Если бы этих элементов не было, работать с полученным каталогом было бы невозможно.

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

Описание

```
uses stdio;

function rmdir(pathname:pchar):integer;
```

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

4.4.2. Открытие и закрытие каталогов

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

Описание

```
uses linux;

Function OpenDir(dirname:pchar):pdir;
Function OpenDir(dirname:string):pdir;
```

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

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

Описание

```
uses linux;

Function CloseDir(dirptr:pdir):integer;
```

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

```
uses linux;
```

```
var
  dp:pdir;
begin
  dp := opendir ('/tmp/dir1');
  if dp = nil then
  begin
    writeln('Ошибка открытия каталога /tmp/dir1');
    halt(1);
  end;
  (*
     Код, работающий с каталогом
     .
     .
     .
  *)
  closedir (dp);
end.
```

4.4.3. Чтение каталогов: вызовы readdir и rewinddir

После открытия каталога из него можно начать считывать записи.

Описание

```
uses linux;

Function ReadDir(dirptr:pdir):pdirent;
```

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

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

Описание

```
uses stdio;

procedure rewinddir(dirptr:pdir);
```

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

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

```
uses linux,stdio;

function my_double_ls(name:pchar):integer;
var
  dp:PDIR;
  d:pdirent;
begin
  (* Открытие каталога с проверкой ошибок *)
  dp:=opendir (name);
  if dp=nil then
  begin
    my_double_ls:=-1;
    exit;
  end;
```

```
  (* Продолжить обход каталога,
   * выводя записи в нем, если
   * индекс остается допустимым
   *)
  d:=readdir(dp);
  while d<>nil do
  begin
    if d^.ino<>0 then
      writeln(d^.name);
    d:=readdir(dp);
  end;

  (* Вернуться к началу каталога ... *)
  rewinddir(dp);
  (* ... и снова вывести его содержимое *)
  d:=readdir(dp);
  while d<>nil do
  begin
    if d^.ino<>0 then
      writeln(d^.name);
    d:=readdir(dp);
  end;
  closedir(dp);
  my_double_ls:=0;
end;
```

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

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

Второй пример: процедура find_entry

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

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

```
uses linux,strings;

function match(s1, s2: pchar):boolean;forward;

function find_entry(dirname:pchar;suffix:pchar;cont:integer):pchar;
const
  dp:pdir=nil;
```

```
var
  d:pdirent;
begin
  if (dp = nil) or (cont = 0) then
  begin
    if dp <> nil then
      closedir (dp);
    dp:=opendir(dirname);
    if dp = nil then
    begin
      find_entry:=nil;
      exit;
    end;
  end;

  d := readdir (dp);
  while d <> nil do
  begin
    if d^.ino = 0 then
      continue;
    if match (d^.name, suffix) then
    begin
      find_entry:=d^.name;
      exit;
    end;
    d := readdir (dp);
  end;

  closedir (dp);
  dp := nil;
  find_entry:=nil;
end;


function match(s1, s2: pchar):boolean;
var
  diff:integer;
begin
  diff := strlen (s1) - strlen (s2);

  if strlen (s1) > strlen (s2) then
    match:=(strcomp (@s1[diff], s2) = 0)
  else
    match:=false;
end;
```

Упражнение 4.1. *Измените функцию* `my_double_ls` *из предыдущего примера так, чтобы она имела второй параметр – целочисленную переменную* `skip`*. Если значение* `skip` *равно нулю, то функция* `my_double_ls` *должна выполняться так же, как и раньше. Если значение переменной* `skip` *равно 1, функция* `my_double_ls` *должна пропускать все имена файлов, которые начинаются сточки (.).*

Упражнение 4.2. *В предыдущей главе мы познакомились с использованием системного вызова* `fstat` *для получения информации о файле. Структура* `tstat`*, возвращаемая вызовом* `fstat`*, содержит поле* `mode`*, режим доступа к файлу. Режим доступа к файлу образуется при помощи выполнения побитовой операции ИЛИ значения кода доступа с константами, определяющими, является ли этот файл обычным файлом, каталогом, специальным файлом, или механизмом межпроцессного взаимодействия, таким*

как именованный канал. Наилучший способ проверить, является ли файл каталогом – использовать макрос S_ISDIR*:*

```
(* Переменная buf получена в результате вызова fstat *)
if S_ISDIR(buf.mode) then
  writeln('Это каталог')
else
  writeln('Это не каталог');
```

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

В дополнение к упражнению приведем пример, демонстрирующий остальные S_-функции:

```
Uses linux;

Var Info : Stat;

begin
  if LStat (paramstr(1),info) then
    begin
    if S_ISLNK(info.mode) then
      Writeln ('File is a link');
    if S_ISREG(info.mode) then
      Writeln ('File is a regular file');
    if S_ISDIR(info.mode) then
      Writeln ('File is a directory');
    if S_ISCHR(info.mode) then
      Writeln ('File is a character device file');
    if S_ISBLK(info.mode) then
      Writeln ('File is a block device file');
    if S_ISFIFO(info.mode) then
      Writeln ('File is a named pipe (FIFO)');
    if S_ISSOCK(info.mode) then
      Writeln ('File is a socket');
    end;
end.
```

4.4.4. Текущий рабочий каталог

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

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

4.4.5. Смена рабочего каталога при помощи вызова chdir

Описание

```
uses stdio;

function chdir(path:pchar):integer;
```

После выполнения системного вызова chdir каталог path становится текущим рабочим каталогом вызывающего процесса. Важно отметить, что эти изменения относятся только к процессу, который выполняет вызов chdir. Смена текущего каталога в программе не затрагивает запустивший программу командный интерпретатор, поэтому после выхода из

программы пользователь окажется в том же рабочем каталоге, в котором он находился перед запуском программы, независимо от перемещений программы.

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

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

```
fd1 := fdopen('/usr/ben/abc', Open_RDONLY);
fd2 := fdopen('/usr/ben/xyz', Open_RDWR);
```

можно использовать:

```
chdir('/usr/ben');
fd1 := fdopen('abc', Open_RDONLY);
fd2 := fdopen('xyz', Open_RDWR);
```

4.4.6. Определение имени текущего рабочего каталога

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

Описание

```
uses stdio;

function getcwd(name:pchar; size:longint):pchar;

uses linux;

Function TellDir(p:pdir):longint;
```

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

Функция TellDir помещает текущий каталог по указателю p, возвращая 0 в случае успешного завершения и -1 – при ошибке.

Альтернативой getcwd является определенная в модуле sysutils функция GetCurrentDir.

Описание

```
uses sysutils;

Function GetCurrentDir:String;
```

Эта короткая программа имитирует команду pwd:

```
(* Программа my_pwd - вывод рабочего каталога *)

uses sysutils;

procedure my_pwd;
```

```
begin
  writeln(GetCurrentDir);
end;


begin
  my_pwd;
end.
```

4.4.7. Обход дерева каталогов

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

Описание

```
uses linux,stdio,strings;

const
  FTW_NS =100;    (* При ошибке stat(2) *)
  FTW_DNR=200;    (* При ошибке opendir(3) *)
  FTW_F  =300;    (* Обычный файл *)
  FTW_D  =400;    (* Каталог *)
  MAXNAMLEN=4000;


(* Удобное сокращение *)
function EQ(a,b:pchar):boolean;
begin
  EQ:=(strcomp(a, b) = 0);
end;

type
  func=function(name:pchar; var status:tstat; _type:integer):integer;


function ftw(directory:pchar; funcptr:func; depth:integer):integer;
var
  dp:pdir;
  p,fullpath:pchar;
  i:integer;
  e:pdirent;
  sb:tstat;
  seekpoint:longint;
begin
  (* При невозможности выполнения fstat, сообщаем пользователю об этом *)
  if not fstat(directory, Sb) then
  begin
    ftw:=funcptr(directory, Sb, FTW_NS);
    exit;
  end;

  (* Если не каталог, вызываем пользовательскую функцию. *)
  if ((Sb.mode and STAT_IFMT) <> STAT_IFDIR) then
  (* Сообщение "FTW_F" может быть некорректным (вдруг это символическая
ссылка? *)
  begin
    ftw:=funcptr(directory, Sb, FTW_F);
```

```
  exit;
end;

(* Открываем каталог; при невозможности - сообщаем пользователю. *)
Dp := opendir(directory);
if dp = nil then
begin
  ftw:=funcptr(directory, Sb, FTW_DNR);
  exit;
end;

(* Определяем, желает ли пользователь продолжать. *)
i := funcptr(directory, Sb, FTW_D);

if i <> 0 then
begin
  closedir(Dp);
  ftw:=i;
  exit;
end;

(* Готовим место для хранения поного пути. *)
i := strlen(directory);
fullpath := stralloc(i + 1 + MAXNAMLEN + 1);
if fullpath = nil then
begin
  closedir(Dp);
  ftw:=-1;
  exit;
end;
strcopy(fullpath, directory);
p := @fullpath[i];
if (i<>0) and (p[-1] <> '/') then
begin
  p^:='/';
  inc(p);
end;

(* Читаем все элементы каталога. *)
E := readdir(Dp);
while E <> nil do
begin
  if not EQ(E^.name, '.') and not EQ(E^.name, '..') then
  begin
    if depth <= 1 then
    begin
      (* Слишком углубились - закрываем этот каталог. *)
      seekpoint := telldir(Dp);
      closedir(Dp);
      Dp := nil;
    end;

    (* Обработка файла. *)
    strcopy(p, E^.name);
    i := ftw(fullpath, funcptr, depth - 1);
    if i<>0 then
    begin
      (* Пользователь завершил; оканчиваем работу. *)
```

```
        strdispose(fullpath);
        if Dp<>nil then
          closedir(Dp);
          ftw:=i;
          exit;
      end;

      (* Повторно отрываем каталог в случае необходимости. *)
      if Dp = nil then
      begin
        Dp := opendir(directory);
        if Dp = nil then
        begin
          (* WTF? *)
          strdispose(fullpath);
          ftw:=-1;
          exit;
        end;
        seekdir(Dp, seekpoint);
      end;
    end;
    E := readdir(Dp);
  end;

  (* Завершающие действия. *)
  strdispose(fullpath);
  closedir(Dp);
  ftw:=0;
end;
```

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

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

```
function func (name:pchar; var status:tstat; _type:integer):integer;
begin
  (* Тело функции *)
end;
```

Целочисленный аргумент `_type` может принимать одно из нескольких возможных значений, описывающих тип встретившегося объекта. Вот эти значения:

`FTW_F`	Объект является файлом
`FTW_D`	Объект является каталогом
`FTW_DNR`	Объект является каталогом, который нельзя прочесть

FTW_SL Объект является символьной ссылкой

FTW_NS Объект не является символьной ссылкой, и для него нельзя успешно выполнить вызов fstat

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

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

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

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

```
function list(name:pchar; var status:tstat; _type:integer):integer;
begin
  (* Если вызов stat завершился неудачей, просто вернуться *)
  if (_type = FTW_NS) then
  begin
    list:=0;
    exit;
  end;

  (*
   * Иначе, вывести имя объекта,
   * права доступа к нему и постфикс "*",
   * если объект является каталогом или символьной ссылкой.
   *)
  if (_type = FTW_F) then
    printf ('%-30s'#9'0%3o'#$a, [name, status.mode and octal(0777)])
  else
    printf ('%-30s*'#9'0%3o'#$a, [name, status.mode and octal(0777)]);

  list:=0;
end;
```

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

```
var
  path:array [0..255] of char;
begin
  if paramcount=0 then
    ftw ('.', @list, 1)
  else
  begin
    strpcopy(path,paramstr(1));
    ftw (path, @list, 1);
  end;
  halt(0);
end.
```

Вывод программы list для простой иерархии каталогов будет выглядеть так:

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

Обратите внимание на порядок обхода каталогов.

В модуле `linux` определен ряд специализированных функций для обхода дерева каталогов.

Описание

```
uses linux;

Function FNMatch(const Pattern, Name:string):Boolean;
Function FSearch(Path:pathstr; DirList:string):Pathstr;
Function Glob(Const Path:Pathstr):PGlob;
Procedure GlobFree(Var P:Pglob);
```

`FNMatch` возвращает `True`, если имя файла в `Name` совпадает с шаблоном в `Pattern`. Шаблон может содержать знаки `*` (совпадение с нулем или более символов) или `?` (совпадение с одиночными символом).

`FSearch` ищет в `DirList`, списке каталогов, разделенных двоеточием, файл, указанный в `Path`, возвращаю путь к найденному файлу или пустую строку.

`Glob` возвращает указатель на структуру `tglob`, содержащую имена всех файлов, отвечающих шаблону в `Path`. Возвращает `nil` при ошибке, устанавливая `LinuxError`.

`GlobFree` освобождает память, занятую структурой `tglob`.

Например:

```
Uses linux;

Var G1,G2 : PGlob;

begin
  G1:=Glob ('*');
  if LinuxError=0 then
    begin
    G2:=G1;
    Writeln ('Files in this directory : ');
    While g2<>Nil do
      begin
      Writeln (g2^.name);
      g2:=g2^.next;
      end;
    GlobFree (g1);
    end;
end.
```

4.5. Файловые системы UNIX

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

Каждая файловая система начинается с каталога в иерархическом дереве. Это свойство позволяет системным администраторам разбивать иерархию файлов UNIX и

отводить под ее части отдельные области на диске или даже распределять файловую структуру между несколькими физическими дисковыми устройствами. В большинстве случаев физическое разбиение файловой системы остается невидимым для пользователей.

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

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

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

Традиционная файловая система разбита на ряд логических частей. Каждая такая файловая система содержит четыре определенных секции: *загрузочная область* (bootstrap area), *суперблок* (superblock), ряд блоков, зарезервированных для структур *индексных дескрипторов* (inode) файловой системы, и области, отведенной для блоков данных, образующих файлы этой файловой системы. Это расположение схематично представлено на рис. 4.6. Первый из этих блоков (блок с нулевым логическим номером, физически он может быть расположен где угодно внутри раздела диска) зарезервирован для использования в качестве загрузочного блока. Это означает, что он может содержать зависящую от оборудования загрузочную программу, которая используется для загрузки ОС UNIX при старте системы.

Рис. 4.6. Расположение традиционной файловой системы

Логический блок 1 в файловой системе называется *суперблоком*. Он содержит всю жизненно важную информацию о системе, например, полный размер файловой системы (r блоков на приведенном рисунке), число блоков, отведенных для индексных дескрипторов (n–2), дату и время последнего обновления файловой системы. Суперблок содержит также два списка. В первом из них находится часть цепочки номеров свободных блоков секции данных, а во втором – часть цепочки номеров свободных индексных дескрипторов. Эти два списка обеспечивают ускорение доступа к файловой системе при выделении новых блоков на диске для хранения дополнительных данных или при создании нового файла или

каталога. Суперблок смонтированной файловой системы находится в памяти для обеспечения быстрого доступа к списку свободных блоков и свободных узлов. Эти списки в памяти пополняются с диска по мере их исчерпания.

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

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

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

4.5.1. Кэширование: вызовы sync и fsync

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

Описание

```
uses stdio;

procedure sync;

function fsync(filedes:integer):integer;
```

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

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

Чтобы убедиться, что содержимое файловых систем на диске не слишком надолго отстает от времени, в системе UNIX регулярно производится вызов `sync`. Обычно период

запуска sync равен 30 секундам, хотя этот параметр может изменяться системным администратором.

4.6. Имена устройств UNIX

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

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

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

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

могут соответствовать трем портам терминалов системы, а имена

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

могут обозначать матричный принтер и два накопителя на магнитной ленте. Имена разделов диска могут иметь разнообразный формат, например:

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

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

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

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

Для доступа к файлам устройств в программе могут использоваться вызовы fdopen, fdclose, fdread и fdwrite, например, программа

```
uses linux;

var
  i,fd:integer;
begin

  fd := fdopen ('/dev/tty', Open_WRONLY);

  for i := 1 to 100 do
    fdwrite(fd, 'x', 1);

  fdclose(fd);
end.
```

приведет к выводу 100 символов x на порт терминала tty00. Конечно, работа с терминалом

является отдельной важной темой, поэтому она подробнее будет рассмотрена в главе 9.

4.6.1. Файлы блочных и символьных устройств

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

- семейство файлов блочных устройств соответствует устройствам класса дисковых накопителей (съемных и встроенных) и накопителей на магнитной ленте. Передача данных между ядром и этими устройствами осуществляется блоками стандартного размера. Все блочные устройства обеспечивают произвольный доступ. Внутри ядра доступ к этим устройствам управляется хорошо структурированным набором процедур и структур ядра. Этот общий интерфейс к блочным устройствам означает, что обычно драйверы блочных устройств очень похожи, различаясь только в низкоуровневом управлении заданным устройством;
- семейство файлов символьных устройств соответствует устройствам терминалов, модемных линий, устройствам печати, то есть тем устройствам, которые не используют блочный механизм структурированной пересылки данных. Произвольный доступ для символьных устройств может как поддерживаться, так и не поддерживаться. Данные передаются не блоками фиксированного размера, а в виде потоков байтов произвольной длины.

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

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

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

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

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

4.6.2. Структура tstat

Структура `tstat`, которую уже была обсуждена в главе 3, позволяет хранить ин формацию о файле устройства в двух полях:

`mode` В случае файла устройства это поле содержит права доступа к

файлу, к которым прибавлено восьмеричное значение 060000 для блочных устройств или 020000 для символьных устройств. В модуле `linux` определены константы `STAT_IFBLK` и `STAT_IFCHR`, которые могут использоваться вместо этих чисел

`rdev` — Это поле содержит старший и младший номера устройства

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

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

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

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

```
if S_ISCHR(buf.mode) then
  writeln('Символьное устройство')
else
  writeln('Не символьное устройство');
```

`S_ISCHR` – это макрос, определенный в модуле `linux`.

4.6.3. Информация о файловой системе

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

Описание

```
uses linux;

Function FSStat(Path:Pathstr; Var buf:statfs):Boolean;
Function FSStat(Fd:longint; Var buf:stat):Boolean;
```

Обе функции возвращают информацию о файловой системе, заданной либо именем файла устройства `path`, либо дескриптором открытого файла `fd`. Параметр `buf` является указателем на структуру `statfs`, определенную модуле `linux`. Структура `statfs` включает, по меньшей мере, следующие элементы:

`bsize:longint;` — Размер блока данных, при котором система имеет наибольшую производительность. Например, значение `bsize` может составлять при этом 8 Кбайт, что означает, что система обеспечивает более эффективный ввод/вывод при операциях с такими порциями данных

`bfree:longint;` — Полное число свободных блоков

`bavail:longint;` — Число свободных блоков, доступных непривилегированным процессам

`files:longint;` — Полное число номеров индексных дескрипторов

`ffree:longint;` — Полное число свободных номеров индексных дескрипторов

`fsid:longint;` — Идентификатор файловой системы

`namelen:longint;` — Максимальная длина файла

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

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

uses linux;
```

```
var
  buf:statfs;

begin
  if paramcount<>1 then
  begin
    writeln('Применение: fsys имя_файла');
    halt(1);
  end;

  if not fsstat(paramstr(1), buf) then
  begin
    writeln('Ошибка вызова fsstat');
    halt(2);
  end;

  writeln(paramstr(1),': свободных блоков ', buf.bfree, ', свободных индексов
', buf.ffree);
  halt(0);
end.
```

4.6.4. Ограничения файловой системы: процедуры pathconf и fpathconf

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

Описание

```
uses stdio;

function pathconf(pathname:pchar;name:longint):integer;

function fpathconf(filedes, name:longint):integer;
```

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

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

```
(* Программа lookup - выводит установки ограничений файлов *)

uses stdio;

type table=record
  val:integer;
  name:pchar;
end;

var
  tb:^table;
const options:array [0..3] of table=(
    (val:_PC_LINK_MAX; name:'Максимальное число ссылок'),
    (val:_PC_NAME_MAX; name:'Максимальная длина имени файла'),
```

```
    (val:_PC_PATH_MAX; name:'Максимальная длина пути'),
    (val:-1; name:nil)
  );
begin
  tb:=options;
  while tb^.name<>nil do
  begin
    printf('%-32.31s%ld'#$a, [tb^.name, pathconf ('/tmp', tb^.val)]);
    inc(tb);
  end;
end.
```

На одной из систем эта программа вывела следующий результат:

```
Максимальное число ссылок        32767
Максимальная длина имени файла   256
Максимальная длина пути          1024
```

Эти значения относятся к каталогу `/tmp`. Максимально возможное число ссылок является характеристикой самого каталога, а максимальная длина имени файла относится к файлам в каталоге. Существуют также *общесистемные ограничения* (system-wide limits), они декларируются в файле `<limits.h>` и их значения могут быть определены при помощи похожей процедуры `sysconf`.

Глава 5. Процесс

5.1. Понятие процесса

Как было уже рассмотрено в главе 1, процессом в терминологии UNIX является просто экземпляр выполняемой программы, соответствующий определению задачи в других средах. Каждый процесс объединяет код программы, значения данных в переменных программы и более экзотические элементы, такие как значения регистров процессора, стек программы и т.д.[1]

Командный интерпретатор для выполнения команд обычно запускает один или несколько процессов. Например, командная строка

```
$ cat file1 file2
```

приведет к созданию процесса для выполнения команды `cat`. Немного более сложная команда

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

приведет к созданию двух процессов для одновременного выполнения команд `ls` и `wc`. (Кроме этого, результат программы `ls`, вывод списка файлов в каталоге, *перенаправляется* с помощью *программного канала* (pipe) на вход программы подсчета числа слов `wc`.)

Так как процессы соответствуют выполняемым программам, не следует путать их с программами, которые они выполняют. Несколько процессов могут выполнять одну и ту же программу. Например, если несколько пользователей выполняют одну и ту же программу редактора, то каждый из экземпляров программы будет отдельным процессом.

Любой процесс UNIX может, в свою очередь, запускать другие процессы. Это придает среде процессов UNIX иерархическую структуру, подобную дереву каталогов файловой системы. На вершине дерева процессов находится единственный управляющий процесс, экземпляр очень важной программы `init`, которая является предком всех системных и пользовательских процессов.

Система UNIX предоставляет программисту набор системных вызовов для создания процессов и управления ими. Если исключить различные средства для межпроцессного взаимодействия, то наиболее важными из оставшихся будут:

`fork`	Используется для создания нового процесса, дублирующего вызывающий. Вызов `fork` является основным примитивом создания процессов
`exec`	Семейство библиотечных процедур и один системный вызов, выполняющих одну и ту же функцию – смену задачи процесса за счет перезаписи его пространства памяти новой программой. Различие между вызовами `exec` в основном лежит в способе задания списка их аргументов
`wait`	Этот вызов обеспечивает элементарную синхронизацию процессов. Он позволяет процессу ожидать завершения другого процесса, обычно логически связанного с ним
`halt`	Используется для завершения процесса

Далее рассмотрим, что представляют собой процессы UNIX в целом и вышеприведенные четыре важных системных вызова в частности.

[1] Не следует путать это понятие с понятием потока выполнения, когда несколько копий кода могут работать с одним набором данных. Потоки выполнения сейчас доступны в некоторых реализациях UNIX, и они охватываются последними расширениями стандарта POSIX и спецификации XSI. Тем не менее мы не будем более подробно описывать сложности многопоточной модели. За дальнейшей информацией обратитесь к справочному руководству системы.

5.2. Создание процессов

5.2.1. Системный вызов fork

Основным примитивом для создания процессов является системный вызов `fork`. Он является механизмом, который превращает UNIX в многозадачную систему.

Описание

```
uses linux;

Function Fork:Longint;
```

В результате успешного вызова `fork` ядро создает новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе, за одним важным исключением, которое вскоре обсудим.

Созданный процесс называется *дочерним процессом* (child process), а процесс, осуществивший вызов `fork`, называется *родителем* (parent).

После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом `fork`.

Идею, заключенную в вызове `fork`, быть может, достаточно сложно понять тем, кто привык к схеме последовательного программирования. Рис. 5.1 иллюстрирует это понятие. На рисунке рассматриваются три строки кода, состоящие из вызова `writeln`, за которым следует вызов `fork`, и еще один вызов `writeln`.

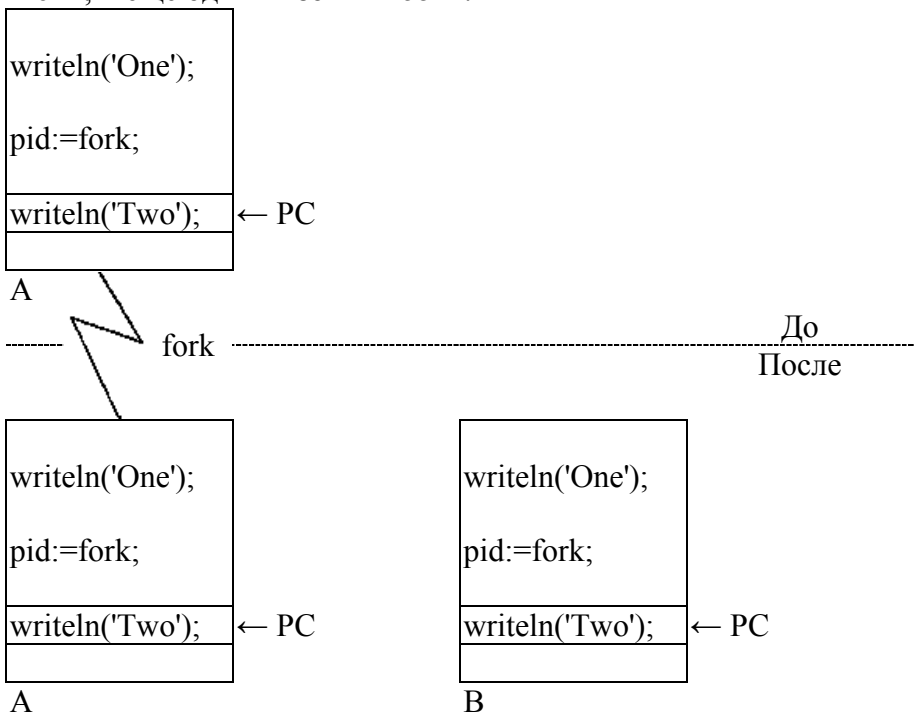


Рис. 5.1. Вызов fork

Рисунок разбит на две части: *До* и *После*. Часть рисунка *До* показывает состояние до вызова `fork`. Существует единственный процесс *A* (его обозначили буквой *A* только для удобства, для системы это ничего не значит). Стрелка, обозначенная *PC* (*program counter* – программный счетчик), указывает на выполняемый в настоящий момент оператор. Так как стрелка указывает на первый оператор `writeln`, на стандартный вывод выдается тривиальное сообщение `One`.

Часть рисунка *После* показывает ситуацию сразу же после вызова `fork`. Теперь

существуют два выполняющихся одновременно процесса: *A* и *B*. Процесс *A* – это тот же самый процесс, что и в части рисунка *До*. Процесс *B* – это новый процесс, порожденный вызовом `fork`. Этот процесс является копией процесса *A* за одним важным исключением – он имеет другое значение идентификатора процесса `pid`, но выполняет ту же самую программу, что и процесс *A*, то есть те же три строки исходного кода, приведенные на рисунке. В соответствии с введенной выше терминологией процесс *A* является родительским процессом, а процесс *B* – дочерним.

Две стрелки с надписью *PC* в этой части рисунка показывают, что следующим оператором, который выполняется родителем и потомком после вызова `fork`, является вызов `writeln`. Другими словами, оба процесса *A* и *B* продолжают выполнение с той же точки кода программы, хотя процесс *B* и является новым процессом для системы. Поэтому сообщение `Two` выводится дважды.

Идентификатор процесса

Как было уже отмечено в начале этого раздела, вызов `fork` не имеет аргументов и возвращает идентификатор процесса `pid` типа `longint`. Пример вызова:

```
uses linux;
var
  pid:longint;

pid := fork;
```

Родитель и потомок отличаются значением переменной `pid`. В родительском процессе значение переменной `pid` будет ненулевым положительным числом, для потомка же оно равно нулю. Так как возвращаемые в родительском и дочернем процессе значения различаются, то программист может задавать различные действия для двух процессов.

Значение, возвращаемое родительскому процессу в переменной `pid`, называется *идентификатором процесса* (process-id) дочернего процесса. Это число идентифицирует процесс в системе аналогично идентификатору пользователя. Поскольку все процессы порождаются при помощи вызова `fork`, то каждый процесс UNIX имеет уникальный идентификатор процесса.

Следующая короткая программа более наглядно показывает работу вызова `fork` и использование идентификатора процесса:

```
 (* Программа spawn - демонстрация вызова fork *)
uses linux;

var
  pid:longint;    (* process-id в родительском процессе *)
begin
  writeln ('Пока всего один процесс');
  writeln ('Вызов fork...');

  pid := fork;    (* создание нового процесса *)

  if pid = 0 then
    writeln ('Дочерний процесс')
  else if (pid > 0) then
    writeln ('Родительский процесс, pid потомка ', pid)
  else
    writeln ('Ошибка вызова fork, потомок не создан');
end.
```

Оператор `if`, следующий за вызовом `fork`, имеет три ветви. Первая определяет дочерний процесс, соответствующий нулевому значению переменной `pid`. Вторая задает действия для родительского процесса, соответствуя положительному значению переменной `pid`. Третья ветвь неявно соответствует отрицательному, а на самом деле равному -1,

значению переменной pid, которое возвращается, если вызову fork не удается создать дочерний процесс. Это может означать, что вызывающий процесс попытался нарушить одно из двух ограничений; первое из них – системное ограничение на число процессов; второе ограничивает число процессов, одновременно выполняющихся и запущенных одним пользователем. В обоих случаях переменная linuxerror содержит код ошибки Sys_EAGAIN. Обратите также внимание на то, что поскольку оба процесса, созданных программой, будут выполняться одновременно без синхронизации, то нет гарантии, что вывод родительского и дочернего процессов не будет смешиваться.

Перед тем как продолжить, стоит обсудить, зачем нужен вызов fork, поскольку сам по себе он может показаться бессмысленным. Существенный момент заключается в том, что вызов fork обретает ценность в сочетании с другими средствами UNIX. Например, возможно, что родительский и дочерний процессы будут выполнять различные, но связанные задачи, организуя совместную работу при помощи одного из механизмов межпроцессного взаимодействия, такого как сигналы или каналы (описываемые в следующих главах). Другим средством, часто используемым совместно с вызовом fork, является системный вызов exec, позволяющий выполнять другие программы, и который будет рассмотрен в следующем разделе.

Упражнение 5.1. *Программа может осуществлять вызов* fork *несколько раз. Аналогично каждый дочерний процесс может вызывать* fork*, порождая своих потомков. Чтобы доказать это, напишите программу, которая создает два подпроцесса, а они, в свою очередь, – свой подпроцесс. После каждого вызова* fork *каждый родительский процесс должен использовать функцию* writeln *для вывода идентификаторов своих дочерних процессов.*

5.3. Запуск новых программ при помощи вызова exec

5.3.1. Семейство вызовов exec

Если бы вызов fork был единственным доступным для программиста примитивом создания процессов, то система UNIX была бы довольно скучной, так как в ней можно было бы создавать копии только одной программы. К счастью, для смены исполняемой программы можно использовать функции семейства exec. На рис. 5.2 показано дерево семейства функций exec. Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рисунка, в конечном итоге все эти функции выполняют один системный вызов execve.

Описание

```
uses stdio;

(* Для семейства вызовов linuxexecl аргументы должны быть списком,
   заканчивающимся NULL *)

function linuxexecl(path:pchar;arg0:pchar;argv:array of const):integer;
function linuxexeclp(fname:pchar;arg0:pchar;argv:array of const):integer;


(* Вызову execl нужно передать полный путь к файлу программы *)
Procedure Execl(Path:pathstr);

(* Вызову execle нужно передать полный путь к файлу программы
   и массив указателей на строки окружения *)
Procedure Execle(Path:pathstr; Envp:ppchar);

(* Вызову execlp нужно только имя файла программы *)
Procedure Execlp(Path:pathstr);
```

```
(* Семейству вызовов execv нужно передать массив аргументов *)

(* Вызову execv нужно передать полный путь к файлу программы *)
Procedure Execv(Path:pathstr;argv:ppchar);

(* Вызову execvp нужно только имя файла программы *)
Procedure Execvp(Path:pathstr;argv:ppchar);

(* Вызову execve нужно передать полный путь к файлу программы
   и массив указателей на строки окружения *)
Procedure Execve(Path:pchar;argv:ppchar;envp:ppchar);
Procedure Execve(Path:pathstr;argv,envp:ppchar);
```

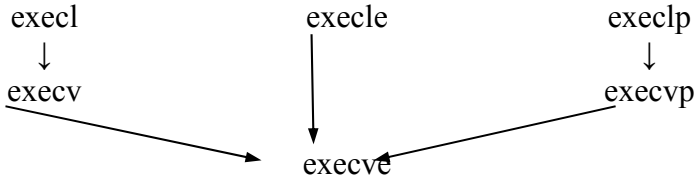


Рис. 5.2. Дерево семейства вызовов exec

Все множество системных вызовов `exec` выполняет одну и ту же функцию: они преобразуют вызывающий процесс, загружая новую программу в его пространство памяти. Если вызов `exec` завершился успешно, то вызывающая программ полностью замещается новой программой, которая запускается с начала. Результат вызова можно рассматривать как запуск нового процесса, который при этом сохраняет идентификатор вызывающего процесса и по умолчанию наследует файловые дескрипторы (см. пункт 5.5.2).

Важно отметить, что вызов `exec` не создает новый подпроцесс, который выполняется одновременно с вызывающим, а вместо этого новая программа загружается на место старой. Поэтому, в отличие от вызова `fork`, успешный вызов `exec` не возвращает значения.

Для простоты осветим только один из вызовов `exec`, а именно `linuxexecl`.

Все аргументы функции `linuxexecl` являются указателями строк. Первый из них, аргумент `path`, задает имя файла, содержащего программу, которая будет запущена на выполнение; для вызова `linuxexecl` это должен быть полный путь к программе, абсолютный или относительный. Сам файл должен содержать программу или последовательность команд оболочки и быть доступным для выполнения. Система определяет, содержит ли файл программу, просматривая его первые байты (обычно первые два байта). Если они содержат специальное значение, называемое *магическим числом* (magic number), то система рассматривает файл как программу. Второй аргумент, `arg0`, является, по соглашению, именем программы или команды, из которого исключен путь к ней. Этот аргумент и оставшееся переменное число аргументов (массив `args`) доступны в вызываемой программе, аналогично аргументам командной строки при запуске программы из оболочки. В действительности командный интерпретатор сам вызывает команды, используя один из вызовов `exec` совместно с вызовом `fork`. Так как список аргументов имеет произвольную длину, он должен заканчиваться нулевым указателем для обозначения конца списка.

Короткий пример ценнее тысячи слов – следующая программа использует вызов `execl` для запуска программы вывода содержимого каталога `ls`:

```
(* Программа runls - использование "execl" для запуска ls *)
uses linux,stdio;

begin
  writeln('Запуск программы ls');
  execl('/bin/ls -l');
  (* Если execl возвращает значение, то вызов был неудачным *)
  perror ('Вызов execl не смог запустить программу ls');
```

```
  halt(1);
end.
```

Работа этой демонстрационной программы показана на рис. 5.3. Часть *До* показывает процесс непосредственно перед вызовом `execl`. Часть *После* показывает измененный процесс после вызова `execl`, который при этом выполняет программу `ls`. Программный счетчик *PC* указывает на первую строку программы `ls`, показывая, что вызов `execl` запускает программу с начала.

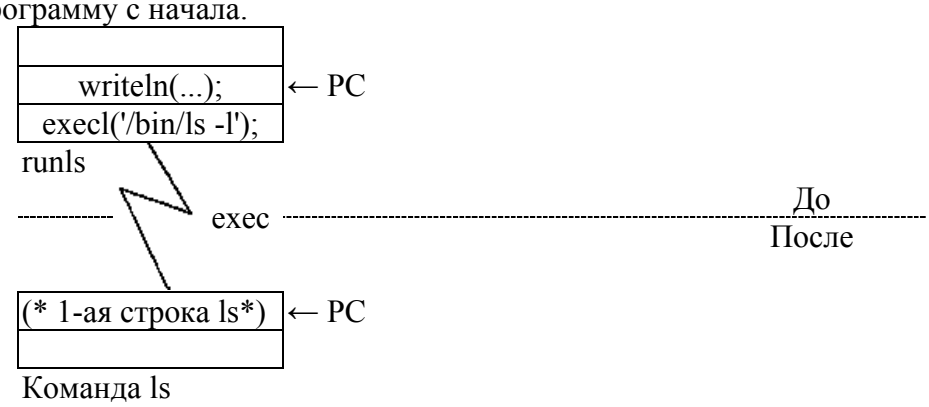


Рис. 5.3. Вызов exec

Обратите внимание, что в примере за вызовом `execl` следует безусловный вызов библиотечной процедуры `perror`. Это отражает то, что успешный вызов функции `execl` (и других родственных функций) стирает вызывающую программу. Если вызывающая программа сохраняет работоспособность и происходит возврат из вызова `execl`, значит, произошла ошибка. Поэтому возвращаемое значение `execl` и родственных функций всегда равно -1.

Вызовы execv, execlp и execvp

Другие формы вызова `exec` упрощают задание списков параметров запуска загружаемой программы. Вызов `execv` принимает два аргумента: первый (`path` в описании применения вызова) является строкой, которая содержит полное имя и путь к запускаемой программе. Второй аргумент (`argv`) является массивом строк, определенным как:

```
argv:ppchar;
```

Первый элемент этого массива указывает, по принятому соглашению, на имя запускаемой программы (исключая префикс пути). Оставшиеся элементы указывают на все остальные аргументы программы. Так как этот список имеет неопределенную длину, он всегда должен заканчиваться нулевым указателем.

Следующий пример использует вызов `execv` для запуска той же программы `ls`, что и в предыдущем примере:

```
(* Программа runls2 - использует вызов execv для запуска ls *)
uses linux,stdio;

const
  av: array [0..2] of pchar=('ls', '-l', nil);
begin
  execv ('/bin/ls', av);
  (* Если мы оказались здесь, то произошла ошибка *)
  perror ('execv failed');
  halt(1);
end.
```

Функции `execlp` и `execvp` почти эквивалентны функциям `execl` и `execv`. Основное отличие между ними состоит в том, что первый аргумент обоих функций `execlp` и `execvp` – просто имя программы, не включающее путь к ней. Путь к файлу находится при помощи поиска в каталогах, заданных в переменной среды `PATH`. Переменная `PATH` может быть легко

задана на уровне командного интерпретатора с помощью следующих команд:

```
$ PATH = /bin:/usr/bin:/usr/keith/mybin
$ export PATH
```

Теперь командный интерпретатор и вызов `execvp` будут вначале искать команды в каталоге `/bin`, затем в `/usr/bin`, и, наконец, в `/usr/keith/mybin`.

Упражнение 5.2. *В каком случае нужно использовать вызов* `execv` *вместо* `execl`?

Упражнение 5.3. *Предположим, что вызовы* `execvp` и `execlp` *не существуют. Напишите эквиваленты этих процедур, используя вызовы* `execl` *и* `execv`. *Параметры этих процедур должны состоять из списка каталогов и набора аргументов командной строки.*

5.3.2. Доступ к аргументам, передаваемым при вызове exec

Любая программа может получить доступ к аргументам активизировавшего ее вызова `exec` через параметры, передаваемые ей. Эти параметры описаны в модуле `syslinux` следующим образом:

```
var
  argc:integer;
  argv:ppchar;
  envp:ppchar;
```

Такое описание должно быть знакомо большинству программистов на Си, так как похожий метод используется для доступа к аргументам командной строки при обычном старте программы – еще один признак того, что командный интерпретатор также использует для запуска процессов вызов `exec`. (Несколько предшествующих примеров и упражнений были составлены с учетом того, что читателям книги известен метод получения программой параметров ее командной строки. Ниже эта тема будет рассмотрена подробнее.)

В вышеприведенном определении значение переменной `argc` равно числу аргументов, переменная `argv` указывает на массив самих аргументов, а переменная `envp` – на массив строк окружения. Поэтому, если программа запускается на выполнение при помощи вызова `execvp` следующим образом:

```
const
  argin:array [0..3] of pchar = ('команда', 'c', 'аргументами', nil);

execvp('prog', argin);
```

то в программе `prog` будут истинны следующие выражения (выражения вида `argv[x] = 'xxx'` следует считать фигуральным равенством, а не выражением языка Паскаль):

При использовании модуля `syslinux`	При использовании модуля `system`
`argc = 3`	`paramcount = 2`
`argv[0] = 'команда'`	`paramstr(0) = 'команда'`
`argv[1] = 'c'`	`paramstr(1) = 'c'`
`argv[2] = 'аргументами'`	`paramstr(2) = 'аргументами'`
`argv[3] = nil`	`paramstr(3) = nil`

В качестве простой иллюстрации этого метода рассмотрим следующую программу, которая печатает свои аргументы, за исключением нулевого, на стандартный вывод:[1]

а) с применением модуля `system`:

```
(* Программа myecho - вывод аргументов командной строки *)
```

[1] Здесь следует отметить, что значение `argv[0]` – не пустая трата памяти, а весьма важный параметр. Во-первых, он напоминает программе ее имя: признаком хорошего стиля программирования считается вывод диагностики от имени программы `argv[0]`, ведь заранее не известно, как впоследствии переименует программу пользователь. Во-вторых, у исполняемого файла может быть несколько имен (вспомните про ссылки), и это можно выгодно использовать. Часто множество утилит на самом деле является одной программой, которая ведет себя по-разному в зависимости от использованного имени. Это кажется странным, но прекрасно работает. Так, программа удаленного выполнения команд `rsh`, будучи названной именем `neibor`, ведет себя так, как будто ей сообщили дополнительный первый аргумент `neibor`. Можно заготовить набор псевдонимов этой программы для запуска команд на соседних системах сети.

```
var
  i:integer;
begin
  for i:=1 to paramcount do
    write(paramstr(i), ' ');
  writeln;
end.
```

б) с применением модуля `syslinux`:

```
(* Программа myecho - вывод аргументов командной строки *)
uses syslinux;

var
  i:integer;
begin
  for i:=1 to argc-1 do
    write(argv[i], ' ');
  writeln;
end.
```

Если вызвать эту программу в следующем фрагменте кода

```
const
  argin:array [0..3] of pchar = ('myecho', 'hello', 'world', nil);
execvp(argin[0], argin);
```

то переменная `argc` в программе `myecho` будет иметь значение 3, и в результат на выходе программы получим:

```
hello world
```

Тот же самый результат можно получить при помощи команды оболочки:

```
$ ./myecho hello world
```

Упражнение 5.4. *Напишите программу* `waitcmd`*, которая выполняет произвольную команду при изменении файла. Она должна принимать в качестве аргументов командной строки имя контролируемой файла и команду, которая должна выполняться в случае его изменения. Для слежения за файлом можно использовать вызов* `fstat`*. Программа не должна расходовать напрасно системные ресурсы, поэтому следует использовать процедуру* `sleep` *(представленную в упражнении 2.16), для приостановки выполнения программы* `waitcmd` *в течение заданного интервала времени, после того как она проверит файл. Как должна действовать программа, если файл изначально не существует?*

5.4. Совместное использование вызовов exec и fork

Системные вызовы `fork` и `exec`, объединенные вместе, представляют мощный инструмент для программиста. Благодаря ветвлению при использовании вызова `exec` во вновь созданном дочернем процессе программа может выполнять другую программу в дочернем процессе, не стирая себя из памяти. Следующий пример показывает, как это можно сделать. В нем мы также представим простую процедуру обработки ошибок `fatal` и системный вызов `wait`. Системный вызов `wait`, описанный ниже, заставляет процесс ожидать завершения работы дочернего процесса.

```
 (* Программа runls3 - выполнить ls как субпроцесс *)
uses linux,stdio;

var
  pid:longint;
begin
  pid := fork;
  case pid of
    -1:
      fatal ('Ошибка вызова fork');
    0:
```

```
    begin
      (* Потомок вызывает exec *)
      execl('/bin/ls -l');
      fatal('Ошибка вызова exec ');
    end;
    else
    begin
      (* Родительский процесс вызывает wait для приостановки
       * работы до завершения дочернего процесса.
       *)
      wait(nil);
      writeln('Программа ls завершилась');
      halt(0);
    end;
  end;
end.
```

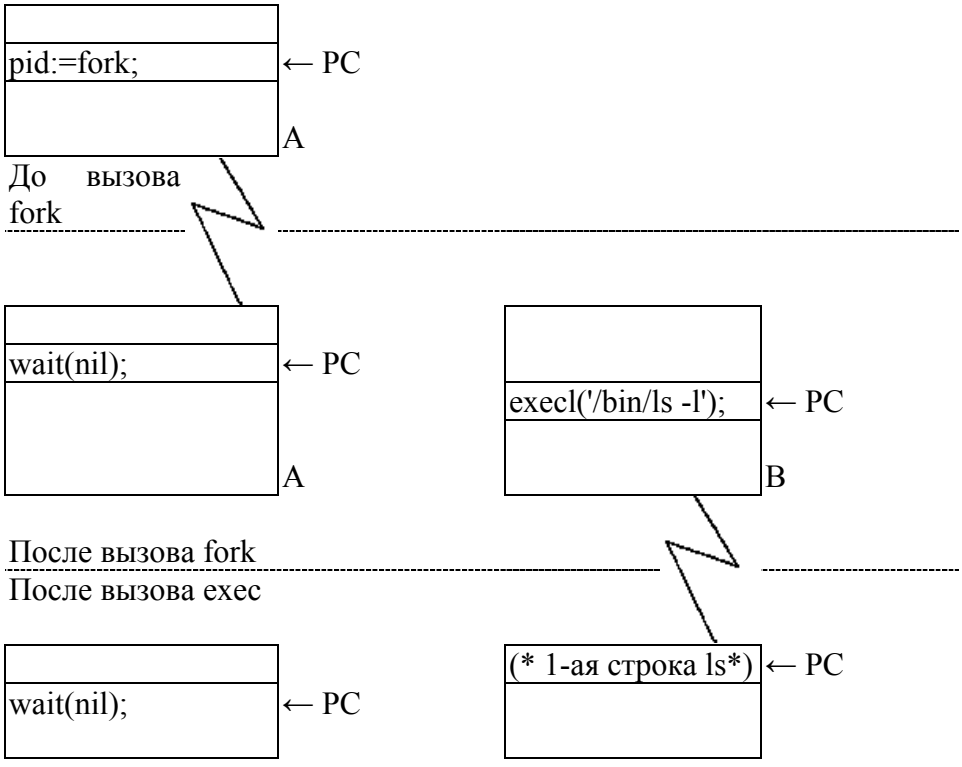
Процедура `fatal` использует функцию `perror` для вывода сообщения, а затем завершает работу процесса. Процедура `fatal` реализована следующим образом:

```
procedure fatal(s:pchar);
begin
  perror(s);
  halt(1);
end;
```

Снова графическое представление, в данном случае рис. 5.4, используется для наглядного объяснения работы программы. Рисунок разбит на три части: *До вызова* `fork`, *После вызова* `fork` и *После вызова* `exec`.

В начальном состоянии, *До вызова* `fork`, существует единственный процесс *A* и программный счетчик *PC* направлен на оператор `fork`, показывая, что это следующий оператор, который должен быть выполнен.

После вызова `fork` существует два процесса, *A* и *B*. Родительский процесс *A* выполняет системный вызов `wait`. Это приведет к приостановке выполнения процесса *A* до тех пор, пока процесс *B* не завершится. В это время процесс *B* использует вызов `execl` для запуска на выполнение команды `ls`.



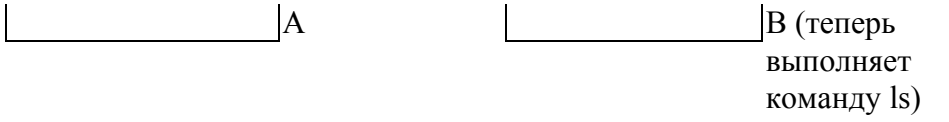


Рис. 5.4. Совместное использование вызовов fork и exec

Что происходит дальше, показано в части *После вызова* `exec` на рис. 5.4. Процесс *B* изменился и теперь выполняет программу `ls`. Программный счетчик процесса *B* установлен на первый оператор команды `ls`. Так как процесс *A* ожидает завершения процесса *B*, то положение его программного счетчика *PC* не изменилось.

Теперь можно увидеть в общих чертах механизмы, используемые командным интерпретатором. Например, при обычном выполнении команды оболочка использует вызовы `fork`, `exec` и `wait` приведенным выше образом. При фоновом исполнении команды вызов wait пропускается, и оба процесса – команда и оболочка – продолжают выполняться одновременно.

Пример docommand

Модуль `stdio` предоставляет библиотечную процедуру `runshell`, которая позволяет выполнить в программе команду оболочки. Для примера создадим упрощенную версию этой процедуры `docommand`, используя вызовы `fork` и `exec`. В качестве посредника вызовем стандартную оболочку (заданную именем `/bin/sh`), а не будем пытаться выполнять программу напрямую. Это позволит программе `docommand` воспользоваться преимуществами, предоставляемыми оболочкой, например, раскрытием шаблонов имен файлов. Задание параметра `-c` вызова оболочки определяет, что команды передаются не со стандартного ввода, а берутся из следующего строчного аргумента.

```
 (* Программа docommand -- запуск команды оболочки, первая версия *)

uses linux,stdio;

function docommand(command:pchar):integer;
var
  pid:longint;
begin
  pid := fork;
  if pid < 0 then
  begin
    docommand:=-1;
    exit;
  end;

  if pid = 0 then            (* дочерний процесс *)
  begin
    linuxexecl('/bin/sh', 'sh', ['-c', command, nil]);
    perror ('execl');
    halt(1);
  end;

  (* Код родительского процесса *)
  (* Ожидание возврата из дочернего процесса *)
  wait(nil);

  docommand:=0;
end;

begin
  docommand('ls -l | wc -l');
end.
```

Следует сказать, что это только первое приближение к настоящей библиотечной процедуре `runshell`. Например, если конечный пользователь программы нажмет клавишу прерывания во время выполнения команды оболочки, то и вызывающая программа, и команда остановятся. Существуют способы обойти это ограничение, которые будут рассмотрены в следующей главе.

5.5. Наследование данных и дескрипторы файлов

5.5.1. Вызов fork, файлы и данные

Созданный при помощи вызова `fork` дочерний процесс является почти точной копией родительского. Все переменные в дочернем процессе будут иметь те же самые значения, что и в родительском (единственным исключением являете значение, возвращаемое самим вызовом `fork`). Так как данные в дочернем процессе являются копией данных в родительском процессе и занимают другое абсолютное положение в памяти, важно понимать, что последующие изменения в одном процессе не будут затрагивать переменные в другом.

Аналогично все файлы, открытые в родительском процессе, также будут открытыми и в потомке; при этом дочерний процесс будет иметь свою копию связанных с каждым файлом дескрипторов. Тем не менее файлы, открытые до вызова `fork`, остаются тесно связанными в родительском и дочернем процессах. Это обусловлено тем, что указатель чтения-записи для каждого из таких файлов используется совместно родительским и дочерним процессами благодаря тому, что он поддерживается системой и существует не только в самом процессе. Следовательно, если дочерний процесс изменяет положение указателя в файле, то в родительском процессе он также окажется в новом положении. Это поведение демонстрирует следующая короткая программа, в которой использована процедура `fatal`, приведенная ранее в этой главе, а также новая процедура `printpos`. Дополнительно введено допущение, что существует файл с именем `data` длиной не меньше 20 символов (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx).

```
(* Программа proc_file -- поведение файлов при ветвлении *)
(* Предположим, что длина файла "data" не менее 20 символов *)
uses linux,stdio;

var
  fd:integer;
  pid:longint;                (* идентификатор процесса *)
  buf:array [0..9] of char;   (* буфер данных для файла *)
begin
  fd := fdopen ('data', Open_RDONLY);
  if fd = -1 then
    fatal ('Ошибка вызова open ');

  fdread (fd, buf, 10);       (* переместить вперед указатель файла *)
  printpos ('До вызова fork', fd);
  (* Создать два процесса *)

  pid := fork;
  case pid of
    1:                  (* ошибка *)
      fatal ('Ошибка вызова fork ');
    0:                  (* потомок *)
    begin
      printpos ('Дочерний процесс до чтения', fd);
      fdread (fd, buf, 10);
      printpos ('Дочерний процесс после чтения', fd);
    end;
```

```
    else                (* родитель *)
    begin
      wait(nil);
      printpos ('Родительский процесс после ожидания', fd);
    end;
  end;
end.
```

Процедура `printpos` просто выводит текущее положение в файле, а также короткое сообщение. Ее можно реализовать следующим образом:

```
 (* Вывести положение в файле *)
procedure printpos(_string:pchar;filedes:integer);
var
  pos:longint;
begin
  pos := fdseek (filedes, 0, SEEK_CUR);
  if pos=-1 then
    fatal ('Ошибка вызова lseek');
  writeln(_string,':',pos);
end;
```

После запуска этого примера получены результаты, которые убедительно подтверждают то, что указатель чтения-записи совместно используется обоими процессами:

```
До вызова fork:10
Дочерний процесс до чтения:10
Дочерний процесс после чтения:20
Родительский процесс после ожидания:20
```

Упражнение 5.5. *Напишите программу, показывающую, что значения переменных программы в родительском и дочернем процессах первоначально совпадают, но не зависят друг от друга.*

Упражнение 5.6. *Определите, что происходит в родительском процессе, если дочерний процесс закрывает файл, дескриптор которого он унаследовал после ветвления. Другими словами, останется ли файл открытым в родительском процессе или же будет закрыт?*

5.5.2. Вызов exec и открытые файлы

Дескрипторы открытых файлов обычно сохраняют свое состояние также во время вызова `exec`, то есть файлы, открытые в исходной программе, остаются открытыми, когда совершенно новая программа запускается при помощи вызова `exec`. Указатели чтения-записи на такие файлы остаются неизменными после вызова `exec`. (Очевидно, не имеет смысла говорить о сохранении значений переменных после вызова `exec`, так как в общем случае новая программа совершенно отличается от старой.)

Тем не менее есть связанный с файловым дескриптором флаг *close-on-exec* (закрывать при вызове `exec`), который может быть установлен с помощью универсальной процедуры `fcntl`. Если этот флаг установлен (по умолчанию он сброшен), то файл закрывается при вызове любой функции семейства `exec`. Следующий фрагмент показывает, как устанавливается флаг close-on-exec:

```
uses linux;

.
.
.
var
  fd:longint;

fd := fdopen('file', Open_RDONLY);
.
```

```
.
.
(* Установить флаг close-on-exec *)
fcntl(fd, F_SETFD, 1);
```

Флаг close-on-exec можно сбросить так:

```
fcntl(fd, F_SETFD, 0);
```

Значение флага можно получить следующим образом:

```
res := fcntl(fd, F_GETFD);
```

Целое `res` будет иметь значение 1, если флаг close-on-exit установлен для дескриптора файла `fd`, и 0 – в противном случае.

5.6. Завершение процессов при помощи системного вызова halt

Описание

```
uses system;

procedure halt(status:word);
```

Системный вызов `halt` уже известен, но теперь следует дать его правильное описание. Этот вызов используется для завершения процесса, хотя это также происходит, когда управление доходит до конца тела главной программы или до процедуры `exit` в теле главной программы.

Единственный целочисленный аргумент вызова `halt` называется *статусом завершения* (exit status) процесса, младшие восемь бит которого доступны родительскому процессу при условии, если он выполнил системный вызов `wait` (подробнее об этом см. в следующем разделе). При этом возвращаемое вызовом `halt` значение обычно используется для определения успешного или неудачного завершения выполнявшейся процессом задачи. По принятому соглашению, нулевое возвращаемое значение соответствует нормальному завершению, а ненулевое значение говорит о том, что что-то случилось.

Кроме завершения вызывающего его процесса, вызов `halt` имеет еще несколько последствий: наиболее важным из них является закрытие всех открытых дескрипторов файлов. Если, как это было в последнем примере, родительский процесс выполнял вызов `wait`, то его выполнение продолжится.

Для полноты изложения следует также упомянуть системный вызов `_exit`, который отличается от вызова `halt` наличием символа подчеркивания в начале. Он используется в точности так же, как и вызов `halt`. Тем не менее он не включает описанные ранее действия по очистке. В большинстве случаев следует избегать использования вызова `_exit`.

Упражнение 5.7. *Статус завершения программы можно получить в командном интерпретаторе при помощи переменной* `$?`*, например:*

```
$ ls nonesuch
  nonesuch: No such file or directory
$ echo $?
  2
```

Напишите программу `fake`*, которая использует целочисленное значение первого аргумента в качестве статуса завершения. Используя намеченный выше метод, выполните программу* `fake`*, задавая различные значения аргументов, включая большие и отрицательные. Есть ли польза от программы* `fake`?

5.7. Синхронизация процессов

5.7.1. Системный вызов wait

Описание

```
uses stdio;

function wait(status:pinteger):longint;
```

Как было уже обсуждено, вызов wait временно приостанавливает выполнение процесса, в то время как дочерний процесс продолжает выполняться. После завершения дочернего процесса выполнение родительского процесса продолжится. Если запущено более одного дочернего процесса, то возврат из вызова wait произойдет после выхода из любого из потомков.

Вызов wait часто осуществляется родительским процессом после вызова fork, например:

```
.
.
.
var
  status:integer;
  cpid:longint;

cpid := fork; (*Создать новый процесс *)

if cpid = 0 then
begin
  (* Дочерний процесс *)
  (* Выполнить какие-либо действия ... *)
end
else
begin
  (* Родительский процесс, ожидание завершения дочернего *)
  cpid := wait(@status);
  writeln('Дочерний процесс ', cpid, ' завершился');
end;
.
.
.
```

Сочетание вызовов fork и wait наиболее полезно, если дочерний процесс предназначен для выполнения совершенно другой программы при помощи вызова exec.

Возвращаемое значение wait обычно является идентификатором дочернего процесса, который завершил свою работу. Если вызов wait возвращает значение -1, это может означать, что дочерние процессы не существуют, и в этом случае переменная linuxerror будет содержать код ошибки Sys_ECHILD. Возможность определить завершение каждого из дочерних процессов по отдельности означает, что родительский процесс может выполнять цикл, ожидая завершения каждого из потомков, а после того, как все они завершатся, продолжать свою работу.

Вызов wait принимает один аргумент, status, – указатель на целое число. Если указатель равен nil, то аргумент просто игнорируется. Если же вызову wait передается допустимый указатель, то после возврата из вызова wait переменная status будет содержать полезную информацию о статусе завершения процесса. Обычно эта информация будет представлять собой код завершения дочернего процесса, переданный при помощи вызова halt.

Следующая программа status показывает, как может быть использован вызов wait:

```
 (* Программа status -- получение статуса завершения потомка *)
uses linux,stdio;

var
  pid:longint;
  status, exit_status:integer;
begin
  pid := fork;
  if pid < 0 then
```

```
    fatal ('Ошибка вызова fork ');
  if pid = 0 then            (* потомок *)
  begin
    (* Вызвать библиотечную процедуру sleep
     * для временного прекращения работы на 4 секунды
     *)
    sleep(4);
    halt(5);                 (* выход с ненулевым значением *)
  end;
  (* Если мы оказались здесь, то это родительский процесс, *)
  (* поэтому ожидать завершения дочернего процесса         *)
  pid := wait(@status);
  if pid = -1 then
  begin
    perror ('Ошибка вызова wait ');
    halt(2);
  end;
  (* Проверка статуса завершения дочернего процесса *)
  if WIFEXITED (status) then
  begin
    exit_status := WEXITSTATUS (status);
    writeln ('Статус завершения ',pid,' равен ', exit_status);
  end;
  halt(0);
end.
```

Значение, возвращаемое родительскому процессу при помощи вызова `halt`, записывается в старшие восемь бит целочисленной переменной `status`. Чтобы оно имело смысл, младшие восемь бит должны быть равны нулю. Функция `WIFEXITED` (определенная в модуле `stdio`) проверяет, так ли это на самом деле. Если `WIFEXITED` возвращает `false`, то это означает, что выполнение дочернего процесса было остановлено (или прекращено) другим процессом при помощи межпроцессного взаимодействия, называемого *сигналом* (signal) и рассматриваемого в главе 6.

Упражнение 5.8. *Переделайте процедуру* `docommand` *так, чтобы она возвращала статус вызова* `halt` *выполняемой команды. Что должно происходить, если вызов* `wait` *возвращает значение -1?*

5.7.2. Ожидание завершения определенного потомка: вызов waitpid

Системный вызов `wait` позволяет родительскому процессу ожидать завершения любого дочернего процесса. Тем не менее, если нужна большая определенность, то можно использовать системный вызов `waitpid` для ожидания завершения определенного дочернего процесса.

Описание

```
uses linux;

Function WaitPid(Pid:longint; Status:pointer; Options:Longint):Longint;
```

Первый аргумент `pid` определяет идентификатор дочернего процесса, завершения которого будет ожидать родительский процесс. Если этот аргумент установлен равным -1, а аргумент `options` установлен равным 0, то вызов `waitpid` ведет себя в точности так же, как и вызов `wait`, поскольку значение -1 соответствует любому дочернему процессу. Если значение `pid` больше нуля, то родительский процесс будет ждать завершения дочернего процесса с идентификатором процесса равным `pid`. Во втором аргументе `status` будет находиться статус дочернего процесса после возврата из вызова `waitpid`.

Последний аргумент, `options`, может принимать константные значения, определенные в модуле `linux`. Наиболее полезное из них – константа `WNOHANG`. Задание

этого значения позволяет вызывать waitpid в цикле без блокирования процесса, контролируя ситуацию, пока дочерний процесс продолжает выполняться. Если установлен флаг WNOHANG, то вызов waitpid будет возвращать 0 в случае, если дочерний процесс еще не завершился.

Функциональные возможности вызова waitpid с параметром WNOHANG можно продемонстрировать, переписав предыдущий пример. На этот раз родительски процесс проверяет, завершился ли уже дочерний процесс. Если нет, он выводит сообщение, говорящее о том, что он продолжает ждать, затем делает секундную паузу и снова вызывает waitpid, проверяя, завершился ли дочерний процесс. Обратите внимание, что потомок получает свой идентификатор процесса при помощи вызова getpid. Об этом вызове расскажем в разделе 5.10.1.

```
(* Программа status2 - получение статуса завершения
 * дочернего процесса при помощи вызова waitpid
 *)
uses linux,stdio,crt;

var
  pid:longint;
  status, exit_status:integer;
begin
  pid := fork;
  if pid < 0 then
    fatal ('Ошибка вызова fork ');

  if pid = 0 then            (* потомок *)
  begin
    (* Вызов библиотечной процедуры sleep
     * для приостановки выполнения на 4 секунды
     *)
    writeln ('Потомок ',getpid,' пауза...');
    sleep(4);
    halt(5);                 (* выход с ненулевым значением *)
  end;

  (* Если мы оказались здесь, то это родительский процесс *)
  (* Проверить, закончился ли дочерний процесс, и если нет,  *)
  (* то сделать секундную паузу, и потом проверить снова     *)
  while (waitpid (pid, @status, WNOHANG) = 0) do
  begin
    writeln ('Ожидание продолжается...\n');
    sleep(1);
  end;

  (* Проверка статуса завершения дочернего процесса *)
  if WIFEXITED (status) then
  begin
    exit_status := WEXITSTATUS (status);
    writeln ('Статус завершения ',pid,' равен ', exit_status);
  end;

  halt(0);
end.
```

При запуске программы получим следующий вывод:

```
Ожидание продолжается...
Потомок 12857 пауза...
Ожидание продолжается...
```

```
Ожидание продолжается...
Ожидание продолжается...
Статус завершения 12857 равен 5
```

5.8. Зомби-процессы и преждевременное завершение программы

До сих пор предполагалось, что вызовы `halt` и `wait` используются правильно, и родительский процесс ожидает завершения каждого подпроцесса. Вместе с тем иногда могут возникать две другие ситуации, которые стоит обсудить:

- в момент завершения дочернего процесса родительский процесс не выполняет вызов `wait`;
- родительский процесс завершается, в то время как один или несколько дочерних процессов продолжают выполняться.

В первом случае завершающийся процесс как бы «теряется» и становится *зомби-процессом* (zombie). Зомби-процесс занимает ячейку в таблице, поддерживаемой ядром для управления процессами, но не использует других ресурсов ядра. В конце концов, он будет освобожден, если его родительский процесс вспомнит о нем и вызовет `wait`. Тогда родительский процесс сможет прочитать статус завершения процесса, и ячейка освободится для повторного использования. Во втором случае родительский процесс завершается нормально. Дочерние процесс (включая зомби-процессы) принимаются процессом `init` (процесс, идентификатор которого pid = 1, становится их новым родителем).

5.9. Командный интерпретатор smallsh

В этом разделе создается простой командный интерпретатор `smallsh`. Этот пример имеет два достоинства. Первое состоит в том, что он развивает понятия, введенные в этой главе. Второе – в том, что подтверждается отсутствие в стандартных командах и утилитах UNIX чего-то особенного. В частности, пример показывает, что оболочка является обычной программой, которая запускается при входе в систему.

Наши требования к программе `smallsh` просты: она должна транслировать и выполнять команды – на переднем плане и в фоновом режиме – а также обрабатывать строки, состоящие из нескольких команд, разделенных точкой с запятой. Другие средства, такие как перенаправление ввода/вывода и раскрытие имен файлов, могут быть добавлены позднее.

Основная логика понятна:

```
while не встретится EOF do
begin
  получить строку команд от пользователя
  оттранслировать аргументы и выполнить
  ожидать возврата из дочернего процесса
end;
```

Дадим имя `userin` функции, выполняющей «получение командной строки от пользователя». Эта функция должна выводить приглашение, а затем ожидать ввода строки с клавиатуры и помещать введенные символы в буфер программы. Функция `userin` реализована следующим образом:

```
uses stdio,linux;

(* Заголовочный файл для примера *)
{$i smallsh.inc}

(* Буферы программы и рабочие указатели *)
var
  inpbuf:array [0..MAXBUF-1] of char;
  tokbuf:array [0..2*MAXBUF-1] of char;
const
  ptr:pchar=@inpbuf[0];
```

```
  tok:pchar=@tokbuf[0];

 (* Вывести приглашение и считать строку *)
function userin(p:pchar):integer;
var
  c, count:integer;
begin
  (* Инициализация для следующих процедур *)
  ptr := inpbuf;
  tok := tokbuf;
  (* Вывести приглашение *)
  write(p);
  count := 0;
  while true do
  begin
    c := getchar;
    if c = EOF then
    begin
      userin:=EOF;
      exit;
    end;
    if count < MAXBUF then
    begin
      inpbuf[count] := char(c);
      inc(count);
    end;
    if (c = $a) and (count < MAXBUF) then
    begin
      inpbuf[count] := #0;
      userin:=count;
      exit;
    end;
    (* Если строка слишком длинная, начать снова *)
    if c = $a then
    begin
      writeln ('smallsh: слишком длинная входная строка');
      count := 0;
      write (p);
    end;
  end;
end;
```

Некоторые детали инициализации можно пока не рассматривать. Главное, что функция `userin` вначале выводит приглашение ввода команды (передаваемое в качестве параметра), а затем считывает ввод пользователя по одному символу до тех пор, пока не встретится символ перевода строки или конец файла (последний случай обозначается символом `EOF`).

Функция `getchar` содержится в стандартной библиотеке ввода/вывода. Она считывает один символ из стандартного ввода программы, который обычно соответствует клавиатуре. Функция `userin` помещает каждый новый символ (если это возможно) в массив символов `inpbuf`. После своего завершения функция `userin` возвращает либо число считанных символов, либо `EOF`, обозначающий конец файла. Обратите внимание, что символы перевода строки не отбрасываются, а добавляются в массив `inpbuf`.

Заголовочный файл `smallsh.inc`, упоминаемый в функции `userin`, содержит определения для некоторых полезных постоянных (например, `MAXBUF`). В действительности файл содержит следующее:

```
(* smallsh.inc - определения для интерпретатора smallsh *)
```

```
{ifndef SMALL_INC}
{define SMALL_INC}

const
  EOL=1;          (* конец строки *)
  ARG=2;          (* обычные аргументы *)
  AMPERSAND=3;          (* символ '&' *)
  SEMICOLON=4;          (* точка с запятой *)
  MAXARG=512;           (* макс. число аргументов *)
  MAXBUF=512;           (* макс. длина строки ввода *)
  FOREGROUND=0;         (* выполнение на переднем плане *)
  BACKGROUND=1;         (* фоновое выполнение *)

{endif} (* SMALL_INC *)
```

Другие постоянные, не упомянутые в функции `userin`, встретятся в следующих процедурах.

Рассмотрим следующую процедуру, `gettok`. Она выделяет *лексемы* (tokens) из командной строки, созданной функцией `userin`. (Лексема является минимальной единицей языка, например, имя или аргумент команды.) Процедура `gettok` вызывается следующим образом:

```
toktype := gettok(@tptr);
```

Целочисленная переменная `toktype` будет содержать значение, обозначающее тип лексемы. Диапазон возможных значений берется из файла `smallsh.inc` и включает символы `EOL` (конец строки), `SEMICOLON` и так далее. Переменная `tptr` является символьным указателем, который будет указывать на саму лексему после вызова `gettok`. Так как процедура `gettok` сама выделяет пространство под строки лексем, нужно передать адрес переменной `tptr`, а не ее значение.

Исходный код процедуры `gettok` приведен ниже. Обратите внимание, что поскольку она ссылается на символьные указатели `tok` и `ptr`, то должна быть включена в тот же исходный файл, что и `userin`. (Теперь должно быть понятно, зачем была нужна инициализация переменных `tok` и `ptr` в начале функции `userin`.)

```
 (* Получить лексему и поместить ее в буфер tokbuf *)
function gettok (outptr:ppchar):integer;
var
  _type:integer;
begin
  (* Присвоить указателю на строку outptr значение tok *)
  outptr^ := tok;
  (* Удалить пробелы из буфера, содержащего лексемы *)
  while (ptr^ = ' ') or (ptr^ = #9) do
    inc(ptr);
  (* Установить указатель на первую лексему в буфере *)
  tok^ := ptr^;
  inc(tok);
  (* Установить значение переменной type в соответствии
   * с типом лексемы в буфере *)
  case ptr^ of
    #$a:
    begin
      _type := EOL;
      inc(ptr);
    end;
    '&':
    begin
      _type := AMPERSAND;
```

```
      inc(ptr);
    end;
    ';':
    begin
      _type := SEMICOLON;
      inc(ptr);
    end;
    else
    begin
      _type := ARG;
      inc(ptr);
      (* Продолжить чтение обычных символов *)
      while inarg (ptr^) do
      begin
        tok^ := ptr^;
        inc(tok);
        inc(ptr);
      end;
    end;
  end;
  tok^ := #0;
  inc(tok);
  gettok:=_type;
end;
```

Функция `inarg` используется для определения того, может ли символ быть частью «обычного» аргумента. Пока можно просто проверять, является ли символ особым для командного интерпретатора команд `smallsh` или нет:

```
const
  special:array [0..5] of char = (' ', #9, '&', ';', #$a, #0);

function inarg(c:char):boolean;
var
  wrk:pchar;
begin
  wrk := special;
  while wrk^<>#0 do
  begin
    if c = wrk^ then
    begin
      inarg:=false;
      exit;
    end;
    inc(wrk);
  end;
  inarg:=true;
end;
```

Теперь можно составить функцию, которая будет выполнять главную работу нашего интерпретатора. Функция `procline` будет разбирать командную строку, используя процедуру `gettok`, создавая тем самым список аргументов процесса. Если встретится символ перевода строки или точка с запятой, то она вызывает для выполнения команды процедуру `runcommand`. При этом она предполагает, что командная строка уже была считана при помощи функции `userin`.

```
{$i smallsh.inc}

function procline:integer;          (* обработка строки ввода *)
var
  arg:array [0..MAXARG] of pchar;  (* массив указателей для runcommand *)
```

```
  toktype:integer;                  (* тип лексемы в команде *)
  narg:integer;                     (* число аргументов *)
  _type:integer;                    (* на переднем плане или в фоне *)
begin
  narg := 0;
  while true do                     (* бесконечный цикл *)
  begin
    (* Выполнить действия в зависимости от типа лексемы *)
    toktype := gettok (@arg[narg]);
    case toktype of
      2://ARG
        if narg < MAXARG then
          inc(narg);
      1,3,4://EOL,SEMICOLON, AMPERSAND:
      begin
        if toktype = AMPERSAND then
          _type := BACKGROUND
        else
          _type := FOREGROUND;
        if narg <> 0 then
        begin
          arg[narg] := nil;
          runcommand (arg, _type);
        end;
        if toktype = EOL then
          exit;
        narg := 0;
      end;
    end;
  end;
end;
```

Следующий этап состоит в определении процедуры `runcommand`, которая в действительности запускает командные процессы. Процедура `runcommand` в сущности, является переделанной процедурой `docommand`, с которой встречались раньше. Она имеет еще один целочисленный параметр `where`. Если параметр `where` принимает значение `BACKGROUND`, определенное в файле `smallsh.inc`, то вызов `waitpid` пропускается, и процедура `runcommand` просто выводит идентификатор процесса и завершает работу.

```
{$i smallsh.inc}

(* Выполнить команду, возможно ожидая ее завершения *)
function runcommand(cline:ppchar;where:integer):integer;
var
  pid:longint;
  status:integer;
begin
  pid := fork;
  case pid of
    -1:
    begin
      perror ('smallsh');
      runcommand:=-1;
      exit;
    end;
    0:
    begin
      execvp (cline^, cline, envp);
      perror (cline^);
```

```
      halt(1);
    end;
  end;

  (* Код родительского процесса *)
  (* Если это фоновый процесс, вывести pid и выйти *)
  if where = BACKGROUND then
  begin
    writeln ('[Идентификатор процесса ',pid,']');
    runcommand:=0;
    exit;
  end;

  (* Ожидание завершения процесса с идентификатором pid *)
  if waitpid (pid, @status, 0) = -1 then
    runcommand:=-1
  else
    runcommand:=status;
end;
```

Обратите внимание, что простой вызов `wait` из функции `docommand` был заменен вызовом `waitpid`. Это гарантирует, что выход из процедуры `docommand` произойдет только после завершения процесса, запущенного в этом вызове `docommand`, и помогает избавиться от проблем с фоновыми процессами, которые завершаются в это время. (Если это кажется не совсем ясным, следует вспомнить, что вызов `wait` возвращает идентификатор первого завершающегося дочернего процесса, а не идентификатор последнего запущенного.)

Процедура `runcommand` также использует системный вызов `execvp`. Это гарантирует, что при запуске программы, заданной командой, выполняется ее поиск во всех каталогах, указанных в переменной окружения `PATH`, хотя, в отличие от настоящего командного интерпретатора, в программе `smallsh` нет никаких средств для работы с переменной `PATH`.

Последний шаг состоит в написании программы, которая связывает вместе остальные функции. Это простое упражнение:

```
(* Программа smallsh - простой командный интерпретатор *)

{$i smallsh.inc}

const
  prompt = 'Command> '; (* приглашение ввода командной строки *)
begin
  while userin (prompt) <> EOF do
    procline;
end.
```

Эта процедура завершает первую версию программы `smallsh`. И снова следует отметить, что это только набросок законченного решения. Так же, как в случае процедуры `docommand`, поведение программы `smallsh` далеко от идеала, когда пользователь вводит символ прерывания, поскольку это приводит к завершению работы программы `smallsh`. В следующей главе будет показано, как можно сделать программу `smallsh` более устойчивой.

Упражнение 5.9. *Включите в программу* `smallsh` *механизм для выключения с помощью символа \ (escaping) специального значения символов, таких как точка с запятой и символ* `&`*, так чтобы они могли входить в список аргументов программы. Программа должна также корректно интерпретировать комментарии, обозначаемые символом* `#` *в начале. Что должно произойти с приглашением командной строки, если пользователь выключил таким способом специальное значение символа возврата строки?*

Упражнение 5.10. *Системный вызов* `dup2` *можно использовать для получения копии дескриптора открытого файла. В этом случае он вызывается следующим образом:*

```
dup2(filedes, reqvalue);
```
где `filedes` *– это исходный дескриптор открытого файла. Значение переменной* `reqvalue` *должно быть небольшим целым числом. Если уже был открыт файл с дескриптором, равным* `reqvalue`*, он закрывается. После успешного вызова переменная* `reqvalue` *будет содержать дескриптор файла, который ссылается на тот же самый файл, что и дескриптор* `filedes`*. Следующий фрагмент программы показывает, как перенаправить стандартный ввод, то есть дескриптор файла со значением 0:*

```
fd := fdopen('somefile', Open_RDONLY);
fdclose (0);
dup2(fd, 0);
```

Используя этот вызов вместе с системными вызовами `fdopen` *и* `fdclose`*, переделайте программу* `smallsh` *так, чтобы она поддерживала перенаправление стандартного ввода и стандартного вывода, используя ту же систему обозначений, что и стандартный командный интерпретатор UNIX. Помните, что стандартный ввод и вывод соответствует дескрипторам 0 и 1 соответственно. Обратите внимание, что существует также близкий по смыслу вызов* `dup`.

5.10. Атрибуты процесса

С каждым процессом UNIX связан набор атрибутов, которые помогают системе управлять выполнением и планированием процессов, обеспечивать защиту файловой системы и так далее. Один из атрибутов, с которым мы уже встречались, – это идентификатор процесса, то есть число, которое однозначно идентифицирует процесс. Другие атрибуты простираются от окружения, которое является набором строк, определяемых программистом и находящихся вне области данных, до действующего идентификатора пользователя, определяющего права доступа процесса к файловой системе. В оставшейся части этой главы рассмотрим наиболее важные атрибуты процесса.

5.10.1. Идентификатор процесса

Как уже отмечено в начале этой главы, система присваивает каждому процессу неотрицательное число, которое называется идентификатором процесса. В любой момент времени идентификатор процесса является уникальным, хотя после завершения процесса он может использоваться снова для другого процесса. Некоторые идентификаторы процесса зарезервированы системой для особых процессов. Процесс с идентификатором 0, хотя он и называется *планировщиком* (scheduler), на самом деле является процессом *подкачки памяти* (swapper). Процесс с идентификатором 1 – это процесс инициализации, выполняющий программу `/etc/init`. Этот процесс, явно или неявно, является предком всех других процессов в системе UNIX.

Программа может получить свой идентификатор процесса при помощи следующего системного вызова:

```
pid := getpid;
```

Аналогично вызов `getppid` возвращает идентификатор родителя вызывающего процесса:

```
ppid := getppid;
```

Например:

```
Uses linux;

begin
  Writeln ('Process Id = ',getpid,' Parent process Id = ',getppid);
end.
```

Следующая процедура `gentemp` использует вызов `getpid` для формирования уникального имени временного файла. Это имя имеет форму:

```
/tmp/tmp<pid>.<no>
```

Суффикс номера `<no>` увеличивается на единицу при каждом вызове процедуры

gentemp. Процедура также вызывает функцию access, чтобы убедиться, что файл еще не существует:

```
uses linux,strings;

const
  num:integer=0;
  namebuf:array [0..19] of char='';
  prefix='/tmp/tmp';

function gentemp:pchar;
var
  length:integer;
  pid:longint;
begin
  pid := getpid;        (* получить идентификатор процесса *)

  (* Стандартные процедуры работы со строками *)
  strcopy (namebuf, prefix);
  length := strlen (namebuf);

  (* Добавить к имени файла идентификатор процесса *)
  itoa (pid, @namebuf[length]);
  strcat (namebuf, '.');
  length := strlen (namebuf);
  repeat
    (* Добавить суффикс с номером *)
    itoa(num, @namebuf[length]);
    inc(num);
  until(not access (namebuf, F_OK));
  gentemp:=namebuf;
end;
```

Процедура itoa просто преобразует целое число в эквивалентную строку:

```
 (* Функция itoa - преобразует целое число в строку *)
function itoa(i:integer;str:pchar):integer;
var
  power, j : integer;
begin
  j := i;
  power := 1;
  while j >= 10 do
  begin
    power := power * 10;
    j := j div 10;
  end;
  while power > 0 do
  begin
    str^ := char(byte('0') + i div power);
    inc(str);
    i := i mod power;
    power := power div 10;
  end;
  str^ := #0;
end;
```

Обратите внимание на способ преобразования цифры в ее символьный эквивалент в первом операторе во втором цикле for – он опирается на знание таблицы символов ASCII. Следует также отметить, что большую часть работы можно было бы выполнить гораздо проще при помощи процедуры sprintf. Описание процедуры sprintf смотрите в главе 11.

Упражнение 5.11. *Переделайте процедуру* `gentemp` *так, чтобы она принимала в качестве аргумента префикс имени временного файла.*

5.10.2. Группы процессов и идентификаторы группы процессов

Система UNIX позволяет легко помещать процессы в группы. Например, если в командной строке задано, что процессы связаны при помощи программного канала, они обычно помещаются в одну группу процессов. На рис. 5.5 показана такая типичная группа процессов, установленная из командной строки.

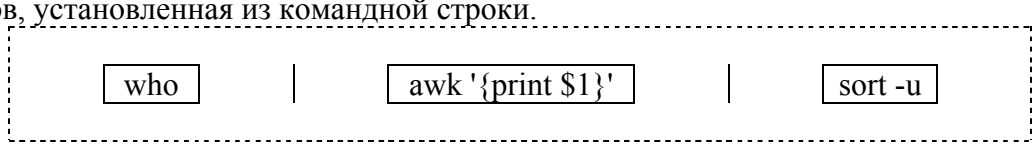


Рис. 5.5. Группа процессов

Группы процессов удобны для работы с набором процессов в целом, с помощью механизма межпроцессного взаимодействия, который называется сигналами, о чем будет сказано подробнее в главе 6. Обычно сигнал «посылается» отдельному процессу и может вызывать завершение этого процесса, но можно послать сигнал и целой группе процессов.

Каждая *группа процессов* (process group) обозначается *идентификатором группы процессов* (process group-id), имеющим тип `pid_t`. Процесс, идентификатор которого совпадает с идентификатором группы процессов, считается *лидером* (leader) группы процессов, и при его завершении выполняются особые действия. Первоначально процесс наследует идентификатор группы во время вызова `fork` или `exec`.

Процесс может получить свой идентификатор группы при помощи системного вызова `getpgrp`:

Описание

```
uses stdio;

function getpgrp:longint;
```

5.10.3. Изменение группы процесса

В оболочке UNIX, поддерживающей управление заданиями, может потребоваться переместить процесс в другую группу процессов. Управление заданиями позволяет оболочке запускать несколько групп процессов (заданий) и контролировать, какие группы процессов должны выполняться на переднем плане и, следовательно, иметь доступ к терминалу, а какие должны выполняться в фоне. Управление заданиями организуется при помощи сигналов.

Процесс может создать новую группу процессов или присоединиться к существующей при помощи системного вызова `setpgid`:

Описание

```
uses stdio;

function setpgid(pid, pgid:longint):longint;
```

Вызов `setpgid` устанавливает идентификатор группы процесса с идентификатором `pid` равным `pgid`. Если `pid` равен 0, то используется идентификатор *вызывающего* процесса. Если значения идентификаторов `pid` и `pgid` одинаковы, то процесс становится лидером группы процессов. В случае ошибки возвращает значение -1. Если идентификатор `pgid` равен нулю, то в качестве идентификатора группы процесса используется идентификатор процесса `pid`.

5.10.4. Сеансы и идентификатор сеанса

В свою очередь, каждая группа процессов принадлежит к сеансу. В действительности сеанс относится к связи процесса с *управляющим терминалом* (controlling terminal). Когда пользователи входят в систему, все процессы и группы процессов, которые они явно или

неявно создают, будут принадлежать к сеансу, связанному с их текущим терминалом. Сеанс обычно представляет собой набор из одной группы процессов переднего плана, использующей терминал, и одной или более групп фоновых процессов. Сеанс обозначается при помощи *идентификатора сеанса* (session-id), который имеет тип `longint`.

Процесс может получить идентификатор сеанса при помощи вызова `getsid`:

Описание

```
uses stdio;

function getsid(pid:longint):longint;
```

Если передать вызову `getsid` значение 0, то он вернет идентификатор сеанса вызывающего процесса, в противном случае возвращается идентификатор сеанса процесса, заданного идентификатором `pid`.

Понятие сеанса полезно при работе с фоновыми процессами или *процессами-демонами* (daemon processes). Процессом-демоном называется просто процесс, не имеющий управляющего терминала. Примером такого процесса является процесс `cron`, запускающий команды в заданное время. Демон может задать для себя сеанс без управляющего терминала, переместившись в другой сеанс при помощи системного вызова `setsid`.

Описание

```
uses stdio;

function setsid:longint;cdecl;
```

Если вызывающий процесс не является лидером группы процессов, то создается новая группа процессов и новый сеанс, и идентификатор вызывающего процесса станет идентификатором созданного сеанса. Он также не будет иметь управляющего терминала. Процесс-демон теперь будет находиться в странном состоянии, так как он будет единственным процессом в группе процессов, содержащейся в новом сеансе, а его идентификатор процесса `pid` – также идентификатором группы и сеанса.

Функция `setsid` может завершиться неудачей и возвратит значение -1, если вызывающий процесс уже является лидером группы.

5.10.5. Переменные программного окружения

Программное *окружение* (environment) процесса – это просто набор строк, заканчивающихся нулевым символом, представленных в программе просто в виде массива указателей на строки. Эти строки называются *переменными окружения* (environment variables). По принятому соглашению, каждая строка окружения имеет следующую форму:

имя переменной = ее содержание

В модуле `dos` определены функция `envcount`, возвращающие количество строк окружения, и функция `envstr`, возвращаю строку окружения с заданным номером. Можно напрямую использовать программное окружение процесса с помощью массива `envp` из модуля `syslinux`.

В качестве простого примера следующая программа просто выводит свое окружение и завершает работу.

а) с использованием модуля `dos`:

```
(* Программа showmyenv.pas - вывод окружения *)
uses dos;

var
  i:integer;
begin
  for i:=1 to envcount do
    writeln(envstr(i));
end.
```

б) с использованием модуля `syslinux`:

```
(* Программа showmyenv.pas - вывод окружения *)
uses syslinux;

var
  i:integer;
begin
  i:=0;
  while envp[i]<>nil do
  begin
    writeln(envp[i]);
    inc(i);
  end;
end.
```

При запуске этой программы на одной из машин были получены следующие результаты:

```
CDPATH=:..:/
HOME=/usr/keith
LOGNAME=keith
MORE=-h -s
PATH=/bin:/etc:/usr/bin:/usr/cbin:/usr/lbin
SHELL=/bin/ksh
TERM=vti00
TZ=GMTOBST
```

Этот список может быть вам знакомым. Это окружение процесса командного интерпретатора (оболочки), вызвавшего программу `showmyenv`, и оно включает такие важные переменные, как `HOME` и `PATH`.

Пример показывает, что по умолчанию окружение процесса совпадает с окружением процесса, создавшего его при помощи вызова `fork` или `exec`. Поскольку окружение передается указанным способом, то можно записывать в окружении информацию, которую иначе пришлось бы задавать заново для каждого нового процесса. Переменная окружения `TERM`, в которой записан тип текущего терминала, является хорошим примером того, насколько полезным может быть использование окружения.

Чтобы задать для процесса новое окружение, необходимо использовать для его запуска один из двух вызовов из семейства `exec`: `execle` или `execve`. Они вызываются следующим образом:

```
execle(path, envp);
```

и:

```
execve(path, argv, envp);
```

Эти вызовы дублируют соответственно вызовы `execv` и `execl`. Единственное различие заключается в добавлении параметра `envp`, который является заканчивающимся нулевым символом массивом строк, задающим окружение новой программы. Следующий пример использует вызов `execle` для передачи нового программного окружения программе `showmyenv`:

```
(* Программа setmyenv.pas - установка окружения программы *)
uses linux,stdio;

const
  argv:array [0..1] of pchar=('showmyenv',nil);
  envp:array [0..2] of pchar=('foo=bar','bar=foo',nil);
begin
  execve ('./showmyen', argv, envp);
  perror ('Ошибка вызова execve');
end.
```

Для поиска в переменной `envp` имени переменной окружения, заданной в форме `name=string`, можно использовать стандартную библиотечную функцию `getenv`.

Описание

```
uses dos;

Function GetEnv(EnvVar:String):String;
```

Аргументом функции `getenv` является имя искомой переменной. При успешном завершении поиска функция `getenv` возвращает указатель на строку переменной окружения, в противном случае – пустая строка. Следующий код показывает пример использования этой функции:

```
(* Найти значение переменной окружения PATH *)

uses dos;

begin
  writeln('PATH=',getenv('PATH'));
end.
```

Для изменения окружения существует парная процедура `putenv`. Она вызывается следующим образом:

```
putenv('НОВАЯ_ПЕРЕМЕННАЯ = значение');
```

В случае успеха процедура `putenv` возвращает нулевое значение. Она изменяет переменную окружения, на которую указывает глобальная переменная `envp`.

5.10.6. Текущий рабочий каталог

Как было установлено в главе 4, с каждым процессом связан текущий рабочий каталог. Первоначально текущий рабочий каталог наследуется во время создавшего процесс вызова `fork` или `exec`. Другими словами, процесс первоначально помещается в тот же каталог, что и родительский процесс.

Важным фактом является то, что текущий рабочий каталог является атрибутом отдельного процесса. Если дочерний процесс меняет каталог при помощи вызова `chdir` (определение которого приведено в главе 4), то текущий рабочий каталог родительского процесса не меняется. Поэтому стандартная команда `cd` на самом деле является «встроенной» командой оболочки, а не программой.

5.10.7. Текущий корневой каталог

С каждым процессом также связан корневой каталог, который используется при поиске абсолютного пути. Так же, как и текущий рабочий каталог, корневым каталогом процесса первоначально является корневой каталог его родительского процесса. Для изменения начала иерархии файловой системы для процесса в ОС UNIX существует системный вызов `chroot`:

Описание

```
uses stdio;

function chroot(path:pchar):longint;
```

Переменная `path` указывает на путь, обозначающий каталог. В случае успешного вызова `chroot` путь `path` становится начальной точкой при поиске в путях, начинающихся с символа / (только для вызывающего процесса, корневой каталог системы при этом не меняется). В случае неудачи вызов `chroot` не меняет корневой каталог и возвращает значение -1. Для изменения корневого каталога вызывающий процесс должен иметь соответствующие права доступа.

Упражнение 5.12. *Добавьте к командному интерпретатору* `smallsh` *команду* `cd`.

Упражнение 5.13. *Напишите собственную версию функции* `getenv`.

5.10.8. Идентификаторы пользователя и группы

С каждым процессом связаны истинные идентификаторы пользователя и группы. Это всегда идентификатор пользователя и текущий идентификатор группы запустившего процесс пользователя.

Действующие идентификаторы пользователя и группы используются для определения возможности доступа процесса к файлу. Чаще всего, эти идентификаторы совпадают с истинными идентификаторами пользователя и группы. Равенство нарушается, если процесс или один из его предков имеет установленные биты доступа set-user-id или set-group-id. Например, если для файла программы установлен бит set-user-id, то при запуске программы на выполнение при помощи вызова `exec` действующим идентификатором пользователя становится идентификатор владельца файла, а не запустившего процесс пользователя.

Для получения связанных с процессом идентификаторов пользователя и группы существует несколько системных вызовов. Следующий фрагмент программы демонстрирует их:

```
uses linux;

var
  uid, euid, gid, egid : longint;
begin
  (* Получить истинный идентификатор пользователя *)
  uid := getuid;
  (* Получить действующий идентификатор пользователя *)
  euid := geteuid;
  (* Получить истинный идентификатор группы *)
  gid := getgid;
  (* Получить действующий идентификатор группы *)
  egid := getegid;
end.
```

Для задания действующих идентификаторов пользователя и группы процесса также существуют два вызова:

```
uses stdio;

var
  newuid, newgid:longint;
.
.
.
(* Задать действующий идентификатор пользователя *)
status := setuid(newuid);

(* Задать действующий идентификатор группы *)
status := setgid(newgid);
```

Процесс, запущенный непривилегированным пользователем (то есть любым пользователем, кроме суперпользователя) может менять действующие идентификаторы пользователя и группы только на истинные.[1] Суперпользователю, как всегда, предоставляется полная свобода. Обе процедуры возвращают нулевое значение в случае успеха, и -1 – в случае неудачи.

Упражнение 5.14. *Напишите процедуру, которая получает истинные идентификаторы пользователя и группы вызывающего процесса, а затем преобразует их в символьную форму и записывает в лог-файл.*

[1] Современные системы, согласно спецификации SUSV2 и стандарту POSIX, должны также позволять возвращаться от истинных идентификаторов к сохраненным (*saved-set-uid*, *saved-set-gid*) действующим идентификаторам пользователя и группы.

5.10.9. Ограничения на размер файла: вызов ulimit

Для каждого процесса существует ограничение на размер файла, который может быть создан при помощи системного вызова fdwrite. Это ограничение распространяется также на ситуацию, когда увеличивается размер существующего файла, имевшего до этого длину, меньшую максимально возможной.

Предельный размер файла можно изменять при помощи системного вызова ulimit.

Описание

```
uses stdio;

function ulimit(cmd:longint;args:array of const):longint;
```

Для получения текущего максимального размера файла можно вызвать ulimit, установив значение параметра cmd равным UL_GETFSIZE. Возвращаемое значение равно числу 512-байтных блоков.

Для изменения максимального размера файла можно присвоить переменной cmd значение UL_SETFSIZE и поместить новое значение максимального размера файла, также в 512-байтных блоках, в переменную newlimit, например:

```
if ulimit(UL_SETFSIZE, newlimit) < 0 then
  perror('Ошибка вызова ulimit');
```

В действительности увеличить максимальный размер файла таким способом может только суперпользователь. Процессы с идентификаторами других пользователей могут только уменьшать этот предел.

5.10.10. Приоритеты процессов

Система приближенно вычисляет долю процессорного времени, отводимую для работы процесса, руководствуясь значением nice (буквально «дружелюбный»). Значения nice находятся в диапазоне от нуля до максимального значения, зависящего от конкретной системы. Чем больше это число, тем ниже приоритет процесса. Процессы, «дружелюбно настроенные», могут понижать свой приоритет, используя системный вызов nice. Этот вызов имеет один аргумент, положительное число, на которое увеличивается текущее значение nice, например:

```
nice(5);
```

Процессы суперпользователя (и только суперпользователя) могут увеличивав свой приоритет, используя отрицательное значение параметра вызова nice. Вызов nice может пригодиться, если есть необходимость, например, вычислить число π с точностью до ста миллионов знаков, не изменяя существенно время реакции системы для остальных пользователей. Вызов nice был введен давно. Современные факультативные расширения реального времени стандарта POSIX определяют гораздо более точное управление планированием параллельной работы процессов.

Описание

```
uses linux;

Function GetPriority(Which,Who:Integer):Integer;
Function SetPriority(Which,Who,Prio:Integer):Integer;
```

GetPriority возвращает приоритет процесса, определяемых переменными Which и Who. Which может принимать значения Prio_Process, Prio_PGrp и Prio_User для идентификаторов процесса, его группы и владельца соответственно.

SetPriority устанавливает приоритет процесса. Значение Prio может быть в диапазоне от -20 до 20.

Программа, демонстрирующая функции Nice и Get/SetPriority:

```
Uses linux;

begin
```

```
  writeln ('Setting priority to 5');
  setpriority (prio_process,getpid,5);
  writeln ('New priority = ',getpriority (prio_process,getpid));
  writeln ('Doing nice 10');
  nice (10);
  writeln ('New Priority = ',getpriority (prio_process,getpid));
end.
```

Глава 6. Сигналы и их обработка

6.1. Введение

Часто требуется создавать программные системы, состоящие не из одной монолитной программы, а из нескольких сотрудничающих процессов. Для этого может быть множество причин: одиночная программа может оказаться слишком громоздкой и не поместиться в памяти; часть требуемых функций часто уже реализована в каких-либо существующих программах; задачу может быть удобнее решать при помощи процесса-сервера, взаимодействующего с произвольным числом процессов-клиентов; есть возможность использовать несколько процессоров и т.д.

К счастью, ОС UNIX имеет развитые механизмы межпроцессного взаимодействия. В этой и следующей главах мы обсудим три наиболее популярных средства: *сигналы* (signals), *программные каналы* (pipes) и *именованные каналы* (FIFO). Данные средства, наряду с более сложными средствами, которым будут посвящены главы 8 и 10, предоставляют разработчику программного обеспечения широкий выбор средств построения многопроцессных систем.

Эта глава будет посвящена изучению первого средства – сигналам. Рассмотрим пример запуска команды UNIX, выполнение которой, вероятно, займет много времени:

```
$ fpc verybigprog.pas
```

Позже становится ясным, что программа содержит ошибку, и ее компиляция не может завершиться успехом. Тогда, чтобы сэкономить время, следует прекратить выполнение команды нажатием специальной *клавиши прерывания задания* (interrupt key) терминала; обычно это клавиша **Del** или клавиатурная комбинация **Ctrl+C**. Выполнение программы прекратится, и программист вернется к приглашению ввода команды командного интерпретатора.

В действительности при этом происходит следующая последовательность событий: часть ядра, отвечающая за ввод с клавиатуры, распознает символ прерывания задания. Затем ядро посылает сигнал `SIGINT` всем процессам, для которых текущий терминал является управляющим терминалом. Среди этих процессов будет и экземпляр компилятора `cc`. Когда процесс компилятора ее получит этот сигнал, он выполнит связанное с сигналом `SIGINT` действие по умолчанию – завершит работу. Интересно отметить, что сигнал `SIGINT` посылается и процессу оболочки, тоже связанному с терминалом. Тем не менее процесс оболочки благоразумно игнорирует этот сигнал, поскольку он должен продолжать работу для интерпретации последующих команд. Как будет рассмотрено далее, пользовательские программы также могут выбирать, нужно ли им перехватывать сигнал `SIGINT`, есть выполнять специальную процедуру реакции на поступление сигнала, и полагаться на действие по умолчанию для данного сигнала.

Сигналы также используются ядром для обработки определенных типов критических ошибок. Например, предположим, что файл программы поврежден и содержит недопустимые машинные инструкции. При выполнении этой программы процессом, ядро определит попытку выполнения недопустимой инструкции и пошлет процессу сигнал `SIGILL` (здесь `ILL` является сокращением от illegal, то есть недопустимый) для завершения его работы. Получившийся диалог может выглядеть примерно так:

```
$ badprog
illegal instruction - core dumped
```

Смысл термина *core dumped* (сброс образа памяти) будет объяснен ниже.

Сигналы могут посылаться не только от ядра к процессу, но и между процессами. Проще всего показать это на примере команды `kill`. Предположим, например, что программист запускает в фоновом режиме длительную команду

```
$ fpc verybigprog.pas &
[1] 1098
```

а затем решает завершить ее работу. Тогда, чтобы послать процессу сигнал `SIGTERM`, можно использовать команду `kill`. Так же, как и сигнал `SIGINT`, сигнал `SIGTERM` завершит процесс,

если в процессе не переопределена стандартная реакция на этот сигнал. В качестве аргумента команды `kill` должен быть задан идентификатор процесса:

```
$ kill 1098
Terminated
```

Сигналы обеспечивают простой метод программного прерывания работы процессов UNIX. Образно можно сравнить его с похлопыванием по плечу, отвлекающим от работы. Из-за своей природы сигналы обычно используются для обработки исключительных ситуаций, а не для обмена данными между процессами.

Процесс может выполнять три действия с сигналами, а именно:

- изменять свою реакцию на поступление определенного сигнала (изменять обработку сигналов);
- блокировать сигналы (то есть откладывать их обработку) на время выполнения определенных критических участков кода;
- посылать сигналы другим процессам.

Каждое из этих действий будет рассмотрено далее в этой главе.

6.1.1. Имена сигналов

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее каждому типу сигналов присвоено мнемоническое имя (например, `SIGINT`), которое указывает, для чего обычно используется сигнал этого типа. Имена сигналов определены в модуле `linux` при помощи директивы `const`. Как и следовало ожидать, эти имена соответствуют небольшим положительным целым числам. Например, сигнал `SIGINT` обычно определяется так:

```
const   SIGINT  =  2; (* прерывание (rubout) *)
```

Большинство типов сигналов UNIX предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу. Ниже приведен описанный в спецификации XSI полный список стандартных сигналов и их значение. Для удобства список сигналов отсортирован в алфавитном порядке. При первом чтении этот список может быть пропущен.

- `SIGABRT` – *сигнал прерывания процесса* (process abort signal). Посылается процессу при вызове им функции abort. В результате сигнала `SIGABRT` произойдет то, что спецификация XSI описывает как *аварийное завершение* (abnormal termination), *авост*. Следствием этого в реализациях UNIX является *сброс образа памяти* (core dump, иногда переводится как «дамп памяти») с выводом сообщения `Quit - core dumped`. Образ памяти процесса сохраняется в файле на диске для изучения с помощью отладчика;
- `SIGALRM` – *сигнал таймера* (alarm clock). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова `alarm` (или установки значения первого параметра в более редко применяющемся вызове `setitimer` равным `ITIMER_REAL`). Вызов `alarm` будет описан в разделе 6.4.2. При необходимости больше узнать о вызове `setitimer` следует обратиться к справочному руководству системы;
- `SIGBUS` – *сигнал ошибки на шине* (bus error). Этот сигнал посылается при возникновении некоторой аппаратной ошибки. Смысл ошибки на шине определяется конкретной реализацией (обычно он генерируется при попытке обращения к допустимому виртуальному адресу, для которого нет физической страницы). Данный сигнал, так же как и сигнал `SIGABRT`, вызывает аварийное завершение;
- `SIGCHLD` – *сигнал останова или завершения дочернего процесса* (child process

terminated or stopped). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему сигнал SIGCHLD. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;

- SIGCONT – *продолжение работы остановленного процесса* (continue executing if stopped). Этот сигнал управления процессом, который продолжит выполнение процесса, если он был остановлен; в противном случае процесс будет игнорировать этот сигнал. Это сигнал обратный сигналу SIGSTOP;
- SIGHUP – *сигнал освобождения линии* (hangup signal). Посылается ядром всем процессам, подключенным к *управляющему терминалу* (control terminal) при отключении терминала. (Обычно управляющий терминал группы процесса является терминалом пользователя, хотя это и не всегда так. Это понятие изучается более подробно в главе 9.) Он также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом. Это гарантирует, что если не были предприняты специальные меры, то при выходе пользователя из системы завершаются все фоновые процессы, запущенные им (подробно об этом написано в разделе 5.10);
- SIGILL – *недопустимая команда процессора* (illegal instruction). Посылается операционной системой, если процесс пытается выполнить недопустимую машинную команду. Иногда этот сигнал может возникнуть из-за того, что программа каким-либо образом повредила свой код, хотя это и маловероятно. Более вероятной представляется попытка выполнения вещественной операции, не поддерживаемой оборудованием. В результате сигнала SIGILL происходит аварийное завершение программы;
- SIGINT – *сигнал прерывания программы* (interrupt). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;
- SIGKILL – *сигнал уничтожения процесса* (kill). Это довольно специфически сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса. Иногда он также посылается системой (например, при завершении работы системы). Сигнал SIGKILL – один из двух сигналов, которые не могут игнорироваться или перехватываться (то есть обрабатываться при помощи определенной пользователем процедуры);
- SIGPIPE – *сигнал о попытке записи в канал или сокет*, для которых принимающий процесс уже завершил работу (write on a pipe or socket when recipient is terminated). Программные каналы и сокеты являются другими средствами межпроцессного взаимодействия, которые обсудим в следующих главах. Там же будет рассмотрен и сигнал SIGPIPE;
- SIGPOLL – *сигнал о возникновении одного из опрашиваемых событий* (pollable event). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода. Тем не менее более удобный способ организации слежения за состояниями некоторого множества открытых файловых дескрипторов заключается в использовании системного вызова select, который подробно описан в главе 7;
- SIGPROF – *сигнал профилирующего таймера* (profiling time expired). Как было уже упомянуто для сигнала SIGALARM, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени

выполнения процесса в пользовательском и системном режимах. Сигнал SIGPROF генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования программы. (Таймер устанавливается заданием первого параметра функции setitimer равным ITIMER_PROF.);

- SIGQUIT – *сигнал о выходе* (quit). Очень похожий на сигнал SIGINT, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. Значение клавиши выхода по умолчанию соответствует символу ASCII FS или **Ctrl-**. В отличие от SIGINT, этот сигнал приводит к аварийному завершению и сбросу образа памяти;
- SIGSEGV – *обращение к некорректному адресу памяти* (invalid memory reference). Сокращение SEGV в названии сигнала означает *нарушение границ сегментов памяти* (segmentation violation). Сигнал генерируется, если процесс пытается обратиться к неверному адресу памяти. Получение сигнала SIGSEGV приводит к аварийному завершению процесса;
- SIGSTOP – *сигнал останова* (stop executing). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал SIGKILL, нельзя проигнорировать или перехватить;
- SIGSYS – *некорректный системный вызов* (invalid system call). Посылается ядром, если процесс пытается выполнить некорректный системный вызов. Это еще один сигнал, приводящий к аварийному завершению;
- SIGTERM – *программный сигнал завершения* (software termination signal). По соглашению, используется для завершения процесса (как и следует из его названия). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведение порядка», прежде чем посылать ему сигнал SIGKILL. Команда kill по умолчанию посылает именно этот сигнал;
- SIGTRAP – *сигнал трассировочного прерывания* (trace trap). Это особый сигнал, который в сочетании с системным вызовом ptrace используется отладчиками, такими как sdb, adb, и gdb. Поскольку он предназначен для отладки, его рассмотрение в рамках данной книги не требуется. По умолчанию сигнал SIGTRAP приводит к аварийному завершению;
- SIGTSTP – *терминальный сигнал остановки* (terminal stop signal). Этот сигнал формируется при нажатии специальной клавиши останова (обычно **Ctrl+Z**). Сигнал SIGTSTP аналогичен сигналу SIGSTOP, но его можно перехватить или игнорировать;
- SIGTTIN – *сигнал о попытке ввода с терминала фоновым процессом* (background process attempting read). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается сигнал SIGTTIN. Действие сигнала по умолчанию – остановка процесса;
- SIGTTOU – *сигнал о попытке вывода на терминал фоновым процессом* (background process attempting write). Аналогичен сигналу SIGTTIN, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. И снова действие по умолчанию – остановка процесса;
- SIGURG – *сигнал о поступлении в буфер сокета срочных данных* (high bandwidth data is available at a socket). Этот сигнал сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;
- SIGUSR1 и SIGUSR2 – *пользовательские сигналы* (user defined signals 1 and 2). Так же, как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя;
- SIGVTALRM – *сигнал виртуального таймера* (virtual timer expired). Как уже упоминалось для сигналов SIGALRM и SIGPROF, каждый процесс может ими не

менее трех таймеров. Последний из этих таймеров можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме. (Таймер устанавливается заданием первого параметра функции `setitimer` равным `ITIMER_VIRTUAL`);

- `SIGXCPU` – *сигнал о превышении лимита процессорного времени* (CPU time limit exceeded). Этот сигнал посылается процессу, если суммарное процессорное время, занятое его работой, превысило установленный предел. Действие по умолчанию – аварийное завершение;
- `SIGXFSZ` – *сигнал о превышении предела на размер файла* (file size limit exceeded). Этот сигнал генерируется, если процесс превысит максимально допустимый размер файла. Действие по умолчанию – аварийное завершение.

Могут встретиться и некоторые другие сигналы, но их наличие зависит от конкретной реализации системы; его не требует спецификация XSI. Большая часть этих сигналов также используется ядром для индикации ошибок, например `SIGEMT` – *прерывание эмулятора* (emulator trap) часто обозначает отказ оборудования и зависит от конкретной реализации.[1]

6.1.2. Нормальное и аварийное завершение

Получение большинства сигналов приводит к *нормальному завершению* (normal termination) процесса. Действие сигнала при этом похоже на неожиданный вызов процессом функции `_exit`. Статус завершения, возвращаемый при этом родительскому процессу, сообщит о причине завершения дочернего процесса. В файле `stdio` определены макросы, которые позволяют родительскому процессу определить причину завершения дочернего процесса (получение сигнала и, собственно, значение сигнала. Следующий фрагмент программы демонстрирует родительский процесс, проверяющий причину завершения дочернего процесса и выводящий соответствующее сообщение:

```
uses stdio;
.
.
pid:=wait(@status);
if pid=-1 then
begin
  perror('ошибка вызова wait');
  halt(1);
end;
(* Проверка нормального завершения дочернего процесса *)
if WIFEXITED(status) then
begin
  exit_status := WEXITSTATUS(status);
  writeln('Статус завершения ', pid, ' был ', exit_status);
end;
(* Проверка, получил ли дочерний процесс сигнал *)
if WIFSIGNALED(status) then
begin
  sig_no := WTERMSIG(status);
  writeln('Сигнал номер ', sig_no, ' завершил процесс ', pid);
end;
(* Проверка остановки дочернего процесса *)
if WIFSTOPPED(status) then
```

[1] Некоторые из упомянутых сигналов могут отсутствовать в используемой системе, тогда компилятор сообщит о неизвестном мнемоническом имени. Иногда имя сигнала определено, а данный сигнал отсутствует в системе. В ряде случаев требуется определить наличие определенного сигнала на стадии выполнения программы. В этих ситуациях можно воспользоваться советом, приведенным в информативной части стандарта POSIX 1003.1: наличие поддержки сигнала сообщает вызов функции `sigaction()` с аргументами `act` и `oact`, имеющими значения NULL.

```
begin
  sig_no := WSTOPSIG(status);
  writeln('Сигнал номер ', sig_no, ' остановил процесс ', pid);
end;
```

Как уже упоминалось, сигналы SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ и SIGFPE приводят к аварийному завершению и обычно сопровождаются сбросом образа памяти на диск. Образ памяти процесса записывается в файл с именем core в текущем рабочем каталоге процесса (термин core, или *сердечник*, напоминает о временах, когда оперативная память состояла из матриц ферритовых сердечников). Файл core будет содержать значения всех переменных программы, регистров процессора и необходимую управляющую информацию ядра на момент завершения программы. Статус завершения процесса после аварийного завершения будет тем же, каким он был бы в случае нормального завершения из-за этого сигнала, только в нем будет дополнительно выставлен седьмой бит младшего байта. В большинстве современных систем UNIX определен макрос WCOREDUMP, который возвращает истинное или ложное значение в зависимости от установки этого бита в своем аргументе. Тем не менее следует учесть, что макрос WCOREDUMP не определен спецификацией XSI. С применением этого макроса предыдущий пример можно переписать так:

```
(* Проверка, получил ли дочерний процесс сигнал *)
if WIFSIGNALED(status) then
begin
  sig_no := WTERMSIG(status);
  writeln('Сигнал номер ', sig_no, ' завершил процесс ', pid);
  if WCOREDUMP(status) then
    writeln('... создан файл дампа памяти');
end;
```

Формат файла core известен отладчикам UNIX, и этот файл можно использовать для изучения состояния процесса в момент сброса образа памяти. Этим можно воспользоваться для определения точки, в которой возникает проблема.

Стоит также упомянуть функцию abort, которая не имеет аргументов:

```
abort;
```

Эта функция посылает вызвавшему ее процессу сигнал SIGABRT, вызывая его аварийное завершение, то есть сброс образа памяти. Процедура abort полезна в качестве средства отладки, так как позволяет процессу записать свое текущее состояние, если что-то происходит не так. Она также иллюстрирует тот факт, что процесс может послать сигнал самому себе.

6.2. Обработка сигналов

При получении сигнала процесс может выполнить одно из трех действий:

- выполнить действие по умолчанию. Обычно действие по умолчанию заключается в прекращении выполнения процесса. Для некоторых сигналов, например, для сигналов SIGUSR1 и SIGUSR2, действие по умолчанию заключается в игнорировании сигнала.[1] Для других сигналов, например, для сигнала SIGSTOP, действие по умолчанию заключается в остановке процесса;
- игнорировать сигнал и продолжать выполнение. В больших программах неожиданно возникающие сигналы могут привести к проблемам. Например, нет смысла позволять программе останавливаться в результате случайного нажатия на клавишу прерывания, в то время как она производит обновление важной базы данных;
- выполнить определенное пользователем действие. Программист может захотеть выполнить при выходе из программы операции по «наведению порядка» (такие

[1] Спецификация SUSV2 приводит для этих сигналов нормальное завершение в качестве действия по умолчанию; лучшими примерами являются сигналы SIGCHLD и SIGURG.

как удаление рабочих файлов), что бы ни являлось причиной этого выхода.

В старых версиях UNIX обработка сигналов была относительно простой. Здесь будут изучены современные процедуры управления механизмом сигналов, и хотя эти процедуры несколько сложнее, их использование дает вполне надежный результат, в отличие от устаревших методов обработки сигналов. Прежде чем перейти к примерам, следует сделать несколько пояснений. Начнем с определена *набора сигналов* (signal set).

6.2.1. Наборы сигналов

Наборы сигналов являются одним из основных параметров, передаваемых работающим с сигналами системным вызовам. Они просто задают список сигналов, которые необходимо передать системному вызову.

Наборы сигналов определяются при помощи типа `sigset_t`, который определен в файле `linux`. Размер типа задан так, чтобы в нем мог поместиться весь набор определенных в системе сигналов. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур `sigemptyset` и `sigfillset` соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур `sigaddset` и `sigdelset`, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание

```
uses stdio;

(* Инициализация *)
function sigemptyset(__set:psigset_t):integer;
function sigfillset(__set:psigset_t):integer;


(* Добавление и удаление сигналов *)
function sigaddset(__set:psigset_t;__signo:integer):integer;
function sigdelset(__set:psigset_t;__signo:integer):integer;
```

Процедуры `sigemptyset` и `sigfillset` имеют единственный параметр – указатель на переменную типа `sigset_t`. Вызов `sigemptyset` инициализирует набор `__set`, исключив из него все сигналы. И наоборот, вызов `sigfillset` инициализирует набор, на который указывает `__set`, включив в него все сигналы. Приложения должны вызывать `sigemptyset` или `sigfillset` хотя бы один раз для каждой переменной типа `sigset_t`.

Процедуры `sigaddset` и `sigdelset` принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, `signo`, может быть символическим именем константы, таким как `SIGINT`, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

В следующем примере создадим два набора сигналов. Первый образуется из пустого набора добавлением сигналов `SIGINT` и `SIGQUIT`. Второй набор изначально заполнен, и из него удаляется сигнал `SIGCHLD`.

```
uses stdio;

var
  mask1, mask2:sigset_t;
.
.
.
(* Создать пустой набор сигналов *)
sigemptyset(@mask1);
```

```
(* Добавить сигналы *)
sigaddset(@mask1, SIGINT);
sigaddset(@mask1, SIGQUIT);

(* Создать полный набор сигналов *)
sigfillset(@mask2);

(* Удалить сигнал *)
sigdelset(@mask2, SIGCHLD);
.
.
.
```

6.2.2. Задание обработчика сигналов: вызов sigaction

После определения списка сигналов можно задать определенный метод обработки сигнала при помощи процедуры `sigaction`.

Описание

```
uses linux;

Procedure SigAction(Signo:Integer; Var Act,OAct:PSigActionRec);
```

Как вскоре увидим, структура `sigactionrec`, в свою очередь, также содержит набор сигналов. Первый параметр `signo` задает отдельный сигнал, для которого нужно определить действие. Чтобы это действие выполнялось, процедура `sigaction` должна быть вызвана до получения сигнала типа `signo`. Значением переменной `signo` может быть любое из ранее определенных имен сигналов, за исключений `SIGSTOP` и `SIGKILL`, которые предназначены только для остановки или завершения процесса и не могут обрабатываться по-другому.

Второй параметр, `act`, определяет обработчика сигнала `signo`. Третий параметр, `oact`, если не равен `nil`, указывает на структуру, куда будет помещено описание старого метода обработки сигнала. Рассмотрим структуру `sigactionrec`, определенную в файле `linux`:

```
SigActionRec  =  packed  record
  Handler    :  record
     case  byte  of
        0:  (Sh:  SignalHandler); (* Функция обработчика *)
        1:  (Sa:  TSigAction);
     end;
  Sa_Mask      :  SigSet;  (* Сигналы, которые блокируются
                              во время обработки сигнала *)
  Sa_Flags     :  Longint; (* Флаги, влияющие
                              на поведение сигнала *)
  Sa_restorer  :  SignalRestorer;  {  Obsolete  -  Don't  use  }
end;
```

Эта структура кажется сложной, но давайте рассмотрим ее поля по отдельности. Первое поле, `handler`, задает обработчик сигнала `signo`. Это поле может иметь три вида значений:

- `SIG_DFL` – константа, сообщающая, что нужно восстановить обработку сигнала по умолчанию (для большинства сигналов это завершение процесса);
- `SIG_IGN` – константа, означающая, что нужно *игнорировать данный сигнал* (ignore the signal). Не может использоваться для сигналов `SIGSTOP` и `SIGKILL`;
- адрес функции, принимающей аргумент типа `int`. Если функция объявлена в тексте программы до заполнения `sigaction`, то полю `handler.sh/handler.sa` можно просто присвоить имя функции. Компилятор поймет, что имелся в виду ее адрес. Эта функция будет выполняться при получении сигнала `signo`, а само значение `signo` будет передано в качестве аргумента вызываемой функции. Управление будет передано функции, как только процесс получит сигнал, какой

бы участок программы при этом не выполнялся. После возврата из функции управление будет снова передано процессу и продолжится с точки, в которой выполнение процесса было прервано. Этот механизм станет понятен из следующего примера.

Функция-обработчик может быть одного из двух типов:

```
type
  TSigAction = procedure(Sig: Longint; SigContext: SigContextRec);cdecl;
  SignalHandler = Procedure (Sig: Integer);cdecl;
```

Второе поле, `sa_mask`, демонстрирует первое практическое использование набора сигналов. Сигналы, заданные в поле `sa_mask`, будут блокироваться во время выполнения функции, заданной полем `handler`. Это не означает, что эти сигналы будут игнорироваться, просто их обработка будет отложена до завершения функции. При входе в функцию перехваченный сигнал также будет неявно добавлен к текущей маске сигналов. Все это делает механизм сигналов более надежным механизмом межпроцессного взаимодействия, чем он был в ранних версиях системы UNIX.[1]

Тип поля `sa_mask`, `sigset`, ограниченно совместим с типом `sigset_t`. Размер типа `sigset_t` определяется константой

```
_SIGSET_NWORDS=1024 div (8 * sizeof (longint));
```

Это в четыре раза больше размера типа `sigset`, поэтому при переходе к нему необходимо использовать поле `__val[0]` типа `sigset_t`. К примеру,

```
sa_mask:=mask.__val[0];
```

Поле `sa_flags` может использоваться для изменения характера реакции на сигнал `signo`. Например, после возврата из обработчика можно вернуть обработчик по умолчанию `SIG_DFL`, установив значение поля `sa_flags` равным `SA_RESETHAND`. Если же значение поля `sa_flags` установлено равным `SA_SIGINFO`, то обработчику сигнала будет передаваться дополнительная информация.

Все это достаточно трудно усвоить, поэтому рассмотрим несколько примеров. В действительности все намного проще, чем кажется.

Пример 1: перехват сигнала SIGINT

Этот пример демонстрирует, как можно перехватить сигнал, а также проясняет лежащий в его основе механизм сигналов. Программа `sigex` просто связывает с сигналом `SIGINT` функцию `catchint`, а затем выполняет набор операторов `sleep` и `writeln`. В данном примере определена структура `act` типа `sigactionrec` как глобальная, поэтому при инициализации структуры все поля, и в частности поле `sa_flags` обнуляются.

```
(* Программа sigex - демонстрирует работу sigaction *)
uses linux,stdio;

(* Простая функция для обработки сигнала SIGINT *)
procedure catchint (signo:integer);cdecl;
begin
  writeln (#$a'сигнал CATCHINT: signo=', signo);
  writeln ('сигнал CATCHINT: возврат'#$a);
end;


var
  act,oact:sigactionrec;
  mask:sigset_t;

begin
```

[1] Тем не менее при написании сложных систем следует знать некоторые дополнительные детали механизма доставки и обработки сигналов (см. стандарт POSIX 1003.1, спецификацию SUSV2, и руководство используемой программистом системы).

```
  (* Определение процедуры обработчика сигнала catchint *)

  (* Задание действия при получении сигнала SIGINT *)
  act.handler.sh := @catchint;
  (* Создать маску, включающую все сигналы *)
  sigfillset (@mask);
  act.sa_mask:=mask.__val[0];
  (* До вызова процедуры sigaction сигнал SIGINT
   * приводил к завершению процесса (действие по умолчанию)
   *)
  sigaction (SIGINT, @act, @oact);
  (* При получении сигнала SIGINT управление
   * будет передаваться процедуре catchint
   *)
  writeln ('вызов sleep номер 1');
  sleep (1);
  writeln ('вызов sleep номер 2');
  sleep (1);
  writeln ('вызов sleep номер 3');
  sleep (1);
  writeln ('вызов sleep номер 4');
  sleep (1);
  writeln ('Выход');
  halt(0);
end.
```

Сеанс обычного запуска `sigex` будет выглядеть так:

```
$ sigex
Вызов sleep номер 1
Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход
```

Пользователь может прервать выполнение программы `sigex`, нажав клавишу прерывания задания. Если она была нажата до того, как в программе `sigex` была выполнена процедура `sigaction`, то процесс просто завершит работу. Если же нажать да клавишу прерывания после вызова `sigaction`, то управление будет передано функции `catchint`:

```
$ sigex
Вызов sleep номер 1
<прерывание> (пользователь нажимает на клавишу прерывания)

Сигнал CATCHINT: signo = 2
Сигнал CATCHINT: возврат

Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход
```

Обратите внимание на то, как передается управление из тела программы процедуре `catchint`. После завершения процедуры `catchint`, управление продолжится с точки, в которой программа была прервана. Можно попробовать прервать программу `sigex` и в другом месте:

```
$ sigex
Вызов sleep номер 1
Вызов sleep номер 2
<прерывание> (пользователь нажимает на клавишу прерывания)

Сигнал CATCHINT: signo = 2
```

```
Сигнал GATCHINT: возврат

Вызов sleep номер 3
Вызов sleep номер 4
Выход
```

Пример 2: игнорирование сигнала SIGINT

Для того чтобы процесс игнорировал сигнал прерывания `SIGINT`, все, что нужно для этого сделать – это заменить следующую строку в программе:

```
act.handler.sh := @catchint;
```

на

```
act.handler.sh := SIG_IGN;
```

После выполнения этого оператора нажатие клавиши прерывания будет безрезультатным. Снова разрешить прерывание можно так:

```
act.handler.sh := SIG_DFL;
sigaction (SIGINT, @act, nil);
```

Можно игнорировать сразу несколько сигналов, например:

```
act.handler.sh := SIG_IGN;
sigaction(SIGINT, @act, nil);
```

При этом игнорируются оба сигнала `SIGINT` и `SIGQUIT`. Это может быть полезно в программах, которые не должны прерываться с клавиатуры.

Некоторые командные интерпретаторы используют этот подход, чтобы гарантировать, что фоновые процессы не останавливаются при нажатии пользователем клавиши прерывания. Это возможно вследствие того, что игнорируемые процессом сигналы будут игнорироваться и после вызова `exec`. Поэтому командный интерпретатор может вызвать `sigaction` для игнорирования сигналов `SIGQUIT` и `SIGINT`, а затем запустить новую программу при помощи вызова `exec`.

Пример 3: восстановление прежнего действия

Как упоминалось выше, в структуре `sigaction` может быть заполнен третий параметр `oact`. Это позволяет сохранять и восстанавливать прежнее состояние обработчика сигнала, как показано в следующем примере:

```
uses linux;

var
  act, oact: sigactionrec;

(* Сохранить старый обработчик сигнала SIGTERM *)
sigaction(SIGTERM, nil, @oact);

(* Определить новый обработчик сигнала SIGTERM *)
act.handler.sh := SIG_IGN;
sigaction(SIGTERM, @act, nil);

(* Выполнить какие-либо действия *)

(* Восстановить старый обработчик *)
sigaction(SIGTERM, @oact, nil);
```

Пример 4: аккуратный выход

Предположим, что программа использует временный рабочий файл. Следующая простая процедура удаляет файл:

```
(* Аккуратный выход из программы *)
uses linux;

procedure g_exit(s:integer);cdecl;
begin
```

```
  unlink ('tempfile');
  writeln (stderr, 'Прерывание - выход из программы');
  halt(1);
end;
```

Можно связать эту процедуру с определенным сигналом:

```
var
  act:sigactionrec;
.
.
  act.handler.sh := @g_exit;
  sigaction (SIGINT, @act, nil);
```

Если после этого вызова пользователь нажмет клавишу прерывания, то управление будет автоматически передано процедуре `g_exit`. Можно дополнить процедуру `g_exit` другими необходимыми для завершения операциями.

6.2.3. Сигналы и системные вызовы

В большинстве случаев, если процессу посылается сигнал во время выполнения им системного вызова, то обработка сигнала откладывается до завершения вызова. Но некоторые системные вызовы ведут себя по-другому, и их выполнение можно прервать при помощи сигнала. Это относится к вызовам ввода/вывода (`fdread`, `fdwrite`, `fdopen`, и т.д.), вызовам `wait` или `pause` (который мы обсудим в свое время). Во всех случаях, если процесс перехватывает вызов, то прерванный системный вызов возвращает значение –1 и помещает в переменную `linuxerror` значение `Sys_EINTR`. Такие ситуации можно обрабатывать при помощи следующего кода:

```
if fdwrite(tfd, buf, size) < 0 then
begin
  if linuxerror = Sys_EINTR then
  begin
    warn('Вызов fdwrite прерван');
    .
    .
    .
  end;
end;
```

В этом случае, если программа хочет вернуться к системному вызову `fdwrite`, то она должна использовать цикл и оператор `continue`. Но процедура `sigactionrec` позволяет автоматически повторять прерванный таким образом системный вызов. Это достигается установкой значения `SA_RESTART` в поле `sa_flags` структуры `sigactionrec`. Если установлен этот флаг, то системный вызов будет выполнен снова, и значение переменной `linuxerror` не будет установлено.

Важно отметить, что сигналы UNIX обычно не могут накапливаться. Другими словами, в любой момент времени только один сигнал каждого типа может ожидать обработки данным процессом, хотя несколько сигналов разных типов могут ожидать обработки одновременно. Фактически то, что сигналы не могут накапливаться, означает, что они не могут использоваться в качестве полностью надежного метода межпроцессного взаимодействия, так как процесс не может быть уверен, что посланный им сигнал не будет «потерян».

Упражнение 6.1. *Измените программу* `smallsh` *из предыдущей главы так, чтобы она обрабатывала клавиши прерывания и завершения как настоящий командный интерпретатор. Выполнение фоновых процессов не должно прерываться сигналами* `SIGINT` *и* `SIGQUIT`. *Некоторые командные интерпретаторы, (а именно* C-shell *и* Korn shell*) помещают фоновые процессы в другую группу процессов. В чем преимущества и недостатки этого подхода? (В последних версиях стандарта POSIX введено накопление сигналов, но в качестве необязательного расширения.)*

6.2.4. Процедуры sigsetjmp и siglongjmp

Иногда при получении сигнала необходимо вернуться на предыдущую позицию в программе. Например, может потребоваться, чтобы пользователь мог вернуться в основное меню программы при нажатии клавиши прерывания. Это можно выполнить, используя две специальные функции `sigsetjmp` и `siglongjmp`. (Существуют альтернативные функции `setjmp` и `longjmp`, но их нельзя использовать совместно с обработкой сигналов.) Процедура `sigsetjmp` «сохраняет» текущие значения счетчика команд, позиции стека, регистров процессора и маски сигналов, а процедура `siglongjmp` передает управление назад в сохраненное таким образом положение. В этом смысле процедура `siglongjmp` аналогична оператору `goto`. Важно понимать, что возврат из процедуры `siglongjmp` не происходит, так как стек возвращается в сохраненное состояние. Как будет показано ниже, при этом происходит выход из соответствующего вызова `sigsetjmp`.

Описание

```
uses stdio;

(* Сохранить текущее положение в программе *)
function sigsetjmp(var env:jmp_buf;savemask:longint):integer;

(* Вернуться в сохраненную позицию *)
procedure siglongjmp(var env:jmp_buf;val:integer);
```

Текущее состояние программы сохраняется в объекте типа `sigjmp_buf`, определенном в файле `stdio`. Если во время вызова `sigsetjmp` значение аргумента `savemask` не равно нулю, то вызов `sigsetjmp` сохранит помимо основного контекста программы также текущую маску сигналов (маску блокированных сигналов и действия, связанные со всеми сигналами). Возвращаемое функцией `sigsetjmp` значение является важным: если в точку `sigsetjmp` управление переходит из функции `siglongjmp`, то она возвращает ненулевое значение – аргумент `val` вызова `siglongjmp`. Если же функция `sigsetjmp` вызывается в обычном порядке исполнения программы, то она возвращает значение 0.

Следующий пример демонстрирует технику использования этих функций:

```
 (* Пример использования процедур sigsetjmp и siglongjmp *)
uses linux,stdio;

var
  position:sigjmp_buf;

procedure domenu;
var
  choice:integer;
begin
  write('Choice menu entry:'#$a' menu 1'#$a' menu 2'#$a' menu 3'#$a'?>');
  scanf('%d',[@choice]);
end;

procedure goback(smth:longint);cdecl;
begin
  fprintf (stderr, #$a'Прерывание'#$a, []);
  (* Вернуться в сохраненную позицию *)
  siglongjmp (position, 1);
end;

var
  act:sigactionrec;

begin
```

```
  (*
     .
     .
     . *)
  (* Сохранить текущее положение *)
  if sigsetjmp(position, 1) = 0 then
  begin
    act.handler.sh := @goback;
    sigaction (SIGINT, @act, nil);
  end;
  domenu;
  (*
   .
   .
   . *)
end.
```

Если пользователь нажимает на клавишу прерывания задания после вызова `sigaction`, то управление передается в точку, положение которой было записано при помощи функции `sigsetjmp`. Поэтому выполнение программы продолжается, как если бы только что завершился соответствующий вызов `sigsetjmp`. В этом случае возвращаемое функцией `sigsetjmp` значение будет равно второму параметру процедуры `siglongjmp`.

6.3. Блокирование сигналов

Если программа выполняет важную задачу, такую как обновление базы данных, то может понадобиться ее защита от прерывания на время выполнения таких критических действий. Как уже упоминалось, вместо игнорирования поступающих сигналов процесс может блокировать сигналы, это будет означать, что их выполнение будет отложено до тех пор, пока процесс не завершит выполнение критического участка.

Блокировать определенные сигналы в процессе позволяет системный вызов `sigprocmask`, определенный следующим образом:

Описание

```
uses linux;

Procedure SigProcMask(How:Integer; SSet,OSSet:PSigSet);
```

Параметр `how` сообщает вызову `sigpromask`, какое действие он должен выполнять. Например, этот параметр может иметь значение `SIG_MASK`, указывающее, что с этого момента будут блокироваться сигналы, заданные во втором параметр `sset`, то есть будет произведена установка маски блокирования сигналов. Третий параметр просто заполняется текущей маской блокируемых сигналов – если не нужно ее знать, просто присвойте этому параметру значение `nil`. Поясним это на примере:

```
var
  set1:sigset_t;

.
.
.
(* Создать полный набор сигналов *)
sigfillset (@set1);

(* Установить блокировку *)
sigprocmask (SIG_SETMASK, @set1, nil);

(* Критический участок кода .. *)

(* Отменить блокировку сигналов *)
```

```
sigprocmask (SIG_UNBLOCK, @set1, nil);
```

Обратите внимание на использование для отмены блокирования сигналов параметра SIG_UNBLOCK. Заметим, что если использовать в качестве первого параметра SIG_BLOCK вместо SIG_SETMASK, то это приведет к *добавлению* заданных в переменной set сигналов к текущему набору сигналов.

Следующий более сложный пример показывает, как сначала выполняется блокирование всех сигналов во время выполнения особенно важного участка программы, а затем, при выполнении менее критического участка, блокируются только сигналы SIGINT и SIGQUIT.

```
 (* Блокировка сигналов - демонстрирует вызов sigprocmask *)
uses linux,stdio;

var
  set1, set2:sigset_t;

begin
  (* Создать полный набор сигналов *)
  sigfillset (@set1);

  (* Создать набор сигналов, не включающий
   * сигналы SIGINT и SIGQUIT
   *)
  sigfillset (@set2);
  sigdelset (@set2, SIGINT);
  sigdelset (@set2, SIGQUIT);
  (* Некритический участок кода ... *)
  (* Установить блокировку всех сигналов *)
  sigprocmask (SIG_SETMASK, @set1, nil);
  (* Более критический участок кода ... *)
  (* Блокировка меньшего числа сигналов. *)
  sigprocmask (SIG_UNBLOCK, @set2, nil);
  (* Менее критический участок кода ... *)
  (* Отменить блокировку для всех сигналов *)
  sigprocmask (SIG_UNBLOCK, @set1, nil);
end.
```

Упражнение 6.2. *Перепишите процедуру* g_exit *в примере 4 из раздела 6.2.2 так, чтобы во время ее выполнения игнорировались сигналы* SIGINT *и* SIGQUIT.

6.4. Посылка сигналов

6.4.1. Посылка сигналов другим процессам: вызов kill

Процесс вызывает процедуру sigaction для установки реакции на поступление сигнала. Обратную операцию, посылку сигнала, выполняет системный вызов kill, описанный следующим образом:

Описание

```
uses linux;

Function Kill(Pid:Longint; Sig:Integer):Integer;
```

Первый параметр pid определяет процесс или процессы, которым посылается сигнал sig. Обычно pid является положительным числом, и в этом случае он рассматривается как идентификатор процесса. Поэтому следующий оператор

```
kill(7421, SIGTERM);
```

означает «*послать сигнал* SIGTERM *процессу с идентификатором 7421*». Так как процесс, посылающий сигнал kill, должен знать идентификатор процесса, которому предназначен сигнал, то вызов kill чаще всего используется для обмена между тесно связанными

процессами, например, родительским и дочерним. Заметим, что процесс может послать сигнал самому себе.

Существуют некоторые ограничения, связанные с правами доступа. Чтобы можно было послать сигнал процессу, действующий или истинный идентификатор пользователя посылающего процесса должен совпадать с действующим или истинным идентификатором пользователя процесса, которому сигнал адресован. Процессы суперпользователя, как обычно, могут посылать сигналы любым другим процессам. Если непривилегированный пользователь пытается послать сигнал процессу, который принадлежит другому пользователю, то вызов `kill` завершится неудачей, вернет значение –1 и поместит в переменную `linuxerror` значение `EPERM`. (Другие возможные значения ошибок в переменной `linuxerror` после неудачного вызова `kill` – это значение `Sys_ESRCH`, указывающее, что процесс с заданным идентификатором не существует, и `Sys_EINVAL`, если `sig` содержит некорректный номер сигнала.)

Параметр `pid` вызова `kill` может также принимать определенные значения, которые имеют особый смысл:

- если параметр `pid` равен нулю, то сигнал будет послан всем процессам, принадлежащим к той же группе, что и процесс, пославший сигнал, в том числе и самому процессу;
- если параметр `pid` равен –1, и действующий идентификатор пользователя является идентификатором суперпользователя, то сигнал посылается всем процессам, истинный идентификатор пользователя которых равен действующему идентификатору пользователя, пославшего сигнал процесса, снова включая и сам процесс, пославший сигнал;
- если параметр `pid` равен –1 и действующий идентификатор пользовать является идентификатором суперпользователя, то сигнал посылается всем процессам, кроме определенных системных процессов (последнее исключение относится ко всем попыткам послать сигнал группе процессов, но наиболее важно это в данном случае);
- и, наконец, если параметр `pid` меньше нуля, но не равен –1, то сигнал посылается всем процессам, идентификатор группы которых равен модулю `pid`, включая пославший сигнал процесс, если для него также выполняется это условие.

Следующий пример – программа `synchro` создает два процесса, которые будут поочередно печатать сообщения на стандартный вывод. Они синхронизирут свою работу, посылая друг другу сигнал `SIGUSR1` при помощи вызова `kill`.

```
 (* Программа synchro -- пример использования вызова kill *)
uses linux,stdio;

const
  ntimes:integer=0;

procedure p_action(sig:integer);cdecl;
begin
  inc(ntimes);
  writeln ('Родительский процесс получил сигнал ', ntimes, ' раз(а)');
end;

procedure c_action(sig:integer);cdecl;
begin
  inc(ntimes);
  writeln ('Дочерний процесс получил сигнал ', ntimes, ' раз(а)');
end;


var
```

```
  pid, ppid:longint;
  pact, cact:sigactionrec;
begin
  (* Задать обработчик сигнала SIGUSR1 в родительском процессе *)
  pact.handler.sh := @p_action;
  sigaction (SIGUSR1, @pact, nil);

  pid := fork;
  case pid of
    -1:                (* ошибка *)
    begin
      perror ('synchro');
      halt(1);
    end;
    0:                 (* дочерний процесс *)
    begin
      (* Задать обработчик в дочернем процессе *)
      cact.handler.sh := @c_action;
      sigaction (SIGUSR1, @cact, nil);
      (* Получить идентификатор родительского процесса *)
      ppid := getppid;
      while true do
      begin
        sleep (1);
        kill (ppid, SIGUSR1);
        pause;
      end;
      (* Бесконечный цикл *)
    end;
    else               (* родительский процесс *)
      while true do
      begin
        pause;
        sleep (1);
        kill (pid, SIGUSR1);
      end;
      (* Бесконечный цикл *)
  end;
end.
```

Оба процесса выполняют бесконечный цикл, приостанавливая работу до получения сигнала от другого процесса. Они используют для этого системный вызов `pause`, который просто приостанавливает работу до получения сигнала (см. раздел 6.4.3). Затем каждый из процессов выводит сообщение и, в свою очередь, посылает сигнал при помощи вызова `kill`. Дочерний процесс начинает вывод сообщений (обратите внимание на порядок операторов в каждом цикле). Оба процесса завершают работу, когда пользователь нажимает на клавишу прерывания. Диалог с программой может выглядеть примерно так:

```
$ synchro
Родительский процесс получил сигнал #1
Дочерний процесс получил сигнал #1
Родительский процесс получил сигнал #2
Дочерний процесс получил сигнал #2
< прерывание >    (пользователь нажал на клавишу прерывания)
$
```

6.4.2. Посылка сигналов самому процессу: вызовы sigraise и alarm

Функция `sigraise` просто посылает сигнал выполняющемуся процессу:

Описание

```
uses linux;

Procedure SigRaise(Sig:integer);
```

Вызывающему процессу посылается сигнал, определенный параметром `sig` и в случае успеха функция `sigraise` возвращает нулевое значение. Например:

```
uses Linux;

Var
   oa,na : PSigActionRec;

Procedure DoSig(sig : Longint);cdecl;

begin
   writeln('Receiving signal: ',sig);
end;

begin
   new(na);
   new(oa);
   na^.handler.sh:=@DoSig;
   na^.Sa_Mask:=0;
   na^.Sa_Flags:=0;
   na^.Sa_Restorer:=Nil;
   SigAction(SigUsr1,na,oa);
   if LinuxError<>0 then
     begin
     writeln('Error: ',linuxerror,'.');
     halt(1);
     end;
   Writeln('Sending USR1 (',sigusr1,') signal to self.');
   SigRaise(sigusr1);
end.
```

Вызов `alarm` – это простой и полезный вызов, который устанавливает таймер процесса. При срабатывании таймера процессу посылается сигнал.

Описание

```
uses linux;

Function Alarm(Secs:longint):Longint;
```

Переменная `secs` задает время в секундах, на которое устанавливается таймер. После истечения заданного интервала времени процессу посылается сигнал `SIGALRM`. Поэтому вызов

```
alarm(60);
```

приводит к посылке сигнала `SIGALRM` через 60 секунд. Обратите внимание, что вызов `alarm` не приостанавливает выполнение процесса, как вызов `sleep`, вместо этого сразу же происходит возврат из вызова `alarm`, и продолжается нормальное выполнение процесса, по крайней мере, до тех пор, пока не будет получен сигнал `SIGALRM`. Установленный таймер будет продолжать отсчет и после вызова `exec`, но вызов `fork` выключает таймер в дочернем процессе.

Выключить таймер можно при помощи вызова `alarm` с нулевым параметром:

```
(* Выключить таймер *)
alarm(0);
```

Вызовы `alarm` не накапливаются: другими словами, если вызвать `alarm` дважды, то второй вызов отменит предыдущий. Но при этом возвращаемое вызовом `alarm` значение будет равно времени, оставшемуся до срабатывания предыдущего таймера, и его можно при

необходимости записать.

Вызов alarm может быть полезен, если нужно ограничить время выполнения какого-либо действия. Основная идея проста: вызывается alarm, и процесс начинает выполнение задачи. Если задача выполняется вовремя, то таймер сбрасывается. Если выполнение задачи отнимает слишком много времени, то процесс прерывается при помощи сигнала SIGTERM и выполняет корректирующие действия.

Следующая функция quickreply использует этот подход для ввода данных от пользователя за заданное время. Она имеет один аргумент, приглашение командной строки, и возвращает указатель на введенную строку, или нулевой указатель, если после пяти попыток ничего не было введено. Обратите внимание, что после каждого напоминания пользователю функция quickreply посылает на терминал символ **Ctrl+G**. На большинстве терминалов и эмуляторов терминала это приводит к подаче звукового сигнала.

Функция quickreply вызывает процедуру gets из *стандартной библиотеки ввода/вывода* (Standard I/O Library). Процедура gets помещает очередную строку из стандартного ввода в массив char. Она возвращает либо указатель на массив, либо нулевой указатель в случае достижения конца файла или ошибки. Обратите внимание на то, что сигнал SIGALRM перехватывается процедурой обработчика прерывания catch. Это важно, так как по умолчанию получение сигнала SIGALRM приводит к завершению процесса. Процедура catch устанавливает флаг timed_out. Функция quickreply проверяет этот флаг, определяя таким образом, не истекло ли заданное время.

```
uses linux,stdio;

const
  TIMEOUT=5;            (* время в секундах *)
  MAXTRIES=5;           (* число попыток *)
  LINESIZE=100;         (* длина строки *)
  CTRL_G=#7;            (* ASCII символ звукового сигнала *)

var
  (* Флаг, определяющий, истекло ли заданное время *)
  timed_out:boolean;
  (* Переменная, которая будет содержать введенную строку *)
  answer:array [0..LINESIZE-1] of char;

(* Выполняется при получении сигнала SIGALRM *)
procedure catch (sig:integer);cdecl;
begin
  (* Установить флаг timed_out *)
  timed_out := TRUE;
  (* Подать звуковой сигнал *)
  write(CTRL_G);
end;


function quickreply(prompt:pchar):pchar;
var
  ntries:integer;
  act, oact:sigactionrec;
begin
  (* Перехватить сигнал SIGALRM и сохранить старый обработчик *)
  act.handler.sh := @catch;
  sigaction (SIGALRM, @act, @oact);
  for ntries:=1 to MAXTRIES do
  begin
    timed_out := FALSE;
```

```
    writeln;
    write(prompt, ' > ');
    (* Установить таймер *)
    alarm (TIMEOUT);
    (* Получить введенную строку *)
    gets (answer);
    (* Выключить таймер *)
    alarm (0);
    (* Если флаг timed_out равен TRUE, завершить работу *)
    if not timed_out then
      break;
  end;
  (* Восстановить старый обработчик *)
  sigaction (SIGALRM, @oact, nil);
  (* Вернуть соответствующее значение *)
  if ntries = MAXTRIES then
    quickreply:=nil
  else quickreply:=answer;
end;


begin
  writeln;
  writeln(quickreply ('Reply'));
end.
```

6.4.3. Системный вызов pause

ОС UNIX также содержит дополняющий вызов `alarm` системный вызов `pause`, который определен следующим образом:

Описание

```
uses linux;

Procedure Pause;
```

Вызов `pause` приостанавливает выполнение вызывающего процесса (так что процесс при этом не занимает процессорного времени) до получения любого сигнала, например, сигнала `SIGALRM`. Если сигнал вызывает нормальное завершена процесса или игнорируется процессом, то в результате вызова `pause` будет просто выполнено соответствующее действие (завершение работы или игнорирована сигнала).

Следующая программа `tml` (сокращение от «tell me later» – *напомнить позднее*) использует оба вызова `alarm` и `pause` для вывода сообщения в течение заданного числа минут. Она вызывается следующим образом:

```
$ tml число_минут текст_сообщения
```

Например:

```
$ tml 10 время идти домой
```

Перед сообщением выводятся три символа **Ctrl+G** (звуковые сигналы) для привлечения внимания пользователя. Обратите внимание на создание в программе `tml` фонового процесса при помощи вызова `fork`. Фоновый процесс выполняет работу, позволяя пользователю продолжать выполнение других задач.

```
 (* tml - программа для напоминания *)
{$mode objfpc}

uses linux, stdio, sysutils;

const
  BELLS=#7#7#7;   (* звуковой сигнал ASCII *)
```

```
  alarm_flag:boolean = FALSE;

(* Обработчик сигнала SIGALRM *)
procedure setflag(sig:integer);cdecl;
begin
  alarm_flag := TRUE;
end;

var
  nsecs, j:integer;
  pid:longint;
  act:sigactionrec;
begin
  if paramcount < 2 then
  begin
    writeln (stderr, 'Применение: tml число_минут сообщение');
    halt(1);
  end;
  try
    nsecs := strtoint(paramstr(1)) * 60;
  except
    on e:econverterror do
    begin
      writeln (stderr, 'Введено нечисловое значение');
      halt(2);
    end;
  end;
  if nsecs <= 0 then
  begin
    writeln (stderr, 'tml: недопустимое время');
    halt(3);
  end;

  (* Вызов fork, создающий фоновый процесс *)
  pid := fork;
  case pid of
    -1:                 (* ошибка *)
    begin
      perror ('tml');
      halt(1);
    end;
    0:                  (* дочерний процесс *)
      ;
    else          (* родительский процесс *)
    begin
      writeln('Процесс tml с идентификатором ', pid);
      halt(0);
    end;
  end;

  (* Установить обработчик таймера *)
  act.handler.sh := @setflag;
  sigaction (SIGALRM, @act, nil);
  (* Установить таймер *)
  alarm (nsecs);
  (* Приостановить выполнение до получения сигнала ... *)
  pause;
```

```
  (* Если был получен сигнал SIGALRM, вывести сообщение *)
  if alarm_flag then
  begin
    write(BELLS);
    for j := 2 to paramcount do
      write(paramstr(j),' ');
    writeln;
  end;
  halt(0);
end.
```

Из этого примера можно получить представление о том, как работает подпрограмма `sleep`, вызывая вначале `alarm`, а затем `pause`.

Упражнение 6.3. *Напишите свою версию подпрограммы* `sleep`. *Она должна сохранить предыдущее состояние таймера и восстанавливать его при выходе. (Посмотрите полное описание процедуры sleep в справочном руководстве системы.)*

Упражнение 6.4. *Перепишите программу* `tml`, *используя свою версию процедуры* `sleep`.

6.4.4. Системные вызовы sigpending и sigsuspend

ОС UNIX также содержит дополняющие вызов `SigProcMask` системные вызовы `SigPending` и `SigSuspend`, которые определены следующим образом:

Описание

```
uses linux;

Function SigPending:SigSet;
Procedure SigSuspend(Mask:SigSet);
```

`SigPending` позволяет узнать, какие из временно заблокированных сигналов необходимо обработать. Возвращаемое значение – маска отложенных сигналов.

`SigSuspend` временно заменяет маску сигналов для процесса на `Mask`, приостанавливая процесс до получения сигнала.

Глава 7. Межпроцессное взаимодействие при помощи программных каналов

Если два или несколько процессов совместно выполняют одну и ту же задачу, то они неизбежно должны использовать общие данные. Хотя сигналы и могут быть полезны для синхронизации процессов или для обработки исключительных ситуаций или ошибок, они совершенно не подходят для передачи данных от одного процесса к другому. Один из возможных путей разрешения этой проблемы заключается в совместном использовании файлов, так как ничто не мешает нескольким процессам одновременно выполнять операции чтения или записи для одного и того же файла. Тем не менее совместный доступ к файлам может оказаться неэффективным и потребует специальных мер предосторожности для избежания конфликтов.

Для решения этих проблем система UNIX обеспечивает конструкцию, которая называется *программными каналами* (pipe). (В следующих главах будут также изучены некоторые другие средства коммуникации процессов.) Программный канал (или просто канал) служит для установления односторонний связи, соединяющей один процесс с другим, и является еще одним видом обобщенного ввода/вывода системы UNIX. Как увидим далее, процесс может посылать данные в канал при помощи системного вызова `fdwrite`, а другой процесс может принимать данные из канала при помощи системного вызова `fdread`.

7.1. Каналы

7.1.1. Каналы на уровне команд

Большинство пользователей UNIX уже сталкивались с конвейерами команд:

```
$ pr doc | lp
```

Этот конвейер организует совместную работу команд `pr` и `lp`. Символ | в командной строке сообщает командному интерпретатору, что необходимо создать канал, соединяющий стандартный вывод команды `pr` со стандартным вводом команды `lp`. В результате этой команды на матричный принтер будет выведена разбитая на страницы версия файла `doc`.

Разобьем командную строку на составные части. Программа `pr` слева от символа, обозначающего канал, ничего не знает о том, что ее стандартный вывод посылается в канал. Она выполняет обычную запись в свой стандартный вывод, не предпринимая никаких особых мер. Аналогично программа `lp` справа выполи чтение точно так же, как если бы она получала свой стандартный ввод с клавиатуры или из обычного файла.[1] Результат в целом будет таким же, как при выполнении следующей последовательности команд:

```
$ pr doc > tmpfile
$ lp < tmpfile
$ rm tmpfile
```

Управление потоком в канале осуществляется автоматически и прозрачно для процесса. Поэтому, если программа `pr` будет выводить информацию слишком быстро, то ее выполнение будет приостановлено. После того как программа `lp` догонит программу `pr`, и количество данных, находящихся в канале, упадет до приемлемого уровня, выполнение программы `pr` продолжится.

Каналы являются одной из самых сильных и характерных особенностей ОС UNIX, доступных даже с уровня командного интерпретатора. Они позволяют легко соединять между собой произвольные последовательности команд. Поэтому программы UNIX могут разрабатываться как простые инструменты, осуществляющие чтение из стандартного ввода, запись в стандартный вывод и выполняют одну, четко определенную задачу. При помощи

[1] На самом деле программа имеет возможность с помощью операций управления ввода/вывода выяснить тип конечного устройства, используемого в качестве стандартного ввода/вывода (файл, терминал или канал). Некоторые стандартные утилиты UNIX ведут себя по-разному в разных ситуаций; сравните вывод `ls` на терминал и в канал.

каналов из этих основных блоков могут быть построены более сложные командные строки, например, команда

```
$ who | wc -l
```

направляет вывод программы who в программу подсчета числа слов wc, а задание параметра -l в программе wc определяет, что необходимо подсчитывать только число строк. Таким образом, в конечном итоге программа wc выводит число находящихся в системе пользователей (иногда нужно исключить из суммы первую строку-заголовок вывода who).

7.1.2. Использование каналов в программе

Каналы создаются в программе при помощи системного вызова AssignPipe. В случае удачного завершения вызов сообщает два дескриптора файла: один для записи в канал, а другой для чтения из него. Вызов AssignPipe определяется следующим образом:

Описание

```
uses linux;

Function AssignPipe(var pipe_in,pipe_out:longint):boolean;
Function AssignPipe(var pipe_in,pipe_out:text):boolean;
Function AssignPipe(var pipe_in,pipe_out:file):boolean;
```

Переменные pipe_in и pipe_out содержат дескрипторы файлов, обозначающие канал. После успешного вызова pipe_in будет открыт для чтения из канала, а pipe_out для записи в канал.

В случае неудачи вызов pipe вернет значение false. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная linuxerror будет содержать значение Sys_EMFILE), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная linuxerror будет содержать значение Sys_ENFILE).

После создания канала с ним можно работать просто при помощи вызовов fdread и fdwrite. Следующий пример демонстрирует это: он создает канал, записывает в него три сообщения, а затем считывает их из канала:

```
uses linux,stdio;

(* Первый пример работы с каналами *)
const
  MSGSIZE=16;

(* Эти строки заканчиваются нулевым символом *)
  msg1:array [0..MSGSIZE-1] of char = 'hello, world #1';
  msg2:array [0..MSGSIZE-1] of char = 'hello, world #2';
  msg3:array [0..MSGSIZE-1] of char = 'hello, world #3';

var
  inbuf:array [0..MSGSIZE-1] of char;
  fdr,fdw,j:longint;
begin
  (* Открыть канал *)
  if not assignpipe(fdr,fdw) then
  begin
    perror ('Ошибка вызова pipe');
    halt (1);
  end;
  (* Запись в канал *)
  fdwrite (fdw, msg1, MSGSIZE);
  fdwrite (fdw, msg2, MSGSIZE);
```

```
  fdwrite (fdw, msg3, MSGSIZE);
  (* Чтение из канала *)
  for j := 1 to 3 do
  begin
    fdread (fdr, inbuf, MSGSIZE);
    writeln(inbuf);
  end;
  halt (0);
end.
```

На выходе программы получим:

```
hello, world #1
hello, world #2
hello, world #3
```

Обратите внимание, что сообщения считываются в том же порядке, в каком они были записаны. Каналы обращаются с данными в порядке «*первый вошел – первым вышел*» (first-in first-out, или сокращенно FIFO). Другими словами, данные, которые помещаются в канал первыми, первыми и считываются на другом конце канала. Этот порядок нельзя изменить, поскольку вызов `fdseek` не работает с каналами.

Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми, хотя в нашем примере это и было так. Можно, например, писать в канал блоками по 512 байт, а затем считывать из него по одному символу, так же как и в случае обычного файла. Тем не менее, как будет показано в разделе 7.2, использование блоков фиксированного размера дает определенные преимущества.

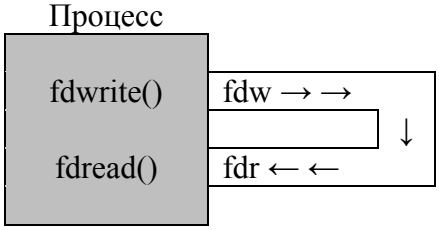


Рис. 7.1. Первый пример работы с каналами

Работа примера показана графически на рис. 7.1. Эта диаграмма позволяет более ясно представить, что процесс только посылает данные сам себе, используя канал в качестве некой разновидности механизма обратной связи. Это может показаться бессмысленным, поскольку процесс общается только сам с собой.

Настоящее значение каналов проявляется при использовании вместе с системным вызовом `fork`, тогда можно воспользоваться тем фактом, что файловые дескрипторы остаются открытыми в обоих процессах. Следующий пример демонстрирует это. Он создает канал и вызывает `fork`, затем дочерний процесс обменивается несколькими сообщениями с родительским:

```
(* Второй пример работы с каналами *)
uses linux, stdio;

const
  MSGSIZE=16;
  msg1:array [0..MSGSIZE-1] of char = 'hello, world #1';
  msg2:array [0..MSGSIZE-1] of char = 'hello, world #2';
  msg3:array [0..MSGSIZE-1] of char = 'hello, world #3';

var
  inbuf:array [0..MSGSIZE-1] of char;
  fdr,fdw,j,pid:longint;
begin
  (* Открыть канал *)
  if not assignpipe (fdr,fdw) then
```

```
  begin
    perror ('Ошибка вызова pipe ');
    halt (1);
  end;
  pid := fork;
  case pid of
    -1:
    begin
      perror ('Ошибка вызова fork');
      halt (2);
    end;
    0:
    begin
      (* Это дочерний процесс, выполнить запись в канал *)
      fdwrite (fdw, msg1, MSGSIZE);
      fdwrite (fdw, msg2, MSGSIZE);
      fdwrite (fdw, msg3, MSGSIZE);
    end;
    else
    begin
      (* Это родительский процесс, выполнить чтение из канала *)
      for j := 1 to 3 do
      begin
        fdread (fdr, inbuf, MSGSIZE);
        writeln (inbuf);
      end;
      wait(nil);
    end;
  end;
  halt (0);
end.
```

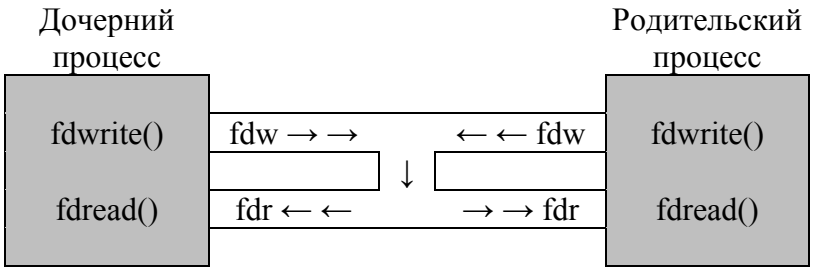


Рис. 7.2. Второй пример работы с каналами

Этот пример представлен графически на рис. 7.2. На нем показано, как канал соединяет два процесса. Здесь видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять запись в канал и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором `fdw` и чтение из файла с дескриптором `fdr`. Это создает определенную проблему. Каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать неприятностей, если посылающий данные процесс закроет дескриптор файла, открытого на запись, – раздел 7.1.4 объясняет возможные последствия. Приведенные до сих пор примеры работают только потому, что принимающий процесс в точности знает, какое количество данных он может ожидать. Следующий пример представляет собой законченное решение:

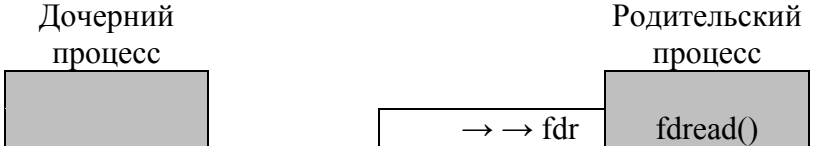
```
 (* Третий пример работы с каналами *)
```

```
uses linux,stdio;

const
  MSGSIZE=16;
  msg1:array [0..MSGSIZE-1] of char = 'hello, world #1';
  msg2:array [0..MSGSIZE-1] of char = 'hello, world #2';
  msg3:array [0..MSGSIZE-1] of char = 'hello, world #3';

var
  inbuf:array [0..MSGSIZE-1] of char;
  fdr,fdw,j,pid:longint;
begin
  (* Открыть канал *)
  if not assignpipe (fdr,fdw) then
  begin
    perror ('Ошибка вызова pipe ');
    halt (1);
  end;
  pid := fork;
  case pid of
    -1:
    begin
      perror ('Ошибка вызова fork');
      halt (2);
    end;
    0:
    begin
      (* Дочерний процесс, закрывает дескриптор файла,
       * открытого для чтения и выполняет запись в канал
       *)
      fdclose (fdr);
      fdwrite (fdw, msg1, MSGSIZE);
      fdwrite (fdw, msg2, MSGSIZE);
      fdwrite (fdw, msg3, MSGSIZE);
    end;
    else
    begin
      (* Родительский процесс, закрывает дескриптор файла,
       * открытого для записи и выполняет чтение из канала
       *)
      fdclose (fdw);
      for j := 1 to 3 do
      begin
        fdread (fdr, inbuf, MSGSIZE);
        writeln (inbuf);
      end;
      wait(nil);
    end;
  end;
  halt (0);
end.
```

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому. Эта упрощенная ситуация показана на рис. 7.3.



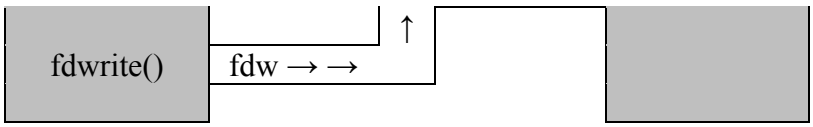


Рис. 7.3. Третий пример работы с каналами

Упражнение 7.1. *В последнем примере канал использовался для установления связи между родительским и дочерним процессами. Но дескрипторы файлов канала могут передаваться и сквозь через несколько вызовов* `fork`*. Это означает, что несколько процессов могут писать в канал и несколько процессов могут читать из него. Для демонстрации этого поведения напишите программу, которая создает три процесса, два из которых выполняют запись в канал, а один – чтение из него. Процесс, выполняющей чтение, должен выводить все получаемые им сообщения на свой стандартный вывод.*

Упражнение 7.2. *Для установления двусторонней связи между процессами можно создать два канала, работающих в разных направлениях. Придумайте возможный диалог между процессами и реализуйте его при помощи двух каналов.*

7.1.3. Размер канала

Пока в примерах передавались только небольшие объемы данных. Важно заметить, что на практике размер буфера канала конечен. Другими словами, только определенное число байтов может находиться в канале, прежде чем следующий вызов `fdwrite` будет заблокирован. Минимальный размер, определенный стандартом POSIX, равен 512 байтам. В большинстве существующих систем это значение намного больше. При программировании важно знать максимальный размер канала для системы, так как он влияет на оба вызова `fdwrite` и `fdread`. Если вызов `fdwrite` выполняется для канала, в котором есть свободное место, то данные посылаются в канал, и немедленно происходит возврат из вызова. Если же в момент вызова `fdwrite` происходит переполнение канала, то выполнение процесса обычно приостанавливается до тех пор, пока место не освободится в результате выполнения другим процессом чтения из канала.

Приведенная в качестве примера следующая программа записывает в канал символ за символом до тех пор, пока не произойдет блокировка вызова `fdwrite`. Программа использует вызов `alarm` для предотвращения слишком долгого ожидания в случае, если вызов `fdread` никогда не произойдет. Необходимо обратить внимание на использование процедуры `fpathconf`, служащей для определения максимального числа байтов, которые могут быть записаны в канал за один прием.

```
(* Запись в канал до возникновения блокировки записи *)
uses linux,stdio;

var
  count:integer;

(* Вызывается при получении сигнала SIGALRM *)
procedure alrm_action(signo:integer);cdecl;
begin
  writeln ('Запись блокируется после вывода ',count,' символов');
  halt (0);
end;

const
  c:char='x';
var
  fdin,fdout,pipe_size:longint;
  act:sigactionrec;
  temp:sigset_t;
begin
```

```
  (* Задать обработчик сигнала *)
  act.handler.sh := @alrm_action;
  sigfillset (@temp);
  act.sa_mask:=temp.__val[0];
  (* Создать канал *)
  if not assignpipe (fdin,fdout) then
  begin
    perror ('Ошибка вызова pipe ');
    halt (1);
  end;
  (* Определить размер канала *)
  pipe_size := fpathconf (fdin, _PC_PIPE_BUF);
  writeln('Максимальный размер канала: ',pipe_size,' байт');
  (* Задать обработчик сигнала *)
  sigaction (SIGALRM, @act, nil);
  while true do
  begin
    (* Установить таймер *)
    alarm (20);
    (* Запись в канал *)
    fdwrite (fdout, c, 1);
    (* Сбросить таймер *)
    alarm (0);
    inc(count);
    if count mod 1024 = 0 then
      writeln (count, ' символов в канале');
  end;
end.
```

Вот результат работы программы на некоторой системе:

```
Максимальный размер канала: 32768 байт
1024 символов в канале
2048 символов в канале
3072 символов в канале
4096 символов в канале
5120 символов в канале
.
.
.
31744 символов в канале
32768 символов в канале
Запись блокируется после вывода 32768 символов
```

Обратите внимание, насколько реальный предел больше, чем заданный стандартом POSIX минимальный размер канала.

Ситуация становится более сложной, если процесс пытается записать за один вызов `fdwrite` больше данных, чем может вместить даже полностью пустой канал. В этом случае ядро вначале попытается записать в канал максимально возможный объем данных, а затем приостанавливает выполнение процесса до тех пор, пока не освободится место под оставшиеся данные. Это важный момент: обычно вызов `fdwrite` для канала выполняется *неделимыми порциями* (atomically), и данные передаются ядром за одну непрерываемую операцию. Если делается попытка записать в канал больше данных, чем он может вместить, то вызов `fdwrite` выполняется поэтапно. Если при этом несколько процессов выполняют запись в канал, то данные могут оказаться беспорядочно перепутанными.

Взаимодействие вызова `fdread` с каналами является более простым. При выполнении вызова `fdread` система проверяет, является ли канал пустым. Если он пуст, то вызов `fdread` будет заблокирован до тех пор, пока другой процесс не запишет в канал какие-либо данные. При наличии в канале данных произойдет возврат из вызова `fdread`, даже если

запрашивается больший объем данных, чем находится в канале.

7.1.4. Закрытие каналов

Что произойдет, если дескриптор файла, соответствующий одному из концов канала, будет закрыт? Возможны два случая:

- *закрывается дескриптор файла, открытого только на запись*. Если существуют другие процессы, в которых канал открыт на запись, то ничего не произойдет. Если же больше не существует процессов, которые могли бы выполнять запись в канал, и канал при этом пуст, то любой процесс, который попытается выполнить чтение из канала, получит пустой блок данных. Процессы, которые были приостановлены и ожидали чтения из канала, продолжат свою работу, вызовы `read` вернут нулевое значение. Для процесса, выполняющего чтение, результат будет похож на достижение конца файла;
- *закрывается дескриптор файла, открытого только на чтение*. Если еще есть процессы, в которых канал открыт на чтение, то снова ничего не произойдет. Если же больше не существует процессов, выполняющих чтение из канала, то ядро посылает всем процессам, ожидающим записи в канал, сигнал `SIGPIPE`. Если этот сигнал не перехватывается в процессе, то процесс при этом завершит свою работу. Если же сигнал перехватывается, то после завершения процедуры обработчика прерывания вызов `fdwrite` вернет значение -1 и переменная `linuxerror` после этого будет содержать значение `Sys_EPIPE`. Процессам, которые будут пытаться после этого выполнить запись в канал, также будет посылаться сигнал `SIGPIPE`.

7.1.5. Запись и чтение без блокирования

Как уже было упомянуто, при использовании и вызова `fdread`, и вызова `fdwrite` может возникнуть блокирование, которое иногда нежелательно. Может, например, понадобиться, чтобы программа выполняла процедуру обработки ошибок или опрашивала несколько каналов до тех пор, пока не получит данные из одного из них. К счастью, есть простые способы пресечения нежелательных остановов внутри `fdread` и `fdwrite`.

Первый метод заключается в использовании для вызова `fstat`. Поле `size` в возвращаемой вызовом структуре `tstat` сообщает текущее число символов, находящихся в канале. Если только один процесс выполняет чтение из канала, такой подход работает прекрасно. Если же несколько процессов выполняют чтение из канала, то за время, прошедшее между вызовами `fstat` и `fdread`, ситуация может измениться, если другой процесс успеет выполнить чтение из канала.

Второй метод заключается в использовании вызова `fcntl`. Помимо других выполняемых им функций этот вызов позволяет процессу устанавливать для дескриптора файла флаг `Open_NONBLOCK`. Это предотвращает блокировку последующих вызовов `fdread` или `fdwrite`. В этом контексте вызов `fcntl` может использоваться следующим образом:

```
uses linux;
.
.
.
fcntl(filedes, F_SETFL, Open_NONBLOCK);
if linuxerror <> 0 then
  perror('fcntl');
```

Если дескриптор `filedes` является открытым только на запись, то следующие вызовы `fdwrite` не будут блокироваться при заполнении канала. Вместо этого они будут немедленно возвращать значение -1 и присваивать переменной `linuxerror` значение `Sys_EAGAIN`. Аналогично, если дескриптор `filedes` соответствует выходу канала, то процесс немедленно вернет значение -1, если в канале нет данных, а не приостановит работу. Так же, как и в случае вызова `fdwrite`, переменной `linuxerror` будет присвоено значение `Sys_EAGAIN`.

(Если установлен другой флаг – Open_NDELAY, то поведение вызова fdread будет другим. Если канал пуст, то вызов вернет нулевое значение. Далее этот случай не будет рассматриваться.)

Следующая программа иллюстрирует применение вызова fcntl. В ней создается канал, для дескриптора чтения из канала устанавливается флаг Open_NONBLOCK, а затем выполняется вызов fork. Дочерний процесс посылает сообщения родительскому, выполняющему бесконечный цикл, опрашивая канал и проверяя, поступили ли данные.

```
 (* Пример использования флага Open_NONBLOCK *)
uses linux,stdio;

const
  MSGSIZE=6;
  msg1:array [0..MSGSIZE-1] of char = 'hello';
  msg2:array [0..MSGSIZE-1] of char = 'bye!!';


function parent(fdin,fdout:integer):integer; (* код родительского процесса *)
var
  nread:longint;
  buf:array [0..MSGSIZE-1] of char;
begin
  fdclose (fdout);
  while true do
  begin
    nread := fdread (fdin, buf, MSGSIZE);
    case nread of
      -1:
      begin
        (* Проверить, есть ли данные в канале. *)
        if linuxerror = Sys_EAGAIN then
        begin
          writeln('(канал пуст)');
          sleep (1);
        end
        else
          fatal ('Ошибка вызова read');
      end;
      0:
      begin
        (* Канал был закрыт. *)
        writeln('Конец связи');
        halt (0);
      end;
      else
        writeln('MSG=', buf);
    end;
  end;
end;


function child (fdin,fdout:integer):integer;
var
  count:longint;
begin
  fdclose (fdin);
  for count := 1 to 3 do
  begin
```

```
    fdwrite (fdout, msg1, MSGSIZE);
    sleep (3);
  end;
  (* Послать последнее сообщение *)
  fdwrite (fdout, msg2, MSGSIZE);
  halt (0);
end;


var
  fdin,fdout:longint;
begin
  (* Открыть канал *)
  if not assignpipe (fdin,fdout) then
    fatal ('Ошибка вызова pipe ');
  (* Установить флаг Open_NONBLOCK для дескриптора fdin *)
  fcntl (fdin, F_SETFL, Open_NONBLOCK);
  if (linuxerror=sys_eagain) or (linuxerror=sys_eacces)   then
    fatal ('ошибка вызова fcntl');
  case fork of
    -1:           (* ошибка *)
      fatal ('ошибка вызова fork');
    0:            (* дочерний процесс *)
      child (fdin,fdout);
    else          (* родительский процесс *)
      parent (fdin,fdout);
  end;
end.
```

Этот пример использует для вывода сообщений об ошибках процедуру `fatal`, описанную в предыдущей главе. Чтобы не возвращаться назад, приведем ее реализацию:

```
(* Вывести сообщение об ошибке и закончить работу *)
function fatal(s:pchar):integer;
begin
  perror (s);
  halt (1);
end;
```

Вывод программы не полностью предсказуем, так как число сообщений «канал пуст» может быть различным. На одном из компьютеров был получен следующий вывод:

```
MSG=hello
(канал пуст)
(канал пуст)
(канал пуст)
MSG=hello
(канал пуст)
(канал пуст)
(канал пуст)
MSG=hello
(канал пуст)
(канал пуст)
(канал пуст)
MSG=bye!!
Конец связи
```

7.1.6. Использование системного вызова select для работы с несколькими каналами

Для простых приложений применение неблокирующих операций чтения и записи работает прекрасно. Для работы с множеством каналов одновременно существует другое

решение, которое заключается в использовании системного вызова `select`.

Представьте ситуацию, когда родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов, как показано на рис. 7.4.

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому. Эта упрощенная ситуация показана на рис. 7.3.

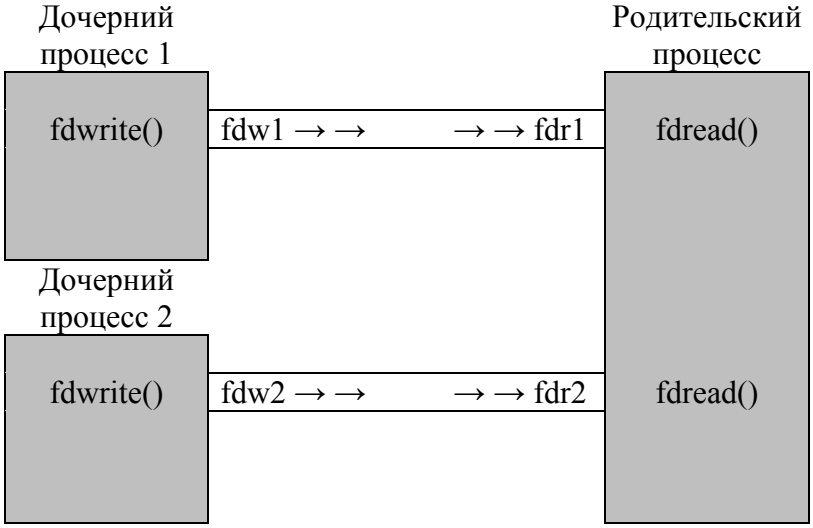


Рис. 7.4. Клиент/сервер с использованием каналов

В этом случае серверный процесс должен как-то справляться с ситуацией, когда одновременно в нескольких каналах может находиться информация, ожидающая обработки. Кроме того, если ни в одном из каналов нет ожидающих данных, то может иметь смысл приостановить работу серверного процесса до их появления, а не опрашивать постоянно каналы. Если информация поступает более чем по одному каналу, то серверный процесс должен знать обо всех таких каналах для того, чтобы работать с ними в правильном порядке (например, согласно их приоритетам).

Это можно сделать при помощи системного вызова `select` (существует также аналогичный вызов `poll`). Системный вызов `select` используется не только для каналов, но и для обычных файлов, терминальных устройств, именованных каналов (которые будут рассмотрены в разделе 7.2) и сокетов (им посвящена глава 10). Системный вызов `select` показывает, какие дескрипторы файлов из заданных наборов готовы для чтения, записи или ожидают обработки ошибок. Иногда серверный процесс не должен совсем прекращать работу, даже если не происходит никаких событий, поэтому в вызове `select` также можно задать предельное время ожидания.

Описание

```
uses linux;

Function Select(Nfds:Longint; var readfds,writefds, errorfds:PFDset;
                Var Timeout): Longint;
```

Первый параметр `nfds` задает число дескрипторов файлов, которые могут представлять интерес для сервера. Например, если дескрипторы файлов с номерами 0, 1 и 2 присвоены потокам `stdin`, `stdout` и `stderr` соответственно, и открыты еще два файла с дескрипторами 3 и 4, то можно присвоить параметру `nfds` значение 5. Программист может определять это значение самостоятельно или воспользоваться постоянной `FD_SETSIZE`, которая определена в файле `stdio`. Значение постоянной `FD_SETSIZE` равно максимальному числу дескрипторов файлов, которые могут быть использованы вызовом `select`.

Второй, третий и четвертый параметры вызова `select` являются указателями на *битовые маски* (bit mask), в которых каждый бит соответствует дескриптору файла. Если бит включен, то это обозначает интерес к соответствующему дескриптору файла. Набор `readfds`

определяет дескрипторы, для которых сервер ожидает возможности чтения; набор `writefds` – дескрипторы, для которых ожидается возможность выполнить запись; набор `errorfds` определяет дескрипторы, для которых сервер ожидает появление ошибки или исключительной ситуации, например, по сетевому соединению могут поступить внеочередные данные. Так как работа с битами довольно неприятна и приводит к немобильности программ, существует абстрактный тип данных `fdset`, а также макросы или функции (в зависимости от конкретной реализации системы) для работы с объектами этого типа. Вот эти макросы для работы с битами файловых дескрипторов:

```
uses linux;

(* Инициализация битовой маски, на которую указывает fds *)
Procedure FD_ZERO(var fds:fdSet);

(* Установка бита fd в маске, на которую указывает fds *)
Procedure FD_Set(fd:longint;var fds:fdSet);

(* Установлен ли бит fd в маске, на которую указывает fds? *)
Function FD_IsSet(fd:longint;var fds:fdSet):boolean;

(* Сбросить бит fd в маске, на которую указывает fds *)
Procedure FD_Clr(fd:longint;var fds:fdSet);
```

Следующий пример демонстрирует, как отслеживать состояние двух открытых дескрипторов файлов:

```
uses linux;
.
.
.
var
  fd1, fd2:longint;
  readset:fdset;

fd1 := fdopen('file1', Open_RDONLY);
fd2 := fdopen('file2', Open_RDONLY);

FD_ZERO(readset);
FD_SET(fd1, readset);
FD_SET(fd2, readset);

case select(5, @readset, nil, nil, nil) of
(* Обработка ввода *)
end;
```

Пример очевиден, если вспомнить, что переменные `fd1` и `fd2` представляют собой небольшие целые числа, которые можно использовать в качестве индексов битовой маски. Обратите внимание на то, что аргументам `writefds` и `errorfds` в вызове `select` присвоено значение `nil`. Это означает, что представляет интерес только чтение из `fd1` и `fd2`.

Пятый параметр вызова `select`, `timeout`, является указателем на следующую структуру `timeval`:

```
uses linux;

TimeVal  =  Record
   sec,               (* Секунды *)
   usec  :  Longint;  (* и микросекунды *)
end;
```

Если указатель является нулевым, как в этом примере, то вызов `select` будет заблокирован до тех пор, пока не произойдет интересующее процесс событие. Если в

структуре timeout задано нулевое время, то вызов select завершится немедленно (без блокирования). И, наконец, если структура timeout содержит ненулевое значение, то возврат из вызова select произойдет через заданное число секунд или микросекунд, если файловые дескрипторы неактивны.

Возвращаемое вызовом select значение равно -1 в случае ошибки, нулю – после истечения временного интервала или целому числу, равному числу «интересующих» программу дескрипторов файлов. Следует сделать предостережение: при возврате из вызова select он переустанавливает битовые маски, на которые указывают переменные readfds, writefds или errorfds, сбрасывая маску и снова задавая в ней дескрипторы файлов, содержащие искомую информацию. Поэтому необходимо сохранять копию исходных масок.[1]

Приведем более сложный пример, в котором используются три канала, связанные с тремя дочерними процессами. Родительский процесс должен также отслеживать стандартный ввод.

```
 (* Программа server - обслуживает три дочерних процесса *)
uses linux,stdio;

const
  MSGSIZE=6;
  msg1:array [0..MSGSIZE-1] of char = 'hello';
  msg2:array [0..MSGSIZE-1] of char = 'bye!!';

type
  tp1=array [0..1] of longint;
  tp3=array [0..2] of tp1;


(* Родительский процесс ожидает сигнала в трех каналах *)
procedure parent(p:tp3);           (* код родительского процесса *)
var
  ch:char;
  buf:array [0..MSGSIZE-1] of char;
  _set, master:fdset;
  i:integer;
begin
  (* Закрыть все ненужные дескрипторы, открытые для записи *)
  for i:=0 to 2 do
    fdclose (p[i][1]);
  (* Задать битовые маски для системного вызова select *)
  FD_ZERO (master);
  FD_SET (0, master);
  for i:=0 to 2 do
    FD_SET (p[i][0], master);
  (* Лимит времени для вызова select не задан, поэтому он
   * будет заблокирован, пока не произойдет событие *)
  _set := master;
  while select (p[2][0] + 1, @_set, nil, nil, nil) > 0 do
  begin
    (* Нельзя забывать и про стандартный ввод,
     * т.е. дескриптор файла fd=0. *)
    if FD_ISSET (0, _set) then
```

[1] В некоторых реализациях вызов select изменяет также содержимое структуры timeout: оно заполняется оставшимся временем до истечения первоначально заданного интервала. Данную возможность следует учитывать при вызове select с нулевыми масками в качестве высокоточного аналога вызова sleep – тогда использование select в цикле может привести к неправильным результатам.

```
    begin
      write('Из стандартного ввода...');
      fdread (0, ch, 1);
      writeln(ch);
    end;
    for i:=0 to 2 do
    begin
      if FD_ISSET (p[i][0], _set) then
      begin
        if fdread (p[i][0], buf, MSGSIZE) > 0 then
        begin
          writeln('Сообщение от потомка', i);
          writeln('MSG=', buf);
        end;
      end;
    end;
    (* Если все дочерние процессы прекратили работу,
     * то сервер вернется в основную программу
     *)
    if waitpid (-1, nil, WNOHANG) = -1 then
      exit;
  _set := master;
  end;
end;


function child (p:tp1):integer;
var
  count:integer;
begin
  fdclose (p[0]);
  for count:=1 to 2 do
  begin
    fdwrite (p[1], msg1, MSGSIZE);
    (* Пауза в течение случайно выбранного времени *)
    sleep (getpid mod 4);
  end;
  (* Послать последнее сообщение *)
  fdwrite (p[1], msg2, MSGSIZE);
  halt (0);
end;


var
  pip:tp3;
  i:integer;
begin
  (* Создать три канала связи, и породить три процесса. *)
  for i:=0 to 2 do
  begin
    if not assignpipe (pip[i][0],pip[i][1]) then
      fatal ('Ошибка вызова pipe');
    case fork of
      -1:         (* ошибка *)
        fatal ('Ошибка вызова fork');
      0:          (* дочерний процесс *)
        child (pip[i]);
    end;
```

```
  end;
  parent (pip);
  halt (0);
end.
```

Результат данной программы может быть таким:

```
Сообщение от потомка 0
MSG=hello
Сообщение от потомка 1
MSG=hello
Сообщение от потомка 2
MSG=hello

d   (пользователь нажимает клавишу d, а затем клавишу Return)
Из стандартного ввода d (повторение символа d)
Из стандартного ввода (повторение символа Return)

Сообщение от потомка 0
MSG=hello
Сообщение от потомка 1
MSG=hello
Сообщение от потомка 2
MSG=hello

Сообщение от потомка 0
MSG=bye
Сообщение от потомка 1
MSG=bye
Сообщение от потомка 2
MSG=bye
```

Обратите внимание, что в этом примере пользователь нажимает клавишу **d**, а затем символ перевода строки (Enter или Return), и это отслеживается в стандартном вводе в вызове `select`.

Функция `SelectText` является модификацией `Select`, предназначенной для работы с текстовыми файлами:

Описание

```
uses linux;

Function SelectText(var T:Text; TimeOut:PTime):Longint;
```

`SelectText` выполняет системный вызов `Select` для файлов типа Text. Время ожидания может быть указано в параметре `TimeOut`. Вызов `SelectText` самостоятельно определяет необходимость проверки чтения и записи в зависимости от того, в каком режиме был открыт файл. При `Reset` выполняется проверка на чтение, при `Rewrite` и `Append` – на запись.

Пример использования `SelectText`:

```
Uses linux;

Var tv : TimeVal;

begin
  Writeln ('Press the <ENTER> to continue the program.');
  { Wait until File descriptor 0 (=Input) changes }
  SelectText (Input,nil);
  { Get rid of <ENTER> in buffer }
  readln;
  Writeln ('Press <ENTER> key in less than 2 seconds...');
  tv.sec:=2;
```

```
  tv.usec:=0;
  if SelectText (Input,@tv)>0 then
    Writeln ('Thank you !')
  else
    Writeln ('Too late !');
end.
```

Связать `SelectText` и `Select` можно с помощью функции `GetFS`, позволяющей из любой файловой переменной получить дескриптор файла.

Описание

```
uses linux;

Function GetFS(Var F:Any File Type):Longint;
```

Например:

```
Uses linux;

begin
  Writeln ('File descriptor of input  ',getfs(input));
  Writeln ('File descriptor of output ',getfs(output));
  Writeln ('File descriptor of stderr ',getfs(stderr));
end.
```

Пример использования `SelectText`:

```
Uses linux;

Var tv : TimeVal;

begin
  Writeln ('Press the <ENTER> to continue the program.');
  { Wait until File descriptor 0 (=Input) changes }
  SelectText (Input,nil);
  { Get rid of <ENTER> in buffer }
  readln;
  Writeln ('Press <ENTER> key in less than 2 seconds...');
  tv.sec:=2;
  tv.usec:=0;
  if SelectText (Input,@tv)>0 then
    Writeln ('Thank you !')
  else
    Writeln ('Too late !');
end.
```

7.1.7. Каналы и системный вызов exec

Вспомним, как можно создать канал между двумя программами с помощью командного интерпретатора:

```
$ ls | wc
```

Как это происходит? Ответ состоит из двух частей. Во-первых, командный интерпретатор использует тот факт, что открытые дескрипторы файлов остаются открытыми (по умолчанию) после вызова `exec`. Это означает, что два файловых дескриптора канала, которые были открыты до выполнения комбинации вызовов `fork/exec`, останутся открытыми и когда дочерний процесс начнет выполнение новой программы. Во-вторых, перед вызовом `exec` командный интерпретатор соединяет стандартный вывод программы `ls` с входом канала, а стандартный ввод программы `wc` – с выходом канала. Это можно сделать при помощи вызова `fcntl` или `dup2`, как было показано в упражнении 5.10. Так как значения дескрипторов файлов, соответствующих стандартному вводу, стандартному выводу и стандартному выводу диагностики, равны 0, 1 и 2 соответственно, то можно, например, соединить стандартный вывод с другим дескриптором файла, используя вызов `dup2`

следующим образом. Обратите внимание, что перед переназначением вызов dup2 закрывает файл, представленный его вторым параметром.

```
(* Вызов dup2 будет копировать дескриптор файла 1 *)
dup2(filedes, 1);
.
.
.
(* Теперь программа будет записывать свой стандартный *)
(* вывод в файл, заданный дескриптором filedes *)
.
.
.
```

Следующий пример, программа join, демонстрирует механизм каналов, задействованный в упрощенном командном интерпретаторе. Программа join имеет два параметра, com1 и com2, каждый из которых соответствует выполняемой команде. Оба параметра в действительности являются массивами строк, которые будут переданы вызову execvp.

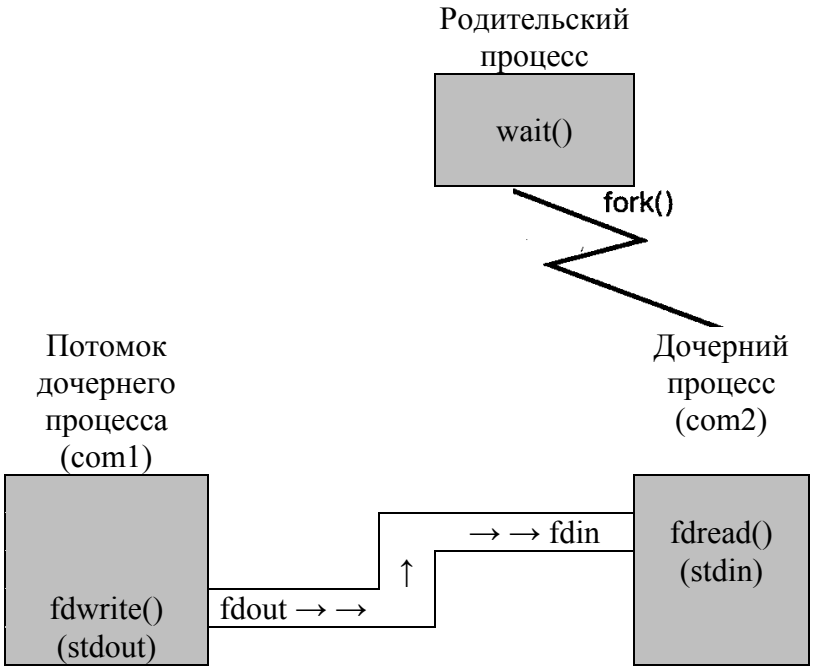


Рис. 7.5. Программа join

Программа join запустит обе программы на выполнение и свяжет стандартный вывод программы com1 со стандартным вводом программы com2. Работа программы join изображена на рис. 7.5 и может быть описана следующей схемой (без учета обработки ошибок):

```
процесс порождает дочерний процесс и ожидает действий от него
дочерний процесс продолжает работу

дочерний процесс создает канал

затем дочерний процесс порождает еще один дочерний процесс

В потомке дочернего процесса:
стандартный вывод подключается
к входу канала при помощи вызова dup2
```

```
ненужные дескрипторы файлов закрываются

при помощи вызова exec запускается программа,
заданная параметром 'com1'

В первом дочернем процессе:
стандартный ввод подключается
к выходу канала при помощи вызова dup2

ненужные дескрипторы файлов закрываются

при помощи вызова exec запускается программа,
заданная параметром 'com2'
```

Далее следует реализация программы `join`; она также использует процедуру `fatal`, представленную в разделе 7.1.5.

```
(* Программа join - соединяет две программы каналом *)
function join (com1, com2:ppchar):integer;
var
  fdin,fdout:longint;
  status:integer;
begin
  (* Создать дочерний процесс для выполнения команд *)
  case fork of
    -1:                 (* ошибка *)
      fatal ('Ошибка 1 вызова fork в программе join');
    0:                  (* дочерний процесс *)
      ;
    else                (* родительский процесс *)
    begin
      wait(@status);
      join:=status;
      exit;
    end;
  end;
  (* Остаток процедуры, выполняемой дочерним процессом *)
  (* Создать канал *)
  if not assignpipe(fdin,fdout) then
    fatal ('Ошибка вызова pipe в программе join');
  (* Создать еще один процесс *)
  case fork of
    -1:
      (* ошибка *)
      fatal ('Ошибка 2 вызова fork в программе join');
    0:
    begin
      (* процесс, выполняющий запись *)
      dup2 (fdout, 1);        (* направить ст. вывод в канал *)
      fdclose (fdin);         (* сохранить дескрипторы файлов *)
      fdclose (fdout);
      execvp (com1[0], com1, envp);
      (* Если execvp возвращает значение, то произошла ошибка *)
      fatal ('Ошибка 1 вызова execvp в программе join');
    end;
    else
    begin
      (* процесс, выполняющий чтение *)
      dup2 (fdin, 0);         (* направить ст. ввод из канала *)
      fdclose (fdin);
```

```
      fdclose (fdout);
      execvp (com2[0], com2, envp);
      fatal ('Ошибка 2 вызова execvp в программе join');
    end;
  end;
end;
```

Эту процедуру можно вызвать следующим образом:

```
uses linux, stdio;

const
  one:array [0..3] of pchar = ('ls', '-l', '/usr/lib', nil);
  two:array [0..2] of pchar = ('grep', '^d', nil);
var
  ret:integer;
begin
  ret := join (one, two);
  writeln ('Возврат из программы join ', ret);
  halt (0);
end.
```

Упражнение 7.3. *Как можно обобщить подход, показанный в программе* `join`*, для связи нескольких процессов при помощи каналов?*

Упражнение 7.4. *Добавьте возможность работы с каналами в командный интерпретатор* `smallsh`*, представленный в предыдущей главе.*

Упражнение 7.5. *Придумайте метод, позволяющий родительскому процессу запускать программу в качестве дочернего процесса, а затем считывать ее стандартный вывод при помощи канала. Стоит отметить, что эта идея лежит в основе процедур* `popen/pipeopen` *и* `pclose/pipeclose`*, которые входят в стандартную библиотеку ввода/вывода. Процедуры* `popen/pipeopen` *и* `pclose/pipeclose` *избавляют программиста от большинства утомительных деталей согласования вызовов* `fork`*,* `exec`*,* `fdclose`*,* `dup` *или* `dup2`*. Эти процедуры обсуждаются в главе 11.*

7.2. Именованные каналы, или FIFO

Каналы являются изящным и мощным механизмом межпроцессного взаимодействия. Тем не менее они имеют ряд недостатков.

Первый, и наиболее серьезный из них, заключается в том, что каналы могут использоваться только для связи процессов, имеющих общее происхождение, таких как родительский процесс и его потомок. Это ограничение становится видным при попытке разработать настоящую «серверную» программу, которая выполняется постоянно, обеспечивая системный сервис. Примерами таких программ являются серверы управления сетью и спулеры печати. В идеале клиентские процессы должны иметь возможность стартовать, подключаться к не связанному с ними серверному процессу при помощи канала, а затем снова отключаться от него. К сожалению, такую модель при помощи обычных каналов реализовать нельзя.

Второй недостаток каналов заключается в том, что они не могут существовать постоянно. Они каждый раз должны создаваться заново, а после завершения обращающегося к ним процесса уничтожаются.

Для восполнения этих недостатков существует разновидность канала, называемая *именованным каналом*, или файлом типа *FIFO* (сокращение от *first-in first-out*, то есть «первый вошел/первым вышел»). В отношении вызовов `fdread` и `fdwrite` именованные каналы идентичны обычным. Тем не менее, в отличие от обычных каналов, именованные каналы являются постоянными и им присвоено имя файла системы UNIX. Именованный канал также имеет владельца, размер и связанные с ним права доступа. Он может быть открыт, закрыт и удален, как и любой файл UNIX, но при чтении или записи ведет себя аналогично каналу.

Прежде чем рассматривать применение каналов FIFO на программном уровне, рассмотрим их использование на уровне команд. Для создания именованного канала используется команда mknod:

```
$ /etc/mknod channel p
```

Первый аргумент channel является именем канала FIFO (в качестве него можно задать любое допустимое имя UNIX). Параметр p команды mknod указывает, что нужно создать именованный канал. Этот параметр необходим, так как команда mknod также используется для создания файлов устройств.

Некоторые атрибуты вновь созданного канала FIFO можно вывести при помощи команды ls:

```
$ ls -l channel
prw-rw-r- 1 ben usr 0 Aug 1 21:05 channel
```

Символ p в первой колонке обозначает, что channel является файлом типа FIFO. Обратите внимание на права доступа к именованному каналу channel (чтение/запись для владельца и группы владельца, только чтение для всех остальных пользователей); владельца и группу владельца (ben, usr); размер (0 байт, то есть в настоящий момент канал пуст) и время создания.

При помощи стандартных команд UNIX можно выполнять чтение из канала FIFO и запись в него, например:

```
$ cat < channel
```

Если выполнить эту команду сразу же после создания именованного канала channel, то она «зависнет». Это происходит из-за того, что процесс, открывающий канал FIFO на чтение, по умолчанию будет блокирован до тех пор, пока другой процесс не попытается открыть канал FIFO для записи. Аналогично процесс, пытающийся открыть канал FIFO для записи, будет блокирован до тех пор, пока другой процесс не попытается открыть его для чтения. Это благоразумный подход, так как он экономит системные ресурсы и облегчает координацию работы программы. Вследствие этого, при необходимости создания одновременно как записывающего, так и читающего процессов, потребуется запустить один из них в фоновом режиме (или с другого терминала, или псевдотерминала xterm графического интерфейса), например:

```
$ cat < channel &
102
$ ls -l > channel; wait
total 17
prw-rw-r- 1 ben usr 0     Aug 1   21:05   channel
-rw-rw-r- 1 ben usr 0     Aug 1   21:06   f
-rw-rw-r- 1 ben usr 937   Jul 27  22:30   fifos
-rw-rw-r- 1 ben usr 7152  Jul 27  22:11   pipes.cont
```

Проанализируем подробнее этот результат. Содержимое каталога вначале выводится при помощи команды ls, а затем записывается в канал FIFO. Ожидающая команда cat затем считывает данные из канала FIFO и выводит их на экран. После этого процесс, выполняющий команду cat, завершает работу. Это происходит из-за того, что канал FIFO больше не открыт для записи, чтение из него будет безуспешным, как и для обычного канала, что команда cat понимает как достижение конца файла. Команда же wait заставляет командный интерпретатор ждать завершения команды cat перед тем, как снова вывести приглашение командной строки.

7.2.1. Программирование при помощи каналов FIFO

Программирование при помощи каналов FIFO, в основном, идентично программированию с использованием обычных каналов. Единственное существенное различие заключается в их инициализации. Вместо использования вызова assignpipe канал FIFO создается при помощи вызова mkfifo. В старых версиях UNIX может потребоваться использование более общего вызова mknod.

Описание

```
uses linux;

Function MkFifo(PathName:String; Mode:Longint):Boolean;
```

Системный вызов `mkfifo` создает файл FIFO с именем, заданным первым параметром `pathname`. Канал FIFO будет иметь права доступа, заданные параметром `mode` и измененные в соответствии со значением `umask` процесса.

После создания канала FIFO он должен быть открыт при помощи вызова `fdopen`. Поэтому, например, фрагмент кода

```
uses linux;
.
.
mkfifo('/tmp/fifo', octal(0666));
.
.
fd := fdopen('/tmp/fifo', Open_WRONLY);
```

открывает канал FIFO для записи. Вызов `fdopen` будет заблокирован до тех пор, пока другой процесс не откроет канал FIFO для чтения (конечно же, если канал FIFO уже был открыт для чтения, то возврат из вызова `open` произойдет немедленно).

Можно выполнить не блокирующий вызов `fdopen` для канала FIFO. Для этого во время вызова должен быть установлен флаг `Open_NONBLOCK` (определенный в файле `linux`) и один из флагов `Open_RDONLY` или `Open_WRONLY`, например:

```
fd := fdopen('/tmp/fifo', Open_WRONLY or Open_NONBLOCK);
if fd = -1 then
  perror('Ошибка вызова open для канала FIFO');
```

Если не существует процесс, в котором канал FIFO открыт для чтения, то этот вызов `fdopen` вернет значение -1 вместо блокировки выполнения, а переменная `linuxerror` будет содержать значение `Sys_ENXIO`. В случае же успешного вызова `fdopen` последующие вызовы `fdwrite` для канала FIFO также будут не блокирующими.

Наступило время привести пример. Представим две программы, которые показывают, как можно использовать канал FIFO для реализации системы обмена сообщениями. Эти программы используют тот факт, что вызовы `fdread` или `fdwrite` для каналов FIFO, как и для программных каналов, являются неделимыми (для небольших порций данных). Если при помощи канала FIFO пересылаются сообщения фиксированного размера, то отдельные сообщения будут сохраняться, даже если несколько процессов одновременно выполняют запись в канал.

Рассмотрим вначале программу `sendmessage`, которая посылает отдельные сообщения в канал FIFO с именем `fifo`. Она вызывается следующим образом:

```
$ sendmessage 'текст сообщения 1' 'текст сообщения 2'
```

Обратите внимание на то, что каждое сообщение заключено в кавычки и поэтому считается просто одним длинным аргументом. Если не сделать этого, то каждое слово будет рассматриваться, как отдельное сообщение. Программа `sendmessage` имеет следующий исходный текст:

```
(* Программа sendmessage - пересылка сообщений через FIFO *)
uses linux,stdio,strings;

const
  MSGSIZ=63;
  fifo  = 'fifo';


var
  fd,j:integer;
  nwrite:longint;
```

```
  msgbuf:array [0..MSGSIZ] of char;
begin
  if paramcount=0 then
  begin
    writeln (stderr, 'Применение: sendmessage сообщение');
    halt (1);
  end;
  (* Открыть канал fifo, установив флаг Open_NONBLOCK *)
  fd := fdopen (fifo, Open_WRONLY or Open_NONBLOCK);
  if fd < 0 then
    fatal ('Ошибка вызова open для fifo');
  (* Посылка сообщений *)
  for j := 1 to paramcount do
  begin
    if length(paramstr(j)) > MSGSIZ then
    begin
      writeln('Слишком длинное сообщение ', paramstr(j));
      continue;
    end;
    strpcopy(msgbuf, paramstr(j));
    nwrite := fdwrite (fd, msgbuf, MSGSIZ + 1);
    if nwrite = -1 then
      fatal ('Ошибка записи сообщения');
  end;
  halt(0);
end.
```

И снова для вывода сообщений об ошибках использована процедура `fatal`. Сообщения посылаются блоками по 64 байта при помощи не блокируемого вызова `fdwrite`. В действительности текст сообщения ограничен 63 символами, а последний символ является нулевым.

Программа `rcvmessage` принимает сообщения при помощи чтения из канала FIFO. Она не выполняет никаких полезных действий и служит только демонстрационным примером:

```
(* Программа rcvmessage - получение сообщений из канала fifo *)
uses linux,stdio;

const
  MSGSIZ=63;
  fifo  = 'fifo';


var
  fd:integer;
  msgbuf:array [0..MSGSIZ] of char;
begin
  (* Создать канал fifo, если он еще не существует *)
  if not mkfifo (fifo, octal(0666)) then
    if linuxerror <> Sys_EEXIST then
      fatal ('Ошибка приемника: вызов mkfifo');

  (* Открыть канал fifo для чтения и записи. *)
  fd := fdopen (fifo, Open_RDWR);
  if fd < 0 then
    fatal ('Ошибка при открытии канала fifo');

  (* Прием сообщений *)
  while true do
```

```
  begin
    if fdread (fd, msgbuf, MSGSIZ + 1) < 0 then
      fatal ('Ошибка при чтении сообщения');
    (*
     * вывести сообщение; в настоящей программе
     * вместо этого могут выполняться какие-либо
     * полезные действия.
     *)

    writeln('Получено сообщение: ', msgbuf);
  end;
end.
```

Обратите внимание на то, что канал FIFO открывается одновременно для чтения и записи (при помощи задания флага `Open_RDWR`). Чтобы понять, для чего это сделано, предположим, что канал FIFO был открыт только для чтения при помощи задания флага `Open_RDONLY`. Тогда выполнение программы `rcvmessage` будет сразу заблокировано в момент вызова `fdopen`. Когда после старта программы `sendmessage` в канал FIFO будет произведена запись, вызов `fdopen` будет разблокирован, программа `rcvmessage` будет читать все посылаемые сообщения. Когда же канал FIFO станет пустым, а процесс `sendmessage` завершит работу, вызов `fdread` начнет возвращать нулевое значение, так как канал FIFO уже не будет открыт на запись ни в одном процессе. При этом программа `rcvmessage` войдет в бесконечный цикл. Использование флага `Open_RDWR` позволяет гарантировать, что, по крайней мере, в одном процессе, то есть самом процессе программы `rcvmessage`, канал FIFO будет открыт для записи. В результате вызов `open` всегда будет блокироваться то тех пор, пока в канал FIFO снова не будут записаны данные.

Следующий диалог показывает, как можно использовать эти две программы. Программа `rcvmessage` выполняется в фоновом режиме для получения сообщений от разных процессов, выполняющих программу `sendmessage`.

```
$ rcvmessage &
40
$ sendmessage 'сообщение 1' 'сообщение 2'
Получено сообщение: сообщение 1
Получено сообщение: сообщение 2
$ sendmessage 'сообщение номер 3'
Получено сообщение: сообщение номер 3
```

Упражнение 7.6. *Программы* `sendmessage` *и* `rcvmessage` *образуют основу простой системы обмена данными. Сообщения, посылаемые программе* `rcvmessage`*, могут, например, быть именами файлов, которые нужно обработать. Проблема заключается в том, что текущие каталоги программ* `sendmessage` *и* `rcvmessage` *могут быть различными, поэтому относительные пути будут восприняты неправильно. Как можно разрешить эту проблему? Можно ли создать, скажем, спулер печати в большой системе, используя только каналы FIFO?*

Упражнение 7.7. *Если программу* `rcvmessage` *нужно сделать настоящей серверной программой, то потребуется гарантия того, что в произвольный момент времени выполняется только одна копия сервера. Существует несколько способов достичь этого. Один из методов состоит в создании файла блокировки. Рассмотрим следующую процедуру:*

```
uses linux;

const
  lck = '/tmp/lockfile';


function makelock:integer;
```

```
var
  fd:integer;
begin
  fd := fdopen (lck, Open_RDWR or Open_CREAT or Open_EXCL, octal(0600));

  if fd < 0 then
  begin
    if linuxerror = SYS_EEXIST then
      halt (1)          (* файл занят другим процессом *)
    else
      halt (127);       (* неизвестная ошибка *)
  end;

  (* Файл блокировки создан, выход из процедуры *)
  fdclose (fd);
  makelock:=0;
end;
```

Эта процедура использует тот факт, что вызов open *осуществляется за один шаг. Поэтому, если несколько процессов пытаются выполнить процедуру* `makelock`*, одному из них это удастся первым, и он создаст файл блокировки и «заблокирует» работу остальных. Добавьте эту процедуру к программе* `sendmessage`*. При этом, если выполнение программы* `sendmessage` *завершается при помощи сигнала* `SIGHUP` *или* `SIGTERM`*, то она должна удалять файл блокировки перед выходом. Как вы думаете, почему мы использовали в процедуре* `makelock` *вызов* `fdopen`*, а не* `fdcreat`*?*

Глава 8. Дополнительные методы межпроцессного взаимодействия

8.1. Введение

Используя средства из глав 6 и 7, можно осуществить основные взаимодействия между процессами. В этой главе рассматриваются усовершенствованные средства межпроцессного взаимодействия, которые позволят использовать более сложные методы программирования.

Первая и наиболее простая тема данной главы – *блокировка записей* (record locking), которая фактически является не формой прямого межпроцессного взаимодействия, а скорее – методом координирования работы процессов. Блокировка позволяет процессу временно резервировать часть файла для исключительного использования при решении некоторых сложных задач управления базами данных. Здесь стоит сделать предупреждение: спецификация XSI определяет блокировку записей как *рекомендательную* (advisory), означающую, что она не препятствует непосредственному выполнению операций файлового ввода/вывода, а вся ответственность за проверку установленных блокировок полностью ложится на процесс.[1]

Другие механизмы межпроцессного взаимодействия, обсуждаемые в этой главе, являются более редкими. В общем случае эти средства описываются как *средства IPC* (IPC facilities, где сокращение IPC означает *inter-process communication* – межпроцессное взаимодействие) и включены в одноименный модуль. Этот общий термин подчеркивает общность их применения и структуры, хотя существуют три определенных типа таких средств:

- *очереди сообщений* (message passing). Они позволяют процессу посылать и принимать сообщения, под которыми понимается произвольная последовательность байтов или символов;
- *семафоры* (semaphores). По сравнению с очередями сообщений семафоры представляют собой низкоуровневый метод синхронизации процессов, малопригодный для передачи больших объемов данных. Их теория берет начало из работ Дейкстры (E.W. Dijkstra, 1968);
- *разделяемая память* (shared memory). Это средство межпроцессного взаимодействия позволяет двум и более процессам совместно использовать данные, содержащиеся в определенных сегментах памяти. Естественно, обычно данные процесса являются недоступными для других процессов. Этот механизм обычно является самым быстрым механизмом межпроцессного взаимодействия.[2]

8.2. Блокировка записей

8.2.1. Мотивация

На первом этапе стоит рассмотреть простой пример демонстрации того, почему блокировка записей необходима в некоторых ситуациях.

Примером будет служить известная корпорация ACME Airlines, использующая ОС UNIX в своей системе заказа билетов. Она имеет два офиса, A и B, в каждом из которых установлен терминал, подключенный к компьютеру авиакомпании. Служащие компании используют для доступа к базе данных, реализованной в виде обычного файла UNIX, программу `acmebook`. Эта программа позволяет пользователю выполнять чтение и обновление базы данных. В частности, служащий может уменьшить на единицу число свободных мест при заказе билета на определенный авиарейс.

---

[1] В некоторых версиях UNIX есть также возможность применения *обязательных блокировок* (mandatory lock).

[2] Три упомянутые средства часто называют *System V IPC*, поскольку впервые это семейство межпроцессных взаимодействий было введено в диалекте System V Unix. Стандарт POSIX 1003.1 их не описывает; более того. аналогичные POSIX-средства имеют другой интерфейс и семантику. Тем не менее *System V IPC* внесены в спецификацию SUSV2 как заимствованные из второй версии спецификации SVID.

Предположим теперь, что осталось всего одно свободное место на рейс ACM501 в Лондон, и миссис Джонс входит в офис A; в то же самое время мистер Смит входит в офис B, и они оба заказывают место на рейс ACM501. При этом возможна такая последовательность событий:

1. Служащий офиса A запускает программу `acmebook`. Назовем стартовавший процесс PA.
2. Сразу же после этого служащий в офисе B также запускает программу `acmebook`. Назовем этот процесс PB.
3. Процесс PA считывает соответствующую часть базы данных при помощи системного вызова `fdread` и определяет, что осталось всего одно свободное место.
4. Процесс PB выполняет чтение из базы данных сразу же после процесса PA и также определяет, что осталось одно свободное место на рейс ACM501.
5. Процесс PA обнуляет счетчик свободных мест для рейса при помощи системного вызова `fdwrite`, изменяя соответствующую часть базы данных. Служащий в офисе A вручает билет миссис Джонс.
6. Сразу же вслед за этим процесс PB также выполняет запись в базу данных, также записывая нулевое значение в счетчик свободных мест. Но на этот раз значение счетчика ошибочно – на самом деле оно должно было бы быть равно -1, то есть хотя процесс PA уже обновил базу данных, процесс PB не знает об этом и спешит выполнить заказ, как если бы место было свободно. Вследствие этого мистер Смит также получит билет, и на самолет будет продано больше билетов, чем число свободных мест в нем.

Эта проблема возникает из-за того, что несколько процессов могут одновременно обращаться к файлу UNIX. Комплексная операция с данными файла, состоящая из нескольких вызовов `fdseek`, `fdread` и `fdwrite`, может быть выполнена двумя или более процессами одновременно, и это, как показывает наш простой пример, будет иметь непредвиденные последствия.

Одно из решений состоит в том, чтобы разрешить процессу выполнить *блокировку* (lock) части файла, с которой он работает. Блокировка, которая нисколько не изменяет содержимое файла, показывает другим процессам, что данные, о которых идет речь, уже используются. Это предотвращает вмешательство другого процесса во время последовательности дискретных физических операций, образующих одну комплексную операцию, или транзакцию. Этот механизм часто называют *блокировкой записи* (record locking), где запись означает просто произвольную часть файла. Для обеспечения корректности сама операция блокировки должна быть атомарной, чтобы она не могла пересечься с параллельной попыткой блокировки в другом процессе.

Для обеспечения нормальной работы блокировка должна выполняться централизованно. Возможно, лучше всего это возложить на ядро, хотя пользовательский процесс, выступающий в качестве агента базы данных, также может служить для этой цели. Блокировка записей на уровне ядра может выполняться при помощи уже известного нам вызова `fcntl`.

Обратите внимание, что возможен также альтернативный способ блокировки записей – при помощи процедуры `lockf`. Этот подход все еще встречается во многих системах – за дополнительными сведениями следует обратиться к справочному руководству системы.

8.2.2. Блокировка записей при помощи вызова fcntl

О системном вызове управления файловым вводом/выводом `fcntl` уже упоминалось ранее. В дополнение к привычным функциям вызов `fcntl` может также использоваться для выполнения блокировки записей. Он предлагает два типа блокировки:

– *блокировка чтения* (read locks) – просто предотвращает установку другими процессами блокировки записи при помощи вызова `fcntl`. Несколько процессов могут одновременно выполнять блокировку чтения для одного и того же участка

файла. Блокировка чтения может быть полезной, если, например, требуется предотвратить обновление данных, не скрывая их от просмотра другими пользователями;

– *блокировка записи* (write locks) – предотвращает установку другими процессами блокировку чтения или записи для файла. Другими словами, для заданного участка файла может существовать только одна блокировка записи одновременно. Блокировка записи может использоваться, например, для скрытия участков файла от просмотра при выполнении обновления.

Следует напомнить, что в соответствии со спецификацией XSI блокировка вызовом `fcntl` является всего лишь рекомендательной. Поэтому процессам необходимо явно согласовывать свои действия, чтобы блокировка вызовом `fcntl` была действенной (процессы не должны производить операции ввода/вывода без предварительного блокирования соответствующей области).

Для блокировки записей вызов `fcntl` используется следующим образом:

Описание

```
uses linux, stdio;

Procedure Fcntl(filedes:longint; Cmd:longint; ldata: pflockrec);
```

Как обычно, аргумент `filedes` должен быть допустимым дескриптором открытого файла. Для блокировки чтения дескриптор `filedes` должен быть открыт при помощи флагов `Open_RDONLY` или `Open_RDWR`, поэтому в качестве него не подойдет дескриптор, возвращаемый вызовом `fdcreat`. Для блокировки записи дескриптор `filedes` должен быть открыт при помощи флагов `Open_WRONLY` или `Open_RDWR`.

Как уже упоминалось, параметр вызова `cmd` определяет выполняемое действие, кодируемое одним из значений, определенных в файле `linux`. Следуют три команды относятся к блокировке записей:

`F_GETLK`	Получить описание блокировки на основе данных, передаваемых в аргументе `ldata`. (Возвращаемая информация описывает первую блокировку, которая препятствует наложению блокировки, описанной структурой `ldata`)
`F_SETLK`	Попытаться применить блокировку к файлу и немедленно возвратить управление, если это невозможно. Используется также для удаления активной блокировки
`F_SETLKW`	Попытаться применить блокировку к файлу и приостановить работу, если блокировка уже наложена другим процессом. Ожидание процесса внутри вызова `fcntl` можно прервать при помощи сигнала

`ldata` содержит описание блокировки. Структура `flockrec` определена в файле `stdio` и включает следующие элементы:

```
flockrec=record
  l_type:word;  (*Описывает тип блокировки: F_RDLCK, F_WRLCK, F_UNLCK. *)
  l_whence:word;    (* Тип смещения, как и в вызове lseek *)
  l_start:longint; (* Смещение в байтах *)
  l_len:longint;    (*Размер сегмента данных; 0 означает до конца файла *)
  l_pid:longint;    (*Устанавливается командой F_GETLK *)
end;
```

Три элемента, `l_whence`, `l_start` и `l_len`, определяют участок файла, который будет заблокирован, проверен или разблокирован. Переменная `l_whence` идентична третьему аргументу вызова `lseek`. Она принимает одно из трех значений: `SEEK_SET`, `SEEK_CUR` или `SEEK_END`, обозначая, что смещение должно вычисляться от начала файла, от текущей позиции указателя чтения-записи или конца файла. Элемент `l_start` устанавливает начальное положение участка файла по отношению к точке, заданной элементом `l_whence`.

Элемент l_len является длиной участка в байтах; нулевое значение обозначает участок с заданной начальной позиции до максимально возможного смещения. На рис. 8.1. показано, как это работает для случая, если значение поля l_whence равно SEEK_CUR. Структура tsemid_ds определена в файле linux.

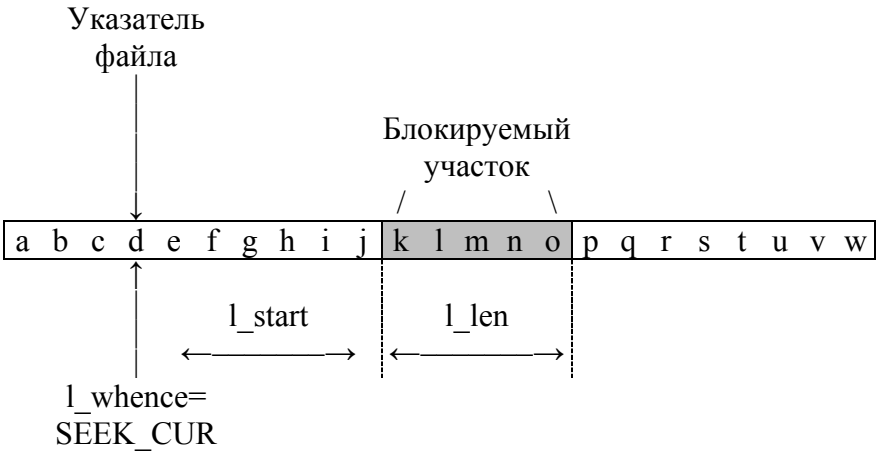


Рис. 8.1. Параметры блокировки

Тип l_type определяет тип блокировки. Он может принимать одно из трех значений, определенных в файле stdio:

F_RDLCK Выполняется блокировка чтения
F_WRLCK Выполняется блокировка записи
F_UNLCK Снимается блокировка заданного участка

Поле l_pid существенно только при выборе команды F_GETLK в вызове fcntl. Если существует блокировка, препятствующая установке блокировки, описанной полями структуры ldata, то значение поля l_pid будет равно значению идентификатора процесса, установившего ее. Другие элементы структуры также будут переустановлены системой. Они будут содержать параметры блокировки, наложенной другим процессом.

Установка блокировки при помощи вызова fcntl

Следующий пример показывает, как можно использовать вызов fcntl для установления блокировки записи.

```
uses linux,stdio;
.
.
.
var
  my_lock:flockrec;

my_lock.l_type := F_WRLCK;
my_lock.l_whence := SEEK_CUR;
my_lock.l_start := 0;
my_lock.l_len := 512;
fcntl (fd, F_SETLKW, longint(@my_lock));
```

При этом будут заблокированы 512 байт, начиная с текущего положения указателя чтения-записи. Заблокированный участок теперь считается «зарезервированным» для исключительного использования процессом. Информация о блокировке помещается в свободную ячейку системной таблицы блокировок.

Если весь участок файла или какая-то его часть уже были заблокированы другим процессом, то вызывающий процесс будет приостановлен до тех пор, пока не будет доступен весь участок. Приостановка работы процесса может быть прервана при помощи сигнала; в частности, для задания времени ожидания может быть использован вызов alarm. Если ожидание не будет прервано, и, в конце концов, участок файла освободится, то процесс

заблокирует его. Если происходит ошибка, например, если вызову `fcntl` передается неверный дескриптор файла или переполнится системная таблица блокировок, то будет возвращено значение -1.

Следующий пример – программа `lockit` открывает файл с именем `locktest` (который должен существовать) и блокирует его первые десять байт при помощи вызова `fcntl`. Затем она порождает дочерний процесс, пытающийся заблокировать первые пять байт файла; родительский процесс в это время делает паузу на пять секунд, а затем завершает работу. В этот момент система автоматически снимает блокировку, установленную родительским процессом.

```
(* Программа lockit - блокировка при помощи вызова fcntl *)
uses linux,stdio;

var
  fd:integer;
  my_lock:flockrec;
begin
  (* Установка параметров блокировки записи *)
  my_lock.l_type := F_WRLCK;
  my_lock.l_whence := SEEK_SET;
  my_lock.l_start := 0;
  my_lock.l_len := 10;
  (* Открыть файл *)
  fd := fdopen ('locktest', Open_RDWR);
  (* Заблокировать первые десять байт *)
  fcntl (fd, F_SETLKW, longint(@my_lock));
  if linuxerror > 0 then
  begin
    perror ('parent: locking');
    halt (1);
  end;
  writeln('Родительский процесс: блокировка установлена');
  case fork of
    -1:                (* ошибка *)
    begin
      perror ('Ошибка вызова fork');
      halt (1);
    end;
    0:
    begin
      my_lock.l_len := 5;
      fcntl (fd, F_SETLKW, longint(@my_lock));
      if linuxerror > 0 then
      begin
        perror ('Дочерний процесс: установка блокировки');
        halt (1);
      end;
      writeln ('Дочерний процесс: блокировка установлена');
      writeln ('Дочерний процесс: выход');
      halt (0);
    end;
  end;
  (* родительский процесс *)
  sleep (5);
  (* Выход, который автоматически снимет блокировку. *)
  writeln ('Родительский процесс: выход');
  halt (0);
end.
```

Вывод программы `lockit` может выглядеть примерно так:

```
Родительский процесс: блокировка установлена
Родительский процесс: выход
Дочерний процесс: блокировка установлена
Дочерний процесс: выход
```

Обратите внимание на порядок, в котором выводятся сообщения. Он показывает, что дочерний процесс не может наложить блокировку до тех пор, пока родительский процесс не завершит работу и не снимет тем самым блокировку, в противном случае сообщение `Дочерний процесс: блокировка установлена` появилось бы вторым, а не третьим. Этот пример показывает, что блокировка, наложенная родительским процессом, также затронула и дочерний, хотя блокируемые участки файла и не совсем совпадали. Другими словами, попытка блокировки завершится неудачей, даже если участок файла лишь частично перекрывает уже заблокированный. Эта программа иллюстрирует несколько интересных моментов. Во-первых, блокировка информации не наследуется при вызове `fork`; дочерний и родительский процессы в данном примере выполняют блокировку независимо друг от друга. Во-вторых, вызов `fcntl` не изменяет положение указателя чтения-записи файла – во время выполнения обоих процессов он указывает на начало файла. В-третьих, все связанные с процессом блокировки автоматически снимаются при его завершении.

Снятие блокировки при помощи вызова fcntl

Можно разблокировать участок файла, который был ранее заблокирован, присвоив при вызове переменной `l_type` значение `F_UNLK`. Снятие блокировки обычно используется через некоторое время после предыдущего запроса `fcntl`. Если еще какие-либо процессы собирались заблокировать освободившийся участок, то один из них прекратит ожидание и продолжит свою работу.

Если участок, с которого снимается блокировка, находится в середине большого заблокированного этим же процессом участка файла, то система создаст две блокировки меньшего размера, исключающие освобождаемый участок. Это означает, что при этом занимается еще одна ячейка в системной таблице блокировок. Вследствие этого запрос на снятие блокировки может завершиться неудачей из-за переполнения системной таблицы, хотя поначалу этого не видно.

Например, в предыдущей программе `lockit` родительский процесс снял блокировку в момент выхода из программы, но вместо этого он мог осуществить эту операцию при помощи следующего кода:

```
(* Родительский процесс снимает блокировку перед выходом *)
writeln('Родительский процесс: снятие блокировки');
my_lock.l_type := F_UNLCK;
fcntl(fd, F_SETLK, longint(@my_lock));
if linuxerror <> 0 then
begin
  perror('ошибка снятия блокировки в родительском процессе');
  halt(1);
end;
```

Задача об авиакомпании ACME Airlines

Теперь удастся разрешить конфликтную ситуацию в примере с авиакомпаний ACME Airlines. Для того, чтобы гарантировать целостность базы данных, нужно построить критический участок кода в программе `acmebook` следующим образом:

```
заблокировать соответствующий участок базы данных на запись

обновить участок базы данных

разблокировать участок базы данных
```

Если ни одна программа не обходит механизм блокировки, то запрос на блокировку гарантирует, что вызывающий процесс имеет исключительный доступ к критической части

базы данных. Запрос на снятие блокировки снова делает доступной эту область для общего использования. Выполнение любой конкурирующей копии программы acmebook, которая пытается получить доступ к соответствующей части базы данных, будет приостановлено на стадии попытки наложения блокировки.

Текст критического участка программы можно реализовать следующим образом:

```
(* Набросок процедуры обновления программы acmebook *)

var
  db_lock:flockrec;
.
.
.
(* Установить параметры блокировки *)
db_lock.l_type := F_WRLCK;
db_lock.l_whence := SEEK_SET;
db_lock.l_start := recstart;
db_lock.l_len := RECSIZE;
.
.
.
(* Заблокировать запись в базе, выполнение приостановится *)
(* если запись уже заблокирована *)
fcntl(fd, F_SETLKW, longint(@db_lock));
if linuxerror <> 0 then
  fatal('Ошибка блокировки');

(* Код для проверки и обновления данных о заказах *)
.
.
.
(* Освободить запись для использования другими процессами *)
db_lock.l_type := F_UNLCK;
fcntl(fd, F_SETLK, longint(@db_lock));
```

Проверка блокировки

При неудачной попытке программы установить блокировку, задав параметр F_SETLK в вызове fcntl, вызов установит значение переменной linuxerror равным Sys_EAGAIN или Sys_EACCESS (в спецификации XSI определены оба эти значения). Если блокировка уже существует, то с помощью команды F_GETLK можно определить процесс, установивший эту блокировку:

```
uses linux, stdio;
.
.
.
fcntl(fd, F_SETLK, longint(@alock));
if linuxerror <> 0 then
begin
  if (linuxerror = Sys EACCES) or (linuxerror = Sys_EAGAIN) then
  begin
    fcntl(fd, F_GETLK, longint(@b_lock));
    writeln(stderr, 'Запись заблокирована процессом ', b_lock.l_pid);
  end
  else
    perror('Ошибка блокировки');
end;
```

Клинч

Предположим, что два процесса, PA и PB, работают с одним файлом. Допустим, что

процесс PA блокирует участок файла SX, а процесс PB – не пересекающийся с ним участок SY. Пусть далее процесс PA попытается заблокировать участок SY при помощи команды F_SETLKW, а процесс PB попытается заблокировать участок SX, также используя команду F_SETLKW. Ни одна из этих попыток не будет успешной, так как процесс PA приостановит работу, ожидая, когда процесс PB освободит участок SY, а процесс PB также будет приостановлен в ожидании освобождения участка SX процессом PA. Если не произойдет вмешательства извне, то будет казаться, что два процесса обречены вечно находиться в этом «смертельном объятии».

Такая ситуация называется *клинчем* (deadlock) по очевидным причинам. Однако UNIX иногда предотвращает возникновение клинча. Если выполнение запроса F_SETLK приведет к очевидному возникновению клинча, то вызов завершается неудачей, и возвращается значение -1, а переменная linuxerror принимает значение Sys_EDEADLK. К сожалению, вызов fcntl может определять только клинч между двумя процессами, в то время как можно создать трехсторонний клинч.[1] Во избежание такой ситуации сложные приложения, использующие блокировки, должны всегда задавать предельное время ожидания.

Следующий пример поможет пояснить изложенное. В точке (*A*) программа блокирует с 0 по 9 байты файла locktest. Затем программа порождает дочерний процесс, который в точках, помеченных как (*B*) и (*C*), блокирует байты с 10 по 14 и пытается выполнить блокировку байтов с 0 по 9. Из-за того, что родительский процесс уже выполнил последнюю блокировку, работа дочернего будет приостановлена. В это время родительский процесс выполняет вызов sleep в течение 10 секунд. Предполагается, что этого времени достаточно, чтобы дочерний процесс выполнил два вызова, устанавливающие блокировку. После того, как родительский процесс продолжит работу, он пытается заблокировать байты с 10 по 14 в точке (*D*), которые уже были заблокированы дочерним процессом. В этой точке возникнет опасность клинча, и вызов fcntl завершится неудачей.

```
(* Программа deadlock - демонстрация клинча *)
uses linux, stdio;

var
  fd:longint;
  first_lock, second_lock:flockrec;
begin
  first_lock.l_type := F_WRLCK;
  first_lock.l_whence := SEEK_SET;
  first_lock.l_start := 0;
  first_lock.l_len := 10;
  second_lock.l_type := F_WRLCK;
  second_lock.l_whence := SEEK_SET;
  second_lock.l_start := 10;
  second_lock.l_len := 5;

  writeln(sizeof(flockrec));
  fd := fdopen ('locktest', Open_RDWR);

  fcntl (fd, F_SETLKW, longint(@first_lock));
  if linuxerror>0 then  (*A *)
    fatal ('A');

  writeln ('A: успешная блокировка (процесс ',getpid,')');

  case fork of
    -1:
```

[1] Клинч иногда является следствием гораздо более запутанной ситуации, если установлению блокировки препятствуют одновременно несколько других блокировок.

```
      (* ошибка *)
      fatal ('Ошибка вызова fork');
    0:
    begin
      (* дочерний процесс *)
      fcntl (fd, F_SETLKW, longint(@second_lock));
      if linuxerror>0 then   (*B *)
        fatal ('B');
      writeln ('B: успешная блокировка (процесс ',getpid,')');
      fcntl (fd, F_SETLKW, longint(@first_lock));
      if linuxerror>0 then   (*C *)
        fatal ('C');
      writeln ('C: успешная блокировка (процесс ',getpid,')');
      halt (0);
    end;
    else
    begin
      (* родительский процесс *)
      writeln ('Приостановка родительского процесса');
      sleep (10);
      fcntl (fd, F_SETLKW, longint(@second_lock));
      if linuxerror>0 then   (*D *)
        fatal ('D');
      writeln ('D: успешная блокировка (процесс ',getpid,')');
    end;
  end;
end.
```

Вот пример работы этой программы:

```
A: успешная блокировка (процесс 1410)
Приостановка родительского процесса
B: успешная блокировка (процесс 1411)
D: Deadlock situation detected/avoided
C: успешная блокировка (процесс 1411)
```

В данном случае попытка блокировки завершается неудачей в точке `(*D*)`, и процедура `perror` выводит соответствующее системное сообщение об ошибке. Обратите внимание, что после того, как родительский процесс завершит работу и его блокировки будут сняты, дочерний процесс сможет выполнить вторую блокировку.

Это пример использует процедуру `fatal`, которая была применена в предыдущих главах.

Упражнение 8.1. *Напишите процедуры, выполняющие те же действия, что и вызовы* `fdread` *и* `fdwrite`*, но которые завершатся неудачей, если уже установлена блокировка нужного участка файла. Измените аналог вызова* `fdread` *так, чтобы он блокировал читаемый участок. Блокировка должна сниматься после завершения вызова* `fdread`*.*

Упражнение 8.2. *Придумайте и реализуйте условную схему блокировок нумерованных логических записей файла. (Совет: можно блокировать участки файла вблизи максимально возможного смещения файла, даже если там нет данных. Блокировки в этом участке файла могут иметь особое значение, например, каждый байт может соответствовать определенной логической записи. Блокировка в этой области может также использоваться для установки различных флагов.)*

8.3. Дополнительные средства межпроцессного взаимодействия

8.3.1. Введение и основные понятия

ОС UNIX предлагает множество дополнительных механизмов межпроцессного

взаимодействия. Их наличие дает UNIX богатые возможности в области связи между процессами и позволяет разработчику использовать различные подходы при программировании многозадачных систем. Дополнительные средства межпроцессного взаимодействия, которые будут рассмотрены, можно разбить на следующие категории:

- передача сообщений;
- семафоры;
- разделяемая память.

Эти средства широко применяются и ведут свое начало от системы UNIX System V, поэтому их иногда называют *IPC System V*. Следует заметить, что вышеназванные дополнительные средства были определены в последних версиях стандарта POSIX.[1]

Ключи средств межпроцессного взаимодействия

Программный интерфейс всех трех средств IPC System V однороден, что отражает схожесть их реализации в ядре. Наиболее важным из общих свойств является *ключ* (key). Ключи – это числа, обозначающие объект межпроцессного взаимодействия в системе UNIX примерно так же, как имя файла обозначает файл. Другими словами, ключ позволяет ресурсу межпроцессного взаимодействия совместно использоваться несколькими процессами. Обозначаемый ключом объект может быть очередью сообщения, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип `TKey`, состав которого зависит от реализации и определяется в файле `ipc`.

Ключи не являются именами файлов и несут меньший смысл. Они должны выбираться осторожно во избежание конфликта между различными программами, в этом помогает применение дополнительной опции – «версии проекта». (Одна известная система управления базами данных использовала для ключа шестнадцатеричное значение типа 0xDB; плохое решение, так как такое же значение мог выбрать и другой разработчик.) В ОС UNIX существует простая библиотечная функция `ftok`, которая образует ключ по указанному файлу.

Описание

```
uses ipc;

Function ftok(Path:String; ID:char):TKey;
```

Эта процедура возвращает номер ключа на основе информации, связанной с файлом `path`. Параметр `id` также учитывается и обеспечивает еще один уровень уникальности – «версию проекта»; другими словами, для одного имени `path` будут получены разные ключи при разных значениях `id`. Процедура `ftok` не слишком удобна: например, если удалить файл, а затем создать другой с таким же именем, то возвращаемый после этого ключ будет другим. Она завершится неудачей и вернет значение `-1` и в случае, если файл `path` не существует. Процедуру `ftok` можно применять в приложениях, использующих функции межпроцессного взаимодействия для работы с определенными файлами или при применении для генерации ключа файла, являющегося постоянной и неотъемлемой частью приложения.

Операция get

Программа применяет ключ для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции вызываются при помощи операции `get`. Результатом операции `get` является его целочисленный *идентификатор* (facility identifier), который может использоваться при вызовах других процедур межпроцессного взаимодействия. Если продолжить аналогию с именами файлов, то операция `get` похожа на вызов `fdcreat` или `fdopen`, а идентификатор средства межпроцессного взаимодействия ведет себя подобно дескриптору файла. В действительности, в отличие от дескрипторов файла, идентификатор средства межпроцессного взаимодействия является уникальным. Различные процессы будут использовать одно и то же значение идентификатора для объекта межпроцессного

[1] В базовом документе POSIX 1003.1 средства *IPC System V* не вводятся.

взаимодействия.

В качестве примера рассмотрим вызов msgget для создания новой очереди сообщений (что представляет собой очередь сообщений, обсудим позже):

```
mqid := msgget(octal(0100), octal(0644) or IPC_CREAT or IPC_EXCL);
```

Первый аргумент вызова, msgget, является ключом очереди сообщений. В случае успеха процедура вернет неотрицательное значение в переменной mqid, которая служит идентификатором очереди сообщений. Соответствующие вызовы для семафоров и объектов разделяемой памяти называются соответственно semget и shmget.

Другие операции

Есть еще два типа операций, которые применимы к средствам межпроцессного взаимодействия. Во-первых, это операции управления, которые используются для опроса и изменения статуса объекта IPC, их функции выполняют вызовы msgctl, semctl и shmctl. Во-вторых, существуют операции, выполняющие основные функции IPC. Для каждого из средств межпроцессного взаимодействия существует набор операций, которые будут обсуждаться ниже в соответствующих пунктах. Например, есть две операции для работы с сообщениями: операция msgsnd помещает сообщение в очередь сообщений, а операция msgrcv считывает из нее сообщение.

Структуры данных статуса

При создании объекта межпроцессного взаимодействия система также создает *структуру статуса средства межпроцессного взаимодействия* (IPC facility status structure), содержащую всю управляющую информацию, связанную с объектом. Для сообщений, семафоров и разделяемой памяти существуют разные типы структуры статуса. Каждый тип содержит информацию, свойственную этому средству межпроцессного взаимодействия. Тем не менее все три типа структуры статуса содержат общую структуру прав доступа. Структура прав доступа tipc_perm содержит следующие элементы:

```
TIPC_Perm  =  record
  key  :  TKey;
  uid,          (* Действующий идентификатор пользователя *)
  gid,          (* Действующий идентификатор группы *)
  cuid,         (* Идентификатор пользователя создателя объекта *)
  cgid,         (* Идентификатор группы создателя объекта *)
  mode,         (* Права доступа *)
  seq  :  Word;
end;
```

Права доступа определяют, может ли пользователь выполнять «чтение» из объекта (получать информацию о нем) или «запись» в объект (работать с ним). Коды прав доступа образуются точно таким же образом, как и для файлов. Поэтому значение 0644 для элемента umode означает, что владелец может выполнить чтение и запись объекта, а другие пользователи – только чтение из него. Обратите внимание, что права доступа, заданные элементом mode, применяются в сочетании с действующими идентификаторами пользователя и группы (записанными в элементах uid и gid).[1] Очевидно также, что права на выполнение в данном случае не имеют значения. Как обычно, суперпользователь имеет неограниченные полномочия. В отличие от других конструкций UNIX, значение переменной umask пользователя не действует при создании средства межпроцессного взаимодействия.

8.3.2. Очереди сообщений

Начнем подробное рассмотрение средств межпроцессного взаимодействия с примитивов очередей сообщений.

В сущности, сообщение является просто последовательностью символов или байтов (необязательно заканчивающейся нулевым символом). Сообщения передаются между

[1] Более точно порядок разрешения доступа к объекту IPC описан в спецификации SUSV2.

процессами при помощи *очередей сообщений* (message queues), которые можно создавать или получать к ним доступ при помощи вызова `msgget`. После создания очереди процесс может помещать в нее сообщения при помощи вызова `msgsnd`, если он имеет соответствующие права доступа. Затем другой процесс может считать это сообщение при помощи примитива `msgrcv`, который извлекает сообщение из очереди. Таким образом, обработка сообщений аналогична обмену данными при помощи вызовов чтения и записи для каналов (рассмотренном в разделе 7.1.2.).

Функция `msgget` определяется следующим образом:

Описание

```
uses ipc;

Function msgget(key:TKey; permflags:longint):longint;
```

Этот вызов лучше всего представить как аналог вызова `fdopen` или `fdcreat`. Как уже упоминалось в разделе 8.3.1, параметр `key`, который, в сущности, является простым числом, идентифицирует очередь сообщений в системе. В случае успешного вызова, после создания новой очереди или доступа к уже существующей, вызов `msgget` вернет ненулевое целое значение, которое называется *идентификатором очереди сообщений* (message queue identifier).

Параметр `permflags` указывает выполняемое вызовом `msgget` действие, которое задается при помощи двух констант, определенных в файле `ipc`; они могут использоваться по отдельности или объединяться при помощи операции побитового ИЛИ:

`IPC_CREAT` При задании этого флага вызов `msgget` создает новую очередь сообщений для данного значения, если она еще не существует. Если продолжить аналогию с файлами, то при задании этого флага вызов `msgget` выполняется в соответствии с вызовом `creat`, хотя очередь сообщений и не будет «перезаписана», если она уже существует. Если же флаг `IPC_CREAT` не установлен и очередь с этим ключом существует, то вызов `msgget` вернет идентификатор существующей очереди сообщений

`IPC_EXCL` Если установлен этот флаг и флаг `IPC_CREAT`, то вызов предназначается только для создания очереди сообщений. Поэтому, если очередь с ключом `key` уже существует, то вызов `msgget` завершится неудачей и вернет значение -1. Переменная `linuxerror` будет при этом содержать значение `Sys_EEXIST`

При создании очереди сообщений младшие девять бит переменной `permflags` используются для задания прав доступа к очереди сообщений аналогично коду доступа к файлу. Они хранятся в структуре `tipc_perm`, создаваемой одновременно с самой очередью.

Теперь можно вернуться к примеру из раздела 8.3.1.

```
mqid := msgget(octal(0100), octal(0644) or IPC_CREAT or IPC_EXCL);
```

Этот вызов предназначен для создания (и только создания) очереди сообщений для значения ключа равного `octal(0100)`. В случае успешного завершения вызова очередь будет иметь код доступа `octal(0644)`. Этот код интерпретируется таким же образом, как и код доступа к файлу, обозначая, что создатель очереди может отправлять и принимать сообщения, а члены его группы и все остальные могут выполнять только чтение. При необходимости для изменения прав доступа или владельца очереди может использоваться вызов `msgctl`.

Работа с очередью сообщений: примитивы msgsnd и msgrcv

После создания очереди сообщений для работы с ней могут использоваться два следующих примитива:

Описание

```
uses ipc;
```

```
Function msgsnd(mqid:longint; message:PMSGBuf; size:longint;
                msg_type:longint; flags:longint): Boolean;
Function msgrcv(mqid:longint; message:PMSGBuf; size:longint;
                msg_type:longint; flags:longint): Boolean;
```

Первый из вызовов, msgsnd, используется для добавления сообщения в очередь, обозначенную идентификатором mqid.

Сообщение содержится в структуре message – шаблоне, определенном пользователем и имеющем следующую форму:

```
PMSGbuf=^TMSGbuf;
TMSGbuf=record
  mtype:longint;                  (* Тип сообщения *)
  mtext:array [0..SOMEVALUE-1] of char;    (* Текст сообщения *)
end;
```

Значение поля mtype может использоваться программистом для разбиения сообщений на категории. При этом значимыми являются только положительные значения; отрицательные или нулевые не могут использоваться (это будет видно из дальнейшего описания операций передачи сообщений). Массив mtext служит для хранения данных сообщения (постоянная SOMEVALUE выбрана совершенно произвольно). Длина посылаемого сообщения задается параметром size вызова msgsnd и может быть в диапазоне от нуля до меньшего из двух значений SOMEVALUE и максимального размера сообщения, определенного в системе.

Параметр flsgs вызова msgsnd может нести только один флаг: IPC_NOWAIT. При неустановленном параметре IPC_NOWAIT вызывающий процесс приостановит работу, если для посылки сообщения недостаточно системных ресурсов. На практике это произойдет, если полная длина сообщений в очереди превысит максимум, заданный для очереди или всей системы. Если флаг IPC_NOWAIT установлен, тогда при невозможности послать сообщение возврат из вызова произойдет немедленно. Возвращаемое значение будет равно -1, и переменная ipcerror будет иметь значение Sys_EAGAIN, означающее необходимость повторения попытки.

Вызов msgsnd также может завершиться неудачей из-за установленных прав доступа. Например, если ни действующий идентификатор пользователя, ни действующий идентификатор группы процесса не связаны с очередью, и установлен код доступа к очереди octal(0660), то вызов msgsnd для этой очереди завершится неудачей. Переменная ipcerror получит значение Sys_EACCES.

Перейдем теперь к чтению сообщений. Для чтения из очереди, заданной идентификатором mqid, используется вызов msgrcv. Чтение разрешено, если процесс имеет права доступа к очереди на чтение. Успешное чтение сообщения приводит к удалению его из очереди.

На этот раз переменная message используется для хранения полученного сообщения, а параметр size задает максимальную длину сообщений, которые могут находиться в этой структуре. Успешный вызов возвращает длину полученного сообщения.

Параметр msg_type определяет тип принимаемого сообщения, он помогает выбрать нужное из находящихся в очереди сообщений. Если параметр msg_type равен нулю, из очереди считывается первое сообщение, то есть то, которое было послано первым. При ненулевом положительном значении параметра msg_type считывается первое сообщение из очереди с заданным типом сообщения. Например, если очередь содержит сообщения со значениями mtype 999, 5 и 1, а параметр msg_type в вызове msgrcv имеет значение 5, то считывается сообщение типа 5. И, наконец, если параметр msg_type имеет ненулевое отрицательное значение, то считывается первое сообщение с наименьшим значением mtype, которое меньше или равно модулю параметра msg_type. Этот алгоритм кажется сложным, но выражает простое правило: если вернуться к нашему предыдущему примеру с тремя

сообщениями со значениями mtype 999, 5 и 1, то при значении параметра msg_type в вызове msgrcv равном -999 и троекратном вызове сообщения будут получены в порядке 1, 5, 999.

Последний параметр flags содержит управляющую информацию. В этом параметре могут быть независимо установлены два флага – IPS_NOWAIT и MSG_NOERROR. Флаг IPC_NOWAIT имеет обычный смысл – если он не задан, то процесс будет приостановлен при отсутствии в очереди подходящих сообщений, и возврат из вызова произойдет после поступления сообщения соответствующего типа. Если же этот флаг установлен, то возврат из вызова при любых обстоятельствах произойдет немедленно.

При установленном флаге MSG_NOERROR сообщение будет усечено, если его длина больше, чем size байт, без этого флага попытка чтения длинного сообщения приводит к неудаче вызова msgrcv. К сожалению, узнать о том, что усечение имело место, невозможно.

Этот раздел может показаться сложным: формулировка средств межпроцессного взаимодействия несколько не соответствует по своей сложности и стилю природе ОС UNIX. В действительности же процедуры передачи сообщений просты в применении и имеют множество потенциальных применений, что попробуем продемонстрировать на следующем примере.

Пример передачи сообщений: очередь с приоритетами

В этом разделе разработаем простое приложение для передачи сообщений. Его целью является реализация очереди, в которой для каждого элемента может быть задан приоритет. Серверный процесс будет выбирать элементы из очереди и обрабатывать их каким-либо образом. Например, элементы очереди могут быть именами файлов, а серверный процесс может копировать их на принтер. Этот пример аналогичен примеру использования FIFO из раздела 7.2.1.

Отправной точкой будет следующий заголовочный файл q.inc:

```
(* q.inc - заголовок для примера очереди сообщений *)

const
  QKEY:tkey=(1 shl 6) + 5{0105};        (* ключ очереди *)
  QPERM=(6 shl 6) + (6 shl 3){0660};    (* права доступа *)
  MAXOBN=50;                            (* макс. длина имени объекта *)
  MAXPRIOR=10;                          (* максимальный приоритет *)

type
  q_entry=record
    mtype:longint;
    mtext:array [0..MAXOBN] of char;
  end;
  pq_entry=^q_entry;
```

Определение QKEY задает значение ключа, которое будет обозначать очередь сообщений в системе. Определение QPERM устанавливает связанные с очередью права доступа. Так как код доступа равен octal(0660), то владелец очереди и члены его группы смогут выполнять чтение и запись. Как увидим позже, определения MAXOBN и MAXPRIOR будут налагать ограничения на сообщения, помещаемые в очередь. Последняя часть этого включаемого файла содержит определение структуры q_entry. Структуры этого типа будут использоваться в качестве сообщений, передаваемых и принимаемых следующими процедурами.

Первая рассматриваемая процедура называется enter, она помещает в очередь имя объекта, заканчивающееся нулевым символом, и имеет следующую форму:

```
{$i q.inc}

(* Процедура enter - поместить объект в очередь *)
function enter (objname:string;priority:longint):boolean;
var
```

```
  len, s_qid:longint;
  s_entry:q_entry;      (* структура для хранения сообщений *)
begin
  (* Проверка длины имени и уровня приоритета *)
  len := length (objname);
  if len > MAXOBN then
  begin
    warn ('слишком длинное имя');
    enter:=false;
    exit;
  end;
  if (priority > MAXPRIOR) or (priority < 0) then
  begin
    warn ('недопустимый уровень приоритета');
    enter:=false;
    exit;
  end;

  (* Инициализация очереди сообщений, если это необходимо *)
  s_qid := init_queue;
  if s_qid = -1 then
  begin
    enter:=false;
    exit;
  end;
  (* Инициализация структуры переменной s_entry *)
  s_entry.mtype := priority;
  strlcopy (s_entry.mtext, @objname[1], MAXOBN);
  (* Посылаем сообщение, выполнив ожидание, если это необходимо *)
  if not msgsnd (s_qid, @s_entry, len, 0) then
  begin
    perror ('Ошибка вызова msgsnd');
    enter:=false;
    exit;
  end
  else
    enter:=true;
end;
```

Первое действие, выполняемое процедурой `enter`, заключается в проверке длины имени объекта и уровня приоритета. Обратите внимание на то, что минимальное значение переменной приоритета `priority` равно 1, так как нулевое значение приведет к неудачному завершению вызова `msgsnd`. Затем процедура `enter` «открывает» очередь, вызывая процедуру `init_queue`, реализацию которой приведем позже.

После завершения этих действий процедура формирует сообщение и пытается послать его при помощи вызова `msgsnd`. Здесь для хранения сообщения использована структура `s_entry` типа `q_entry`, и последний параметр вызова `msgsnd` равен нулю. Это означает, что система приостановит выполнение текущего процесса, если очередь заполнена (так как не задан флаг `IPC_NOWAIT`).

Процедура `enter` сообщает о возникших проблемах при помощи функции `warn` или библиотечной функции `perror`. Для простоты функция `warn` реализована следующим образом:

```
procedure warn (s:pchar);
begin
  writeln(stderr, 'Предупреждение: ', s);
end;
```

В реальных системах функция `warn` должна записывать сообщения в специальный

файл протокола.

Назначение функции `init_queue` очевидно. Она инициализирует идентификатор очереди сообщений или возвращает идентификатор очереди сообщений, который с ней уже связан.

```
{$i q.inc}

(* Инициализация очереди - получить идентификатор очереди *)
function init_queue:longint;
var
  queue_id:longint;
begin
  (* Попытка создания или открытия очереди сообщений *)
  queue_id := msgget (QKEY, IPC_CREAT or QPERM);
  if queue_id = -1 then
    perror ('Ошибка вызова msgget');
  init_queue:=queue_id;
end;
```

Следующая процедура, `serve`, используется серверным процессом для получения сообщений из очереди и противоположна процедуре `enter`.

```
{$i q.inc}

(* Процедура serve - принимает и обрабатывает сообщение обслуживает
 * объект очереди с наивысшим приоритетом
 *)
function serve:integer;
var
  r_qid:longint;
  r_entry:q_entry;
begin
  (* Инициализация очереди сообщений, если это необходимо *)
  r_qid := init_queue;
  if r_qid = -1 then
  begin
    serve:=-1;
    exit;
  end;

  (* Получить и обработать следующее сообщение *)
  while true do
  begin
    if not msgrcv(r_qid, @r_entry, MAXOBN, -1*MAXPRIOR, MSG_NOERROR) then
    begin
      perror ('Ошибка вызова msgrcv');
      serve:=-1;
      exit;
    end
    else
    begin
      (* Обработать имя объекта *)
      proc_obj (@r_entry);
    end;
  end;
end;
```

Обратите внимание на вызов `msgrcv`. Так как в качестве параметра типа задано отрицательное значение `(-1 * MAXPRIOR)`, то система вначале проверяет очередь на наличие сообщений со значением `mtype` равным 1, затем равным 2 и так далее, до значения `MAXPRIOR` включительно. Другими словами, сообщения с наименьшим номером будут иметь

наивысший приоритет. Процедура `proc_obj` работает с объектом. Для системы печати она может просто копировать файл на принтер.

Две следующих простых программы демонстрируют взаимодействие этих процедур: программа `etest` помещает элемент в очередь, а программа `stest` обрабатывает его (в действительности она всего лишь выводит содержимое и тип сообщения).

Программа etest

```
(* Программа etest - ввод имен объектов в очередь. *)
{$mode objfpc}
uses ipc,linux,stdio,sysutils;
{$i q.inc}


var
  priority:longint;
begin
  if paramcount <> 2 then
  begin
    writeln(stderr, 'Применение: ',paramstr(0),' имя приоритет');
    halt (1);
  end;
  try
    priority:=strtoint(paramstr(2));
  except
    on e:econverterror do
    begin
      warn ('Нечисловой приоритет');
      halt (2);
    end;
  end;
  if (priority <= 0) or (priority > MAXPRIOR) then
  begin
    warn ('Недопустимый приоритет');
    halt (2);
  end;
  if not enter (paramstr(1), priority) then
  begin
    warn ('Ошибка в процедуре enter');
    halt (3);
  end;
  halt (0);
end.
```

Программа stest

```
(* Программа stest - простой сервер для очереди *)
uses ipc,linux,stdio,sysutils;
{$i q.inc}


function proc_obj (msg:pq_entry):integer;
begin
  writeln(#$a'Приоритет: ',msg^.mtype,' имя: ',msg^.mtext);
end;


var
  pid:longint;
begin
  pid := fork;
```

```
  case pid of
    0:                  (* дочерний процесс *)
      serve;
    -1:                  (* сервер не существует *)
      warn ('Не удалось запустить сервер');
    else
      writeln('Серверный процесс с идентификатором ', pid);
  end;
  if pid <> -1 then
    halt (0)
  else
    halt (1);
end.
```

Ниже следует пример использования этих двух простых программ. Перед запуском программы `stest` в очередь вводятся четыре простых сообщения при помощи программы `etest`. Обратите внимание на порядок, в котором выводятся сообщения:

```
$ etest objname1 3
$ etest objname2 4
$ etest objname3 1
$ etest objname4 9
$ stest
Серверный процесс с идентификатором 2545
$
Приоритет 1 имя objname3

Приоритет 3 имя objname1

Приоритет 4 имя objname2

Приоритет 9 имя objname4
```

Упражнение 8.3. *Измените процедуры* `enter` *и* `serve` *так, чтобы можно было посылать серверу управляющие сообщения. Зарезервируйте для таких сообщений единичный тип сообщения (как это повлияет на расстановку приоритетов?). Реализуйте следующие возможности:*

1. Остановка сервера.

2. Стирание всех сообщений из очереди.

3. Стирание сообщений с заданным уровнем приоритета.

Системный вызов msgctl

Процедура `msgctl` служит трем целям: она позволяет процессу получать информацию о статусе очереди сообщений, изменять некоторые из связанных с очередью ограничений или удалять очередь из системы.

Описание

```
uses ipc;

Function msgctl(mqid:longint; cmd:longint; msg_stat:PMSQid_ds): Boolean;
```

Переменная `mqid` должна быть допустимым идентификатором очереди. Пропуская пока параметр `cmd`, обратимся к третьему параметру `msg_stat`, который содержит адрес структуры `TMSQid_ds`. Эта структура определяется в файле `ipc` и содержит следующие элементы:

```
PMSQid_ds = ^TMSQid_ds;
TMSQid_ds = record
  msg_perm   : TIPC_perm; (* Владелец/права доступа *)
  msg_first  : PMsg;
  msg_last   : PMsg;
  msg_stime  : Longint;   (* Время посл. вызова msgsnd *)
```

```
  msg_rtime  : Longint;    (* Время посл. вызова msgrcv *)
  msg_ctime  : Longint;    (* Время посл. изменения *)
  wwait      : Pointer;
  rwait      : pointer;
  msg_cbytes : word;
  msg_qnum   : word;       (* Число сообщений в очереди *)
  msg_qbytes : word;       (* Макс. число байтов в очереди *)
  msg_lspid  : word;       (* Идентификатор процесса,
                              последним вызвавшего msgsnd *)
  msg_lrpid  : word;       (* Идентификатор процесса,
                              последним вызвавшего msgrcv *)
end;
```

Структура `TIPC_perm`, с которой уже встречались ранее, содержит связанную с очередью информацию о владельце и правах доступа. Переменные `msg_stime`, `msg_rtime`, `msg_ctime` содержат число секунд, прошедшее с 00:00 по гринвичскому времени 1 января 1970 г. (Следующий пример покажет, как можно преобразовать такие значения в удобочитаемый формат.)

Параметр `cmd` в вызове `msgctl` сообщает системе, какую операцию она должна выполнить. Существуют три возможных значения этого параметра, каждое из которых может быть применено к одному из трех средств межпроцессного взаимодействия. Они обозначаются следующими константами, определенными в файле `ipc`.

`IPC_STAT` — Сообщает системе, что нужно поместить информацию о статусе объекта в структуру `msg_stat`

`IPC_SET` — Используется для задания значений управляющих параметров очереди сообщений, содержащихся в структуре `msg_stat`. При этом могут быть изменены только следующие поля:

```
msq_stat.msg_perm.uid
msq_stat.msg_perm.gid
msq_stat.msg_perm.mode
msq_stat.msg_qbytes
```

Операция `IPC_SET` завершится успехом только в случае ее выполнения суперпользователем или текущим владельцем очереди, заданным параметром `msq_stat.msg_perm.uid`. Кроме того, только суперпользователь может увеличивать значение `msg_qbytes` – максимальное количество байтов, которое может находиться в очереди

`IPC_RMID` — Эта операция удаляет очередь сообщений из системы. Она также может быть выполнена только суперпользователем или владельцем очереди. Если параметр `command` принимает значение `IPC_RMID`, to параметр `msg_stat` задается равным `nil`

Следующий пример, программа `show_msg`, выводит часть информации о статусе объекта очереди сообщений. Программа должна вызываться так:

```
$ show_msg значение_ключа
```

Программа `show_msg` использует библиотечную процедуру `ctime` для преобразования значений структуры `time_t` в привычную запись. (Процедура `ctime` и другие функции для работы с временными значениями будут обсуждаться в главе 12.) Текст программы `show_msg`:

```
(* Программа showmsg - выводит данные об очереди сообщений *)
{$mode objfpc}
uses ipc,stdio,sysutils;

procedure mqstat_print(mkey:tkey; msq_id:longint; mstat:pmsqid_ds);
begin
  writeln (#$a'Ключ ', mkey, ', msg_qid ', msq_id, #$a);
```

```
  writeln(mstat^.msg_qnum, ' сообщений в очереди'#$a);

  writeln('Последнее сообщение послано процессом ', mstat^.msg_lspid, ' в ',
         ctime(mstat^.msg_stime));
  writeln('Последнее сообщение принято процессом ', mstat^.msg_lrpid, ' в ',
         ctime(mstat^.msg_rtime));
end;


var
  mkey:tkey;
  msq_id:longint;
  msq_status:tmsqid_ds;
begin
  if paramcount<>1 then
  begin
    writeln(stderr, 'Применение: showmsg значение_ключа');
    halt(1);
  end;

  (* Получаем идентификатор очереди сообщений *)
  try
    mkey:=tkey(strtoint(paramstr(1)));
  except
    on e:econverterror do
    begin
      writeln(stderr, 'Нечисловой идентификатор очереди сообщений');
      halt (2);
    end;
  end;

  msq_id := msgget(mkey, 0);
  if msq_id = -1 then
  begin
    perror('Ошибка вызова msgget');
    halt(2);
  end;

  (* Получаем информацию о статусе *)
  if not msgctl(msq_id, IPC_STAT, @msq_status) then
  begin
    perror('Ошибка вызова msgctl');
    halt(3);
  end;

  (* Выводим информацию о статусе *)
  mqstat_print(mkey, msq_id, @msq_status);
  halt(0);
end.
```

Упражнение 8.4. *Измените процедуру* `show_msg` *так, чтобы она выводила информацию о владельце и правах доступа очереди сообщений.*

Упражнение 8.5. *Взяв за основу программу* `chmod`*, напишите программу* `msg_chmod`*, которая изменяет связанные с очередью права доступа. Очередь сообщений также должна указываться значением ее ключа.*

8.3.3. Семафоры

Семафор как теоретическая конструкция

В информатике понятие *семафор* (semaphore) был впервые введено голландским теоретиком Э.В. Дейкстрой (E.W. Dijkstra) для решения задач синхронизации процессов. Семафор *sem* может рассматриваться как целочисленная переменная, для которой определены следующие операции:

```
p(sem) или wait (sem)

if sem<>0 then
  уменьшить sem на единицу
else
  ждать, пока sem не станет ненулевым, затем вычесть единицу

v(sem) или signal(sem)

увеличить sem на единицу
if очередь ожидающих процессов не пуста then
  продолжить выполнение первого процесса в очереди ожидания
```

Обратите внимание, что обозначения `p` и `v` происходят от голландских терминов для понятий *ожидания* (wait) и *сигнализации* (signal), причем последнее понятие не следует путать с обычными сигналами UNIX.

Действия проверки и установки в обеих операциях должны составлять одно атомарное действие, чтобы только один процесс мог изменять семафор *sem* в каждый момент времени.

Формально удобство семафоров заключается в том, что утверждение

```
(начальное значение семафора
+ число операций v
- число завершившихся операций p) >= 0
```

всегда истинно. Это – *инвариант семафора* (semaphore invariant). Теоретикам нравятся такие инварианты, так как они делают возможным систематическое и строгое доказательство правильности программ.

Семафоры могут использоваться несколькими способами. Наиболее простой из них заключается в обеспечении *взаимного исключения* (mutual exclusion), когда только один процесс может выполнять определенный участок кода одновременно. Рассмотрим схему следующей программы:

```
p(sem);

  какие-либо действия

v(sem);
```

Предположим далее, что начальное значение семафора *sem* равно единице. Из инварианта семафора можно увидеть, что:

```
(число завершенных операций p -
  число завершенных операций v) <= начального значения семафора
```

или:

```
(число завершенных операций p -
  число завершенных операций v) <= 1
```

Другими словами, в каждый момент времени только один процесс может выполнять группу операторов, заключенных между определенными операциями `p` и `v`. Такая область программы часто называется *критическим участком* (critical section).

Реализация семафоров в ОС UNIX основана на этой теоретической идее, хотя в действительности предлагаемые средства являются более общими (и, возможно, чрезмерно сложными). Вначале рассмотрим процедуры `semget` и `semctl`.

Системный вызов semget

Описание

```
uses ipc;

Function semget(key:Tkey; nsems:longint; permflags:longint):longint;
```

Вызов `semget` аналогичен вызову `msgget`. Дополнительный параметр `nsems` задает требуемое число семафоров в наборе семафоров; это важный момент – семафорные операции в System V IPC приспособлены для работы с наборами семафоров, а не с отдельными объектами семафоров. На рис. 8.2 показан набор семафоров. Ниже увидим, что использование целого набора семафоров усложняет интерфейс процедур работы с семафорами.

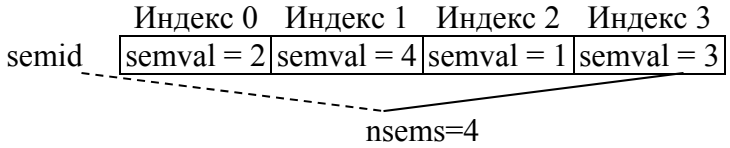


Рис.8.2. Набор семафоров

Значение, возвращаемое в результате успешного вызова `semget`, является *идентификатором набора семафоров* (semaphore set identifier), который ведет себя почти так же, как идентификатор очереди сообщений. Идентификатор набора семафоров обозначен на рис. 8.2 как `semid`. Следуя обычной практике, индекс семафора в наборе может принимать значения от 0 до `nsems-1`.

С каждым семафором в наборе связаны следующие значения:

semval Значение семафора, положительное целое число. Устанавливается при помощи системных вызовов работы с семафорами, то есть к значениям семафоров нельзя получить прямой доступ из программы, как к другим объектам данных

sempid Идентификатор процесса, который последним работал с семафором

semcnt Число процессов, ожидающих увеличения значения семафора

semzcnt Число процессов, ожидающих обнуления значения семафора

Системный вызов semctl

Описание

```
uses ipc;

Function semctl(semid:longint; sem_num:longint; command:longint;
                var ctl_arg:tsemun):longint;
```

Из определения видно, что функция `semctl` намного сложнее, чем `msgctl`. Параметр `semid` должен быть допустимым идентификатором семафора, возвращенным вызовом `semget`. Параметр `command` имеет тот же смысл, что и в вызове `msgctl`, – задает требуемую команду. Команды распадаются на три категории: стандартные команды управления средством межпроцессного взаимодействия (такие как `IPC_STAT`); команды, которые воздействуют только на один семафор; и команды, действующие на весь набор семафоров. Все доступные команды приведены в табл. 8.1.

Таблица 8.1. Коды функций вызова semctl

Стандартные функции межпроцессного взаимодействия	
`IPC_STAT`	Поместить информацию о статусе в поле `ctl_arg.stat`
`IPC_SET`	Установить данные о владельце/правах доступа
`IPC_RMID`	Удалить набор семафоров из системы
Операции над одиночными семафорами (относятся к семафору `sem_num`, значение возвращается вызовом `semctl`)	
`GETVAL`	Вернуть значение семафора (то есть `setval`)

SETVAL	Установить значение семафора равным ctl_arg.val
GETPID	Вернуть значение sempid
GETNCNT	Вернуть semncnt (см. выше)
GETZCNT	Вернуть semzcnt (см. выше)
Операции над всеми семафорами	
GETALL	Поместить все значения setval в массив ctl_arg.array
SETALL	Установить все значения setval из массива ctl_arg.array

Параметр sem_num используется со второй группой возможных операций вызова semctl для задания определенного семафора. Последний параметр ctl_arg является объединением (записью с вариантами), определенным следующим образом:

```
PSEMun = ^TSEMun;
TSEMun = record
  case longint of
    0 : (val    : longint);
    1 : (buf    : PSEMid_ds);
    2 : (arr    : PWord);
    3 : (padbuf : PSeminfo);
    4 : (padpad : pointer);
  end;
```

Каждый элемент объединения представляет некоторый тип значения, передаваемого вызову semctl при выполнении определенной команды. Например, если значение command равно SETVAL, то будет использоваться элемент ctl_arg.val.

Одно из важных применений функции setval заключается в установке начальных значений семафоров, так как вызов semget не позволяет процессу сделать это. Приведенная в качестве примера функция initsem может использоваться для создания одиночного семафора и получения связанного с ним идентификатора набора семафоров. После создания семафора (если семафор еще не существовал) функция semctl присваивает ему начальное значение, равное единице.

```
{$i pv.inc}


(* Функция initsem - инициализация семафора *)
function initsem(semkey:tkey):longint;
var
  status, semid:longint;
  arg:tsemun;
begin
  status := 0;
  semid := semget (semkey, 1,
                   SEMPERM or IPC_CREAT or IPC_EXCL);
  if semid = -1 then
  begin
    if ipcerror = Sys_EEXIST then
      semid := semget (semkey, 1, 0);
  end
  else
    (* если семафор создается ... *)
  begin
    arg.val := 1;
    status := semctl (semid, 0, SETVAL, arg);
  end;
  if (semid = -1) or (status = -1) then
  begin
    perror ('ошибка вызова initsem');
    initsem:=-1;
```

```
    exit;
  end;
  (* Все в порядке *)
  initsem:=semid;
end;
```

Включаемый файл `pv.inc` содержит следующие определения:

```
(* Заголовочный файл для примера работы с семафорами *)
const
  SEMPERM=6 shl 6{0600};
```

Функция `initsem` будет использована в примере следующего раздела.

Операции над семафорами: вызов semop

Вызов `semop` выполняет основные операции над семафорами.

Описание

```
uses ipc;

Function semop(semid:longint;op_array:pointer;num_ops:cardinal):Boolean;
```

Переменная `semid` является идентификатором набора семафоров, полученным с помощью вызова `semget`. Параметр `op_array` является массивом структур `TSEMbuf`, определенных в файле `ipc`. Каждая структура `TSEMbuf` содержит описание операций, выполняемых над семафором.

И снова основной акцент делается на операции с наборами семафоров, при этом функция `semop` позволяет выполнять группу операций как атомарную операцию. Это означает, что пока не появится возможность одновременного выполнения всех операций с отдельными семафорами набора, не будет выполнена ни одна из этих операций. Если не указано обратного, процесс приостановит работу до тех пор, пока он не сможет выполнить все операции сразу.

Рассмотрим структуру `TSEMbuf`. Она включает в себя следующие элементы:

```
TSEMbuf=record
  sem_num : word;
  sem_op  : integer;
  sem_flg : integer;
end;
```

Поле `sem_num` содержит индекс семафора в наборе. Если, например, набор содержит всего один элемент, то значение `sem_num` должно быть равно нулю. Поле `sem_op` содержит целое число со знаком, значение которого сообщает функции `semop`, что необходимо сделать. При этом возможны три случая:

Случай 1: отрицательное значение sem_op

Это обобщенная форма команды для работы с семафорами `p()`, которая обсуждалась ранее. Действие функции `semop` можно описать при помощи псевдокода следующим образом (обратите внимание, что *ABS()* обозначает модуль переменной):

```
if semval >= ABS(sem_op) then
begin
  semval := semval - ABS(sem_op)
end
else
begin
  if (sem_flg and IPC_NOWAIT) <> 0 then
    немедленно вернуть -1
  else
  behin
    ждать, пока semval не станет больше или равно ABS(sem_op)
    затем, как и выше, вычесть ABS(sem_op)
  end;
end;
```

Основная идея заключается в том, что функция semop вначале проверяет значение semval, связанное с семафором sem_num. Если значение semval достаточно велико, то оно сразу уменьшается на указанную величину. В противном случае процесс будет ждать, пока значение semval не станет достаточно большим. Тем не менее, если в переменной sem_flg установлен флаг IPC_NOWAIT, то возврат из вызова sem_op произойдет немедленно, и переменная ipcerror будет содержать код ошибки Sys_EAGAIN.

Случай 2: положительное значение sem_op

Это соответствует традиционной операции v(). Значение переменной sem_op просто прибавляется к соответствующему значению semval. Если есть процессы, ожидающие изменения значения этого семафора, то они могут продолжить выполнение, если новое значение семафора удовлетворит их условия.

Случай 3: нулевое значение sem_op

В этом случае вызов sem_op будет ждать, пока значение семафора не станет равным нулю; значение semval этим вызовом не будет изменяться. Если в переменной sem_flg установлен флаг IPC_NOWAIT, а значение semval еще не равно нулю, то функция semop сразу же вернет сообщение об ошибке.

Флаг SEM_UNDO

Это еще один флаг, который может быть установлен в элементе sem_flg структуры sembuf. Он сообщает системе, что нужно автоматически «отменить» эту операцию после завершения процесса. Для отслеживания всей последовательности таких операций система поддерживает для семафора целочисленную переменную semadj. Важно понимать, что переменная semadj связана с процессами, и для разных процессов один и тот же семафор будет иметь различные значения semadj. Если при выполнении операции semop установлен флаг SEM_UNDO, то значение переменной sem_num просто вычитается из значения semadj. При этом важен знак переменной sem_num: значение semadj уменьшается, если значение sem_num положительное, и увеличивается, если оно отрицательное. После выхода из процесса система прибавляет все значения semadj к соответствующим семафорам и, таким образом, сводит на нет эффект от всех вызовов semop. В общем случае флаг SEM_UNDO должен быть всегда установлен, кроме тех случаев, когда значения, устанавливаемые процессом, должны сохраняться после завершения процесса.

Пример работы с семафорами

Теперь продолжим пример, который начали с процедуры initsem. Он содержит две процедуры p() и v(), реализующие традиционные операции над семафорами. Сначала рассмотрим p():

```
{$i pv.inc}


(* Процедура p.pas - операция p для семафора *)
function p (semid:longint):longint;
var
  p_buf:tsembuf;
begin
  p_buf.sem_num := 0;
  p_buf.sem_op := -1;
  p_buf.sem_flg := SEM_UNDO;
  if not semop (semid, @p_buf, 1) then
  begin
    perror ('ошибка операции p(semid)');
    halt (1);
  end;
  p:=0;
end;
```

Обратите внимание на то, что здесь использован флаг SEM_UNDO. Теперь рассмотрим текст процедуры v().

```
{$i pv.inc}


(* Процедура v.pas - операция v для семафора *)
function v (semid:longint):longint;
var
  v_buf:tsembuf;
begin
  v_buf.sem_num := 0;
  v_buf.sem_op := 1;
  v_buf.sem_flg := SEM_UNDO;
  if not semop (semid, @v_buf, 1) then
  begin
    perror ('Ошибка операции v(semid)');
    halt (1);
  end;
  v:=0;
end;
```

Можно продемонстрировать использование этих довольно простых процедур для реализации взаимного исключения. Рассмотрим следующую программу:

```
(* Программа testsem - проверка процедур работы с семафорами *)
uses ipc,stdio,linux;
{$i pv.inc}


procedure handlesem (skey:tkey);
var
  semid, pid:longint;
begin
  pid := getpid;

  semid := initsem (skey);
  if semid < 0 then
    halt (1);

  writeln (#$a'Процесс ',pid,' перед критическим участком');
  p (semid);
  writeln ('Процесс ',pid,' выполняет критический участок');

  (* В реальной программе здесь выполняется нечто осмысленное *)
  sleep (10);

  writeln ('Процесс ',pid,' покинул критический участок');
  v (semid);
  writeln ('Процесс ',pid,' завершает работу');

  halt (0);
end;


const
  semkey:tkey = $200;
var
  i:integer;
begin
```

```
  for i := 1 to 3 do
    if fork = 0 then
      handlesem (semkey);
end.
```

Программа `testsem` порождает три дочерних процесса, которые используют вызовы `p()` и `v()` для того, чтобы в каждый момент времени только один из них выполнял критический участок. Запуск программы `testsem` может дать следующий результат:

```
Процесс 799 перед критическим участком
Процесс 799 выполняет критический участок
Процесс 800 перед критическим участком
Процесс 801 перед критическим участком
Процесс 799 покинул критический участок
Процесс 801 выполняет критический участок
Процесс 799 завершает работу
Процесс 801 покинул критический участок
Процесс 801 завершает работу
Процесс 800 выполняет критический участок
Процесс 800 покинул критический участок
Процесс 800 завершает работу
```

8.3.4. Разделяемая память

Операции с разделяемой памятью позволяют двум и более процессам совместно использовать область физической памяти (общеизвестно, что обычно области данных любых двух программ совершенно отделены друг от друга). Чаще всего разделяемая память является наиболее производительным механизмом межпроцессного взаимодействия.

Для того, чтобы сегмент памяти мог использоваться совместно, он должен быть сначала создан при помощи системного вызова `shmget`. После создания сегмента разделяемой памяти процесс может подключаться к нему при помощи вызова `shmat` и затем использовать его для своих частных целей. Когда этот сегмент памяти больше не нужен, процесс может отключиться от него при помощи вызова `shmdt`.

Системный вызов shmget

Сегменты разделяемой памяти создаются при помощи вызова `shmget`.

Описание

```
uses ipc;

Function shmget(key:Tkey; Size:longint; permflags:longint):longint;
```

Этот вызов аналогичен вызовам `msgget` и `semget`. Наиболее интересным параметром вызова является `size`, который задает требуемый минимальный размер (в байтах) сегмента памяти. Параметр `key` является значением ключа сегмента памяти, параметр `permflags` задает права доступа к сегменту памяти и, кроме того, может содержать флаги `IPC_CREAT` и `IPC_EXCL`.

Операции с разделяемой памятью: вызовы shmat и shmdt

Сегмент памяти, созданный вызовом `shmget`, является участком *физической* памяти и не находится в *логическом* пространстве данных процесса. Для использования разделяемой памяти текущий процесс, а также все другие процессы, взаимодействующие с этим сегментом, должны явно подключать этот участок памяти к логическому адресному пространству при помощи вызова `shmat`:

Описание

```
uses ipc;

Function shmat(shmid:longint; daddr:pchar; shmflags:longint):pchar;
```

Вызов `shmat` связывает участок памяти, обозначенный идентификатором `shmid` (который был получен в результате вызова `shmget`) с некоторым допустимым адресом

логического адресного пространства вызывающего процесса. Этот адрес является значением, возвращаемым вызовом shmat.

Параметр daddr позволяет программисту до некоторой степени управлять выбором этого адреса. Если этот параметр равен nil, то участок подключается к первому доступному адресу, выбранному системой. Это наиболее простой случай использования вызова shmat. Если параметр daddr не равен nil, то участок будет подключен к содержащемуся в нем адресу или адресу в ближайшей окрестности в зависимости от флагов, заданных в аргументе shmflags. Этот вариант сложнее, так как при этом необходимо знать расположение программы в памяти.

Аргумент shmflag может содержать два флага, SHM_RDONLY и SHM_RND, определенные в заголовочном файле ipc. При задании флага SHM_RDONLY участок памяти подключается только для чтения. Флаг SHM_RND определяет, если это возможно, способ обработки в вызове shmat ненулевого значения daddr.

В случае ошибки вызов shmat вернет значение:

```
(pchar) -1
```

Вызов shmdt противоположен вызову shmat и отключает участок разделяемой памяти от логического адресного пространства процесса (это означает, что процесс больше не может использовать его). Он вызывается очень просто:

```
retval := shmdt(memptr);
```

Возвращаемое значение retval является логическим значением и равно true в случае успеха и false – в случае ошибки.

Системный вызов shmctl

Описание

```
uses ipc;

Function shmctl(shmid:longint; command:longint; shm_stat: pshmid_ds):
        Boolean;
```

Этот вызов в точности соответствует вызову msgctl, и параметр command может, наряду с другими, принимать значения IPC_STAT, IPC_SET и IPC_RMID. В следующем примере этот вызов будет использован с аргументом command равным IPC_RMID.

Пример работы с разделяемой памятью: программа shmcopy

В этом разделе создадим простую программу shmcopy для демонстрации практического использования разделяемой памяти. Программа shmcopy просто копирует данные со своего стандартного ввода на стандартный вывод, но позволяет избежать лишних простоев в вызовах fdread и fdwrite. При запуске программы shmcopy создаются два процесса, один из которых выполняет чтение, а другой – запись, и которые совместно используют два буфера, реализованных в виде сегментов разделяемой памяти. Когда первый процесс считывает данные в первый буфер, второй записывает содержимое второго буфера, и наоборот. Так как чтение и запись выполняются одновременно, пропускная способность возрастает. Этот подход используется, например, в программах, которые выводят информацию на ленточный накопитель.

Для согласования двух процессов (чтобы записывающий процесс не писал в буфер до тех пор, пока считывающий процесс его не заполнит) будем использовать два семафора. Почти во всех программах, использующих разделяемую память, требуется дополнительная синхронизация, так как механизм разделяемой памяти не содержит собственных средств синхронизации.

Программа shmcopy использует следующий заголовочный файл share_ex.inc:

```
(* Заголовочный файл для примера работы с разделяемой памятью *)

const
  SHMKEY1:tkey=$10; (* ключ разделяемой памяти *)
  SHMKEY2:tkey=$15; (* ключ разделяемой памяти *)
```

```
  SEMKEY :tkey=$20; (* ключ семафора *)

  (* Размер буфера для чтения и записи *)
  BUFSIZ=8192;
  SIZ=5*BUFSIZ;

(* В этой структуре будут находиться данные и счетчик чтения *)
type
  databuf=record
    d_nread:integer;
    d_buf:array [0..SIZ-1] of char;
  end;
  pdatabuf=^databuf;
```

Напомним, что постоянная `BUFSIZ` определена в файле `stdio` и задает оптимальный размер порций данных при работе с файловой системой. Шаблон `databuf` показывает структуру, которая связывается с каждым сегментом разделяемой памяти. В частности, элемент `d_nread` позволит процессу, выполняющему чтение, передавать другому, осуществляющему запись, через участок разделяемой памяти число считанных символов.

Следующий файл содержит процедуры для инициализации двух участков разделяемой памяти и набора семафоров. Он также содержит процедуру `remobj`, которая удаляет различные объекты межпроцессного взаимодействия в конце работы программы. Обратите внимание на способ вызова `shmat` для подключения участков разделяемой памяти к адресному пространству процесса.

```
(* Процедуры инициализации *)

{$i share_ex.inc}

const
  IFLAGS=IPC_CREAT or IPC_EXCL;
  ERR:pdatabuf=pdatabuf(-1);

var
  shmid1, shmid2, semid : longint;

procedure getseg (var p1,p2:pdatabuf);
begin
  (* Создать участок разделяемой памяти *)
  shmid1 := shmget (SHMKEY1, sizeof (databuf), octal(0600) or IFLAGS);
  if shmid1 = -1 then
    fatal ('shmget');

  shmid2 := shmget (SHMKEY2, sizeof (databuf), octal(0600) or IFLAGS);
  if shmid2 = -1 then
    fatal ('shmget');

  (* Подключить участки разделяемой памяти. *)
  p1 := pdatabuf( shmat (shmid1, 0, 0));
  if p1 = ERR then
    fatal ('shmat');

  p2 := pdatabuf( shmat (shmid2, 0, 0));
  if p2 = ERR then
    fatal ('shmat');
end;


function getsem:longint;      (* Получить набор семафоров *)
```

```
var
  x:tsemun;
begin
  x.val := 0;

  (* Создать два набора семафоров *)
  semid := semget (SEMKEY, 2, octal(0600) or IFLAGS);
  if semid = -1 then
    fatal ('semget');

  (* Задать начальные значения *)

  if semctl (semid, 0, SETVAL, x) = -1 then
    fatal ('semctl');

  if semctl (semid, 1, SETVAL, x) = -1 then
    fatal ('semctl');

  getsem:=semid;
end;


(* Удаляет идентификаторы разделяемой памяти
  * и идентификатор набора семафоров
  *)
procedure remobj;
var
  x:tsemun;
begin
  if not shmctl (shmid1, IPC_RMID, nil) then
    fatal ('shmctl');

  if not shmctl (shmid2, IPC_RMID, nil) then
    fatal ('shmctl');

  if semctl (semid, 0, IPC_RMID, x) = -1 then
    fatal ('semctl');
end;
```

Ошибки в этих процедурах обрабатываются при помощи процедуры `fatal`, которая использовалась в предыдущих примерах. Она просто вызывает `perror`, а затем `halt`.

Ниже следует главная функция для программы `shmcopy`. Она вызывает процедуры инициализации, а затем создает процесс для чтения (родительский) и для записи (дочерний). Обратите внимание на то, что именно выполняющий запись процесс вызывает процедуру `remobj` при завершении программы.

```
(* Программа shmcopy - главная функция *)
uses ipc,stdio,linux;
{$i share_ex.inc}


var
  pid : longint;
  buf1, buf2 : pdatabuf;
begin
  (* Инициализация набора семафоров. *)
  semid := getsem;

  (* Создать и подключить участки разделяемой памяти. *)
```

```
  getseg (buf1, buf2);

  pid := fork;
  case pid of
    -1:
      fatal ('fork');
    0:                  (* дочерний процесс *)
    begin
      writer (semid, buf1, buf2);
      remobj;
    end;
    else                 (* родительский процесс *)
      reader (semid, buf1, buf2);
  end;

  halt (0);
end.
```

Программа создает объекты межпроцессного взаимодействия до вызова `fork`. Обратите внимание на то, что адреса, определяющие сегменты разделяемой памяти (которые находятся в переменных `buf1` и `buf2`), будут заданы в обоих процессах.

Процедура `reader` принимает данные со стандартного ввода, то есть из дескриптора файла 0, и является первой функцией, представляющей интерес. Ей передается идентификатор набора семафоров в параметре `semid` и адреса двух участков разделяемой памяти в переменных `buf1` и `buf2`.

```
{$i share_ex.inc}

(* Определения процедур p() и v() для двух семафоров *)
const
  p1:tsembuf=(sem_num:0;sem_op:-1;sem_flg:0);
  p2:tsembuf=(sem_num:1;sem_op:-1;sem_flg:0);
  v1:tsembuf=(sem_num:0;sem_op:1;sem_flg:0);
  v2:tsembuf=(sem_num:1;sem_op:1;sem_flg:0);

(* Процедура reader - выполняет чтение из файла *)
procedure reader(semid:longint;buf1,buf2:pdatabuf);
begin
  while true do
  begin
    (* Считать в буфер buf1 *)
    buf1^.d_nread := fdread (0, buf1^.d_buf, SIZ);

    (* Точка синхронизации *)
    semop (semid, @v1, 1);
    semop (semid, @p2, 1);

    (* Чтобы процедура writer не была приостановлена. *)
    if buf1^.d_nread <= 0 then
      exit;

    buf2^.d_nread := fdread (0, buf2^.d_buf, SIZ);

    semop (semid, @v2, 1);
    semop (semid, @p1, 1);

    if buf2^.d_nread <= 0 then
      exit;
  end;
```

```
end;
```

Структуры `sembuf` просто определяют операции `p()` и `v()` для набора из двух семафоров. Но на этот раз они используются не для блокировки критических участков кода, а для синхронизации процедур, выполняющих чтение и запись. Процедура `reader` использует операцию `v2` для сообщения о том, что она завершила чтение и ожидает, вызвав `semop` с параметром `p1`, пока процедура `writer` сообщит о завершении записи. Это станет более очевидным при описании процедуры `writer`. Возможны другие подходы, включающие или четыре бинарных семафора, или семафоры, имеющие более двух значений.

Последней процедурой, вызываемой программой `shmcopy`, является процедура `writer`:

```
{$i share_ex.inc}

(* Процедура writer - выполняет запись *)
procedure writer(semid:longint;buf1,buf2:pdatabuf);
begin
  while true do
  begin
    semop (semid, @p1, 1);
    semop (semid, @v2, 1);

    if buf1^.d_nread <= 0 then
      exit;

    fdwrite (1, buf1^.d_buf, buf1^.d_nread);
    semop (semid, @p2, 1);
    semop (semid, @v1, 1);

    if buf2^.d_nread <= 0 then
      exit;

    fdwrite (1, buf2^.d_buf, buf2^.d_nread);
  end;
end;
```

И снова следует обратить внимание на использование набора семафоров для согласования работы процедур `reader` и `writer`. На этот раз процедура `writer` использует операцию `v2` для сигнализации и ждет `p1`. Важно также отметить, что значения `buf1^.d_nread` и `buf2^.d_nread` устанавливаются процессом, выполняющим чтение.

После компиляции можно использовать программу `shmcopy` при помощи подобной команды:

```
$ shmcopy < big > /tmp/big
```

Упражнение 8.6. *Усовершенствуйте обработку ошибок и вывод сообщений в программе* `shmcopy` *(в особенности для вызовов* `fdread` *и* `fdwrite`*). Сделайте так, чтобы программа* `shmcopy` *принимала в качестве аргументов имена файлов в форме команды* `cat`*. Какие последствия возникнут при прерывании программы* `shmcopy`*? Можете ли вы улучшить поведение программы?*

Упражнение 8.7. *Придумайте систему передачи сообщений, использующую разделяемую память. Измерьте ее производительность и сравните с производительностью стандартных процедур передачи сообщений.*

8.3.5. Команды ipcs и ipcrm

Существуют две команды оболочки для работы со средствами межпроцессного взаимодействия. Первая из них – команда `ipcs`, которая выводит информацию о текущем статусе средства межпроцессного взаимодействия. Вот простой пример ее применения:

```
$ ipcs
```

```
IPC status from /dev/kmem as of Wed Feb 26 18:31:31 1998
T ID KEY  MODE    OWNER GROUP
Message Queues:
Shared Memory:
Semaphores
s 10 0x00000200 --ra------- keith   users
```

Другая команда `ipcrm` используется для удаления средства межпроцессного взаимодействия из системы (если пользователь является его владельцем или суперпользователем), например, команда

```
$ ipcrm -s 0
```

удаляет семафор, связанный с идентификатором 0, а команда

```
$ ipcrm -S 200
```

удаляет семафор со значением ключа равным 200.

За дополнительной информацией о возможных параметрах этих команд следует обратиться к справочному руководству системы.

Глава 9. Терминал

9.1. Введение

Когда пользователь взаимодействует с программой при помощи терминала, происходит намного больше действий, чем может показаться на первый взгляд. Например, если программа выводит строку на терминальное устройство, то она вначале обрабатывается в разделе ядра, которое будем называть *драйвером терминала* (terminal driver). В зависимости от значения определенных флагов состояния системы строка может передаваться буквально или как-то изменяться драйвером. Одно из обычных изменений заключается в замене символов `line-feed` (перевод строки) или `newline` (новая строка) на последовательность из двух символов `carriage-return` (возврат каретки) и `newline`. Это гарантирует, что каждая строка всегда будет начинаться с левого края экрана терминала или открытого окна.

Аналогично драйвер терминала обычно позволяет пользователю редактировать ошибки в строке ввода при помощи текущих символов `erase` (стереть) и `kill` (уничтожить). Символ `erase` удаляет последний напечатанный символ, а символ `kill` – все символы до начала строки. Только после того, как вид строки устроит пользователя и он нажмет на клавишу **Return** (ввод), драйвер терминам передаст строку программе.

Но это еще не все. После того, как выводимая строка достигает терминала, аппаратура терминала может либо просто вывести ее на экран, либо интерпретировать ее как *escape-последовательность* (escape-sequence), посланную для управления экраном. В результате, например, может произойти не вывод сообщения, а очистка экрана.

На рис. 9.1 более ясно показаны различные компоненты связи между компьютером и терминалом.

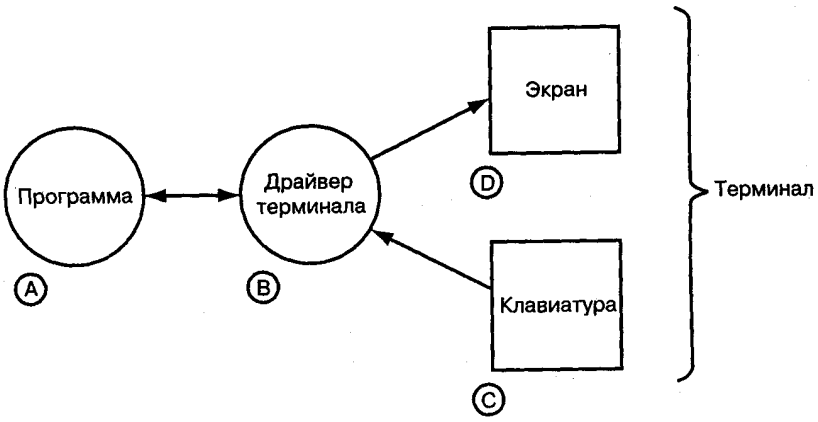


Рис. 9.1. Связь между процессом UNIX и терминалом

Эта связь включает четыре элемента:

– *программы (A)*. Программа генерирует выходные последовательности волов и интерпретирует входные. Она может взаимодействовать с терминалом при помощи системных вызовов (`fdread` или `fdwrite`), стандартной библиотеки ввода/вывода или специального библиотечного пакета, разработанного для управления экраном. Разумеется, в конечном счете весь ввод/вывод будет осуществляться при помощи вызовов `fdread` и `fdwrite`, так как высокоуровневые библиотеки могут вызывать только эти основные примитивы;

– *драйвер терминала (B)*. Основная функция драйвера терминала заключается в передаче данных от программы к периферийному устройству и наоборот. В самом ядре UNIX терминал обычно состоит из двух основных программных компонентов – *драйвера устройства* (device driver) и *дисциплины линии связи* (line discipline). Драйвер устройства является низкоуровневым программным

обеспечением, написанным для связи с определенным аппаратным обеспечением, которое позволяет компьютеру взаимодействовать с терминалом. На самом деле чаще всего драйверы устройств нужны для того, чтобы работать с разными типами аппаратного обеспечения. Над этим нижним слоем надстроены средства, которые полагаются на то, что основные свойства, поддерживаемые драйвером устройства, являются общими независимо от аппаратного обеспечения. Кроме этой основной функции передачи данных, драйвер терминала будет также выполнять некоторую логическую обработку входных и выходных данных, преобразуя одну последовательность символов в другую. Это осуществляется дисциплиной линии связи. Она также может обеспечивать множество функций для помощи конечному пользователю, таких как редактирование строки ввода. Точная обработка и преобразование данных зависят от флагов состояния, которые хранятся в дисциплине линии связи для каждого порта терминала. Они могут устанавливаться при помощи группы системных вызовов, которая будут рассмотрена в следующих разделах;

– *клавиатура и экран (C и D)*. Эти два элемента представляют сам терминал и подчеркивают его двойственную природу. Узел (C) означает клавиатуру терминала и служит источником ввода. Узел (D) представляет экран терминала и выступает в качестве назначения вывода. Программа может получить доступ к терминалу и как к устройству ввода, и как к устройству вывода при помощи общего имени терминала, и, в конечном счете, единственного дескриптора файла. Для того чтобы это было возможно, дисциплина лини связи имеет раздельные очереди ввода и вывода для каждого терминала. Эта схема показана на рис. 9.2.

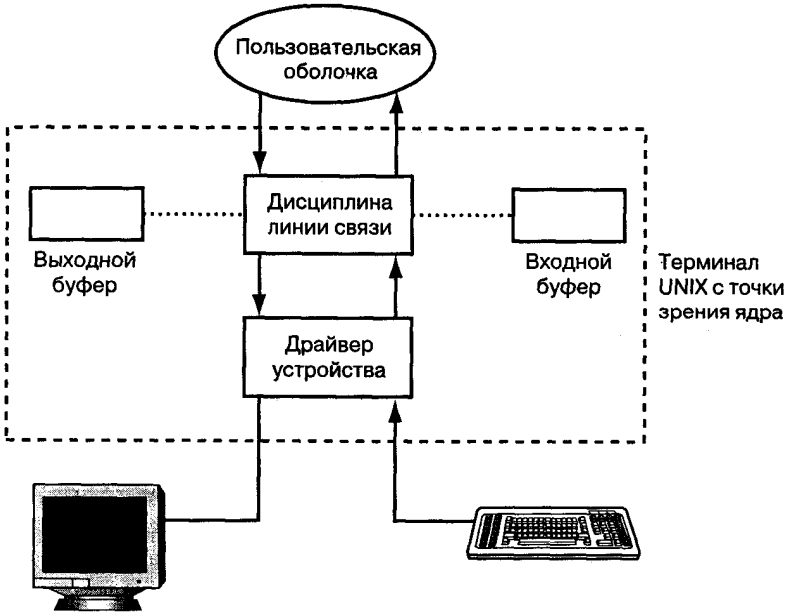


Рис. 9.2. Реализация терминала

До сих пор предполагалось, что подключенное к терминалу периферийное устройство является стандартным дисплеем. Вместе с тем периферийное устройство может быть принтером, плоттером, сетевым адаптером или даже другим компьютером. Тем не менее, независимо от природы периферийного устройства, оно может служить и в качестве источника, и в качестве назначения для входного и выходного потоков данных соответственно.

Эта глава будет в основном посвящена узлам (A) и (B) схемы. Другими словами, будет рассмотрено взаимодействие между программой и драйвером устройства на уровне системных вызовов. Не будем касаться совершенно отдельного вопроса работы с экраном, поскольку драйвер терминала не принимает участия в создании соответствующих escape-

последовательностей, управляющих экраном.

Перед тем как продолжить дальше, следует сделать два предостережения. Во-первых, будут рассматриваться только «обычные» терминалы, а не графические, построенные на оконных системах X Window System или MS Windows. Для них характерны свои проблемы, которых касаться не будем. Во-вторых, работа с терминалом в UNIX является областью, печально известной своей несовместимостью. Тем не менее спецификация XSI обеспечивает стандартный набор системных вызовов. Именно на них и сфокусируем внимание.

9.2. Терминал UNIX

Как уже упоминалось в главе 4, терминалы обозначаются файлами устройств (из-за природы терминалов они рассматриваются как символьные устройства). Вследствие этого доступ к терминалам, а точнее к портам терминалов, обычно можно получить при помощи имен файлов в каталоге `dev`. Типичные имена терминалов могут быть такими:

```
/dev/console
/dev/tty01
/dev/tty02
/dev/tty03
...
```

Обозначение `tty` является широко используемым в UNIX синонимом терминала.

Из-за универсальности понятия файла UNIX к терминалам можно получить доступ при помощи стандартных примитивов доступа к файлам, таких как `fdread` или `fdwrite`. Права доступа к файлам сохраняют свои обычные значения и поэтому управляют доступом к терминалам в системе. Чтобы эта схема работала, владелец терминала меняется при входе пользователя в систему, при этом все пользователи являются владельцами терминала, за которым они работают.

Обычно процессу не нужно явно открывать файл терминала для взаимодействия с пользователем. Это происходит из-за того, что его стандартный ввод и вывод, если они не переопределены, будут по умолчанию связаны с терминалом пользователя. Поэтому, если предположить, что стандартный вывод не назначен в файл, то следующий фрагмент кода приведет к выводу данных на экран терминала:

```
const FD_STDOUT=1;
.
.
.
fdwrite(FD_STDOUT, mybuffer, somesize);
```

В традиционном окружении UNIX терминалы, обеспечивающие вход в систему, обычно первоначально открываются при старте системы программой управления процессами `init`. Дескрипторы файла терминала передаются потомка программы `init`, и, в конечном итоге, каждый процесс пользовательской оболочки будет наследовать три дескриптора файла, связанные с терминалом пользователя. Эти дескрипторы будут представлять стандартный ввод, стандартный вывод и стандартный вывод диагностики оболочки. Они в свою очередь передаются всем запущенным из оболочки программам.

9.2.1. Управляющий терминал

При обычных обстоятельствах терминал, связанный с процессом при помощи его стандартных дескрипторов файлов, является *управляющим терминалом* (control terminal) этого процесса и его сеанса. Управляющий терминал является важным атрибутом процесса, который определяет обработку генерируемых с клавиатуры прерываний. Например, если пользователь нажимает текущую клавишу прерывания, то все процессы, которые считают терминал своим управляющим терминалом, получат сигнал `SIGINT`. Управляющие терминалы, как и другие атрибуты процесса, наследуются при вызове `fork`. (Более конкретно, терминал становится управляющим терминалом для сеанса, когда его открывает лидер сеанса, при условии, что терминал еще не связан с сеансом и лидер сеанса еще не

имеет управляющего терминала. Вследствие этого процесс может разорвать свою связь с управляющим терминалом, изменив свой сеанс при помощи вызова `setsid`. Этот аспект был рассмотрен в главе 5, хотя, возможно, там это было несколько преждевременно. Теперь же следует получить понимание того, как процесс `init` инициализирует систему при старте.)

Если процесс должен получить доступ к своему управляющему терминалу независимо от состояния его стандартных дескрипторов файлов, то можно использовать имя файла

```
/dev/tty
```

которое всегда интерпретируется как определяющее текущий управляющий терминал процесса. Следовательно, терминал, обозначаемый в действительности этим файлом, различается для разных процессов.

9.2.2. Передача данных

Основной задачей драйвера терминала является передача данных между процессом и его терминальным устройством. На самом деле это достаточно сложное требование, так как пользователь может печатать символы в любое время, даже во время вывода. Чтобы лучше понять эту ситуацию, вернемся к рис. 9.1 и представим, что по путям от (C) к (B) и от (B) к (D) одновременно передаются данные. Напомним, что программа, представленная на схеме узлом (A), может выполнять только последовательные вызовы `fdread` или `fdwrite`.

Поскольку в каждый момент времени программа может обрабатывать только один поток данных терминала, то для одновременного управления двумя потоками дисциплина линии связи запоминает входные и выходные данные во внутренних буферах. Входные данные передаются программе пользователя, когда он выполняет вызов `fdread`. Входные символы могут быть потеряны при переполнении поддерживаемых ядром буферов или если число символов, поступивших на терминал, превысит ограничение `MAX_INPUT`, определенное в файле `stdio`. Это предельное значение обычно равно 255, что достаточно велико для того, чтобы потери данных при обычном использовании были достаточно редки. Тем не менее не существует способа определить, что данные были потеряны, так как система просто молча отбрасывает лишние символы.

Ситуация с выводом несколько проще. Каждый вызов `fdwrite` для терминала помещает символы в очередь вывода. Если очередь переполняется, то следующий вызов `fdwrite` будет «заблокирован» (то есть процесс будет приостановлен) до тех пор, пока очередь вывода не уменьшится до нужного уровня.

9.2.3. Эхо-отображение вводимых символов и опережающий ввод с клавиатуры

Поскольку терминалы используются для взаимодействия между людьми и компьютерными программами, драйвер терминала UNIX поддерживает множество дополнительных средств, облегчающих жизнь пользователям.

Возможно, самым элементарным из этих дополнительных средств является «эхо», то есть отображение вводимых с клавиатуры символов на экране. Оно позволяет увидеть на экране символ «A», когда вы печатаете «A» на клавиатуре. Подключенные к системам UNIX терминалы обычно работают в *полнодуплексном* (full-duplex) режиме; это означает, что за эхо-отображение символов на экране отвечает система UNIX, а не терминал. Следовательно, при наборе символа он вначале передается терминалом системе UNIX. После его получения дисциплина линии связи сразу же помещает его копию в очередь вывода терминала. Затем символ выводится на экран терминала. Если вернуться к рис. 9.1, то увидим, что символ при этом пересылается по пути от (C) к (B), а затем сразу же направляется по пути от (B) к (D). Все может произойти до того, как программа (A) успеет считать символ. Это приводит к интересному явлению, когда символы, вводимые с клавиатуры в то время, когда программа осуществляет вывод на экран, отображаются в середине вывода. В операционных системах некоторых семейств (не UNIX) отображение вводимых символов на экране может

подавляться до тех пор, пока программа не сможет прочитать их.

9.2.4. Канонический режим, редактирование строки и специальные символы

Терминал может быть настроен на самые разные режимы в соответствии с типом работающих с ним программ; работа этих режимов поддерживается соответствующей дисциплиной линии связи. Например, экранному редактору может понадобиться максимальная свобода действий, поэтому он может перевести терминал в режим *прямого доступа* (raw mode), при котором дисциплина линии связи просто передает символы программе по мере их поступления, без всякой обработки.

Вместе с тем ОС UNIX не была бы универсальной программной средой, если бы всем программам приходилось бы иметь дело с деталями управления терминалом. Поэтому стандартная дисциплина линии связи обеспечивает также режим работы, специально приспособленный для простого интерактивного использования на основе построчного ввода. Этот режим называется *каноническим режим терминала* (canonical mode) и используется командным интерпретатором, редактором `ed` и аналогичными программами.

В каноническом режиме драйвер терминала выполняет специальные действия при нажатии специальных клавиш. Многие из этих действий относятся к редактированию строки. Вследствие этого, если терминал находится в каноническом режиме, то ввод в программе осуществляется целыми строками (подробнее об этом ниже).

Наиболее часто используемой в каноническом режиме клавишей редактирования является клавиша `erase`, нажатие которой приводит к стиранию предыдущего символа в строке. Например, следующий клавиатурный ввод

```
$ whp<erase>o
```

за которым следует перевод строки, приведет к тому, что драйвер терминала передаст командному интерпретатору строку *who*. Если терминал настроен правильно, то символ *p* должен быть стерт с экрана.

В качестве символа `erase` пользователем может быть задано любое значение ASCII. Наиболее часто для этих целей используется символ ASCII `backspace` (возврат).

На уровне оболочки проще всего поменять этот символ при помощи команды `stty`, например

```
$ stty erase "^h"
```

задает в качестве символа erase комбинацию **Ctrl+H**, которая является еще одним именем для символа возврата. Обратите внимание, что зачастую можно набрать строку *"^h"*, а не нажимать саму комбинацию клавиш **Ctrl+H**.

Ниже следует систематическое описание других символов, определенных в спецификации XSI и имеющих особое значение для оболочки в каноническом режиме. Если не указано иначе, их точные значения могут быть установлены пользователем или администратором системы.

kill Приводит к стиранию всех символов до начала строки. Поэтому входная последовательность

```
$ echo < kill > who
```

за которой следует новая строка, приводит к выполнению команды `who`. По умолчанию значение символа **kill** равно **Ctrl+?**, часто также используются комбинации клавиш **Ctrl+X** и **Ctrl+U**. Можно поменять символ **kill** при помощи команды:

```
$ stty kill <новый_символ>
```

intr Символ прерывания. Если пользователь набирает его, то программе, выполняющей чтение с терминала, а также всем другим процессам, которые считают терминал своим управляющим терминалом, посылается сигнал `SIGINT`. Такие программы, как командный интерпретатор, перехватывают этот сигнал, поскольку по умолчанию

сигнал SIGINT приводит к завершению программы. Одно из значений для символа **intr** по умолчанию равно символу ASCII **delete** (удалить), который часто называется **DEL**. Вместо него может также использоваться символ **Ctrl+C**. Для изменения текущего значения символа **intr** на уровне оболочки используйте команду:

$ stty intr <новый_символ>

quit Обратитесь к главе 6 за описанием обработки сигналов **quit**.

Если пользователь набирает этот символ, то группе процессов, связанной с терминалом, посылается сигнал SIGQUIT. Командный интерпретатор перехватывает этот сигнал, остальные пользовательские процессы также имеют возможность перехватить и обработать этот сигнал. Как уже описывалось в главе 6, действием этого сигнала по умолчанию является сброс образа памяти процесса на диск и аварийное завершение, сопровождаемое выводом сообщения «Quit – core dumped». Обычно символ **quit** – это символ ASCII **FS**, или **Ctrl+**\. Его можно изменить, выполнив команду:

$ stty quit <новый_символ>

eof Этот символ используется для обозначения окончания входного потока с терминала (для этого он должен быть единственным символом в начале новой строки). Стандартным начальным значением для него является символ ASCII **eof**, известный также как **Ctrl+D**. Его можно изменить, выполнив команду:

$ stty eof <новый_символ>

nl Это обычный разделитель строк. Он всегда имеет значение ASCII символа **line-feed** (перевод строки), который соответствует символу **newline** (новая строка) в языке C. Он не может быть изменен или установлен пользователем. На терминалах, которые посылают вместо символа **line-feed** символ **carriage-return** (возврат каретки), дисциплина линии связи может быть настроена так, чтобы преобразовывать возврат каретки в перевод строки

eol Еще один разделитель строк, который действует так же, как и **nl**. Обычно не используется и поэтому имеет по умолчанию значение символа ASCII **NULL**

stop Обычно имеет значение **Ctrl+S** и в некоторых реализациях может быть изменен пользователем. Используется для временной приостановки записи вывода на терминал. Этот символ особенно полезен при применении старомодных терминалов, так как он может использоваться для того, чтобы приостановить вывод прежде, чем он исчезнет за границей экрана терминала

start Обычно имеет значение **Ctrl+Q**. Может ли он быть изменен, также зависит от конкретной реализации. Он используется для продолжения вывода, приостановленного при помощи комбинации клавиш **Ctrl+S**. Если комбинация **Ctrl+S** не была нажата, то **Ctrl+Q** игнорируется

susp Если пользователь набирает этот символ, то группе процессов, связанной с терминалом, посылается сигнал SIGTSTP. При этом выполнение группы приоритетных процессов останавливается, и она переводится в фоновый режим. Обычное значение символа **suspend** равно **Ctrl+Z**. Его также можно изменить, выполнив команду:

$ stty susp <новый_символ>

Специальное значение символов **erase**, **kill**, и **eof** можно отменить, набрав перед ними символ обратной косой черты (\). При этом связанная с символом функция не выполняется, и символ посылается программе без изменений. Например, строка

aa\<erase> b <erase> c

приведет к тому, что программе, выполняющей чтение с терминала, будет передана строка *aa\<erase> c*.

9.3. Взгляд с точки зрения программы

До сих пор изучались функции драйвера терминала, относящиеся к пользовательскому интерфейсу. Теперь же мы рассмотрим их с точки зрения программы, использующей терминал.

9.3.1. Системный вызов fdopen

Вызов `fdopen` может использоваться для открытия дисциплины линии связи терминала так же, как и для открытия обычного дискового файла, например:

```
fd := fdopen('/dev/tty0a', Open_RDWR);
```

Однако при попытке открыть терминал возврата из вызова не произойдет до тех пор, пока не будет установлено соединение. Для терминалов с модемным управлением это означает, что возврат из вызова не произойдет до тех пор, пока не будут установлены сигналы управления модемом и не получен сигнал «детектирования несущей», что может потребовать значительного времени либо вообще не произойти.

Следующая процедура использует вызов `alarm` (представленный в главе 6) для задания интервала ожидания, если возврат из вызова `fdopen` не произойдет за заданное время:

```
(* Процедура ttyopen - вызов fdopen с интервалом ожидания *)
uses stdio,linux;

const
  TIMEOUT=10;
  timeout_flag:boolean=FALSE;
  termname:pchar='';

procedure settimeout(value:longint);cdecl;
begin
  writeln(stderr, 'Превышено время ожидания ', termname);
  timeout_flag := TRUE;
end;


function ttyopen(filename:pchar; flags:longint):longint;
var
  fd:longint;
  act, oact:sigactionrec;
  mask:sigset_t;
begin
  fd := -1;

  termname := filename;

  (* Установить флаг таймаута *)
  timeout_flag := FALSE;

  (* Установить обработчик сигнала SIGALRM *)
  act.handler.sh := @settimeout;
  sigfillset(@mask);
  act.sa_mask:=mask.__val[0];
  sigaction(SIGALRM, @act, @oact);

  alarm(TIMEOUT);
```

```
  fd := fdopen(filename, flags);

  (* Сброс установок *)
  alarm(0);
  sigaction(SIGALRM, @oact, @act);
  if timeout_flag then
    ttyopen:=-1
  else
    ttyopen:=0;
end;
```

9.3.2. Системный вызов fdread

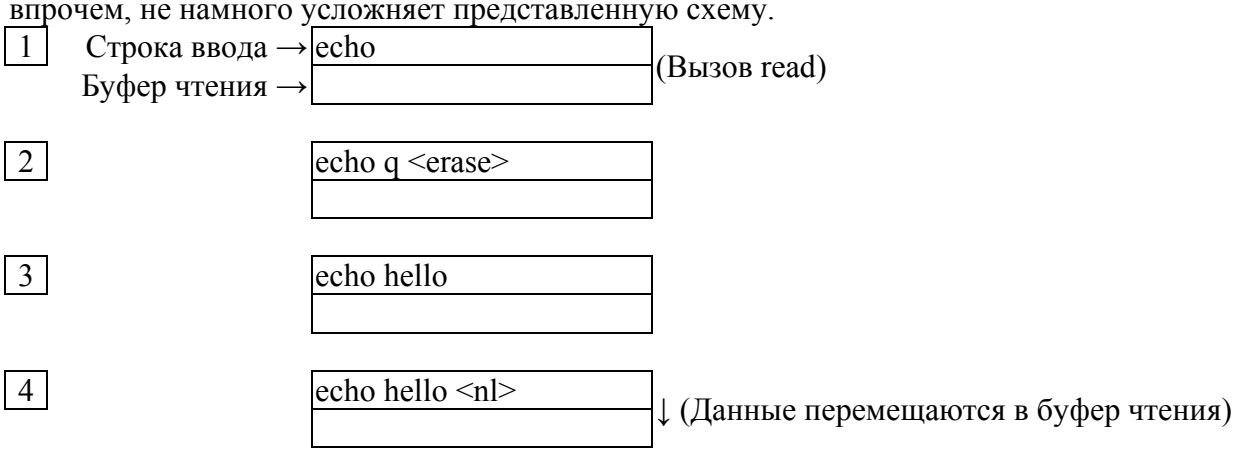
При использовании файла терминального устройства вместо обычного дискового файла изменения больше всего затрагивают работу системного вызова `read`. Это особенно проявляется, если терминал находится в каноническом режиме, устанавливаемом по умолчанию для обычного интерактивного использования. В этом режиме основной единицей ввода становится строка. Следовательно, программа не может считать символы из строки, пока пользователь не нажмет на клавишу **Return**, которая интерпретируется системой как переход на новую строку. Важно также, что после ввода новой строки всегда происходит возврат из вызова `fdread`, даже если число символов в строке меньше, чем число символов, запрошенное вызовом `fdread`. Если вводится только **Return** и системе посылается пустая строка, то соответствующий вызов `fdread` вернет значение 1, так как сам символ новой строки тоже доступен программе. Поэтому нулевое значение, как и обычно, может использоваться для определения конца файла (то есть ввода символа **eof**).

Использование вызова `fdread` для чтения из терминала в программе `io` уже рассматривалось в главе 2. Тем не менее эта тема нуждается в более подробном объяснении, поэтому рассмотрим следующий вызов:

```
nread := fdread(0, buffer, 256);
```

При извлечении стандартного ввода процесса из обычного файла вызов интерпретируется просто: если в файле более 256 символов, то вызов `fdread` вернет в точности 256 символов в массиве `buffer`. Чтение из терминала происходит в несколько этапов – чтобы продемонстрировать это, обратимся к рис. 9.3, который показывает взаимодействие между программой и пользователем в процессе вызова `fdread`.

Эта схема иллюстрирует возможную последовательность действий, инициированных чтением с терминала с помощью вызова `read`. Для каждого шага изображены два прямоугольника. Верхний из них показывает текущее состояние строки ввода на уровне драйвера терминала; нижний, обозначенный как буфер чтения, показывает данные, доступные для чтения процессом в настоящий момент. Необходимо подчеркнуть, что схема показывает только логику работы с точки зрения пользовательского процесса. Кроме того, обычная реализация драйвера терминала использует не один, а два буфера (очереди), что, впрочем, не намного усложняет представленную схему.



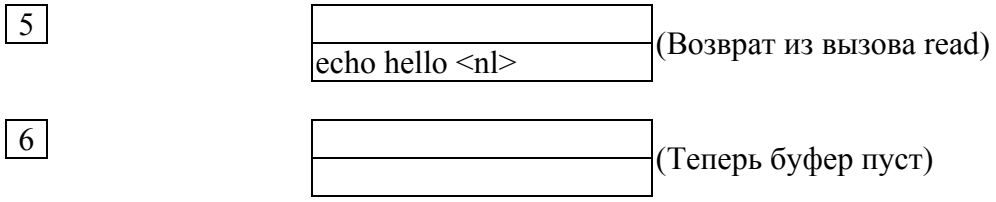


Рис. 9.3. Этапы чтения из терминала в каноническом режиме

Шаг 1 представляет ситуацию в момент выполнения программой вызова `read`. В этот момент пользователь уже напечатал строку *echo*, но поскольку строка еще не завершена символом перевода строки, то в буфере чтения нет данных, и выполнение процесса приостановлено.

На шаге 2 пользователь напечатал *q*, затем передумал и набрал символ **erase** для удаления символа из строки ввода. Эта часть схемы подчеркивает, что редактирование строки ввода может выполняться, не затрагивая программу, выполняющую вызов `read`.

На шаге 3 строка ввода завершена, только еще не содержит символ перехода на новую строку. Часть схемы, обозначенная как шаг 4, показывает состояние в момент ввода символа перехода на новую строку, при этом драйвер терминала передает строку ввода, включая символ перевода строки, в буфер чтения. Это приводит к шагу 5, на котором вся строка ввода становится доступной для чтения. В процессе, выполнившем вызов `read`, происходит возврат из этого вызова, при этом число введенных символов `nread` равно 11. Шаг 6 показывает ситуацию сразу же после такого завершения вызова, при этом и строка ввода, и буфер чтения снова временно пусты.

Следующий пример подкрепляет эти рассуждения. Он основан на простой программе `read_demo`, которая имеет лишь одну особенность: она использует небольшой размер буфера для приема байтов из стандартного ввода.

```
(* Программа read_demo - вызов fdread для терминала *)
uses linux;
const
  SMALLSZ=10;

var
  nread:longint;
  smallbuf:array [0..SMALLSZ] of char;
begin
  nread := fdread (0, smallbuf, SMALLSZ);
  while nread > 0 do
  begin
    smallbuf[nread] := #0;
    writeln('nread: ',nread,' ', smallbuf);
    nread := fdread (0, smallbuf, SMALLSZ);
  end;
end.
```

Если подать на вход программы следующий терминальный ввод

```
1
1234
Это более длинная строка
<EOF>
```

то получится такой диалог:

```
1
nread: 2 1
1234
nread: 5 1234
Это более длинная строка
```

```
nread: 10 Это более
nread: 10 длинная с
nread: 6 трока
```

Обратите внимание, что для чтения самой длинной строки требуется несколько последовательных операций чтения. Заметим также, что значения счетчика `nread` включают также символ перехода на новую строку. Это не показано для простоты изложения.

Что происходит, если терминал не находится в каноническом режиме? В таком случае для полного управления процессом ввода программа должна устанавливать дополнительные параметры состояния терминала. Это осуществляется при помощи семейства системных вызовов, которые будут описаны позже.

Упражнение 9.1. *Попробуйте запустить программу* `read_demo`*, перенаправив ее ввод на чтение файла.*

9.3.3. Системный вызов fdwrite

Этот вызов намного проще в отношении взаимодействия с терминалом. Единственный важный момент заключается в том, что вызов `fdwrite` будет блокироваться при переполнении очереди вывода терминала. Программа продолжит работу, только когда число символов в очереди станет меньше некоторого заданного порогового уровня.

9.3.4. Функции ttyname и isatty

Теперь представим две полезных функции, которые будем использовать в следующих примерах. Функция `ttyname` возвращает имя терминального устройства, связанного с дескриптором открытого файла, а функция `isatty` возвращает значение `true` (то есть *истинно* в терминах языка Паскаль), если дескриптор файла описывает терминальное устройство, и `false` (*ложно*) – в противном случае.

Описание

```
uses linux;

Function TTYName(var f):String;

Function IsATTY(var f):Boolean;
```

В обоих случаях параметр `f` является дескриптором открытого файла либо файловой переменной. Если `f` не соответствует терминалу, то функция `ttyname` вернет пустую строку.

Следующий пример – процедура `what_tty` выводит имя терминала, связанного с дескриптором файла, если это возможно:

```
(* Процедура what_tty - выводит имя терминала *)

procedure what_tty(fd:longint);
begin
  if isatty(fd) then
    writeln('fd ',fd,' =>> ', ttyname(fd));
  else
    writeln ('fd ',fd, ' не является терминалом!');
end;
```

Упражнение 9.2. *Измените процедуру* `ttyopen` *предыдущего раздела так, чтобы она возвращала дескриптор файла только для терминалов, а не для дисковых файлов или других типов файлов. Для выполнения проверки используйте функцию* `isatty`*. Существуют ли еще какие-либо способы сделать это?*

9.3.5. Изменение свойств терминала: структура termios

На уровне оболочки пользователь может вызвать команду `stty` для изменения свойств дисциплины линии связи терминала. Программа может сделать практически то же самое, используя структуру `termios` вместе с соответствующими функциями. Обратите

внимание, что в более старых системах для этого использовался системный вызов ioctl (сокращение от *I/O control* – управление вводом/выводом), его применение было описано в первом издании этой книги. Вызов ioctl предназначен для более общих целей и теперь разделен на несколько конкретных вызовов. Совокупность этих вызовов обеспечивает общий программный интерфейс ко всем асинхронным портам связи, независимо от свойств их оборудования.

Структуру termios можно представлять себе как объект, способный описать общее состояние терминала в соответствии с набором флагов, поддерживаемым системой для любого терминального устройства. Точное определение структуры termios будет вскоре рассмотрено. Структуры termios могут заполняться текущими установками терминала при помощи вызова tcgetattr, определенного следующим образом:

Описание

```
uses linux;

Function TCGetAttr(ttyfd:longint; var tsaved:TermIOS):Boolean;
```

Эта функция сохраняет текущее состояние терминала, связанного с дескриптором файла ttyfd в структуре tsaved типа termios. Параметр ttyfd должен быть дескриптором файла, описывающим терминал.

Описание

```
uses linux;

Function TCSetAttr(ttyfd:longint; actions:longint; var tnew:TermIOS):
         Boolean;
```

Вызов tcsetattr установит новое состояние дисциплины связи, заданное структурой tnew. Второй параметр вызова tcsetattr, переменная actions, определяет, как и когда будут установлены новые атрибуты терминала. Существует три возможных варианта, определенных в файле linux:

TCSANOW	Немедленное выполнение изменений, что может вызвать проблемы, если в момент изменения флагов драйвер терминала выполняет вывод на терминал
TCSADRAIN	Выполняет ту же функцию, что и TCSANOW, но перед установкой новых параметров ждет опустошения очереди вывода
TCSAFLUSH	Аналогично TCSADRAIN ждет, пока очередь вывода не опустеет, а затем также очищает и очередь ввода перед установкой для параметров дисциплины линии связи значений, заданных в структуре tnew

Следующие две функции используют описанные вызовы. Функция tsave сохраняет текущие параметры, связанные с управляющим терминалом процесса, а функция tback восстанавливает последний набор сохраненных параметров. Флаг saved используется для предотвращения восстановления установок функцией tback, если перед этим не была использована функция tsave.

```
(* Структура tsaved будет содержать параметры терминала *)
var
  tsaved:termios;

(* Равно TRUE если параметры сохранены *)
const
  saved:boolean=false;

function tsave:boolean;
begin
  if isatty(0) and tcgetattr(0,tsaved) then
  begin
    saved := true;
```

```
    tsave := true;
    exit;
  end;
  tsave := false;
end;


function tback:boolean; (* Восстанавливает состояние терминала *)
begin
  if not isatty(0) or not saved then
    tback:=false
  else
    tback:=tcsetattr(0, TCSAFLUSH, tsaved);
end;
```

Между этими двумя процедурами может быть заключен участок кода, который временно изменяет состояние терминала, например:

```
uses linux;

begin
  if not tsave then
  begin
    writeln(stderr, 'Невозможно сохранить параметры терминала');
    halt(1);
  end;
  (* Интересующий нас участок *)

  tback;
  halt(0);
end.
```

Определение структуры termios

Теперь изучим состав структуры `termios`. Определение структуры `termios` находится в файле `linux` и содержит следующие элементы:

```
termios = record
   c_iflag,               (* Режимы ввода *)
   c_oflag,               (* Режимы вывода *)
   c_cflag,               (* Управляющие режимы *)
   c_lflag : Cardinal;    (* Режимы дисциплины линии связи *)
   c_line  : char;        (* Дисциплина линии связи *)
   c_cc    : array [0..NCCS-1] of char; (* Управляющие символы *)
end;
```

Проще всего рассматривать структуру, начав с ее последнего элемента `c_cc`.

Массив c_cc

Символы редактирования строки, которые рассматривались в разделе 9.2.4, находятся в массиве `c_cc`. Их позиции в этом массиве задаются константами, определенными в файле `stdio`. Все определенные в спецификации XSI значения приведены в табл. 9.1. Размер массива определяется константой `NCCS`, определенной в файле `linux`.

Следующий фрагмент программы показывает, как можно изменить значение символа **quit** для терминала, связанного со стандартным вводом (дескриптор файла со значением 0):

```
var
  tdes:termios;

(* Получить исходные настройки терминала *)
tcgetattr(0, tdes);
tdes.c_cc[VQUIT] := char(octal(031)); (* CTRL-Y *)
(* Изменить установки терминала *)
tcsetattr(0, TCSAFLUSH, tdes);
```

Таблица 9.1. Коды управляющих символов

Константа	Значение
VINTR	Клавиша прерывания (Interrupt key)
VQUIT	Клавиша завершения (Quit key)
VERASE	Символ стирания (Erase character)
VKILL	Символ удаления строки (Kill character)
VEOF	Символ конца файла (EOF character)
VEOL	Символ конца строки (End of line marker – необязательный)
VSTART	Символ продолжения передачи данных (Start character)
VSTOP	Символ остановки передачи данных (Stop character)
VSUSP	Символ временной приостановки выполнения (Suspend character)

Этот пример иллюстрирует наиболее безопасный способ изменения состояния терминала. Сначала нужно получить текущее состояние терминала. Далее следует изменить только нужные параметры, не трогая остальные. И, наконец, изменить состояние терминала при помощи модифицированной структуры `termios`. Как уже было упомянуто, сохранять исходные значения полезно также для восстановления состояния терминала перед завершением программы, иначе нестандартное состояние терминала может оказаться сюрпризом для остальных программ.

Поле c_cflag

Поле `c_cflag` содержит параметры, управляющие состоянием порта терминала. Обычно процессу не нужно изменять значения поля `c_cflag` своего управляющего терминала. Изменения этого поля могут понадобиться специальным коммуникационным приложениям, или программам, открывающим коммуникационные линии для дополнительного оборудования, например, для принтера. Значения поля `c_cflag` образуются объединением при помощи операции ИЛИ битовых констант, определенных в файле `stdio`. В общем случае каждая константа представляет бит поля `c_cflag`, который может быть установлен или сброшен. Не будем объяснять назначение всех битов этого поля (за полным описанием обратитесь к справочному руководству системы). Тем не менее есть четыре функции, которые позволяют опрашивать и устанавливать скорость ввода и вывода, закодированную в этом поле, не беспокоясь о правилах манипулирования битами.

Описание

```
uses linux;

(* Установить скорость ввода *)
Procedure CFSetISpeed(var tdes:TermIOS; Speed:Longint);

(* Установить скорость вывода *)
Procedure CFSetOSpeed(var tdes:TermIOS; Speed:Longint);


uses stdio;

(* Получить скорость ввода *)
function cfgetispeed(var tdes:TermIOS):longint;

(* Получить скорость вывода *)
function cfgetospeed(var tdes:TermIOS):longint;
```

Следующий пример устанавливает в структуре `tdes` значение скорости терминала равное 9600 бод. Постоянная `B9600` определена в файле `stdio`.

```
var
  tdes:termios;

(* Получает исходные настройки терминала * )
```

```
tcgetattr(0, tdes);

(* Изменяет скорость ввода и вывода *)
cfsetispeed(tdes, B9600);
cfsetospeed(tdes, B9600);
```

Конечно, эти изменения не будут иметь эффекта, пока не будет выполнен вызов `tcsetattr`:

```
tcsetattr(0, TCSAFLUSH, tdes);
```

Следующий пример устанавливает режим контроля четности, напрямую устанавливая необходимые биты:

```
tdes.c_cflag := tdes.c_cflag or PARENB or PARODD;
tcsetattr(0, TCSAFLUSH, tdes);
```

В этом примере установка флага `PARENB` включает проверку четности. Установленный флаг `PARODD` сообщает, что ожидаемый контроль – контроль нечетности. Если флаг `PARODD` сброшен и установлен флаг `PARENB`, то предполагается, что используется контроль по четности. (Термин *четность*, parity, относится к использованию битов проверки при передаче данных. Для каждого символа задается один такой бит. Это возможно благодаря тому, что набор символов ASCII занимает только семь бит из восьми, используемых для хранения символа на большинстве компьютеров. Значение бита проверки может использоваться для того, чтобы полное число битов в байте было либо четным, либо нечетным. Программист также может полностью выключить проверку четности.)

Поле c_iflag

Поле `c_iflag` в структуре `termios` служит для общего управления вводом с терминала. Не будем рассматривать все возможные установки, а выберем из них только наиболее общие.

Три из связанных с этим полем флага связаны с обработкой символа возврата каретки. Они могут быть полезны в терминалах, которые посылают для обозначения конца строки последовательность, включающую символ возврата каретки **CR**. ОС UNIX, конечно же, ожидает в качестве символа конца строки символ **LF** (line feed) или символ перевода строки ASCII, символ **NL** (new line). Следующие флаги могут исправить ситуацию:

`INLCR` Преобразовывать символ новой строки в возврат каретки

`IGNCR` Игнорировать возврат каретки

`ICRNL` Преобразовывать возврат каретки в символ новой строки

Три других поля `c_iflag` связаны с управлением потоком данных:

`IXON` Разрешить старт/стопное управление выводом

`IXANY` Продолжать вывод при нажатии любого символа

`IXOFF` Разрешить старт/стопное управление вводом

Флаг `IXON` дает пользователю возможность управления выводом. Если этот флаг установлен, то пользователь может прервать вывод, нажав комбинацию клавиш **Ctrl+S**. Вывод продолжится после нажатия комбинации **Ctrl+Q**. Если также установлен параметр `IXANY`, то для возобновления приостановленного вывода достаточно нажатия любой клавиши, хотя для остановки вывода должна быть нажата именно комбинация клавиш **Ctrl+S**. Если установлен флаг `IXOFF`, то система сама пошлет терминалу символ остановки (как обычно, **Ctrl+S**), когда буфер ввода будет почти заполнен. После того как система будет снова готова к приему данных, для продолжения ввода будет послана комбинация символов **Ctrl+Q**.

Поле c_oflag

Поле `c_oflag` позволяет управлять режимом вывода. Наиболее важным флатом в этом поле является флаг `OPOST`. Если он не установлен, то выводимые символы передаются без изменений. В противном случае символы подвергаются обработке, заданной остальными флагами, устанавливаемыми в поле `c_oflag`. Некоторые из них вызывают подстановку символа возврата каретки (**CR**) при выводе на терминал:

ONLCR	Преобразовать символ возврата каретки (**CR**) в символ возврата каретки (**CR**) и символ перевода строки (**NL**)
OCRNL	Преобразовать символ возврата каретки (**CR**) в символ перевода строки (**NL**)
ONOCR	Не выводить символ возврата каретки (**CR**) в нулевом столбце
ONLRET	Символ перевода строки (**NL**) выполняет функцию символа возврата каретки (**CR**)

Если установлен флаг ONLCR, то символы перевода строки **NL** преобразуются в последовательность **CR+NL** (символ возврата каретки и символ перевода строки). Это гарантирует, что каждая строка будет начинаться с левого края экрана. И наоборот, если установлен флаг OCRNL, то символ возврат каретки будет преобразовываться в символ перевода строки. Установка флага ONLRET сообщает драйверу терминала, что для используемого терминала символы перевода строки будут автоматически выполнять и возврат каретки. Если установлен флаг ONOCR, то символ возврата каретки не будет посылаться при выводе строки нулевой длины.

Большинство остальных флагов поля c_oflag относятся к задержкам в передаче, связанным с интерпретацией специальных символов, таких как перевод строки, табуляция, перевод страницы и др. Эти задержки учитывают время позиционирования указателя знакоместа, где должен быть выведен следующий символ на экране или принтере. Подробное описание этих флагов должно содержать справочное руководство системы.

Поле c_lflag

Возможно, наиболее интересным элементом структуры termios для программиста является поле c_lflag. Оно используется текущей дисциплиной линии связи для управления функциями терминала. Это поле содержит следующие флаги:

ICANON	Канонический построчный ввод
ISIG	Разрешить обработку прерываний
IEXTEN	Разрешить дополнительную (зависящую от реализации) обработку вводимых символов
ECHO	Разрешить отображение вводимых символов на экране
ECHOE	Отображать символ удаления как возврат-пробел-возврат
ECHOK	Отображать новую строку после удаления строки
ECHONL	Отображать перевод строки
NOFLSH	Отменить очистку буфера ввода после прерывания
TOSTOP	Посылать сигнал SIGTTOU при попытке вывода фонового процесса

Если установлен флаг ICANON, то включается канонический режим работы терминала. Как уже было видно выше, это позволяет использовать символы редактирования строки в процессе построчного ввода. Если флаг ICANON не установлен, то терминал находится в режиме прямого доступа (raw mode), который чаще всего используется полноэкранными программами и коммуникационными пакетами. Вызовы read будут при этом получать данные непосредственно из очереди ввода. Другими словами, основной единицей ввода будет одиночный символ, а не логическая строка. Программа при этом может считывать данные по одному символу (что обязательно для экранных редакторов) или большими блоками фиксированного размера (что удобно для коммуникационных программ). Но для того чтобы полностью управлять поведением вызова fdread, программист должен задать еще два дополнительных параметра. Это параметр VMIN, наименьшее число символов, которые должны быть приняты до возврата из вызова read, и параметр VTIME, максимальное время ожидания для вызова fdread. Оба параметра записываются в массиве c_cc. Это важная тема, которая будет подробно изучена в следующем разделе. А пока просто обратим внимание на то, как в следующем примере сбрасывается флаг ICANON.

```
uses stdio, linux;
```

```
var
  tdes:termios;
.
.
.
tcgetattr(0, tdes);

tdes.c_lflag := tdes.c_lflag and (not ICANON);

tcsetattr(0, TCSAFLUSH, tdes);
```

Если установлен флаг ISIG, то разрешается обработка клавиш прерывания (**intr**) и аварийного завершения (**quit**). Обычно это позволяет пользователю завершить выполнение программы. Если флаг ISIG не установлен, то проверка не выполняется, и символы **intr** и **quit** передаются программе без изменений.

Если установлен флаг ECHO, то символы будут отображаться на экране по мере их набора. Сброс этого флага полезен для процедур проверки паролей и программ, которые используют клавиатуру для особых функций, например, для перемещения курсора или команд экранного редактора.

Если одновременно установлены флаги ECHOE и ECHO, то символ удаления будет отображаться как последовательность символов **backspace-space-backspace** (возврат–пробел–возврат). При этом последний символ на терминале немедленно стирается с экрана, и пользователь видит, что символ действительно был удален. Если флаг ECHOE установлен, а флаг ECHO нет, то символ удаления будет отображаться как **space-backspace**, тогда при его вводе будет удаляться символ в позиции курсора алфавитно-цифрового терминала.

Если установлен флаг ECHONL, то перевод строки будет всегда отображать на экране, даже если отображение символов отключено, что может быть полезным при выполнении самим терминалом локального отображения вводима символов. Такой режим часто называется *полудуплексным режимом* (half-duplex mode).

Последним флагом, заслуживающим внимания в этой группе флагов, является флаг NOFLSH, который подавляет обычную очистку очередей ввода и вывод при нажатии клавиш **intr** и **quit** и очистку очереди ввода при нажатии клавиши **susp**.

Альтернативой TCGetAttr может быть вызов IOCtl:

```
uses Linux;

var
  tios : Termios;
begin
  IOCtl(1,TCGETS,@tios);
  WriteLn('Input Flags  : $',hexstr(tios.c_iflag,8));
  WriteLn('Output Flags : $',hexstr(tios.c_oflag,8));
  WriteLn('Line Flags   : $',hexstr(tios.c_lflag,8));
  WriteLn('Control Flags: $',hexstr(tios.c_cflag,8));
end.
```

Для удобства изменения параметров терминала в файле linux определена функция CFMakeRaw:

Описание

```
uses linux;

Procedure CFMakeRaw(var Tios:TermIOS);
```

CFMakeRaw устанавливает флаги в структуре Termios в состояние, соответствующее переводу терминала в неканонический режим. Пример:

```
uses Linux;
```

```
procedure ShowTermios(var tios:Termios);
begin
  WriteLn('Input Flags  : $',hexstr(tios.c_iflag,8)+#13);
  WriteLn('Output Flags : $',hexstr(tios.c_oflag,8));
  WriteLn('Line Flags   : $',hexstr(tios.c_lflag,8));
  WriteLn('Control Flags: $',hexstr(tios.c_cflag,8));
end;

var
  oldios,
  tios : Termios;
begin
  WriteLn('Old attributes:');
  TCGetAttr(1,tios);
  ShowTermios(tios);
  oldios:=tios;
  Writeln('Setting raw terminal mode');
  CFMakeRaw(tios);
  TCSetAttr(1,TCSANOW,tios);
  WriteLn('Current attributes:');
  TCGetAttr(1,tios);
  ShowTermios(tios);
  TCSetAttr(1,TCSANOW,oldios);
end.
```

Упражнение 9.3. *Напишите программу* `ttystate`*, которая выводит текущее состояние терминала, связанного со стандартным вводом. Эта программа должна использовать имена констант, описанные в этом разделе (*`ICANON`*, *`ECHOE`*, и т.д.). Найдите в справочном руководстве системы полный список этих имен.*

Упражнение 9.4. *Напишите программу* `ttyset`*, которая распознает выходной формат программы* `ttystate` *и настраивает терминал, связанный с ее стандартным выводом в соответствии с описанным состоянием. Есть ли какая-то польза от программ* `ttystate` *и* `ttyset`*, вместе или по отдельности?*

9.3.6. Параметры MIN и TIME

Параметры `MIN` и `TIME` имеют значение только при выключенном флаге `ICANON`. Они предназначены для тонкой настройки управления вводом данных. Параметр `MIN` задает минимальное число символов, которое должен получить драйвер терминала для возврата из вызова `fdread`. Параметр `TIME` задает значение максимального интервала ожидания; этот параметр обеспечивает еще один уровень управления вводом с терминала. Время ожидания измеряется десятыми долями секунды.

Значения параметров `MIN` и `MAX` находятся в массиве `c_cc` структуры `termios`, описывающей состояние терминала. Их индексы в массиве определяются постоянными `VMIN` и `VTIME` из файла `stdio`. Следующий фрагмент программы показывает, как можно задать их значения:

```
uses stdio, linux;

var
  tdes:termios;
  ttyfd:longint;

(* Получает текущее состояние *)
tcgetattr(ttyfd, tdes);

tdes.c_lflag := tdes.c_lflag and (not ICANON); (* Отключает канонический режим
*)
```

```
tdes.c_cc[VMIN] := 64;    (* В символах *)
tdes.c_cc[VTIME] := 2;    (* В десятых долях секунды *)

tcsetattr(0, TCSAFLUSH, &tdes);
```

Константы VMIN и VTIME обычно имеют те же самые значения, что и постоянные VEOF и VEOL. Это означает, что параметры MIN и TIME занимают то же положение, что и символы **eof** и **eol**. Следовательно, при переключении из канонического в неканонический режим нужно обязательно задавать значения параметров MIN и TIME, иначе может наблюдаться странное поведение терминала. (В частности, если символу **eof** соответствует комбинация клавиш **Ctrl+D**, то программа будет читать ввод блоками по четыре символа.) Аналогичная опасность возникает при возврате в канонический режим.

Существуют четыре возможных комбинации параметров MIN и TIME:

- *оба параметра* MIN *и* TIME *равны нулю.* При этом возврат из вызова fdread обычно происходит немедленно. Если в очереди ввода терминала присутствуют символы (напомним, что попытка ввода может быть осуществлена в любой момент времени), то они будут помещены в буфер процесса. Поэтому, если программа переводит свой управляющий терминал в режим прямого доступа при помощи сброса флага ICANON и оба параметра MIN и TIME равны нулю, то вызов
  ```
  nread := fdread(0, buffer, SOMESZ);
  ```
  вернет произвольное число символов от нуля до SOMESZ в зависимости от того, сколько символов находится в очереди в момент выполнения вызова;
- *параметр* MIN *больше нуля, а параметр* TIME *равен нулю.* В этом случае таймер не используется. Вызов fdread завершится только после того, как будут считаны MIN символов. Это происходит даже в том случае, если вызов read запрашивал меньше, чем MIN символов.

  В самом простом варианте параметр MIN равен единице, а параметр TIME – нулю, что приводит к возврату из вызова fdread после получения каждого символа из линии терминала. Это может быть полезно при чтении с клавиатуры терминала, хотя могут возникнуть проблемы с клавишами, посылающими последовательности из нескольких символов;
- *параметр* MIN *равен нулю, а параметр* TIME *больше нуля.* В этом случае параметр MIN не используется. Таймер запускается сразу же после выполнения вызова fdread. Возврат из вызова read происходит, как только вводится первый символ. Если заданный интервал времени истекает (то есть проход время, заданное в параметре TIME в десятых долях секунды), то вызов read возвращает нулевые символы;
- *оба параметра* MIN *и* TIME *больше нуля.* Это, возможно, наиболее полезный и гибкий вариант. Таймер запускается после получения первого символа, а не при входе в вызов fdread. Если MIN символов будут получены до истечения заданного интервала времени, то происходит возврат из вызова fdread. Если таймер срабатывает раньше, то в программу пользователя возвращаются только символы, находящиеся при этом в очереди ввода. Этот режим работы полезен при поступлении ввода пакетами, которые посылаются в течение коротких интервалов времени. При этом упрощается программирование и уменьшается число необходимых системных вызовов. Такой режим также полезен, например, при работе с функциональными клавишами, которые посылают при нажатии на них сразу несколько символов.

9.3.7. Другие системные вызовы для работы с терминалом

Есть несколько дополнительных системных вызовов для работы с терминалом, позволяющих программисту до некоторой степени управлять очередями ввода и вывода, поддерживаемыми драйвером терминала. Эти вызовы определены следующим образом.

Описание

```
uses linux;

Function TCFlush(ttyfd, queue:longint):Boolean;

Function TCDrain(ttyfd:longint):Boolean;

Function TCFlow(ttyfd, actions:longint):Boolean;

Function TCSendBreak(ttyfd, duration:longint):longint;

Function TCGetPGrp(Fd:longint;var Id:longint):boolean;

Function TCSetPGrp(Fd,Id:longint):boolean;
```

Вызов `tcflush` очищает заданную очередь. Если параметр `queue` имеет значение `TCIFLUSH` (определенное в файле `stdio`), то очищается очередь ввода. Это означает, что все символы в очереди ввода сбрасываются. Если параметр `queue` имеет значение `TCOFLUSH`, то очищается очередь вывода. При значении `TCIOFLUSH` параметра `queue` очищаются и очередь ввода, и очередь вывода.

Вызов `tcdrain` приводит к приостановке работы процесса до тех пор, пока текущий вывод не будет записан в терминал `ttyfd`.

Вызов `tcflow` обеспечивает старт/стопное управление драйвером терминала. Если параметр `actions` равен `TCOOFF`, то вывод приостанавливается. Он может быть возобновлен при помощи еще одного вызова `tcflow` со значением параметра `actions` равным `TCOON`. Вызов `tcflow` также может использоваться для посылки драйверу терминала специальных символов `START` и `STOP`, это происходит при задании значения параметра `actions` равного `TCIOFF` или `TCION` соответственно. Специальные символы `START` и `STOP` служат для приостановки и возобновления ввода с терминала.

Вызов `TCSendBreak` используется для посылки сигнала прерывания сеанса связи, которому соответствует посылка нулевых битов в течение времени, заданного параметром `duration`. Если параметр `duration` равен 0, то биты посылаются в течение не менее четверти секунды и не более полсекунды. Если параметр `duration` не равен нулю, то биты будут посылаться в течение некоторого промежутка времени, длительность которого зависит от значения параметра `duration` и конкретной реализации.

`TCSetPGrp` устанавливает, а `TCGetPGrp` – получает идентификатор группы фоновых процессов, сохраняя его в `Id`.

9.3.8. Сигнал разрыва соединения

В главе 6 упоминалось, что сигнал разрыва соединения `SIGHUP` посылается членам сеанса в момент завершения работы лидера сеанса (при условии, что он имеет управляющий терминал). Этот сигнал также имеет другое применение в средах, в которых соединение между компьютером и терминалом может быть разорвано (тогда пропадает сигнал несущей в линии терминала). Это может происходить, например, если терминалы подключены через телефонную сеть или с помощью некоторых типов локальных сетей. В этих обстоятельствах драйвер терминала должен послать сигнал `SIGHUP` всем процессам, которые считают данный терминал своим управляющим терминалом. Если этот сигнал не перехватывается, то он приводит к завершению работы программы. (В отличие от сигнала `SIGINT`, сигнал `SIGHUP` обычно завершает и процесс оболочки. В результате пользователь автоматически отключается от системы при нарушении его связи с системой – такой подход необходим для обеспечения безопасности.)

Обычно программисту не нужно обрабатывать сигнал `SIGHUP`, так как он служит нужным целям. Тем не менее может потребоваться перехватывать его для выполнения

некоторых операций по «наведению порядка» перед выходом из программы; вот как это можно сделать:

```
uses linux;

procedure hup_action(sig:integer);cdecl;forward;

var
  act:sigactionrec;
.
.
.
act.handler.sh:=@hup_action;
sigaction(SIGHUP, @act, nil);
```

Этот подход используется некоторыми редакторами, сохраняющими редактируемый файл и отсылающими пользователю сообщение перед выходом. Если сигнал `SIGHUP` полностью игнорируется (установкой значения `act.handler.sh` равного `SIG_IGN`) и терминал разрывает соединение, то следующие попытки чтения из терминала будут возвращать 0 для обозначения «конца файла».

9.4. Псевдотерминалы

Еще одно применение модуля дисциплины линии связи заключается в поддержке работы так называемого *псевдотерминала* (pseudo terminal), применяемого для организации дистанционного доступа через сеть. Псевдотерминал предназначен для обеспечения соединения терминала одного компьютера с командным интерпретатором другого компьютера. Пример такого соединения приведен на рис. 9.4. На нем пользователь подключен к компьютеру A (клиенту), но использует командный интерпретатор на компьютере B (сервере). Стрелки на схеме показывают направление пересылки вводимых с клавиатуры символов. Здесь схема несколько упрощена за счет исключения стеков сетевых протоколов на клиентской и серверной системе.

Когда пользователь подключается к командному интерпретатору на другом компьютере (обычно при помощи команды `rlogin`), то локальное терминальное соединение должно быть изменено. По мере того как данные считываются с локального терминала, они должны без изменений передаваться через модуль дисциплины линии связи клиентскому процессу `rlogin`, выполняющемуся на локальном компьютере. Поэтому дисциплина линии связи на локальном компьютере должна работать в режиме прямого доступа. Следующий фрагмент должен напомнить, как включается такой режим:

```
uses stdio, linux;

var
  attr:termios;
.
.
.
(* Получает текущую дисциплину линии связи *)
tcgetattr(0, attr);

(* Возврат из вызова fdread разрешен только после считывания одного символа *)
attr.c_cc[VMIN] := 1;
attr.c_cc(VTIME] := 0;
attr.c_lflag := attr.c_lflag and not (ISIG or ECHO or ICANON);

(* Устанавливает новую дисциплину линии связи *)
tcsetattr(0, TCSAFLUSH, attr);
```

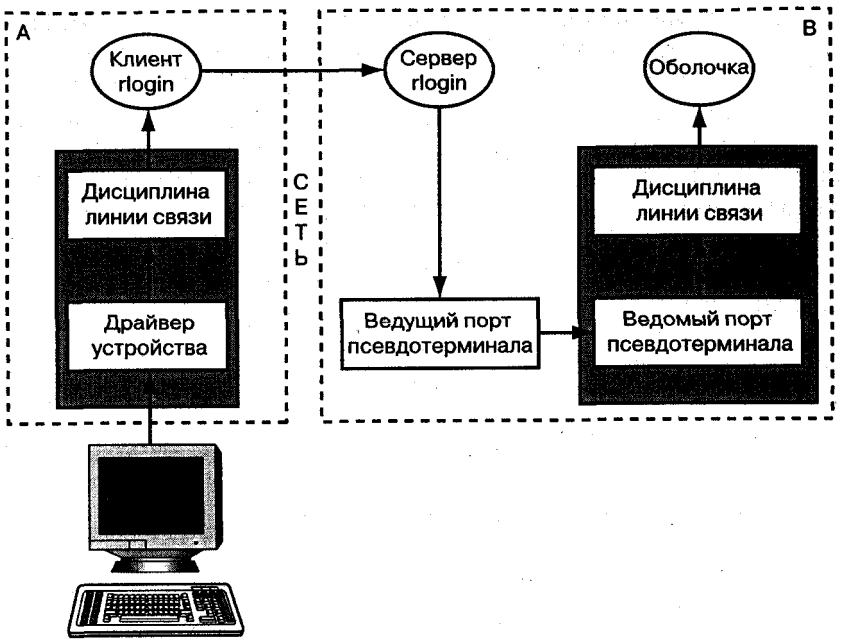


Рис. 9.4. Удаленный вход в систему в ОС UNIX

Теперь процесс клиента `rlogin` может передавать данные по сети в исходном виде.

Когда сервер (B) получает первоначальный запрос на вход в систему, он выполняет вызовы `fork` и `exec`, порождая новый командный интерпретатор. С новым командным интерпретатором не связан управляющий терминал, поэтому создается псевдотерминал, который имитирует обычный драйвер устройства терминала. Псевдотерминал (*pseudo tty*) действует подобно двунаправленному каналу и просто позволяет двум процессам передавать данные. В рассматриваемом примере псевдотерминал связывает командный интерпретатор с соответствующим сетевым процессом. Псевдотерминал является парой устройств, которые называются ведущим и ведомым устройствами, или портами псевдотерминала. Сетевой процесс открывает ведущий порт устройства псевдотерминала, а затем выполняет запись и чтение. Процесс командного интерпретатора открывает ведомый порт, а затем работает с ним (через дисциплину линии связи). Данные, записываемые в ведущий порт, попадают на вход ведомого порта, и наоборот. В результате пользователь на клиентском компьютере (A) как бы напрямую обращается к командному интерпретатору, который на самом деле выполняется на сервере (B). Аналогично, по мере того как данные выводятся командным интерпретатором на сервере, они обрабатываются дисциплиной линии связи на сервере (которая работает в каноническом режиме) и затем передаются без изменений клиентскому терминалу, не подвергаясь модификации со стороны дисциплины линии связи клиента.

Хотя средства, при помощи которых выполняется инициализация псевдотерминалов, были улучшены в новых версиях ОС UNIX и спецификации XSI, они все еще остаются довольно громоздкими. Система UNIX обеспечивает конечное число псевдотерминалов, и процесс командного интерпретатора должен открыть следующий доступный псевдотерминал. В системе SVR4 это выполняется и помощи открытия устройства `/dev/ptmx`, которое определяет и открывает первое неиспользуемое ведущее устройство псевдотерминала. С каждым ведущим устройством связано ведомое устройство. Для того, чтобы предотвратить открытие ведомого устройства другим процессом, открытие устройства `/dev/ptmx` также блокирует соответствующее ведомое устройство.

```
uses linux;
var
  mfd:longint;
.
.
.
```

```
  (* Открыть псевдотерминал -
   * получить дескриптор файла главного устройства *)
  masterfd := fdopen ('/dev/ptmx', Open_RDWR);
  if masterfd = -1 then
  begin
    perror('Ошибка при открытии главного устройства');
    halt(1);
  end;
```

Перед тем как открыть и «разблокировать» ведомое устройство, необходимо убедиться, что только один процесс с соответствующими правами доступа сможет выполнять чтение из устройства и запись в него. Функция `grantpt` изменяет режим доступа и идентификатор владельца ведомого устройства в соответствии с параметрами связанного с ним главного устройства. Функция `unlockpt` снимает флаг, блокирующий ведомое устройство (то есть делает его доступным). Далее нужно открыть ведомое устройство. Но его имя пока еще не известно. Функция `ptsname` возвращает имя ведомого устройства, связанного с заданным ведущим устройством, которое обычно имеет вид `/dev/pts/pttyXX`. Следующий фрагмент демонстрирует последовательность необходимых действий:

```
uses stdio, linux;

var
  mfd, sfd:longint;
  slavenm:pchar;
.
.
.
(* Открываем ведущее устройство, как и раньше *)
mfd := fdopen ('/dev/ptmx', Open_RDWR);
if mfd = -1 then
begin
  perror('Ошибка при открытии ведущего устройства');
  halt(1);
end;
(* Изменяем права доступа ведомого устройства *)
if grantpt (mfd) = -1 then
begin
  perror('Невозможно разрешить доступ к ведомому устройству');
  halt(1);
end;
(* Разблокируем ведомое устройство, связанное с mfd *)
if unlockpt(mfd) = -1 then
begin
  perror('Невозможно разблокировать ведомое устройство');
  halt(1);
end;
(* Получаем имя ведомого устройства и затем пытаемся открыть его *)
slavenm := ptsname (mfd);
if slavenm = nil then
begin
  perror('Невозможно получить имя ведомого устройства');
  halt(1);
end;
sfd := fdopen (slavenm, Open_RDWR);
if slavefd = -1 then
begin
  perror('Ошибка при открытии ведомого устройства');
  halt(1);
end;
```

Теперь, когда получен доступ к драйверу устройства псевдотерминала, нужно установить для него дисциплину линии связи. До сих пор дисциплина линии связи рассматривалась как единое целое, тогда как в действительности она состоит из набора внутренних модулей ядра, известных как модули *STREAM*. Стандартная дисциплина линии связи псевдотерминала состоит из трех модулей: `ldterm` (модуль дисциплины линии связи терминала), `ptem` (модуль эмуляции псевдотерминала) и ведомой части псевдотерминала. Вместе они работают как настоящий терминал. Эта конфигурация показана на рис. 9.5.

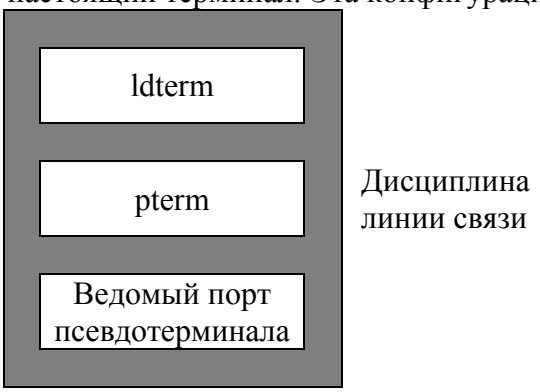


Рис. 9.5. Дисциплина линии связи в виде модулей STREAM для устройства псевдотерминала

Для создания дисциплины линии связи нужно «вставить» дополнительные модули STREAM в ведомое устройство. Это достигается при помощи многоцелевой функции `ioctl`, например:

```
(*
 * Заголовочный файл stdio содержит интерфейс STREAMS
 * и определяет макрокоманду I_PUSH, используемую в качестве
 * второго аргумента функции ioctl().
 *)

uses stdio;
.
.
.
(* Открываем ведущее и ведомое устройства, как и раньше *)
(* Вставляем два модуля в ведомое устройство *)
ioctl(sfd, I_PUSH, 'ptem');
ioctl(sfd, I_PUSH, 'ldterm');
```

Обратимся теперь к главному примеру, программе `tscript`, которая использует псевдотерминал в пределах одного компьютера для перехвата вывода командного интерпретатора в процессе интерактивного сеанса, не влияя на ход этого сеанса. (Эта программа аналогична команде UNIX script.) Данный пример можно расширить и для дистанционного входа через сеть.

9.5. Пример управления терминалом: программа tscript

Программа `tscript` устроена следующим образом: при старте она выполняет вызовы `fork` и `exec` для запуска пользовательской оболочки. Далее все данные, записываемые на терминал оболочкой, сохраняются в файле, при этом оболочка ничего об этом не знает и продолжает вести себя так, как будто она полностью управляет дисциплиной линии связи и, следовательно, терминалом. Логическая структура программы `tscript` показана на рис. 9.6.

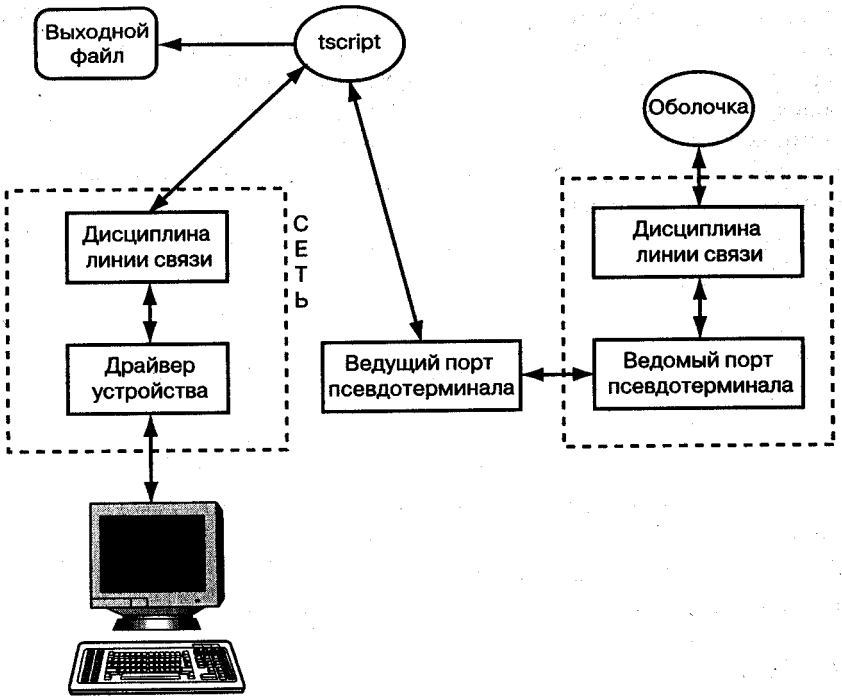


Рис. 9.6. Использование псевдотерминала в программе tscript

Основные элементы схемы:

`tscript` Первый запускаемый процесс. После инициализации псевдотерминала и дисциплины линии связи этот процесс использует вызовы `fork` и `exec` для создания оболочки `shell`. Теперь программа `tscript` играет две роли. Первая состоит в чтении из настоящего терминала и записи всех данных в порт ведущего устройства псевдотерминала. (Все данные, записываемые в ведущее устройство псевдотерминала, непосредственно передаются на ведомое устройство псевдотерминала.) Вторая роль состоит в чтении вывода программы оболочки `shell` при помощи псевдотерминала и копировании этих данных на настоящий терминал и в выходной файл

`shell` Пользовательская оболочка. Перед запуском процесса `shell` модули дисциплины линии связи STREAM вставляются в ведомое устройство. Стандартный ввод, стандартный вывод и стандартный вывод диагностики оболочки перенаправляются в ведомое устройство псевдотерминала

Первая задача программы `tscript` состоит в установке обработчика сигнала `SIGCHLD` и в открытии псевдотерминала. Затем программа создает процесс `shell`. И, наконец, вызывается процедура `script`. Эта процедура отслеживает два потока данных: ввод с клавиатуры (стандартный ввод), который она передает ведущему устройству псевдотерминала, и ввод с ведущего устройства псевдотерминала, передаваемый на стандартный вывод и записываемый в выходной файл.

```
(* Программа tscript управление терминалом *)
(* Хотя в Linux этот пример и не работает... *)
uses linux,stdio;

var
  dattr:termios;

var
```

```
  act:sigactionrec;
  mfd, sfd:longint;
  err:integer;
  buf:array [0..511] of char;
  mask:sigset_t;
begin
  (* Сохранить текущие установки терминала *)
  tcgetattr (0, dattr);
  (* Открыть псевдотерминал *)
  err := pttyopen (mfd, sfd);
  if err <> 1 then
  begin
    writeln (stderr, 'pttyopen: ', err);
    perror ('Ошибка при открытии псевдотерминала');
    halt (1);
  end;
  (* Установить обработчик сигнала SIGCHLD *)
  act.handler.sh := @catch_child;
  sigfillset (@mask);
  act.sa_mask:=mask.__val[0];
  sigaction (SIGCHLD, @act, nil);
  (* Создать процесс оболочки *)
  case fork of
    -1:                (* ошибка *)
    begin
      perror ('Ошибка вызова оболочки');
      halt (2);
    end;
    0:                 (* дочерний процесс *)
    begin
      fdclose (mfd);
      runshell (sfd);
    end;
    else          (* родительский процесс *)
    begin
      fdclose (sfd);
      script (mfd);
    end;
  end;
end.
```

Основная программа использует четыре процедуры. Первая из них называется `catch_child`. Это обработчик сигнала `SIGCHLD`. При получении сигнала `SIGCHLD` процедура `catch_child` восстанавливает атрибуты терминала и завершает работу.

```
procedure catch_child (signo:integer);cdecl;
begin
  tcsetattr (0, TCSAFLUSH, dattr);
  halt (0);
end;
```

Вторая процедура, `pttyopen`, открывает псевдотерминал.

```
function pttyopen (var masterfd, slavefd:longint):integer;
var
  slavenm:pchar;
begin
  (* Открыть псевдотерминал -
   * получить дескриптор файла главного устройства *)
  masterfd := fdopen ('/dev/ptmx', Open_RDWR);
  if masterfd = -1 then
```

```
  begin
    pttyopen:=-1;
    exit;
  end;
  (* Изменить права доступа для ведомого устройства *)
  if grantpt (masterfd) = -1 then
  begin
    fdclose (masterfd);
    pttyopen:=-2;
    exit;
  end;
  (* Разблокировать ведомое устройство, связанное с mfd *)
  if unlockpt (masterfd) = -1 then
  begin
    fdclose (masterfd);
    pttyopen:=-3;
    exit;
  end;
  (* Получить имя ведомого устройства и затем открыть его *)
  slavenm := ptsname (masterfd);
  if slavenm = nil then
  begin
    fdclose (masterfd);
    pttyopen:=-4;
    exit;
  end;
  slavefd := fdopen (slavenm, Open_RDWR);
  if slavefd = -1 then
  begin
    fdclose (masterfd);
    pttyopen:=-5;
    exit;
  end;
  (* Создать дисциплину линии связи *)
  ioctl (slavefd, I_PUSH, pchar('ptem'));
  if linuxerror>0 then
  begin
    fdclose (masterfd);
    fdclose (slavefd);
    pttyopen:=-6;
    exit;
  end;
  ioctl (slavefd, I_PUSH, pchar('ldterm'));
  if linuxerror>0 then
  begin
    fdclose (masterfd);
    fdclose (slavefd);
    pttyopen:=-7;
    exit;
  end;
  pttyopen:=1;
end;
```

Третья процедура – процедура `runshell`. Она выполняет следующие задачи:

- вызывает `setpgrp`, чтобы оболочка выполнялась в своей группе процессов. Это позволяет оболочке полностью управлять обработкой сигналов, в особенности в отношении управления заданиями;
- вызывает системный вызов `dup2` для перенаправления дескрипторов `stdin`,

stdout и stderr на дескриптор файла ведомого устройства. Это особенно важный шаг;

– запускает оболочку при помощи вызова exec, которая выполняется до тех пор, пока не будет прервана пользователем.

```
procedure runshell (sfd:longint);
begin
  setpgrp;
  dup2 (sfd, 0);
  dup2 (sfd, 1);
  dup2 (sfd, 2);
  execl ('/bin/sh -i');
end;
```

Теперь рассмотрим саму процедуру script. Первым действием процедуры script является изменение дисциплины линии связи так, чтобы она работала в режиме прямого доступа. Это достигается получением текущих атрибутов терминала и изменением их при помощи вызова tcsetattr. Затем процедура script открывает файл output и использует системный вызов select (обсуждавшийся в главе 7) для обеспечения одновременного ввода со своего стандартного ввода и ведущего устройства псевдотерминала. Если данные поступают со стандартного ввода, то процедура script передает их без изменений ведущему устройству псевдотерминала. При поступлении же данных с ведущего устройства псевдотерминала процедура script записывает эти данные в терминал пользователя и в файл output.

```
procedure script(mfd:longint);
var
  nread, ofile:longint;
  _set, master:fdset;
  attr:termios;
  buf:array [0..511] of char;
begin
  (* Перевести дисциплину линии связи в режим прямого доступа *)
  tcgetattr (0, attr);
  attr.c_cc[VMIN] := 1;
  attr.c_cc[VTIME] := 0;
  attr.c_lflag := attr.c_lflag and not (ISIG or ECHO or ICANON);
  tcsetattr (0, TCSAFLUSH, attr);

  (* Открыть выходной файл *)
  ofile := fdopen ('output', Open_CREAT or Open_WRONLY or Open_TRUNC,
octal(0666));

  (* Задать битовые маски для системного вызова select *)
  FD_ZERO (master);
  FD_SET (0, master);
  FD_SET (mfd, master);

  (* Вызов select осуществляется без таймаута,
   * и будет заблокирован до наступления события. *)
  _set := master;
  while select (mfd + 1, @_set, nil, nil, nil) > 0 do
  begin
    (* Проверить стандартный ввод *)
    if FD_ISSET (0, _set) then
    begin
      nread := fdread (0, buf, 512);
      fdwrite (mfd, buf, nread);
    end;
```

```
    (* Проверить главное устройство *)
    if FD_ISSET (mfd, _set) then
    begin
      nread := fdread (mfd, buf, 512);
      write (ofile, buf, nread);
      write (1, buf, nread);
    end;
    _set := master;
  end;
end;
```

Следующий сеанс демонстрирует работу программы `tscript`. Комментарии, обозначенные символом #, показывают, какая из оболочек выполняется в данный момент.

```
$ ./tscript

$ ls -l tscript     # работает новая оболочка
-rwxr-xr-x   1 spate    fcf 6984 Jan 22 21:57 tscript

$ head -2 /etc/passwd # выполняется в новой оболочке
root:x:0:1:0000-Admin(0000):/:/bin/ksh
daemon:x:1:1:0000-Admin(0000):/:

$ exit         # выход из новой оболочки

$ cat output       # работает исходная оболочка
-rwxr-xr-x   1 spate     fcf 6984 Jan 22 21:57 tscript
root:x:0:1:0000-Admin(0000):/:/bin/ksh
daemon:x:1:1:0000-Admin(0000):/:
```

Упражнение 9.5. *Добавьте к программе обработку ошибок и возможность задания в качестве параметра имени выходного файла. Если имя не задано, используйте по умолчанию имя* `output`.

Упражнение 9.6. *Эквивалентная стандартная программа UNIX* `script` *позволяет задать параметр* `-a`*, который указывает на необходимость дополнения файла* `output` *(содержимое файла не уничтожается). Реализуйте аналогичную возможность в программе* `tscript`.

Глава 10.Сокеты

10.1. Введение

В предыдущих главах был изучен ряд механизмов межпроцессного взаимодействия системы UNIX. В настоящее время пользователями и разработчиками часто используются и сетевые среды, предоставляющие возможности технологи клиент/сервер. Такой подход позволяет совместно использовать данные, дисковое пространство, периферийные устройства, процессорное время и другие ресурсы. Работа в сетевой среде по технологии клиент/сервер предполагает взаимодействие процессов, находящихся на клиентских и серверных системах, разделенных средой передачи данных.

Исторически сетевые средства UNIX развивались двумя путями. Разработчики Berkeley UNIX создали в начале 80-х годов известный и широко применяемый интерфейс сокетов, а разработчики System V выпустили в 1986 г. Transport Level Interface (интерфейс транспортного уровня, сокращенно TLI). Раздел сетевого программирования в документации по стандарту X/Open часто называют спецификацией XTI. Спецификация XTI включает и интерфейс сокетов, и интерфейс TLI. Основные понятия являются общими для обеих реализации, но интерфейс TLI использует намного больше структур данных, и его реализация гораздо сложнее, чем реализация интерфейса сокетов. Поэтому в этой главе будет рассмотрен хорошо известный и испытанный интерфейс сокетов. Они обеспечивают простой программный интерфейс, применимый как для связи процессов на одном компьютере, так и для связи процессов через сети. Целью сокетов является обеспечение средства межпроцессного взаимодействия для двунаправленного обмена сообщениями между двумя процессами независимо от того, находятся ли они на одном или на разных компьютерах.

Глава будет посвящена краткому ознакомлению с основными понятиями и средствами работы с сокетами. При необходимости продолжить их изучение следует обратиться к более подробному руководству (например, книге «Advanced Programming in the UNIX Environment» У.Р. Стивенса) или к специалисту по сетевому программированию.

Кроме этого, необходимо отметить, что во время компоновки программ может понадобиться подключение сетевых библиотек – за рекомендациями обратитесь к справочному руководству системы.

10.2. Типы соединения

Если процессам нужно передать данные по сети, они могут выбрать для этого один из двух способов связи. Процесс, которому нужно посылать неформатированный, непрерывный поток символов одному и тому же абоненту, например, процессу удаленного входа в систему, может использовать *модель соединения* (connection oriented model) или *виртуальное соединение* (virtual circuit). В других же случаях (например, если серверу нужно разослать сообщение клиентам, не проверяя его доставку) процесс может использовать *модель дейтаграмм* (connectionless oriented model). При этом процесс может посылать сообщения (*дейтаграммы*) по произвольным адресам через один и тот же сокет без предварительного установления связи с этими адресами. Термин *дейтаграмма* (datagram) обозначает пакет пользовательского сообщения, посылаемый через сеть. Для облегчения понимания приведем аналогию: модель виртуальных соединений напоминает телефонную сеть, а модель дейтаграмм – пересылку писем по почте. Поэтому в последнем случае нельзя быть абсолютно уверенным, что сообщение дошло до адресата, а если необходимо получить ответ на него, то нужно указать свой обратный адрес на конверте. Модель соединений будет более подходящей при необходимости получения тесного взаимодействия между системами, когда обмен сообщениями и подтверждениями происходит в определенном порядке. Модель без соединений является более эффективной и лучше подходит в таких случаях, как рассылка широковещательных сообщений большому числу компьютеров.

Для того чтобы взаимодействие между процессами на разных компьютерах стало

возможным, они должны быть связаны между собой как на аппаратном уровне при помощи сетевого оборудования – кабелей, сетевых карт и различных устройств маршрутизации, так и на программном уровне при помощи стандартного набора сетевых протоколов. Протокол представляет собой просто набор правил, в случае сетевого протокола – набор правил обмена сообщениями между компьютерами. Поэтому в системе UNIX должны существовать наборы правил для обеих моделей – как для модели соединений, так и для модели дейтаграмм. Для модели соединений используется *протокол управления передачей* (Transmission Control Protocol, сокращенно TCP), а для модели дейтаграмм – *протокол пользовательских дейтаграмм* (User Datagram Protocol, сокращенно UDP).[1]

10.3. Адресация

Чтобы процессы могли связаться по сети, должен существовать механизм определения *сетевого адреса* (network address) компьютера, на котором находится другой процесс. В конечном счете адрес определяет физическое положение компьютера в сети. Обычно адреса состоят из нескольких частей, соответствующих различным уровням сети. Далее будут затронуты только те вопросы, без ответов на которые не обойтись при программировании с использованием сокетов.

10.3.1. Адресация Internet

Сейчас почти во всех глобальных сетях применима *адресация IP* (сокращение от Internet Protocol – межсетевой протокол, протокол сети Интернет).

Адрес IP состоит из четырех десятичных чисел, разделенных точками, например:

```
197.124.10.1
```

Эти четыре числа содержат достаточную информацию для определения сети назначения, а также компьютера в этой сети; собственно, термин Internet и означает «сеть сетей».

Сетевые вызовы UNIX не могут работать с IP адресами в таком формате. На программном уровне IP адреса хранятся в структуре типа `in_addr_t`. Обычно программистам не нужно знать внутреннее представление этого типа, так как для преобразования IP адреса в структуру типа `in_addr_t` предназначена процедура `inet_addr`.

Описание

```
uses stdio;

function inet_addr(ip_address:pchar):in_addr_t;
```

Процедура `inet_addr` принимает IP адрес в форме строки вида `1.2.3.4` и возвращает адрес в виде структуры соответствующего типа. Если вызов процедуры завершается неудачей из-за неверного формата IP адреса, то возвращаемое значение будет равно `in_addr_t(-1)`, например:

```
var
  server:in_addr_t;

server := inet_addr('197.124.10.1');
```

Для того чтобы процесс мог ссылаться на адрес своего компьютера, в заголовочном файле `stdio` определена постоянная `INADDR_ANY`, содержащая локальный адрес компьютера в формате `in_addr_t`.

10.3.2. Порты

Кроме адреса компьютера, клиентская программа должна иметь возможность

[1] Протокол UDP не гарантирует доставку дейтаграмм; кроме того, может быть нарушен исходный порядок сообщений, допускается также случайное дублирование дейтаграмм. Приложения, использующие протокол UDP, должны обеспечивать контроль данных на прикладном уровне, для этого может потребоваться организация подтверждения доставки и повторной пересылки данных.

подключения к нужному серверному процессу. Серверный процесс ждет подключения к заданному *номеру порта* (port number). Поэтому клиентский процесс должен выполнить запрос на подключение к определенному порту на заданном компьютере. Если продолжить аналогию с пересылкой писем по почте, то это равносильно дополнению адреса номером комнаты или квартиры.

Некоторые номера портов по соглашению считаются отведенными для стандартных сервисов, таких как `ftp` или `rlogin`. Эти номера портов записаны в файле `/etc/services`. В общем случае порты с номерами, меньшими 1024, считаются зарезервированными для системных процессов UNIX. Все остальные порты доступны для пользовательских процессов.

10.4. Интерфейс сокетов

Для хранения информации об адресе и порте адресата (абонента) существуют стандартные структуры. Обобщенная структура адреса сокета определяется в модуле `sockets` следующим образом:

```
TSockAddr=packed Record
  family:word;                    (* Семейство адресов *)
  data  :array [0..13] of char;   (* Адрес сокета *)
end;
```

Эта структура называется *обобщенным сокетом* (generic socket), так как в действительности применяются различные типы сокетов в зависимости от того, используются ли они в качестве средства межпроцессного взаимодействия на одном и том же компьютере или для связи процессов через сеть. Сокеты для связи через сеть имеют следующую форму:

```
uses sockets;


TInetSockAddr = packed Record
  family  : Word;                 (* Семейство адресов *)
  port    : Word;                 (* Номер порта *)
  addr    : Cardinal;             (* IP-адрес *)
  pad     : array [1..8] of byte; (* Поле выравнивания *)
end;
```

10.4.1. Создание сокета

При любых моделях связи клиент и сервер должны создать *абонентские точки* (transport end points,), или сокеты, которые являются дескрипторами, используемыми для установки связи между процессами в сети. Они создаются при помощи системного вызова `socket`.

Описание

```
uses sockets;

Function Socket(Domain,SocketType,Protocol:Longint):Longint;
```

Параметр `domain` определяет коммуникационный домен, в котором будет использоваться сокет. Например, значение `AF_INET` определяет, что будет использоваться домен Internet. Интерес может представлять также другой домен, `AF_UNIX`, который используется, если процессы находятся на одном и том же компьютере.

Параметр `SocketType` определяет тип создаваемого сокета. Значение `SOCK_STREAM` указывается при создании сокета для работы в режиме виртуальных соединений, а значение `SOCK_DGRAM` – для работы в режиме пересылок дейтаграмм. Последний параметр `protocol` определяет используемый протокол. Этот параметр обычно задается равным нулю, при этом по умолчанию сокет типа `SOCK_STREAM` будет использовать протокол TCP, а сокет типа `SOCK_DGRAM` – протокол UDP. Оба данных протокола являются стандартными протоколами

UNIX. Поэтому виртуальное соединение часто называют TCP-соединением, а пересылку дейтаграмм – работой с UDP-сокетами.

Системный вызов `socket` обычно возвращает неотрицательное целое число, которое является дескриптором файла сокета, что позволяет считать механизм сокетов разновидностью обобщенного файлового ввода/вывода UNIX.

10.5. Программирование в режиме TCP-соединения

Для того чтобы продемонстрировать основные системные вызовы для работы с сокетами, рассмотрим пример, в котором клиент посылает серверу поток строчных символов через TCP-соединение. Сервер преобразует строчные символы в прописные и посылает их обратно клиенту. В следующих разделах этой главы приведем тот же самый пример, но использующий сокеты UDP-протокола.

Сначала составим план реализации серверного процесса:

```
(* Серверный процесс *)

(* Включает нужные заголовочные файлы *)
uses sockets,stdio,linux;


var
  sockfd:longint;
begin
  (* Установить абонентскую точку сокета *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

(* 'Связывание' адреса сервера с сокетом

Ожидание подключения

Цикл
  установка соединения
  создание дочернего процесса для работы с соединением
  если это дочерний процесс,
  то нужно в цикле принимать данные от клиента и посылать ему ответы
*)
end.
```

План клиентского процесса выглядит следующим образом:

```
(* Клиентский процесс *)

(* Включает нужные заголовочные файлы *)

var
  sockfd:longint;
begin
  (* Создает сокет *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;
```

```
  (* Соединяет сокет с адресом серверного процесса *)
  (* В цикле посылает данные серверу и принимает от него ответы *)
end.
```

Далее будем постепенно превращать эти шаблоны в настоящие программы, начиная с реализации сервера.

10.5.1. Связывание

Системный вызов `bind` связывает сетевой адрес компьютера с идентификатором сокета.

Описание

```
uses sockets;

Function Bind(sockfd:Longint; Var address; add_len:Longint):Boolean;
Function Bind(sockfd:longint; const address:string):boolean;
```

Первый параметр, `sockfd`, является дескриптором файла сокета, созданным с помощью вызова `socket`, а второй – указателем на обобщенную структуру адреса сокета или адрес в форме строки. В рассматриваемом примере данные пересылаются по сети, поэтому в действительности в качестве этого параметра будет задан адрес структуры `TInetSockAddr`, содержащей информацию об адресе нашего сервера. Последний параметр содержит размер указанной структуры адреса сокета. В случае успешного завершения вызова `bind` он возвращает значение 0. В случае ошибки, например, если сокет для этого адреса уже существует, вызов `bind` возвращает значение -1. Переменная `linuxerror` будет иметь при этом значение `Sys_EADDRINUSE`.

10.5.2. Включение приема TCP-соединений

После выполнения связывания с адресом и перед тем, как какой-либо клиент сможет подключиться к созданному сокету, сервер должен включить прием соединений. Для этого служит вызов `listen`.

Описание

```
uses sockets;

Function Listen(sockfd, queue_size:Longint):Boolean;
```

Параметр `sockfd` имеет то же значение, что и в предыдущем вызове. В очереди сервера может находиться не более `queue_size` запросов на соединение. (Спецификация XSI определяет минимальное ограничение сверху на длину очереди равное пяти.)

10.5.3. Прием запроса на установку TCP-соединения

Когда сервер получает от клиента запрос на соединение, он должен создать новый сокет для работы с новым соединением. Первый же сокет используется только для установки соединения. Дополнительный сокет создается при помощи вызова `accept`, принимающего очередное соединение.

Описание

```
uses sockets;

Function Accept(sockfd:Longint;Var address;Var add_len:Longint):Longint;
Function Accept(sockfd:longint;var address:string;
                var SockIn,SockOut:text):Boolean;
Function Accept(sockfd:longint;var address:string;
                var SockIn,SockOut:File):Boolean;
Function Accept(sockfd:longint;var address:TInetSockAddr;
                var SockIn,SockOut:File):Boolean;
```

Системному вызову `accept` передается дескриптор сокета, для которого ведется прием соединений. Возвращаемое значение соответствует идентификатору нового сокета,

который будет использоваться для связи. Параметр address заполняется информацией о клиенте. Так как связь использует соединение, адрес клиента знать не обязательно, поэтому можно присвоить параметру address значение nil. Если значение address не равно nil, то переменная, на которую указывает параметр add_len, первоначально должна содержать размер структуры адреса, заданной параметром address. После возврата из вызова accept переменная add_len будет содержать реальный размер записанной структуры.

Вторая, третья и четвертая формы вызова accept эквивалентны вызову первой с последующим использованием функции Sock2Text, преобразующей сокет sockfd в две файловые переменные типа Text, одна из которых отвечает за чтение из сокета (SockIn), а другая – за запись в сокет (SockOut).

После подстановки вызовов bind, listen и accept текст программы сервера примет вид:

```
(* Серверный процесс *)
uses sockets,stdio,linux;

const
  SIZE=sizeof(tinetsockaddr);
  (* Инициализация сокета Internet с номером порта 7000
   * и локальным адресом, заданным в постоянной INADDR_ANY *)
  server:tinetsockaddr = (family:AF_INET; port:7000; addr:INADDR_ANY);

var
  newsockfd:longint;
  sockfd:longint;

begin
  (* Создает сокет *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

  (* Связавает адрес с сокетом *)
  if not bind (sockfd, server, SIZE) then
  begin
    perror ('Ошибка вызова bind');
    halt (1);
  end;

  (* Включает прием соединений *)
  if not listen (sockfd, 5) then
  begin
    perror ('ошибка вызова listen');
    halt (1);
  end;

  while true do
  begin
    (* Принимает очередной запрос на соединение *)
    newsockfd := accept (sockfd, client, clientaddrlen);
    if newsockfd = -1 then
    begin
      perror ('Ошибка вызова accept');
      continue;
```

```
    end;
    (*
    Создает дочерний процесс для работы с соединением.
    Если это дочерний процесс,
      то в цикле принимает данные от клиента
      и посылает ему ответы.
    *)
  end;
end.
```

Обратите внимание на то, что сервер использует константу `INADDR_ANY`, соответствующую адресу локального компьютера.

Теперь имеется серверный процесс, способный переходить в режим приёма соединений и принимать запросы на установку соединений. Рассмотрим, как клиент должен обращаться к серверу.

10.5.4. Подключение клиента

Для выполнения запроса на подключение к серверному процессу клиент использует системный вызов `connect`.

Описание

```
uses sockets;

Function Connect(csockfd:Longint; Var address; add_len:Longint): Longint;
Function Connect(csockfd:longint; const address:string;
                 var SockIn,SockOut:text):Boolean;
Function Connect(csockfd:longint; const address:string;
                 var SockIn,SockOut:file):Boolean;
Function Connect(csockfd:longint; const address:TInetSockAddr;
                 var SockIn,SockOut:file):Boolean;
```

Первый параметр `csockfd` является дескриптором сокета клиента и не имеет отношения к дескриптору сокета на сервере. Параметр `address` указывает на структуру, содержащую адрес сервера, либо на адрес в формате строки. Параметр `add_len` определяет размер используемой структуры адреса.

Вторая, третья и четвертая формы вызова `connect` эквивалентны вызову первой с последующим использованием функции `Sock2Text`, преобразующей сокет `sockfd` в две файловые переменные типа `Text`, одна из которых отвечает за чтение из сокета (`SockIn`), а другая – за запись в сокет (`SockOut`).

Продолжая составление рассматриваемого примера, запишем следующий вариант текста программы клиента:

```
(* Клиентский процесс *)
uses sockets,stdio,linux;

const
  SIZE=sizeof(tinetsockaddr);
  server:tinetsockaddr=(family:AF_INET; port:7000);

var
  sockfd:longint;


begin
  (* Преобразовать и сохранить IP address сервера *)
  server.addr := inet_addr ('127.0.0.1');

  (* Создать сокет *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
```

```
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

  (* Соединяет сокет с сервером *)
  if not connect (sockfd, server, SIZE) then
  begin
    perror ('Ошибка вызова connect');
    halt (1);
  end;

  (* Обмен данными с сервером *)
end.
```

Адрес сервера преобразуется в нужный формат при помощи вызова `inet_addr`. Адреса известных компьютеров локальной сети обычно можно найти в файле `/etc/hosts`.

10.5.5. Пересылка данных

Теперь уже освоена процедура установления соединения между клиентом и сервером. Для сокетов типа `SOCK_STREAM` и клиент, и сервер получают дескрипторы файлов, которые могут использоваться для чтения или записи. В большинстве случаев для этого годятся обычные вызовы `fdread` и `fdwrite`. Если же необходимо задавать дополнительные параметры пересылки данных по сети, то можно использовать два новых системных вызова – `send` и `recv`. Эти вызовы имеют схожий интерфейс и ведут себя точно так же, как вызовы `fdread` и `fdwrite`, если их четвертый аргумент равен нулю.

Описание

```
uses sockets;

Function Recv(sockfd:Longint; Var buffer; length,Flags:Longint):Longint;
Function Send(sockfd:Longint; Var buffer; length,Flags:Longint):Longint;
```

Вызов `recv` имеет четыре параметра: дескриптор файла `filedes`, из которого читаются данные, буфер `buffer`, в который они помещаются, размер буфера `length` и поле флагов `flags`.

Параметр `flags` указывает дополнительные опции получения данных. Его возможные значения определяются комбинациями следующих констант:

`MSG_PEEK`	Процесс может просматривать данные, не «получая» их
`MSG_OOB`	Обычные данные пропускаются. Процесс принимает только срочные данные, например, сигнал прерывания
`MSG_WAITALL`	Возврат из вызова `recv` произойдет только после получения всех данных

При аргументе `flags` равном нулю вызов `send` работает точно так же, как и вызов `write`, пересылая массив данных буфера `buffer` в сокет `sockfd`. Параметр `length` задает размер массива данных. Аналогично вызову `recv` параметр `flags` определяет опции передачи данных. Его возможные значения определяются комбинациями следующих констант:

`MSG_OOB`	Передать *срочные* (out of band) данные
`MSG_DONTROUTE`	При передаче сообщения игнорируются условия маршрутизации протокола более низкого уровня. Обычно это означает, что сообщение посылается по прямому, а не по самому быстрому маршруту (самый быстрый маршрут не обязательно прямой и может зависеть от текущего распределения нагрузки сети)

Теперь с помощью этих вызовов можно реализовать обработку данных на серверной стороне:

```
(* Серверный процесс *)

var
  c:char;
  client:tinetsockaddr;
  clientaddrlen:longint;

begin
  (* Приведенная выше инициализация сокета *)
  .
  .
  .

  while true do
  begin
    (* Принимает запрос на установку соединения *)
    newsockfd := accept (sockfd, client, clientaddrlen);
    if newsockfd = -1 then
    begin
      perror ('Ошибка вызова accept');
      continue;
    end;

    (* Создает дочерний процесс для работы с соединением *)
    if fork = 0 then
    begin
      (* Принимает данные *)
      while recv (newsockfd, c, 1, 0) > 0 do
      begin
        (* Преобразовывает строчный символ в прописной *)
        c := upcase (c);
        (* Пересылает символ обратно *)
        send (newsockfd, c, 1, 0);
      end;
    end;
  end;
end.
```

Напомним, что использование вызова `fork` позволяет серверу обслуживать несколько клиентов. Цикл работы клиентского процесса может быть реализован так:

```
(* Клиентский процесс *)

var
  sockfd:longint;
  c,rc:char;

begin
  (* Приведенная выше инициализация сокета и запрос
   * на установку соединения *)

  (* Обмен данными с сервером *)
  rc := #$a;
  while true do
  begin
    if rc = #$a then
      writeln ('Введите строчный символ');
```

```
    c:=char(getchar);
    send (sockfd, c, 1, 0);
    recv (sockfd, rc, 1, 0);
    write (rc)
  end;
end.
```

10.5.6. Закрытие TCP-соединения

При работе с сокетами важно корректно реагировать на завершение работы абонентского процесса. Так как сокет является двусторонним механизмом связи, то нельзя предсказать заранее, когда произойдет разрыв соединения – во время чтения или записи. Поэтому нужно учитывать оба возможных варианта.

Если процесс пытается записать данные в оборванный сокет при помощи вызова `fdwrite` или `send`, то он получит сигнал `SIGPIPE`, который может быть перехвачен соответствующим обработчиком сигнала. При чтении обрыв диагностируется проще.

В случае разорванной связи вызов `fdread` или `recv` возвращает нулевое значение. Поэтому для вызовов `fdread` и `recv` необходимо всегда проверять возвращаемое значение, чтобы не зациклиться при приеме данных.

Закрываются сокеты так же, как и обычные дескрипторы файлового ввода/вывода, – при помощи системного вызова `fdclose`. Для сокета типа `SOCK_STREAM` ядро гарантирует, что все записанные в сокет данные будут переданы принимающему процессу. Это может вызвать блокирование операции закрытия сокета до тех пор, пока данные не будут доставлены. (Если сокет имеет тип `SOCK_DGRAM`, то сокет закрывается немедленно.)

Теперь можно привести полный текст примера клиента и сервера, добавив в серверный процесс обработку сигналов и вызов `fdclose` в обе программы. В данном случае эти меры могут показаться излишними, но в реальном клиент/серверном приложении обязательна надежная обработка всех исключительных ситуаций. Приведем окончательный текст программы сервера:

```
(* Серверный процесс *)
uses sockets,stdio,linux;

const
  SIZE=sizeof(tinetsockaddr);
  server:tinetsockaddr = (family:AF_INET; port:7000; addr:INADDR_ANY);

var
  newsockfd:longint;

procedure catcher (sig:integer);cdecl;
begin
  fdclose (newsockfd);
  halt (0);
end;

var
  sockfd:longint;
  c:char;
  act:sigactionrec;
  mask:sigset_t;
  client:tinetsockaddr;
  clientaddrlen:longint;
begin
  act.handler.sh := @catcher;
  sigfillset (@mask);
  act.sa_mask:=mask.__val[0];
```

```
  sigaction (SIGPIPE, @act, nil);

  (* Установить абонентскую точку сокета *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

  (* Связать адрес с абонентской точкой *)
  if not bind (sockfd, server, SIZE) then
  begin
    perror ('Ошибка вызова bind');
    halt (1);
  end;

  (* Включить прием соединений *)
  if not listen (sockfd, 5) then
  begin
    perror ('ошибка вызова listen');
    halt (1);
  end;

  while true do
  begin
    (* Прием запроса на соединение *)
    newsockfd := accept (sockfd, client, clientaddrlen);
    if newsockfd = -1 then
    begin
      perror ('Ошибка вызова accept');
      continue;
    end;

    (* Создать дочерний процесс для работы с соединением *)
    if fork = 0 then
    begin
      while recv (newsockfd, c, 1, 0) > 0 do
      begin
        c := upcase (c);
        send (newsockfd, c, 1, 0);
      end;

      (* После того, как клиент прекратит передачу данных,
       * сокет может быть закрыт и дочерний процесс
       * завершает работу *)
      fdclose (newsockfd);
      halt (0);
    end;

    (* В родительском процессе newsockfd не нужен *)
    fdclose (newsockfd);
  end;
end.
```

И клиента:

```
(* Клиентский процесс *)
uses sockets,stdio,linux;
```

```
const
  SIZE=sizeof(tinetsockaddr);
  server:tinetsockaddr=(family:AF_INET; port:7000);

var
  sockfd:longint;
  c,rc:char;

begin
  (* Преобразовать и сохранить IP address сервера *)
  server.addr := inet_addr ('127.0.0.1');

  (* Установить абонентскую точку сокета *)
  sockfd := socket (AF_INET, SOCK_STREAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

  (* Подключить сокет к адресу сервера *)
  if not connect (sockfd, server, SIZE) then
  begin
    perror ('Ошибка вызова connect');
    halt (1);
  end;

  (* Обмен данными с сервером *)
  rc := #$a;
  while true do
  begin
    if rc = #$a then
      writeln ('Введите строчный символ');
    c:=char(getchar);
    send (sockfd, c, 1, 0);
    if recv (sockfd, rc, 1, 0) > 0 then
      write (rc)
    else
    begin
      writeln ('Сервер не отвечает');
      fdclose (sockfd);
      halt (1);
    end;
  end;
end.
```

Упражнение 10.1. *Запустите приведенную программу сервера и несколько клиентских процессов. Что произойдет после того, как все клиентские процессы завершат работу?*

Упражнение 10.2. *Измените код программ так, чтобы после того, как все клиентские процессы завершат свою работу, сервер также завершал работу после заданного промежутка времени, если не поступят новые запросы на соединение.*

Упражнение 10.3. *Измените код программ так, чтобы два взаимодействующих процесса выполнялись на одном и том же компьютере. В этом случае сокет должен иметь коммуникационный домен* `AF_UNIX`.

10.6. Программирование в режиме пересылок UDP-дейтаграмм

Перепишем теперь пример, используя модель дейтаграмм. Основное отличие будет

заключаться в том, что дейтаграммы (UDP-пакеты), передаваемые между клиентом и сервером, могут достигать точки назначения в произвольном порядке. К тому же, как уже упоминалось, протокол UDP не гарантирует доставку пакетов. При работе с UDP-сокетами процесс клиента должен также сначала создать сокет и связать с ним свой локальный адрес при помощи вызова `bind`. После этого процесс сможет использовать этот сокет для посылки и приема UDP-пакетов. Чтобы послать сообщение, процесс должен знать адрес назначения, который может быть как конкретным адресом, так и шаблоном, называемым «широковещательным адресом» и обозначающим сразу несколько компьютеров.

10.6.1. Прием и передача UDP-сообщений

Для сокетов UDP есть два новых системных вызова – `sendto` и `recvfrom`. Параметр `sockfd` в обоих вызовах задает связанный с локальным адресом сокет, через который принимаются и передаются пакеты.

Описание

```
uses stdio;


function recvfrom(sockfd:longint; var message; length, flags:longint;
                  var send_addr:tsockaddr; var add_len:longint):longint;

function sendto(sockfd:longint; var message; length, flags:longint;
                var dest_addr:tsockaddr; dest_len:longint):longint;
```

Если параметр `send_addr` равен `nil`, то вызов `recvfrom` работает точно так же, как и вызов `recv`. Параметр `message` указывает на буфер, в который помещается принимаемая дейтаграмма, а параметр `length` задает число байтов, которые должны быть считаны в буфер. Параметр `flags` принимает те же самые значения, что и в вызове `recv`. Два последних параметра помогают установить двустороннюю связь с помощью UDP-сокета. В структуру `send_addr` будет помещена информация об адресе и порте, откуда пришел прочитанный пакет. Это позволяет принимающему процессу направить ответ пославшему пакет процессу. Последний параметр является указателем на целочисленную переменную типа `longint`, в которую помещается длина записанного в структуру `send_addr` адреса.

Вызов `sendto` противоположен вызову `recvfrom`. В этом вызове параметр `dest_addr` задает адрес узла сети и порт, куда должно быть передано сообщение, а параметр `dest_len` определяет длину адреса.

Адаптируем пример для модели дейтаграммных посылок.

```
(* Сервер *)
uses sockets,linux,stdio;

const
  SIZE=sizeof(tinetsockaddr);
  (* Локальный серверный порт *)
  server:tinetsockaddr = (family:AF_INET; port:7000; addr:INADDR_ANY);
  client_len:longint=SIZE;

var
  sockfd:longint;
  c:char;
  (* Структура, которая будет содержать адрес процесса 2 *)
  client:tinetsockaddr;

begin
  (* Установить абонентскую точку сокета *)
  sockfd := socket (AF_INET, SOCK_DGRAM, 0);
  if sockfd = -1 then
```

```
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;

  (* Связать локальный адрес с абонентской точкой *)
  if not bind (sockfd, server, SIZE) then
  begin
    perror ('Ошибка вызова bind');
    halt (1);
  end;

  (* Бесконечный цикл ожидания сообщений *)
  while true do
  begin
    (* Принимает сообщение и записывает адрес клиента *)
    if recvfrom (sockfd, c, 1, 0, tsockaddr(client), client_len) = -1 then
    begin
      perror ('Сервер: ошибка при приеме');
      continue;
    end;
    c := upcase (c);
    (* Посылает сообщение обратно *)
    if sendto (sockfd, c, 1, 0, tsockaddr(client), client_len) = -1 then
    begin
      perror ('Сервер: ошибка при передаче');
      continue;
    end;
  end;
end.
```

Новый текст клиента:

```
(* Клиентский процесс *)
uses sockets,stdio,linux;

const
  SIZE=sizeof(tinetsockaddr);

  (* Локальный порт на клиенте *)
  client:tinetsockaddr = (family:AF_INET; port:INADDR_ANY; addr:INADDR_ANY);

  (* Адрес удаленного сервера *)
  server:tinetsockaddr = (family:AF_INET; port:7000);

var
  sockfd:longint;
  c:char;
begin
  (* Преобразовать и записать IP адрес *)
  server.addr := inet_addr ('127.0.0.1');

  (* Установить абонентскую точку сокета *)
  sockfd := socket (AF_INET, SOCK_DGRAM, 0);
  if sockfd = -1 then
  begin
    perror ('Ошибка вызова socket');
    halt (1);
  end;
```

```
  (* Связать локальный адрес с абонентской точкой сокета. *)
  if not bind (sockfd, client, SIZE) then
  begin
    perror ('Ошибка вызова bind');
    halt (1);
  end;

  (* Считать символ с клавиатуры *)
  while fdread (0, c, 1) <> 0 do
  begin
    (* Передать символ серверу *)
    if sendto (sockfd, c, 1, 0, tsockaddr(server), SIZE) = -1 then
    begin
      perror ('Клиент: ошибка передачи');
      continue;
    end;

    (* Принять вернувшееся сообщение *)
    if recv (sockfd, c, 1, 0) = -1 then
    begin
      perror ('Клиент: ошибка приема');
      continue;
    end;

    fdwrite (1, c, 1);
  end;
end.
```

Упражнение 10.4. *Запустите сервер и несколько клиентов. Как сервер определяет, от какого клиента он принимает сообщение?*

10.7. Различия между двумя моделями

Обсудим различия между двумя реализациями рассматриваемого примера с точки зрения техники программирования.

В обеих моделях сервер должен создать сокет и связать свой локальный адрес с этим сокетом. В модели TCP-соединений серверу следует после этого включить прием соединений. В модели UDP-сокетов этот шаг не нужен, зато на клиента возлагается больше обязанностей.

С точки зрения клиента в модели TCP-соединений достаточно простого подключения к серверу. В модели UDP-сокетов клиент должен создать сокет и связать свой локальный адрес с этим сокетом.

И, наконец, для передачи данных обычно используются различные системные вызовы. Системные вызовы `sendto` и `recvfrom` могут использоваться в обеих моделях, но все же они обычно используются в UDP-модели, чтобы сервер мог получить информацию об отправителе и отправить обратно ответ.

Глава 11. Стандартная библиотека ввода/вывода

11.1. Введение

В последних главах книги рассмотрим некоторые из стандартных библиотек процедур системы UNIX (а также большинства сред поддержки языка C в других операционных системах).

Начнем с изучения очень важной стандартной библиотеки ввода/вывода, образующей основную часть стандартной библиотеки C, поставляемой со всеми системами UNIX. Интерфейс этой библиотекой составляет основную часть приведенного в приложении модуля `stdio`.

Читатели кратко ознакомились со стандартным вводом/выводом во второй главе и уже встречались с некоторыми из входящих в его состав процедур, например, процедурами `getchar` и `printf`. Основная цель стандартной библиотеки ввода/вывода состоит в предоставлении эффективных, развитых и переносимых средств доступа к файлам. Эффективность процедур, образующих библиотеку, достигается за счет обеспечения механизма автоматической буферизации, который невидим для пользователя и минимизирует число действительных обращений к файлам и число выполняемых низкоуровневых системных вызовов. Библиотека предлагает широкий выбор функций, таких как форматированный вывод и преобразование данных. Процедуры стандартного ввода/вывода являются переносимыми, так как они не привязаны к особым свойствам системы UNIX и на самом деле являются частью независимого от UNIX стандарта ANSI языка C. Любой полноценный компилятор языка C предоставляет доступ к стандартной библиотеке ввода/вывода независимо от используемой операционной системы. Компилятор Free Pascal позволяет нам использовать эту библиотеку, как и многие другие, простым экспортом её функций.

11.2. Структура TFILE

Процедуры *буферизованного ввода/вывода* идентифицируют открытые файлы (каналы, сокеты, устройства и другие объекты) при помощи указателя на структуру типа `FILE`. Процедуры этого семейства также называют процедурами *стандартного ввода/вывода*, так как они содержатся в стандартной библиотеке языка C. Указатель на объект `FILE` часто называется также *потоком ввода/вывода* и является аналогом файловых дескрипторов базового ввода/вывода.

Определение структуры `TFILE` находится в заголовочном файле `stdio`. Следует отметить, что программисту нет необходимости знать устройство структуры `TFILE`, более того, ее определение различно в разных системах.

Все данные, считываемые из файла или записываемые в файл, передаются через буфер структуры `TFILE`. Например, стандартная процедура вывода сначала будет лишь заполнять символ за символом буфер. Только после заполнения буфета очередной вызов библиотечной процедуры вывода автоматически запишет его содержимое в файл вызовом `fdwrite`. Эти действия невидимы для пользовательской программы. Размер буфера составляет `BUFSIZ` байтов. Постоянная `BUFSIZ` определена в файле `stdio` и, как уже описывалось во второй главе, обычно задает размер блоков на диске. Как правило, ее значение равно 512 или 1024 байта.

Аналогично процедура ввода извлекает данные из буфера, связанного со структурой `TFILE`. Как только буфер опустеет, для его заполнения автоматически считывается еще один фрагмент файла. Эти действия также не видимы для пользовательской программы.

Механизм буферизации стандартной библиотеки ввода/вывода гарантирует, что данные всегда считываются и записываются блоками стандартного размера. В результате число обращений к файлам и число внутренних системных вызовов поддерживаются на оптимальном уровне. Но поскольку вся буферизация скрыта внутри процедур ввода/вывода,

программист может их использовать для чтения или записи произвольных порций данных, даже по одному байту. Поэтому программа может быть составлена исходя из требований простоты и наглядности, а проблему общей эффективности ввода/вывода решат стандартные библиотеки. Далее увидим, что стандартные библиотечные процедуры также обеспечиваю простые в использовании средства форматирования. Поэтому для большинства приложений стандартный ввод/вывод является предпочтительным методом доступа к файлам.

11.3. Открытие и закрытие потоков: процедуры fopen и fclose

Описание

```
uses stdio;

function fopen(filename:pchar; _type:pchar):pfile;

function fclose(_stream:pfile):integer;
```

Библиотечные процедуры `fopen` и `fclose` являются эквивалентами вызовов `fdopen` и `fdclose`. Процедура `fopen` открывает файл, заданный параметром `filename`, и связывает с ним структуру `TFILE`. В случае успешного завершения процедура `fopen` возвращает указатель на структуру `TFILE`, идентифицирующую открытый файл, объект `PFILE` также часто называют открытым потоком ввода/вывода (эта структура `FILE` является элементом внутренней таблицы). Процедура `fclose` закрывает файл, заданный параметром `_stream`, и, если этот файл использовался для вывода, также сбрасывает на диск все данные из внутреннего буфера.

В случае неудачи процедура `fopen` возвращает нулевой указатель `nil`, определенный в файле `system`. В этом случае, так же как и для вызова `fdopen`, переменная `linuxerror` будет содержать код ошибки, указывающий на ее причину.

Второй параметр процедуры `fopen` указывает на строку, определяющую режим доступа. Она может принимать следующие основные значения:

- `r` Открыть файл `filename` только для чтения. (Если файл не существует, то вызов завершится неудачей и процедура `fopen` вернет нулевой указатель `nil`)
- `w` Создать файл `filename` и открыть его только для записи. (Если файл уже существует, то он будет усечен до нулевой длины)
- `a` Открыть файл `filename` только для записи. Все данные будут добавляться в конец файла. Если файл не существует, он создается

Файл может быть также открыт для обновления, то есть программа может выполнять чтение из файла и запись в него. Другими словами, программа может одновременно выполнять для файла и операции ввода, и операции вывода без необходимости открывать его заново. В то же время из-за механизма буферизации такой ввод/вывод будет более ограниченным, чем режим чтения/записи, поддерживаемый вызовами `fdread` и `fdwrite`. В частности, после вывода нельзя осуществить ввод без вызова одной из стандартных процедур ввода/вывода `fseek` или `rewind`. Эти процедуры изменяют положение внутреннего указателя чтения/ записи и обсуждаются ниже. Аналогично нельзя выполнить вывод после ввода без вызова процедур `fseek` или `rewind` или процедуры ввода, которая перемещает указатель в конец файла. Режим обновления обозначается символом `+` в конце аргумента, передаваемого процедуре `fopen`. Вышеприведенные режимы можно дополнить следующим образом:

- `r+` Открыть файл `filename` для чтения и записи. Если файл не существует, то вызов снова завершится неудачей
- `w+` Создать файл `filename` и открыть его для чтения и записи. (Если файл уже существует, то он будет усечен до нулевой длины)
- `a+` Открыть файл `filename` для чтения и записи. При записи данные будут добавляться в конец файла. Если файл не существует, то он

создается

В некоторых системах для доступа к двоичным, а не текстовым файлам, к строке также нужно добавлять символ `b`, например, `rb`.

Если файл создается при помощи процедуры `fopen`, для него обычно устанавливается код доступа `octal(0666)`. Это позволяет всем пользователям выполнять чтение из файла и запись в него. Эти права доступа по умолчанию могут быть изменены установкой ненулевого значения атрибута процесса `umask`. (Системный вызов `umask` был изучен в главе 3.)

Следующий пример программы показывает использование процедуры `fopen` и ее связь с процедурой `fclose`. При этом, если файл `indata` существует, то он открывается для чтения, а файл `outdata` создается (или усекается до нулевой длины, если он существует). Процедура `fatal` предназначена для вывода сообщения об ошибке, ее описание было представлено в предыдущих главах. Она просто передает свой аргумент процедуре `perror`, а затем вызывает `halt` для завершения работы программы.

```
uses stdio;

const
  inname:pchar = 'indata';
  outname:pchar = 'outdata';

function fatal(s:pchar):integer;
begin
  perror (s);
  halt (1);
end;

var
  inf,outf:pfile;
begin
  inf := fopen (inname, 'r');
  if inf = nil then
    fatal ('Невозможно открыть входной файл');
  outf := fopen (outname, 'w');
  if outf = nil then
    fatal ('Невозможно открыть выходной файл');
  (* Выполняются какие-либо действия ... *)
  fclose (inf);
  fclose (outf);
  halt (0);
end.
```

На самом деле, в данном случае оба вызова `fclose` не нужны. Дескрипторы, связанные с файлами `inf` и `outf`, будут автоматически закрыты при завершении работы процесса, и вызов `halt` автоматически сбросит данные из буфера указателя `outf` на диск, записав их в файл `outdata`.

С процедурой `fclose` тесно связана процедура `fflush`:

Описание

```
uses stdio;

function fflush(_stream:pfile):integer;
```

Выполнение этой процедуры приводит к сбросу на диск содержимого буфера вывода, связанного с потоком `_stream`. Другими словами, данные из буфера записываются в файл немедленно, независимо от того, заполнен буфер или нет. Это гарантирует, что содержимое файла на диске будет соответствовать тому, как он выглядит с точки зрения процесса. (Процесс считает, что данные записаны в файл с того момента, как они оказываются в

буфере, поскольку механизм буферизации прозрачен.) Любые данные из буфера ввода этим вызовом предусмотрительно отбрасываются.

Поток `_stream` остается открытым после завершения процедуры `fflush`. Как и процедура `fclose`, процедура `fflush` возвращает постоянную `EOF` в случае ошибки и нулевое значение – в случае успеха. (Значение постоянной `EOF` задано в файле `stdio` равным –1. Оно обозначает конец файла, но может также использоваться для обозначения ошибок.)

11.4. Посимвольный ввод/вывод: процедуры getc и putc

Описание

```
uses stdio;

function getc(inf:pfile):integer;

function putc(c:integer; outf:pfile):integer;
```

Наиболее простыми из процедур стандартной библиотеки ввода/вывода являются процедуры `getc` и `putc`. Процедура `getc` возвращает очередной символ из входного потока `inf`. Процедура `putc` помещает символ, обозначенный параметром `c`, в выходной поток `outf`.

В обеих процедурах символ `c` имеет тип `integer`, а не `char`, что позволяет процедурам использовать наборы 16-битовых «широких» символов. Это также позволяет процедуре `getc` возвращать значение –1, находящееся вне диапазона возможных значений типа `char`. Постоянная `EOF` используется процедурой `getc` для обозначения того, что либо достигнут конец файла, либо произошла ошибка. Процедура `putc` также может возвращать значение `EOF` в случае ошибки.

Следующий пример является новой версией процедуры `copyfile`, представленной в главе 2; в данном случае вместо использования вызовов `fdread` и `fdwrite` используются процедуры `getc` и `putc`:

```
uses stdio;

(* Скопировать файл f1 в файл f2
 * при помощи стандартных процедур ввода/вывода
 *)
function copyfile(const f1, f2:pchar):integer;
var
  inf, outf:pfile;
  c:longint;
begin
  inf := fopen (f1, 'r');
  if inf = nil then
  begin
    copyfile:=-1;
    exit;
  end;
  outf := fopen (f2, 'w');
  if outf = nil then
  begin
    fclose (inf);
    copyfile:=-2;
    exit;
  end;
  c := getc (inf);
  while c <> EOF do
  begin
    putc (c, outf);
    c := getc (inf);
```

```
  end;
  fclose (inf);
  fclose (outf);
  copyfile:=0;
end;
```

Копирование выполняет внутренний цикл `while`. Снова обратите внимание на то, что переменная `c` имеет тип `longint`, а не `char`.

Упражнение 11.1. *В упражнениях 2.4 и 2.5 мы описали программу* `count`*, которая выводит число символов, слов и строк во входном файле. (Напомним, что слово определялось, как любая последовательность алфавитно-цифровых символов или одиночный пробельный символ.) Перепишите программу* `count`*, используя процедуру* `getc`.

Упражнение 11.2. *Используя процедуру* `getc`*, напишите программу, выводящую статистику распределения символов в файле, то есть число раз, которое встречается в файле каждый символ. Один из способов сделать это состоит в использовании массива целых чисел типа* `long`*, который будет содержать счетчики числа символов, а затем рассматривать значение каждого символа в качестве индекса увеличиваемого счетчика массива. Программа также должна рисовать простую гистограмму полученного распределения при помощи процедур* `printf` *и* `putc`.

11.5. Возврат символов в поток: процедура ungetc

Описание

```
uses stdio;

function ungetc(c:integer; _stream:pfile):integer;
```

Процедура `ungetc` возвращает символ `c` в поток `_stream`. Это всего лишь логическая операция. Входной файл не будет при этом изменяться. В случае успешного завершения процедуры `ungetc` символ `c` будет следующим символом, который будет считан процедурой `getc`. Гарантируется возврат только одного символа. В случае неудачной попытки вернуть символ `c` процедура `ungetc` возвращает значение `EOF`. Попытка вернуть сам символ `EOF` должна всегда завершаться неудачей. Но это обычно не представляет проблемы, так как все последующие вызовы процедуры `getc` после достижения конца файла приведут к возврату символа `EOF`.

Обычно процедура `ungetc` используется для восстановления исходного состояния входного потока после чтения лишнего символа для проверки условия. Следующая процедура `getword` применяет это простой подход для ввода строки, которая содержит либо непрерывную последовательность алфавитно-цифровых символов, либо одиночный нетекстовый символ. Конец файла кодируется возвращенным значением `nil`. Процедура `getword` принимает в качестве аргумента указатель на структуру `TFILE`. Она использует для проверки два макроса, определенные в файле `stdio`. Первый из них, `isspace`, определяет, является ли символ пробельным символом, таким как символ пробела, табуляции или перевода строки. Второй, `isalnum`, проверяет, является ли символ алфавитно-цифровым, то есть цифрой или буквой.

```
(* В этом файле определены isspace и isalnum *)
uses stdio;

const
  MAXTOK=256;

var
  inbuf:array [0..MAXTOK] of char;


function getword (inf:pfile):pchar;
```

```
var
  c,count:integer;
begin
  count:=0;
  (* Удалить пробельные символы *)
  repeat
    c := getc (inf);
  until not isspace (c);
  if c = EOF then
  begin
    getword:=nil;
    exit;
  end;
  if not isalnum (c) then    (* символ не является алфавитно-цифровым *)
  begin
    inbuf[count] := char(c);
    inc(count);
  end
  else
  begin
    (* Сборка "слова" *)
    repeat
      if count < MAXTOK then
      begin
        inbuf[count] := char(c);
        inc(count);
      end;
      c := getc (inf);
    until not isalnum (c);
    ungetc (c, inf);          (* вернуть символ *)
  end;
  inbuf[count] := #0;
  (* нулевой символ в конце строки *)
  getword:=inbuf;
end;


var
  word:pchar;
begin
  while true do
  begin
    word := getword (stdin);
    if word <> nil then
      puts (word)
    else
      break;
  end;
end.
```

Если подать на вход программы следующий ввод

```
Это данные
  на входе
 программы!!!
```

то процедура `getword` вернет следующую последовательность строк:

```
 Это
 данные
 на
 входе
```

программы
!
!
!

Упражнение 11.3. *Измените процедуру* `getword` *так, чтобы она распознавала также числа, которые могут начинаться со знака минус или плюс и могут содержать десятичную точку.*

11.6. Стандартный ввод, стандартный вывод и стандартный вывод диагностики

Стандартная библиотека ввода/вывода обеспечивает две структуры `TFILE`, связанные со стандартным вводом и стандартным выводом, и переменная типа `TEXT`, связанная со стандартным выводом диагностики. (Еще раз напомним, что не следует путать эти потоки с одноименными дескрипторами ввода/вывода 0, 1 и 2.) Эти стандартные структуры не требуют открытия и задаются предопределенными указателями:

`stdin`	Соответствует стандартному вводу
`stdout`	Соответствует стандартному выводу
`stderr`	Соответствует стандартному выводу диагностики

Следующий вызов получает очередной символ из структуры `stdin`, которая так же, как и дескриптор файла со значением 0, по умолчанию соответствует клавиатуре:

```
inchar := getc (stdin);
```

Так как ввод и вывод через потоки `stdin` и `stdout` используются очень часто, для удобства определены еще две процедуры – `getchar` и `putchar`. Процедура `getchar` возвращает очередной символ из `stdin`, а процедура `putchar` выводит символ в `stdout`. Они аналогичны процедурам `getc` и `putc`, но не имеют аргументов.

Следующая программа `io2` использует процедуры `getchar` и `putchar` для копирования стандартного ввода в стандартный вывод:

```
(* Программа io2 - копирует stdin в stdout *)
uses stdio;

var
  c:integer;
begin
  c := getchar;
  while c <> EOF do
  begin
    putchar (c);
    c := getchar;
  end;
end.
```

Программа `io2` ведет себя почти аналогично приведенному ранее примеру – программе `io` из главы 2.

Так же, как `getc` и `putc`, `getchar` и `putchar` могут быть макросами. Фактически `getchar` часто просто определяется как `getc(stdin)`, а `putchar` – как `putc(stdout)`.

`stderr` обычно предназначена для вывода сообщений об ошибках, поэтому вывод в `stderr` обычно не буферизуется. Другими словами, символ, который посылается в `stderr`, будет немедленно записан в файл или устройство, соединенное со стандартным выводом диагностики. При включении отладочной печати в код для тестирования рекомендуется выполнять вывод в `stderr`. Вывод в `stdout` буферизуется и может появиться через несколько шагов после того, как он в действительности произойдет. (Вместо этого можно использовать процедуру `fflush(stdout)` после каждого вывода для записи всех сообщений

из буфера stdout.)[1]

Упражнение 11.4. *При помощи стандартной команды* time *сравните производительность программы* io2 *и программы* io*, разработанной в главе 2. Измените исходную версию программы* io *так, чтобы она использовала вызовы* fdread *и* fdwrite *для посимвольного ввода и вывода. Снова сравните производительность полученной программы и программы* io2.

Упражнение 11.5. *Перепишите программу* io2 *так, чтобы она более соответствовала команде* cat. *В частности, сделайте так, чтобы она выводила на экран содержимое файлов, заданных в качестве аргументов командной строки. При отсутствии аргументов она должна принимать ввод из* stdin.

11.7. Стандартные процедуры опроса состояния

Для опроса состояния структуры TFILE существует ряд простых процедур. Они, например, позволяют программе определять, вернула ли процедура ввода, такая как getc, символ EOF из-за того, что достигнут конец файла или в результате возникновения ошибки. Эти процедуры описаны ниже:

Описание

```
uses stdio;

function ferror(_stream:pfile):integer;

function feof(_stream:pfile):integer;

procedure clearerr(_stream:pfile);

function fileno(_stream:pfile):longint;
```

Функция ferror является предикатом, который возвращает ненулевое значение, если в потоке _stream возникла ошибка во время последнего запроса на ввод или вывод. Ошибка может возникать в результате вызова примитивов доступа к файлам (fdread, fdwrite и др.) внутри процедуры стандартного ввода/вывода. Если же функция ferror возвращает нулевое значение, значит, ошибок не было. Функция ferror может использоваться следующим образом:

```
if ferror(_stream) <> 0 then
begin
  (* Обработка ошибок *)
end
else
begin
  (* Ошибок нет *)
end;
```

Функция feof является предикатом, возвращающим ненулевое значение, если для потока _stream достигнут конец файла. Возврат нулевого значения просто означает, что этого еще не произошло.

Функция clearerr используется для сброса индикаторов ошибки и флага достижения конца файла для потока _stream. При этом гарантируется, что последующие вызовы функций ferror и feof для этого файла вернут нулевое значение, если за это время не произошло что-нибудь еще. Очевидно, что функция clearerr бывает необходима редко.

Функция fileno является вспомогательной и не связана с обработкой ошибок. Она возвращает целочисленный дескриптор файла, содержащийся в структуре TFILE, на которую

[1] Потоки stdin, stdout и stderr не следует закрывать: это может привести к аварийному завершению процесса, так как соответствующие структуры TFILE часто размещены в статической, а не динамической памяти.

указывает параметр _stream. Это может быть полезно, если нужно передать какой-либо процедуре дескриптор файла, а не идентификатора потока TFILE. Однако не следует использовать процедуру fileno для смешивания вызовов буферизованного и небуферизованного ввода/вывода. Это почти неизбежно приведет к хаосу.

Следующий пример – процедура egetc использует функцию ferror, чтобы отличить ошибку от достижения конца файла при возврате процедурой getc значения EOF.

```
(* Процедура egetc - getc с проверкой ошибок *)
uses stdio;

function egetc (stream:pfile):longint;
var
  c:longint;
begin
  c := getc (stream);
  if c = EOF then
  begin
    if ferror (stream) <> 0 then
    begin
      writeln (stderr, 'Фатальная ошибка: ошибка ввода');
      halt (1);
    end
    else
      writeln (stderr, 'Предупреждение: EOF');
  end;
  egetc:=c;
end;
```

11.8. Построчный ввод и вывод

Существует также набор простых процедур для ввода и вывода строк (под которыми понимается последовательность символов, завершаемая символом перевода строки). Эти процедуры удобно использовать в интерактивных программах, которые выполняют чтение с клавиатуры и вывод на экран терминала. Основные процедуры для ввода строк называются gets и fgets.

Описание

```
uses stdio;

function gets(buf:pchar):pchar;

function fgets(buf:pchar; nsize:integer; inf:pfile):pchar;
```

Процедура gets считывает последовательность символов из потока стандартного ввода (stdin), помещая все символы в буфер, на который указывает аргумент buf. Символы считываются до тех пор, пока не встретится символ перевода строки или конца файла. Символ перевода строки newline отбрасывается, и вместо него в буфер buf помещается нулевой символ, образуя завершенную строку. В случае возникновения ошибки или при достижении конца файла возвращается значение nil.

Процедура fgets является обобщенной версией процедуры gets. Она считывает символы из потока inf в буфер buf до тех пор, пока не будет считано nsize-1 символов или не встретится раньше символ перевода строки newline или не будет достигнут конец файла. В процедуре fgets символы перевода строки newline не отбрасываются, а помещаются в конец буфера (это позволяет вызывающей функции определить, в результате чего произошел возврат из процедуры fgets). Как и процедура gets, процедура fgets возвращает указатель на буфер buf в случае успеха и nil – в противном случае.

Процедура gets является довольно примитивной. Так как она не знает размер передаваемого буфера, то слишком длинная строка может привести к возникновению

внутренней ошибки в процедуре. Чтобы избежать этого, можно использовать процедуру `fgets` (для стандартного ввода `stdin`).

Следующая процедура `yesno` использует процедуру `fgets` для получения положительного или отрицательного ответа от пользователя; она также вызывает макрос `isspace` для пропуска пробельных символов в строке ответа:

```
(* Процедура yesno - получить ответ от пользователя *)
uses stdio;

const
  YES=1;
  NO=0;
  ANSWSZ=80;
  pdefault:pchar = 'Наберите "y" (YES), или "n" (NO)';
  error:pchar = 'Неопределенный ответ';


function yesno (prompt:pchar):integer;
var
  buf:array [0..ANSWSZ-1] of char;
  p_use, p:pchar;
begin
  (* Вывести приглашение, если он не равно nil.
   * Иначе использовать приглашение по умолчанию
   * pdefault *)
  if prompt <> nil then
    p_use := prompt
  else
    p_use := pdefault;
  (* Бесконечный цикл до получения правильного ответа. *)
  while true do
  begin
    (* Вывести приглашение *)
    printf ('%s > ', [p_use]);
    if fgets (buf, ANSWSZ, stdin) = nil then
    begin
      yesno:=EOF;
      exit;
    end;
    (* Удалить пробельные символы *)
    p := buf;
    while isspace (byte(p^)) do
      inc(p);
    case p^ of
      'Y','y':
      begin
        yesno:=YES;
        exit;
      end;
      'N','n':
      begin
        yesno:=NO;
        exit;
      end;
      else
        printf (#$a'%s'#$a, [error]);
    end;
  end;
```

```
end;


var
  ans:integer;
begin
  ans := yesno (nil);
  printf ('Получен ответ: ',[]);
  if ans = YES then
    printf ('Да'#$a,[])
  else
    printf ('Нет'#$a,[]);
end.
```

В этом примере предполагается, что `stdin` связан с терминалом. Как можно сделать эту процедуру более безопасной?

Обратными процедурами для `gets` и `fgets` будут соответственно процедуры `puts` и `fputs`.

Описание

```
uses stdio;

function puts(str:pchar):integer;

function fputs(str:pchar; outf:pfile):integer;
```

Процедура `puts` записывает все символы (кроме завершающего нулевого символа) из строки `str` на стандартный вывод (`stdout`). Процедура `fputs` записывает строку `str` в поток `outf`. Для обеспечения совместимости со старыми версиями системы процедура `puts` добавляет в конце символ перевода строки, процедура же `fputs` не делает этого. Обе функции возвращают в случае ошибки значение `EOF`.

Следующий вызов процедуры `puts` приводит к выводу сообщения `Hello, world` на стандартный вывод, при этом автоматически добавляется символ перевода строки `newline`:

```
puts('Hello, world');
```

11.9. Ввод и вывод бинарных данных: процедуры fread и fwrite

Описание

```
uses stdio;

function fread(buffer:pointer; size, nitems:longint; inf:pfile):longint;

function fwrite(buffer:pointer; size,nitems:longint; outf:pfile):longint;
```

Эти две полезные процедуры обеспечивают ввод и вывод произвольных нетекстовых данных. Процедура `fread` считывает `nitems` объектов данных из входного файла, соответствующего потоку `inf`. Считанные байты будут помещены в массив `buffer`. Каждый считанный объект представляется последовательностью байтов длины `size`. Возвращаемое значение дает число успешно считанных объектов.

Процедура `fwrite` является точной противоположностью процедуры `fread`. Она записывает данные из массива `buffer` в поток `outf`. Массив `buffer` содержит `nitems` объектов, размер которых равен `size`. Возвращаемое процедурой значение дает число успешно записанных объектов.

Эти процедуры обычно используются для чтения и записи содержимого произвольных структур данных языка Паскаль. При этом параметр `size` часто содержит конструкцию `sizeof`, которая возвращает размер структуры в байтах.

Следующий пример показывает, как все это работает. В нем используется шаблон структуры `dict_elem`. Экземпляр этой структуры может представлять собой часть записи

простой базы данных. Используя терминологию баз данных, структура dict_elem представляет собой запись, или атрибут, базы данных. Мы поместили определение структуры dict_elem в заголовочный файл dict.inc, который выглядит следующим образом:

```
(* dict.inc - заголовочный файл для writedict и readdict *)
uses stdio;

 (* Структура dict_elem элемент данных *)
 (* (соответствует полю базы данных) *)
type dict_elem=record
  d_name:array [0..14] of char;  (* имя элемента словаря *)
  d_start:integer;               (* начальное положение записи *)
  d_length:integer;              (* длина поля *)
  d_type:integer;                (* обозначает тип данных *)
end;
pdict_elem=^dict_elem;

const
  ERROR=-1;
  SUCCESS=0;
```

Не вдаваясь в смысл элементов структуры, введем две процедуры writedict и readdict, которые соответственно выполняют запись и чтение массива структур dict_elem. Файлы, создаваемые при помощи этих двух процедур, можно рассматривать как простые словари данных для записей в базе данных.

Процедура writedict имеет два параметра, имя входного файла и адрес массива структур dict_elem. Предполагается, что этот список заканчивается первой структурой массива, в которой элемент d_length равен нулю.

```
{$i dict.inc}


function writedict (const dictname:pchar; elist:pdict_elem):integer;
var
  j:integer;
  outf:pfile;
begin
  (* Открыть входной файл *)
  outf := fopen (dictname, 'w');
  if outf = nil then
  begin
    writedict:=ERROR;
    exit;
  end;
  (* Вычислить размер массива *)
  j:=0;
  while elist[j].d_length <> 0 do
    inc(j);
  (* Записать список структур dict_elem *)
  if fwrite (elist, sizeof (dict_elem), j, outf) < j then
  begin
    fclose (outf);
    writedict:=ERROR;
    exit;
  end;
  fclose (outf);
  writedict:=SUCCESS;
end;
```

Обратите внимание на использование `sizeof(dict_elem)` для сообщения процедуре `fwrite` размера структуры `dict_elem` в байтах.

Процедура `readdict` использует процедуру `fread` для считывания списка структур из файла. Она имеет три параметра: указатель на имя файла словаря `indictname`, указатель `inlist` на массив структур `dict_elem`, в который будет загружен список структур из файла, и размер массива `maxlength`.

```
function readdict (const indictname:pchar;inlist:pdict_elem;
                   maxlength:integer):pdict_elem;
var
  i:integer;
  inf:pfile;
begin
  (* Открыть входной файл *)
  inf := fopen (indictname, 'r');
  if inf = nil then
  begin
    readdict:=nil;
    exit;
  end;
  (* Считать структуры dict_elem из файла *)
  for i:=0 to maxlength - 1 do
    if fread (@inlist[i], sizeof (dict_elem), 1, inf) < 1 then
      break;
  fclose (inf);
  (* Обозначить конец списка *)
  inlist[i].d_length := 0;
  (* Вернуть начало списка *)
  readdict:=inlist;
end;


const
  delem1:array [0..1] of dict_elem=(
    (d_name:('d','n','a','m','e', #0,#0,#0,#0,#0,#0,#0,#0,#0,#0);
     d_start:2; d_length:15; d_type:3),
    (d_name:(#0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0)
     d_start:0; d_length:0; d_type:0)
  );
var
  delem2:array [0..1] of dict_elem;
begin
  printf ('delem1: d_name=%s, d_start=%d, d_length=%d, d_type=%d'#$a,
         [pchar(delem1[0].d_name), delem1[0].d_start, delem1[0].d_length,
         delem1[0].d_type]);
  writedict ('dictionary', @delem1[0]);
  if readdict ('dictionary', @delem2[0], 2)<>nil then
    printf ('delem2: d_name=%s, d_start=%d, d_length=%d, d_type=%d'#$a,
         [pchar(delem2[0].d_name), delem2[0].d_start, delem2[0].d_length,
         delem2[0].d_type]);
end.
```

И снова обратите внимание на приведение типа и использование конструкции `sizeof`.

Необходимо сделать важную оговорку. Бинарные данные, записываемые в файл при помощи процедуры `fwrite`, отражают внутреннее представление данных в системной памяти. Так как это представление зависит от архитектуры компьютера и различается порядком байтов в слове и выравниванием слов, то данные, записанные на одном компьютере, могут не читаться на другом, если не предпринять специальные усилия для

того, чтобы они были записаны в машинно-независимом формате. По тем же причинам почти всегда бессмысленно выводить значения адресов и указателей.

И последний момент: можно было бы получить практически тот же результат, напрямую используя вызовы `fdread` или `fdwrite`, например:

```
fdwrite(fd, ptr, sizeof(dict_elem));
```

Основное преимущество версии, основанной на стандартной библиотеке ввода/вывода, снова заключается в ее лучшей эффективности. Данные при этом будут читаться и записываться большими блоками, независимо от размера структуры `dict_elem`.

Упражнение 11.6. *Представленные версии процедур* `writedict` *и* `readdict` *работают с файлами словаря, которые могут содержать только один тип записей. Измените их так, чтобы в одном файле можно было хранить информацию о нескольких типах записей. Другими словами, нужно, чтобы файл словаря мог содержать несколько независимых именованных списков структур* `dict_elem`. *(Совет: включите в начало файла «заголовок»; содержащий информацию о числе записей и типе полей.)*

11.10. Произвольный доступ к файлу: процедуры fseek, rewind и ftell

Стандартная библиотека ввода/вывода содержит процедуры для позиционирования `fseek`, `rewind` и `ftell`, которые позволяют программисту перемещать указатель файла, а также опрашивать его текущее положение. Они могут использоваться только для потоков, допускающих произвольный доступ (в число которых, например, не входят терминалы).

Описание

```
use stdio;

function fseek(_stream:pfile; offset:longint; direction:integer):longint;

procedure rewind(_stream:pfile);

function ftell(_stream:pfile):longint;
```

Процедура `fseek` аналогична низкоуровневой функции `lseek`, она устанавливает указатель файла, связанный с потоком `_stream`, изменяя позицию следующей операции ввода или вывода. Параметр `direction` определяет начальную точку, от которой отсчитывается новое положение указателя. Если значение этого параметра равно `SEEK_SET` (обычно 0), то отсчет идет от начала файла; если оно равно `SEEK_CUR` (обычно 1), то отсчет идет от текущего положения; для значения `SEEK_END` (обычно 2) отсчет ведется от конца файла.

Процедура `rewind(stream)` равносильна оператору:

```
fseek(stream, 0, SEEK_SET);
```

Другими словами, она устанавливает указатель чтения/записи на начало файла.

Процедура `ftell` сообщает текущее положение указателя в файле – число байтов от начала файла (началу файла соответствует нулевая позиция).

11.11. Форматированный вывод: семейство процедур printf

Описание

```
uses stdio;

function printf(fmt:pchar; args:array of const):integer;

function fprintf(outf:pfile; fmt:pchar; args:array of const):integer;

function sprintf(str:pchar; fmt:pchar; args:array of const):integer;
```

Каждая из этих процедур получает строку формата вывода `fmt` и переменное число аргументов произвольного типа (обозначенных как массив констант `args`), используемых

для формирования выходной строки вывода. В выходную строку выводится информация из параметров `args` согласно формату; заданному аргументом `fmt`. В случае процедуры `printf` эта строка затем копируется в `stdout`. Процедура `fprintf` направляет выходную строку в файл `outf`. Процедура `sprintf` вывода не производит, а копирует строку в символьный массив, заданный указателем `str`. Процедура `sprintf` также автоматически добавляет в конец строки нулевой символ.

Строка формата `fmt` похожа на строки, задающие формат вывода языка Fortran. Она состоит из обычных символов, которые копируются без изменений, и набора *спецификаций формата* (conversion specifications). Это подстроки, которые начинаются с символа % (если нужно напечатать сам символ процента, то нужно записать два таких символа: %%).

Для каждого из аргументов `args` должна быть задана своя спецификация формата, которая указывает тип соответствующего аргумента и способ его преобразования в выходную последовательность символов ASCII.

Прежде чем обсудить общую форму этих спецификаций, рассмотрим пример, демонстрирующий использование формата процедуры `printf` в двух простых случаях. В первом из них нет других аргументов, кроме строки `fmt`. Во втором есть один параметр форматирования: целочисленная переменная `iarg`.

```
var
  iarg:integer=34;
.
.
.
printf('Hello, world!'#$a, []);
printf('Значение переменной iarg равно %d'#$a, [iarg]);
```

Так как в первом вызове нет аргументов, которые нужно было бы преобразовать, то в строке формата не заданы спецификации формата, а массив констант пуст. Этот оператор просто приводит к выводу сообщения

```
Hello, world!
```

на стандартный вывод, за которым следует символ перевода строки (символ `#$a` в строке интерпретируется в языке Паскаль как символ перевода строки). Второй оператор `printf` содержит еще один аргумент `iarg` и поэтому в строке формата есть спецификация `%d`. Это сообщает процедуре `printf`, что дополнительный аргумент является целым числом, которое должно быть выведено в десятичной форме (поэтому используется символ `d`). Вывод этого оператора будет выглядеть так:

```
Значение переменной iarg равно 34
```

Приведем возможные типы спецификаций (кодов) формата:

Целочисленные форматы

`%d` Как уже было видно из примеров, это общеупотребительный код формата для значений типа `integer`. Если значение является отрицательным, то будет автоматически добавлен знак минуса

`%u` Аргумент имеет тип `word` и будет выводиться в десятичной форме

`%o` Аргумент имеет тип `word` и будет выводиться как восьмеричное число без знака

`%x` Аргумент имеет тип `word` и будет выводиться как шестнадцатеричное число без знака. В качестве дополнительных шестнадцатеричных цифр будут использоваться символы `a`, `b`, `c`, `d`, `e` и `f`. Если задан код `%X`, то будут использоваться символы `A`, `B`, `C`, `D`, `E` и `F`

`%ld` Аргумент имеет тип `longint` со знаком и будет выводиться в десятичной форме. Можно также использовать спецификации `%lo`, `%lu`, `%lx`, `%lX`

Форматы вещественных чисел

`%f` Аргумент имеет тип `single` или `double` и будет выводиться в стандартной десятичной форме

`%e` Аргумент имеет тип `single` или `double` и будет выводиться в экспоненциальной форме, принятой в научных приложениях. Для обозначения экспоненты будет использоваться символ `e`. Если задана спецификация `%E`, то будет использоваться символ `E`

`%g` Это объединение спецификаций `%e` и `%f`. Аргумент имеет тип `single` или `double`. В зависимости от величины числа, оно будет выводиться либо в обычном формате, либо в формате экспоненциальной записи (как для спецификации `%e`). Если задана спецификация `%G`, то экспонента будет обозначаться, как при задании спецификации `%E`

Форматирование строк и символов

`%c` Аргумент имеет тип `char` и будет выводиться без изменений, даже если он является «непечатаемым» символом. Численное значение символа можно вывести, используя код формата для целых чисел. Это может понадобится при невозможности отображения символа на терминале

`%s` Соответствующий аргумент считается строкой (то есть указателем на массив символов). Содержимое строки передается дословно в выходной поток. Строка должна заканчиваться нулевым символом

Следующий пример, процедура `warnuser`, демонстрирует использование кодов `%c` и `%s`. Она использует процедуру `fprintf` для вывода предупреждения на стандартный вывод – поток `stdout`. Если `stdout` соответствует терминалу, то процедура также пытается подать три звуковых сигнала, послав символ **Ctrl+G** (символ ASCII **BEL**, который имеет шестнадцатеричное значение `$7`). Эта процедура использует функцию `isatty`, определяющую, соответствует ли дескриптор файла терминалу, и процедуру `fileno`, возвращающую дескриптор файла, связанный с потоком. Функция `isatty` является стандартной функцией UNIX, представленной в главе 9, а процедура `fileno` является частью стандартной библиотеки ввода/вывода и описана в разделе 11.7.

```
(* Процедура warnuser — вывод сообщения и звукового сигнала *)

uses stdio, linux;

(* Этот код на большинстве терминалов вызывает *)
(* подачу звукового сигнала *)
const
  bel:char=$7;

procedure warnuser (const str:pchar);
begin
  (* Это терминал?? *)
  if isatty(fileno(stderr)) then
    fprintf(stdout, '%c%c%c', [bel, bel, bel]);
  fprintf(stdout, 'Предупреждение: %s'#$a, [string]);
end;
```

Задание ширины поля и точности

Спецификации формата могут также включать информацию о минимальной *ширине* (width) поля, в котором выводится аргумент, и *точности* (precision). В случае целочисленного аргумента под точностью понимается максимальное число выводимых цифр. Если аргумент имеет тип `single` или `double`, то точность задает число цифр после десятичной точки. Для строчного аргумента этот параметр определяет число символов, которые будут взяты из строки.

Значения ширины поля и точности находятся в спецификации формата сразу же после знака процента и разделены точкой, например, спецификация

```
%10.5d
```

означает: вывести соответствующий аргумент типа `integer` в поле шириной 10 символов;

если аргумент имеет меньше пяти цифр, то дополнить его спереди нулями до пяти знаков. Спецификация

```
%.5f
```

означает: вывести соответствующий аргумент типа `single` или `double` с точностью до пяти десятичных знаков после запятой. Этот пример также показывает, что можно опускать параметр ширины поля. Аналогично можно задавать только ширину поля, поэтому спецификация

```
%10s
```

показывает: вывести соответствующую строку в поле длиной не менее 10 символов.

Во всех приведенных примерах вывод будет выравниваться по правой границе заданного поля. Для выравнивания по левой границе нужно задать знак минус сразу же за символом процента. Например, спецификация

```
%-30s
```

означает, что соответствующий строчный аргумент будет выведен в левой части поля шириной не менее 30 символов.

Может случиться, что ширина спецификации формата не может быть вычислена до запуска программы. В этом случаем можно заменить спецификацию ширины поля символом (`*`). Тогда процедура `printf` будет ожидать для этого кода дополнительный аргумент типа `integer`, определяющий ширину поля. Поэтому выполнение оператора

```
var
  width, iarg:integer;
.
.
.
printf('%*d', [width, iarg]);
```

приведет к тому, что целочисленная переменная `iarg` будет выведена в поле шириной `width`.

Комплексный пример

Число возможных комбинаций различных форматов огромно, поэтому для экономии места в одну программу были включены сразу несколько примеров. Функция `arctan` является стандартной функцией арктангенса из математической библиотеки `math`.

```
(* Программа cram - демонстрация процедуры printf *)
uses stdio,math;

const
  weekday:pchar = 'Воскресенье';
  month:pchar = 'Сентября';
  str:pchar = 'Hello, world';
  i:longint = 11058;
  day:longint = 15;
  hour:longint = 16;
  minute:longint = 25;
begin
  (* Вывести дату *)
  printf ('Дата %s, %d %s, %d:%.2d'#$a,
          [weekday, day, month, hour, minute]);
  (* Перевод строки *)
  putchar ($a);
  (* Демонстрация различных комбинаций ширины поля и точности *)
  printf ('>>%s<<'#$a, [str]);
  printf ('>>%30s<<'#$a, [str]);
  printf ('>>%-30s<<'#$a, [str]);
  printf ('>>%30.5s<<'#$a, [str]);
  printf ('>>%-30.5s<<'#$a, [str]);
  putchar ($a);
  (* Вывести число i в разных форматах *)
```

```
  printf ('%d, %u, %o, %x, %X'#$a, [i, i, i, i, i]);
  (* Вывести число пи с точностью 5 знаков после запятой *)
  printf ('пи равно %.5f '#$a, [4 * arctan (1.0)]);
end.
```

Программа генерирует следующий вывод:

```
Дата Воскресенье, 15 Сентября, 16:25
>>Hello, world<<
>>  Hello, world<<
>>Hello, world  <<
>>              Hello<<
>>He11o         <<

11058, 11058, 25462, 2b32, 2B32
пи равно 3.14159
```

Специальные символы

Спецификации формата вывода могут быть еще более сложными и содержать дополнительные символы, одним из которых является знак `#`. Он должен задаваться сразу же за значением ширины поля. Для спецификаций формата беззнаковых целых чисел, включающих коды формата `o`, `x` и `X`, это приводит к выводу префикса `0`, `0x` и `0X` соответственно. Поэтому такой фрагмент программы

```
var
  arg:integer = $FF;
printf('В восьмеричной форме, %#o'#$a, [arg]);
```

приведет к выводу строки:

```
В восьмеричной форме, 0377
```

Для вывода вещественных чисел задание знака `#` приведет к выводу десятичной точки, даже если задано нулевое число знаков после запятой.

В спецификации может также содержаться знак плюса (`+`) для принудительного вывода символа `+` даже для положительных чисел. (Этот символ имеет смысл только для вывода целых чисел со знаком или вещественных чисел.) Знак `+` располагается в спецификации на особом месте, следуя сразу же после знака минус, который обозначает выравнивание влево, или после знака процента, если знак минуса отсутствует. Следующие строки кода

```
var
  farg:single=57.88;
printf('Значение farg равно <%-+10.2f>'#$a, [farg]);
```

приведут к выводу:

```
Значение farg равно <+57.88>
```

Обратите внимание на комбинацию символов минус и плюс. Можно также заменить символ `+` пробелом. В этом случае процедура `printf` выведет на месте знака плюс пробел. Это позволяет правильно выравнивать таблицы, содержащие положительные и отрицательные числа.

Процедура sprintf

Прежде всего нужно отметить еще один момент, касающийся процедуры `sprintf`. Дело в том, что не следует думать о процедуре `sprintf` как о процедуре вывода. На самом деле она представляет собой наиболее гибкую из библиотечных процедур, работающих со строками и преобразующих форматы данных. Следующий текст показывает использование этой функции:

```
(* Процедура genkey - генерация ключа базы данных *)
uses stdio;

(* Длина ключа всегда будет равна 20 символам *)

function genkey (buf:pchar; const suppcode:pchar; orderno:longint):pchar;
```

```
begin
  (* Проверка размера ключа *)
  if strlen (suppcode) <> 10 then
  begin
    genkey:=nil;
    exit;
  end;

  sprintf (buf, '%s_%.9d', [suppcode, orderno]);
  genkey:=buf;
end;

var
  buf:array [0..99] of char;
begin
  if genkey (buf, 'supplement', 12)<>nil then
    printf ('Key: %s'#$a, [pchar(buf)]);
end.
```

Тогда вызов процедуры `genkey`

```
printf('%s'#$a, [genkey(buf, 'abcdefghij', 12)]);
```

выведет такую строку ключа:

```
abcdefghij_000000012
```

11.12. Форматированный ввод: семейство процедур scanf

Описание

```
uses stdio;
(* Все параметры массива args являются указателями.
 * Переменные, на которые они указывают, могут
 * иметь произвольный тип.
 *)


function scanf(fmt:pchar; args:array of const):integer;

function fscanf(inf:pfile; fmt:pchar; args:array of const):integer;

function sscanf(str:pchar; fmt:pchar; args:array of const):integer;
```

Процедуры семейства `scanf` противоположны по смыслу процедурам семейства `printf`. Все они принимают ввод из файла (или из строки в случае процедуры `sscanf`), декодируют его в соответствии с информацией формата `fmt` и помещают полученные данные в переменные, заданные указателями в массиве `args`. Указатель файла перемещается на число обработанных символов.

Процедура `scanf` всегда выполняет чтение из `stdin`; процедура `fscanf` выполняет чтение из потока `inf`; а процедура `sscanf` выделяется в этом семействе процедур тем, что декодирует строку `str` и не осуществляет ввода данных. Поскольку последняя процедура работает со строкой в памяти, то она особенно полезна, если некоторую строку ввода нужно анализировать несколько раз.

Строка формата `fmt` имеет ту же структуру, что и строка формата процедуры `printf`. Например, следующий оператор считывает очередное целое число из потока стандартного ввода:

```
var
  inarg:integer;
scanf('%d', [@inarg]);
```

Важно, что функции `scanf` передается адрес переменной `inarg`. Это связано с тем, что, если нужно, чтобы процедура `scanf` изменяла переменную, которая находится в

вызывающей процедуре, следует передать указатель, содержащий адрес этой переменной. Можно очень легко забыть про символ `@`, что приведет к ошибке записи в память. Новичкам также приходится бороться с искушением помещать знак `@` перед всеми указателями, такими как имена символьных массивов.

В общем случае строка формата процедуры `scanf` может содержать:

- *пробельные символы*, то есть пробелы, символы табуляции, перевода строки и страницы. Обычно они соответствуют любым пробельным символам с текущей позиции во входном потоке, до первого не пробельного символа;
- *обычные, не пробельные символы*. Они должны точно совпадать с соответствующими символами во входном потоке;
- *спецификации формата*. Как упоминалось ранее, они в основном аналогичны спецификациям, используемым в процедуре `printf`.

Следующий пример показывает использование процедуры `scanf` с несколькими переменными различных типов:

```
(* Демонстрационная программа для процедуры scanf *)
uses stdio;

var
  i1, i2:integer;
  fit:float;
  str1, str2:array [0..9] of char;

begin
  scanf('%2d %2d %f %s %s', [@i1, @i2, @flt, pchar(str1), pchar(str2)]);
  .
  .
  .
end.
```

Первые две спецификации в строке формата сообщают процедуре `scanf`, что она должна считать два целых числа (в десятичном формате). Так как в обоих случаях ширина поля равна двум символам, предполагается, что первое число должно находиться в двух считанных первыми символах, а второе – в двух следующих (в общем случае ширина поля обозначает максимальное число символов, которое может занимать значение). Спецификации `%f` соответствует переменная типа `single`. Спецификация `%s` означает, что ожидается строка, ограниченная пробельными символами. Поэтому, если подать на вход программы последовательность

```
11 12 34.07 keith ben
```

то в результате получится следующее:

переменная `i1` будет иметь значение `11`
переменная `i2` будет иметь значение `12`
переменная `fit` будет иметь значение `34.07`
строка `str1` будет содержать значение `keith`
строка `str2` будет содержать значение `ben`

Обе строки будут заканчиваться нулевым символом. Обратите внимание, что переменные `str1` и `str2` должны иметь достаточно большую длину, чтобы в них поместились вводимые строки и нулевой символ в конце. Нельзя передавать процедуре `scanf` неинициализированный указатель.

Если задана спецификация формата `%s`, то предполагается, что строка должна быть ограничена пробельными символами. Для считывания строки целиком, включая пробельные символы, необходимо использовать код формата `%c`. Например, оператор

```
scanf ('%10c', [pchar(s1)]);
```

считает любые 10 символов из входного потока и поместит их в массив символов `s1`. Так как код формата `c` соответствует пробельным символам, то для получения следующего не

пробельного символа должна использоваться спецификация %1s, например:

```
(* Считать 2 символа, начиная с первого не пробельного*)
scanf('%1s%1c',[@c1,@c2])
```

Другим способом задания формата строчных данных, не имеющим аналога в формате процедуры printf, является *шаблон* (scan set). Это последовательность символов, заключенных в квадратные скобки: [и]. Входное поле составляется их максимальной последовательности символов, которые попадают в шаблон (в этом случае пробельные символы не игнорируются и не попадают в поле, если они не являются частью шаблона). Например, оператор

```
scanf('%[ab12]%s',[str1,str2]);
```

при задании входной строки

```
2bbaa1other
```

поместит в строку str1 значение 2bbaa1, а в строку str2 – значение other.

Существует несколько соглашений, используемых при создании шаблона, которые должны быть знакомы пользователям grep или ed. Например, диапазон символов задается строкой вида a-z, то есть [a-d] равносильно [abcd]. Если символ тире (-) должен входить в шаблон, то он должен быть первым или последним символом. Аналогично, если в шаблон должна входить закрывающая квадратная скобка], то она должна быть первым символом после открывающей квадратной скобки [. Если первым символом шаблона является знак ^, то при этом выбираются только символы, *не* входящие в шаблон.

Для присваивания переменных типа longint или double после символа процента в спецификации формата должен находиться символ l. Это позволяет процедуре scanf определять размер параметра, с которым она работает. Следующий фрагмент программы показывает, как можно считать из входного потока переменные обоих типов:

```
var
  l:longint;
  d:double;

scanf('%ld %lf', [@l, @d]);
```

Другая ситуация часто возникает, если входной поток содержит больше данных, чем необходимо. Для обработки этого случая спецификация формата может содержать символ (*) сразу же после символа процента, обозначающий данное поле лишним. В результате поле ввода, соответствующее спецификации, будет проигнорировано. Вызов

```
scanf('%d %*s %*d %s', [@ivar, str]);
```

для строки ввода

```
131 cat 132 mat
```

приведет к присвоению переменной ivar значения 131, пропуску следующих двух полей, а затем присвоению строке str значения mat.

И, наконец, какое значение возвращают функции семейства scanf? Они обычно возвращают число преобразованных и присвоенных полей. Возвращаемое значение может быть равно нулю в случае несоответствия строки формата и вводимых данных. Если ввод прекращается до первого успешно (или неуспешно) введенного поля, то возвращается значение EOF.[1]

Упражнение 11.7. *Напишите программу, выводящую в шестнадцатеричной и восьмеричной форме свои аргументы, которые должны быть десятичными целыми числами.*

---

[1] Опытные программисты не советуют использовать функцию fscanf (scanf) для ввода данных, кроме случаев простого интерактивного ввода. Вместо функции fscanf (scanf) предлагается использовать комбинацию вызовов fgets (gets) и sscanf. Недостаток функции fscanf (scanf) в данном случае состоит в том, что при случайном нарушении формата строки вводимых данных эта функция может перейти к чтению следующей строки, поскольку функция fscanf (scanf) не отличает символ окончания строки от других разделителей полей. В такой ситуации сложно обработать ошибку входных данных корректно, а кроме того, ввод следующей строки тоже будет нарушен.

Упражнение 11.8. *Напишите программу* `savematrix`*, которая должна сохранять в файле матрицу целых чисел произвольного размера в удобочитаемом формате, и программу* `readmatrix`*, которая загружает матрицу из файла. Используйте для этого только процедуры* `fprintf` *и* `fscanf`*. Постарайтесь свести к минимуму число пробельных символов (пробелы, символы табуляции и др.) в файле. Совет: используйте для задания формата записи в файл символ переменной ширины (*).*

11.13. Запуск программ при помощи библиотек стандартного ввода/вывода

Стандартная библиотека ввода/вывода содержит несколько процедур для запуска одних программ из других. Основной из них является уже известная процедура `runshell`.

Описание

```
uses stdio;

function runshell(comstring:pchar):longint;

uses linux;

function shell(comstring:pchar):longint;
```

Функция `runshell` из файла `stdio`, как и `shell` из `linux`, выполняет команду, заданную строкой `comstring`. Вначале она создает дочерний процесс, который, в свою очередь, осуществляет вызов `exec` для запуска стандартного командного интерпретатора UNIX с командной строкой `comstring`. В это время процедура `runshell` в первом процессе выполняет вызов `wait`, гарантируя тем самым, что выполнение продолжится только после того, как запущенная команда завершится. Возвращаемое после этого значение `retval` содержит статус выхода командного интерпретатора, по которому можно определить, было ли выполнение программы успешным или нет. В случае неудачи любого из вызовов `fork` или `exec` значение переменной `retval` будет равно –1.

Поскольку в качестве посредника выступает командный интерпретатор, строка `comstring` может содержать любую команду, которую можно набрать на терминале. Это позволяет программисту воспользоваться такими преимуществами командного интерпретатора, как перенаправление ввода/вывода, поиск файлов в пути и т.д. Следующий оператор использует процедуру `runshell` для создания подкаталога при помощи программы `mkdir`:

```
retval := runshell('mkdir workdir');
if retval <> 0 then
  writeln(stderr, 'Процедура runshell вернула значение ', retval);
```

Остановимся на некоторых важных моментах. Во-первых, процедура `runshell` в вызывающем процессе будет игнорировать сигналы `SIGINT` и `SIGQUIT`. Это позволяет пользователю прерывать выполнение команды, не затрагивая родительский процесс. Во-вторых, команда, выполняемая процедурой `runshell`, будет наследовать из вызывающего процесса некоторые открытые дескрипторы файлов. В частности, стандартный ввод команды будет получен из того же источника, что и в родительском процессе. При вводе из файла могут возникнуть проблемы, если процедура `runshell` используется для запуска интерактивной программы, так как ввод программы также будет производиться из файла.

Процедура `runshell` имеет один серьезный недостаток. Он не позволяет программе получать доступ к выводу запускаемой программы. Для этого можно использовать две другие процедуры из стандартной библиотеки ввода/вывода: `pipeopen/popen` и `pipeclose/pclose`.

Описание

```
uses stdio;
```

```
function pipeopen(comstring, _type:pchar):pfile;

function pipeclose(strm:pfile):integer;

Procedure POpen(Var F:FileType; comstring:pathstr; _type:char);

Function PClose(Var F:FileType):longint;
```

Как и процедура `runshell`, процедуры `popen` и `pipeopen` создает дочерний процесс командного интерпретатора для запуска команды, заданной параметром `comstring`. Но, в отличие от процедуры `runshell`, она также создает канал между вызывающим процессом и командой. При этом `pipeopen` возвращает структуру `TFILE`, связанную с этим каналом, а `popen` – переменную файлового типа. Если значение параметра `_type` равно `w`, то программа может выполнять запись в стандартный ввод при помощи структуры `TFILE`. Если же значение параметра `_type` равно `r`, то программа сможет выполнять чтение из стандартного вывода программы. Таким образом, процедуры `popen` и `pipeopen` представляют простой и понятный метод взаимодействия с другой программой.

Для закрытия потока, открытого при помощи процедуры `popen`, должна всегда использоваться процедура `pclose`. Она будет ожидать завершения команды, после чего вернет статус ее завершения.

Пример использования `POpen`:

```
uses linux;

var f : text;
    i : longint;

begin
  writeln ('Creating a shell script to which echoes its arguments');
  writeln ('and input back to stdout');
  assign (f,'test21a');
  rewrite (f);
  writeln (f,'#!/bin/sh');
  writeln (f,'echo this is the child speaking.... ');
  writeln (f,'echo got arguments \*"$*"\*');
  writeln (f,'cat');
  writeln (f,'exit 2');
  writeln (f);
  close (f);
  chmod ('test21a',octal (755));
  popen (f,'./test21a arg1 arg2','W');
  if linuxerror<>0 then
     writeln ('error from POpen : Linuxerror : ', Linuxerror);
  for i:=1 to 10 do
    writeln (f,'This is written to the pipe, and should appear on stdout.');
  Flush(f);
  Writeln ('The script exited with status : ',PClose (f));
  writeln;
  writeln ('Press <return> to remove shell script.');
  readln;
  assign (f,'test21a');
  erase (f)
end.
```

Следующий пример, процедура `getlist`, использует процедуру `popen` и команду `ls` для вывода списка элементов каталога. Каждое имя файла затем помещается в двухмерный массив символов, адрес которого передается процедуре `getlist` в качестве параметра.

```
(* getlist - процедура для получения списка файлов в каталоге *)
```

```
uses stdio, strings;

const
  MAXLEN=255; (* Максимальная длина имени файла *)
  MAXCMD=100; (* Максимальная длина команды *)
  ERROR=-1;
  SUCCESS=0;

type
  sarray=array [0..MAXLEN] of char;
  darray=array [0..MAXCMD] of sarray;


function getlist(namepart:pchar; var dirnames:darray;
                 maxnames:integer):integer;
var
  cmd:array [0..MAXCMD] of char;
  in_line:array [0..MAXLEN+1] of char;
  i:integer;
  lsf:pfile;
begin
  (* Основная команда *)
  strcopy(cmd, 'ls ');

  (* Дополнительные параметры команды *)
  if namepart <> nil then
    strlcat(cmd, namepart, MAXCMD - strlen(cmd));

  lsf := pipeopen(cmd, 'r'); (* Запускаем команду *)
  if lsf = nil then
  begin
    getlist:=ERROR;
    exit;
  end;

  for i:=0 to maxnames-1 do
  begin
    if fgets(in_line, MAXLEN+2, lsf) = nil then
      break;

    (* Удаляем символ перевода строки *)
    if in_line[strlen(in_line)-1] = #$a then
      in_line[strlen(in_line)-1] := #0;

    strcopy(dirnames[i], in_line);
  end;
  if i < maxnames then
    dirnames[i][0] := #0;

  pipeclose (lsf);
  getlist:=SUCCESS;
end;

var
  namebuf:darray;
  i:integer;
begin
  getlist('*.pas', namebuf, 100);
```

```
  i:=0;
  while namebuf[i][0]<>#0 do
  begin
    writeln(namebuf[i]);
    inc(i);
  end;
end.
```

Процедура `getlist` может быть вызвана следующим образом:

```
getlist('*.pas', namebuf, 100);
```

при этом в переменную `namebuf` будут помещены имена всех Паскаль-программ в текущем каталоге.

Следующий пример разрешает обычную проблему, с который часто сталкиваются администраторы UNIX: как быстро «освободить» терминал, который был заблокирован какой-либо программой, например, неотлаженной программой, работающей с экраном. Программа `unfreeze` принимает в качестве аргументов имz терминала и список программ. Затем она запускает команду вывода списка процессов `ps` при помощи процедуры `popen` для получения списка связанных с терминалом процессов и выполняет поиск указанных программ в этом списке процессов. Далее программа `unfreeze` запрашивает разрешение пользователя на завершение работы каждого из процессов, удовлетворяющих критерию.

Программа `ps` сильно привязана к конкретной системе. Это связано с тем, что она напрямую обращается к ядру (через специальный файл, представляющий образ системной памяти) для получения системной таблицы процессов. На системе, использованной при разработке этого примера, команда `ps` имеет синтаксис

```
$ ps -t ttyname
```

где `ttyname` является именем специального файла терминала в каталоге `/dev`, например, `tty1`, `console`, `pts/8` и др. Выполнение этой команды `ps` дает следующий вывод:

```
PID TTY TIME  COMMAND
29  co  0:04  sh
39  co  0:49  vi
42  co  0:00  sh
43  co  0:01  ps
```

Первый столбец содержит идентификатор процесса. Второй – имя терминала, в данном случае `co` соответствует консоли. В третьем столбце выводится суммарное время выполнения процесса. В последнем, четвертом, столбце выводится имя выполняемой программы. Обратите внимание на первую строку, которая является заголовком. В программе `unfreeze`, текст которой приведен ниже, нам потребуется ее пропустить.

```
(* Программа unfreeze - освобождение терминала *)
uses stdio,linux,strings;

const
  LINESZ =150;
  SUCCESS=0;
  ERROR  =(-1);

const
  killflag:integer=0;
  (* Инициализация этой переменной зависит от вашей системы *)
  pspart:pchar = 'ps t ';
  fmt:pchar = '%d %*s %*s %*s %s';
var
  comline, inbuf, header, name:array [0..LINESZ-1] of char;
  f:pfile;
  j:integer;
  pid:longint;
begin
```

```
  if paramcount <2 then
  begin
    writeln (stderr, 'синтаксис: ',paramstr(0),' терминал программа ...');
    halt (1);
  end;

  (* Сборка командной строки *)
  strcopy (comline, pspart);
  strcat (comline, argv[1]);

  (* Запуск команды ps *)
  f := pipeopen (comline, 'r');
  if f = nil then
  begin
    writeln (stderr, paramstr(0),': не могу запустить команду ps ');
    halt (2);
  end;

  (* Получить первую строку от ps и игнорировать ее *)
  if fgets (header, LINESZ, f) = nil then
  begin
    writeln (stderr, paramstr(0),': нет вывода от ps?');
    halt (3);
  end;

  (* Поиск программы, которую нужно завершить *)
  while fgets (inbuf, LINESZ, f) <> nil do
  begin
    if sscanf (inbuf, fmt, [@pid, pchar(name)]) < 2 then
      break;
    for j := 2 to argc-1 do
    begin
      if strcomp (name, argv[j]) = 0 then
      begin
        if dokill (pid, inbuf, header) = SUCCESS then
          inc(killflag);
      end;
    end;
  end;
  (* Это предупреждение, а не ошибка *)
  if killflag=0 then
    writeln(stderr, paramstr(0),': работа программы не завершена ',
            paramstr(1));
  pipeclose(f);
  halt (0);
end.
```

Ниже приведена реализация процедуры `dokill`, вызываемой программой `unfreeze`. Обратите внимание на использование процедуры `readln` для чтений первого не пробельного символа (вместо нее можно было бы использовать и функцию `yesno`, представленную в разделе 11.8).

```
(* Получить подтверждение, затем завершить работу программы *)
function dokill(procid:longint;line,hd:pchar):integer;
var
  c:char;
begin
  writeln (#$a'Найден процесс, выполняющий заданную программу :');
  writeln (#9,hd,#9,line);
```

```
  writeln ('Нажмите `y` для завершения процесса ', procid);
  write (#$a'Yes\No? > ');
  (* Введите следующий не пробельный символ *)
  readln (c);
  if (c = 'y') or (c = 'Y') then
  begin
    kill (procid, SIGKILL);
    dokill:=SUCCESS;
    exit;
  end;
  dokill:=ERROR;
end;
```

Упражнение 11.9. *Напишите свою версию процедуры* `getcwd`*, которая возвращает строку с именем текущего рабочего каталога. Назовите вашу программу* `wdir`*. Совет: используйте стандартную команду* `pwd`*.*

Упражнение 11.10. *Напишите программу* `arrived`*, которая запускает программу* `who` *при помощи процедуры* `popen` *для проверки (с 60-секундными интервалами), находится ли в системе пользователи из заданного списка. Список пользователей должен передаваться программе* `arrived` *в качестве аргумента командной строки. При обнаружении кого-нибудь из перечисленных в списке пользователей, программа* `arrived` *должна выводить сообщение. Программа должна быть достаточно эффективной, для этого используйте вызов* `sleep` *между выполнением проверок. Программа* `who` *должна быть описана в справочном руководстве системы.*

11.14. Вспомогательные процедуры

Этот раздел будет посвящен краткому описанию различных дополнительных процедур стандартной библиотеки ввода/вывода. Более подробная информация содержится в справочном руководстве системы.

11.14.1. Процедуры freopen и fdopen

Описание

```
uses stdio;

function freopen(filename:pchar; _type:pchar; oldstream:pfile):pfile;

function fdopen(fildes:longint; _type:pchar):pfile;
```

Процедура `freopen` закрывает поток `oldstream`, а затем открывает его для ввода из файла `filename`. Параметр `_type` определяет режим доступа к новой структуре `TFILE` и принимает те же значения, что и аналогичный аргумент процедуры `fopen` (строки `r`, `w` и др.). Процедура `freopen` обычно используется для перенаправления `stdin`, `stdout` или `stderr`, например:

```
if freopen('new.input', 'r', stdin) = nil then
  fatal('Невозможно перенаправить stdin');
```

Процедура `fdopen` связывает новую структуру `TFILE` с целочисленным дескриптором файла `filedes`, полученным при выполнении одного из системных вызовов `fdcreat`, `fdopen`, `assignpipe` или `dup2`.

В случае ошибки обе процедуры возвращают `nil`.

11.14.2. Управление буфером: процедуры setbuf и setvbuf

Описание

```
uses stdio;

procedure setbuf(stream:pfile; buf1:pchar);
```

```
function setvbuf(stream:pfile; buf1:pchar; _type:longint; size:longint):
        integer;
```

Эти процедуры позволяют программисту в некоторой степени управлять буферами потоков. Они должны использоваться после открытия файла, но до первых операций чтения или записи.

Процедура `setbuf` подставляет буфер `buf1` вместо буфера, выделяемого стандартной библиотекой ввода/вывода. Размер сегмента памяти, на который указывает параметр `buf1`, должен быть равен константе `BUFSIZ`, определенной в файле `stdio`.

Процедура `setvbuf` позволяет осуществлять более тонкое управление буферизацией, чем процедура `setbuf`. Параметр `buf2` задает адрес нового буфера, а параметр `size` – его размер. Если вместо адреса буфера передается значение `nil`, то используется буферизация по умолчанию. Параметр `_type` в процедуре `setvbuf` определяет метод буферизации потока `stream`. Он позволяет настроить поток для использования с конкретным типом устройства, например, для дисковых файлов или терминальных устройств. Возможны три значения `_type`, они определены в файле `stdio`:

`_IOFBF` Поток файла буферизуется полностью. Этот режим включен по умолчанию для всех потоков ввода/вывода, не связанных с терминалом. Данные при этом будут записываться или считываться блоками размером `BUFSIZ` байтов для обеспечения максимальной эффективности

`_IOLBF` Вывод буферизируется построчно, и буфер сбрасывается при записи символа перевода строки. Он также очищает буфер вывода при его заполнении или при поступлении запроса на ввод. Этот режим включен по умолчанию для терминалов и служит для поддержки интерактивного использования

`_IOBNF` Ввод и вывод не буферизуются. В этом случае параметры `buf2` и `size` игнорируются. Этот режим иногда необходим, например, для записи диагностических сообщений в файл протокола

Обратите внимание, что при задании недопустимого значения любого из параметров `type` или `size` процедура `setvbuf` возвращает ненулевое значение. В случае успеха возвращается 0.

Глава 12. Разные дополнительные системные вызовы и библиотечные процедуры

12.1. Введение

В предпоследней главе будут рассмотрены несколько системных вызовов и некоторые полезные библиотечные процедуры, которые не соответствовали тематике всех предыдущих глав, такие как управление памятью, работа с временными значениями, предикаты типов символов, функции работы со строками.

12.2. Управление динамическим распределением памяти

Каждая из программ, рассмотренных ранее, использовала структуры данных, определенные при помощи стандартных объявлений языка Паскаль, например:

```
var
  x, y: something;
  z: ^something;
  a: array [0..19] of something;
```

Другими словами, расположение данных в наших примерах определялось во время компиляции. Однако многие вычислительные задачи удобнее решать при помощи динамического создания и уничтожения структур данных, а это значит, что расположение данных в программе определяется окончательно только во время выполнения программы. Семейство библиотечных функций ОС UNIX `malloc` (от *memory allocation* – выделение памяти) позволяет создавать объекты в области динамической памяти на стадии выполнения программы, которая в англоязычной литературе часто называется *heap* – «куча», «кипа» (книг). Функция `malloc` определяется следующим образом:

Описание

```
uses stdio;

function malloc(nbytes:longint):pointer;
```

В результате этого вызова функция `malloc` обычно возвращает указатель на участок памяти размером `nbytes`. При этом программа получает указатель на массив байтов, которые она может использовать по своему усмотрению. Если памяти недостаточно и система не может выделить запрошенный вызовом `malloc` объем памяти, то вызов возвращает нулевой указатель.

Обычно вызов `malloc` используется для выделения памяти под одну или несколько структур данных, например:

```
var
  p:^item;

  p := malloc(sizeof(item));
```

В случае успеха вызов `malloc` создает новую структуру `item`, на которую ссылается указатель `p`. Заметим, что возвращаемое вызовом `malloc` значение приводится к соответствующему типу указателя. Это помогает избежать вывода предупреждений компилятором или такими программами, как `lint`. Приведение типа в данном случае целесообразно, поскольку вызов `malloc` реализован так, чтобы отводить память под объекты любого типа при условии, что запрашивается достаточный для хранения объекта ее объем. Такие задачи, как выравнивание по границе слова, решаются самим алгоритмом функции `malloc`. Обратите внимание на то, что размер структуры `item` задается при помощи конструкции `sizeof`, которая возвращает размер объекта в байтах.

Функция `free` противоположна по своему действию функции `malloc` и возвращает отведенную память, позволяя использовать ее повторно. Функции `free` передается указатель, который был получен во время вызова `malloc`:

```
var
```

```
  ptr:^item;
.
.
.
ptr := malloc(sizeof(item));

(* Выполнить какие-либо действия .. *)
free(ptr);
```

После подобного вызова функции `free` освобожденный участок памяти, на который указывает `ptr`, уже нельзя использовать, так как функция `malloc` может позднее снова его отдать процессу (целиком или частично). Очень важно, чтобы, функции `free` передавался указатель, который был получен от функции `malloc`, или от одной из функций `calloc` и `realloc`, принадлежащих тому же семейству. Если указатель не удовлетворяет этому требованию, то почти наверняка возникнут ошибки в механизме распределения динамической памяти, которые приведут к некорректной работе программы или к ее аварийному завершению. Неправильное применение функции `free` – очень часто встречающаяся и трудноуловимая ошибка.[1]

Еще две функции в семействе `malloc` непосредственно связаны с выделением памяти. Первой из них является функция `calloc`.

Описание

```
uses stdio;

function calloc(nelem, nbytes:longint):pointer;
```

Функция `calloc` отводит память под массив из `nelem` элементов, размер каждого из которых равен `nbytes`. Она обычно используется следующим образом:

```
(* Выделить память под массив структур *)
var
  aptr:^item;
.
.
.
aptr := calloc(nitem, sizeof(item));
```

В отличие от функции `malloc`, память, отводимая функцией `calloc`, заполняется нулями, что приводит к задержке при ее выполнении, но может быть полезно в тех случаях, когда такая инициализация необходима.

Последней функцией выделения памяти из семейства `malloc` является функция `realloc`.

Описание

```
uses stdio;

function realloc(oldptr:pointer; newsize:longint):pointer;
```

Функция `realloc` используется для изменения размера блока памяти, на который указывает параметр `oldptr` и который был получен ранее в результате вызова `malloc`, `calloc` или `realloc`. При этом блок памяти может переместиться, и возвращаемый указатель задает новое положение его начала. После изменения размера блока сохраняется содержимое его части, соответствующей меньшему из старого и нового размеров.

Пример использования функции malloc: связные списки

В информатике существует множество типов динамических структур данных. Одним из классических примеров таких структур является связный список, в котором группа

[1] Для отладки программ, интенсивно использующих динамическую память, существуют специальные библиотеки, подменяющие стандартный механизм процедур семейства `malloc`. Эти библиотеки менее производительны, зато выполняют функцию «раннего предупреждения» подобных ошибок, то есть незамедлительно фиксируют нарушения правил работы с динамической памятью.

одинаковых объектов связывается в единый логический объект. В этом разделе составляется простой пример, использующий односвязный список для демонстрации применения функций семейства malloc.

```
(* list.inc - заголовочный файл для примера. *)
uses strings,stdio;

(* Определение основной структуры *)
type
pMEMBER=^MEMBER;
MEMBER=record
  m_data:pchar;
  m_next:pMEMBER;
end;
ppMEMBER=^pMEMBER;

(* Определение функций *)
function new_member(data:pchar):pMEMBER;forward;
procedure add_member(head:ppMEMBER; newmem:pMEMBER);forward;
procedure free_list (head:ppMEMBER);forward;
procedure printlist (listhead:pMEMBER);forward;
```

Оператор type вводит тип MEMBER, имеющий два поля. Первое поле, m_data, в экземпляре типа MEMBER будет указывать на произвольную строку. Второе поле, m_next, указывает на другой объект типа MEMBER.

Каждый элемент в связном списке структур типа MEMBER будет указывать на следующий в списке объект MEMBER, то есть если известен один элемент списка, то следующий можно найти при помощи указателя m_next. Поскольку на каждый объект MEMBER в списке ссылается только один указатель, список можно обходить только в одном направлении. Такие списки называются *односвязными* (singly linked). Если бы был определен еще один указатель m_prev, то список можно было бы обходить и в обратном направлении, и в этом случае он был бы *двусвязным* (doubly linked).

Адрес начала, или головы, списка обычно записывается в отдельном указателе, определенным следующим образом:

```
const
  head:pmember = nil;
```

Конец списка обозначается нулевым значением поля m_next последнего элемента списка.

На рис. 12.1 показан простой список из трех элементов. Его начало обозначено указателем head.

Теперь представим небольшой набор процедур для работы с этими структурами. Первая функция называется new_member. Она отводит память под структуру MEMBER с помощью вызова malloc. Обратите внимание, что указателю m_next присвоено нулевое значение, в данном случае обозначенное как nil. Это связано с тем, что функция malloc не обнуляет выделяемую память. Поэтому при создании структуры MEMBER поле m_text может содержать ложный, но правдоподобный адрес.

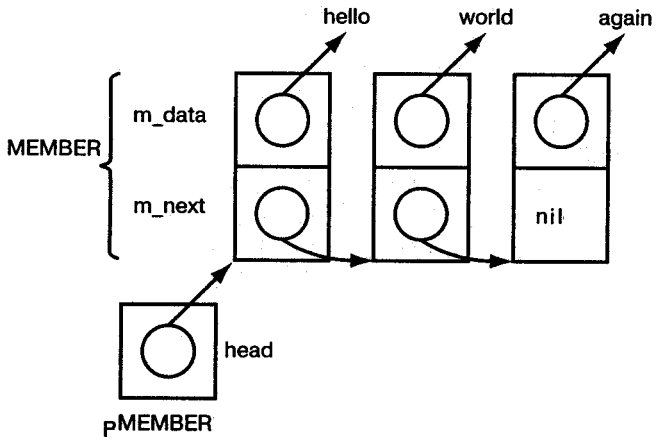


Рис. 12.1. Связный список объектов MEMBER

```
{$i list.inc}

(* Функция new_member - выделить память для нового элемента *)
function new_member (data:pchar):pMEMBER;
var
  newmem:pMEMBER;
begin
  newmem := malloc (sizeof (MEMBER));
  if newmem = nil then
    writeln (stderr, 'new_member: недостаточно памяти')
  else
  begin
    (* Выделить память для копирования данных *)
    newmem^.m_data := malloc (strlen (data) + 1);
    (* Скопировать данные в структуру *)
    strcopy (newmem^.m_data, data);
    (* Обнулить указатель в структуре *)
    newmem^.m_next := nil;
  end;
  new_member:=newmem;
end;
```

Следующая процедура `add_member` добавляет в список, на который указывает `head`, новый элемент `MEMBER`. Как видно из текста процедуры, элемент `MEMBER` добавляется всегда в начало списка.

```
{$i list.inc}

(* Процедура add_member - добавить новый элемент MEMBER *)
procedure add_member (head:ppMEMBER; newmem:pMEMBER);
begin
  (* Эта простая процедура вставляет новый
   * элемент в начало списка
   *)
  newmem^.m_next := head^;
  head^ := newmem;
end;
```

Последняя процедура – `free_list`. Она принимает указатель на начало списка `head^` и освобождает память, занятую всеми структурами `MEMBER`, образующими список. Она также обнуляет указатель `head^`, гарантируя, что в указателе `head^` не содержится прежнее значение (иначе при случайном использовании `head^` могла бы возникать трудноуловимая ошибка).

```
{$i list.inc}
```

```
(* Процедура free_list - освободить занятую списком память *)
procedure free_list (head:ppMEMBER);
var
  curr, next:pMEMBER;
begin
  curr := head^;
  while curr <> nil do
  begin
    next := curr^.m_next;
    (* Освободить память, занятую данными *)
    free (curr^.m_data);
    (* Освободить память, отведенную под структуру списка *)
    free (curr);
    curr := next;
  end;
  (* Обнулить указатель на начало списка *)
  head^ := nil;
end;
```

Следующая простая программа использует описанные процедуры. Она создает односвязный список, изображенный на рис. 12.1, а затем удаляет его. Обратите внимание на способ обхода списка процедурой `printlist`. Используемый в ней цикл `while` типичен для программ, в которых применяются связные списки.

```
{$i list.inc}

(* Обход и вывод списка *)
procedure printlist (listhead:pMEMBER);
var
  m:pMEMBER;
begin
  writeln (#$a'Содержимое списка:');
  if listhead = nil then
    writeln (#9'(пусто)')
  else
  begin
    m := listhead;
    while m <> nil do
    begin
      writeln (#9, m^.m_data);
      m := m^.m_next;
    end;
  end;
end;


const
  strs:array [0..2] of pchar=('again', 'world', 'Hello');

(* Программа для проверки процедур работы со списком *)

var
  head, newm:pMEMBER;
  j:integer;
begin
  (* Инициализация списка *)
  head := nil;
  (* Добавление элементов к списку *)
  for j:=0 to 2 do
```

```
  begin
    newm := new_member (strs[j]);
    add_member (@head, newm);
  end;
  (* Вывести элементы списка *)
  printlist (head);
  (* Удалить список *)
  free_list (@head);
  (* Вывести элементы списка *)
  printlist (head);
end.
```

Следует обратить внимание и на то, как в начале программы был инициализирован список присвоением указателю `head` значения `nil`. Это важно, так как иначе список мог оказаться заполненным «мусором», что неизбежно привело бы к ошибочной работе или к аварийному завершению программы.

Рассматриваемая программа должна выдать такой результат:

```
Содержимое списка:
        Hello world again

Содержимое списка:
        (пусто)
```

Вызовы brk и sbrk

Для полноты изложения необходимо упомянуть вызовы `brk` и `sbrk`. Это базовые низкоуровневые вызовы UNIX для динамического выделения памяти. Они изменяют размер сегмента данных процесса или, если быть более точным, смещают верхнюю границу сегмента данных процесса. Вызов `brk` устанавливает абсолютное значение границы сегмента, а вызов `sbrk` – ее относительное смещение. В большинстве ситуаций для выделения динамической памяти рекомендуется использовать функции семейства `malloc`, а не эти вызовы.

Упражнение 12.1. *Односвязный список нашего примера может использоваться для реализации стека, в котором первым используется последний добавленный элемент. Процедура* `add_member` *будет соответствовать операции вставки (*`push`*) данных в стек. Напишите процедуру, реализующую обратную операцию извлечения (*`pop`*) данных из стека за счет удаления первого элемента списка.*

Упражнение 12.2. *Напишите программу, использующую функции семейства* `malloc` *для выделения памяти для целого числа, массива из трех переменных типа* `integer` *и массива указателей на переменные типа* `char`*.*

12.3. Ввод/вывод с отображением в память и работа с памятью

При выполнении процессом ввода/вывода больших объемов данных с диска, возникают накладные расходы, связанные с двойным копированием данных сначала с диска во внутренние буферы ядра, а потом из этих буферов в структуры данных процесса. Ввод/вывод с отображением в память увеличивает скорость доступа к файлу, напрямую «отображая» дисковый файл в пространство памяти процесса. Как уже было видно на примере работы с разделяемой памятью в разделе 8.3.4, работа с данными в адресном пространстве процесса является наиболее эффективной формой доступа. (Но хотя использование ввода/вывода с отображением в память и приводит к ускорению доступа к файлам, его интенсивное использование может уменьшить объем памяти, доступный другим средствам, использующим память процесса.)

Вскоре будут рассмотрены системные вызовы `mmap` и `munmap`. Но вначале опишем несколько простых процедур для работы с блоками памяти.

Описание

```
uses stdio;

function memset(buf:pointer; character:longint; size:longint):pointer;
function memcpy(buf1:pointer; const buf2:pointer; size:longint):pointer;
function memmove(buf1:pointer; const buf2:pointer; size:longint):pointer;

function memcmp(const buf1, buf2:pointer; size:longint):longint;

function memchr(const buf:pointer; character:longint; size:longint):
         pointer;
```

Для инициализации массивов данных можно использовать процедуру `memset`, записывающую значения `character` в первые `size` байтов массива памяти `buf`.

Для прямого копирования одного участка памяти в другой можно использовать любую из процедур `memcpy` или `memmove`. Обе эти функции перемещают `size` байт памяти, начиная с адреса `buf1` в участок, начинающийся с адреса `buf2`. Разница между этими двумя функциями состоит в том, что функция `memmove` гарантирует, что если области источника и адресата копирования `buf1` и `buf2` перекрываются, то при перемещении данные не будут искажены. Для этого функция `memmove` вначале копирует сегмент `buf2` во временный массив, а затем копирует данные из временного массива в сегмент `buf1` (или использует более разумный алгоритм).

Функция `memcmp` работает аналогично функции `strcmp`. Если первые `size` байтов `buf1` и `buf2` совпадают, то функция `memcmp` вернет значение 0.

Функция `memchr` проверяет первые `size` байтов `buf` и возвращает либо адрес первого вхождения символа `character`, либо значение `nil`. Процедуры `memset`, `memcpy` и `memmove` в случае успеха возвращают значение первого параметра.

Системные вызовы mmap и munmap

Вернемся к отображению файлов в память. Ввод/вывод с отображением в память реализуется при помощи системного вызова `mmap`. Вызов `mmap` работает со страницами памяти, и отображение файла в памяти будет выровнено по границе страницы. Вызов `mmap` определяется следующим образом:

Описание

```
uses linux;

type
  tmmapargs=record
    address : longint;
    size    : longint;
    prot    : longint;
    flags   : longint;
    fd      : longint;
    offset  : longint;
  end;

Function MMap(const m:tmmapargs):longint;
```

Файл должен быть заранее открыт при помощи системного вызова `fdopen`. Полученный дескриптор файла используется в качестве поля `fd` в структуре `tmmapargs`.

Поле `address` позволяет программисту задать начало отображаемого файла в адресном пространстве процесса. Как и для вызова `shmat`, рассмотренного в разделе 8.3.4, в этом случае программе нужно знать расположение данных и кода процесса в памяти. Из соображений безопасности и переносимости лучше, конечно, позволить выбрать начальный адрес системе, и это можно сделать, присвоив параметру `address` значение 0. При этом возвращаемое вызовом `mmap` значение является адресом начала отображения. В случае

ошибки вызов mmap вернет значение -1.

Поле offset определяет смещение в файле, с которого начинается отображение данных. Обычно нужно отображать в память весь файл, поэтому поле offset часто равно 0, что соответствует началу файла. Если поле offset не равно нулю, то оно должно быть кратно размеру страницы памяти.

Число отображаемых в память байтов файла задается полем size. Если это поле не кратно размеру страницы памяти, то size байтов будут взяты из файла, а оставшаяся часть страницы будет заполнена нулями.

Поле prot определяет, можно ли выполнять чтение, запись или выполнение содержимого адресного пространства отображения. Поле prot может быть комбинацией следующих значений, определенных в файле linux:

PROT_READ	Разрешено выполнять чтение данных из памяти
PROT_WRITE	Разрешено выполнять запись данных в память
PROT_EXEC	Разрешено выполнение кода, содержащегося в памяти
PROT_NONE	Доступ к памяти запрещен

Значение поля prot не должно противоречить режиму, в котором открыт файл.

Поле flags влияет на доступность изменений отображенных в память данных файла для просмотра другими процессами. Наиболее полезны следующие значения этого параметра:

MAP_SHARED	Все изменения в области памяти будут видны в других процессах, также отображающих файл в память, изменения также записываются в файл на диске
MAP_PRIVATE	Изменения в области памяти не видны в других процессах и не записываются в файл

После завершения процесса отображение файла автоматически отменяется. Чтобы отменить отображение файла в память до завершения программы, можно использовать системный вызов munmap.

Описание

```
uses linux;

function MUnMap(address:Pointer; length:Longint):Boolean;
```

Если был задан флаг MAP_SHARED, то в файл вносятся все оставшиеся изменения, при флаге MAP_PRIVATE все изменения отбрасываются.

Следует иметь в виду, что эта функция только отменяет отображение файла в память, но не закрывает файл. Файл требуется закрыть при помощи системного вызова fdclose.

Следующий пример повторяет программу copyfile, последний вариант которой был рассмотрен в разделе 11.4. Эта программа открывает входной и выходной файлы и копирует один из них в другой. Для простоты опущена часть процедур обработки ошибок.

```
uses linux;

var
  i, input, output, filesize:longint;
  source, target:pchar;
  args:tmmapargs;
const
  endchar:char=#0;
type
  oearray=array [0..0] of char;
  poearray=^oearray;
begin
  (* Проверка числа входных параметров *)
  if paramcount <> 2 then
  begin
```

```
    writeln(stderr, 'Синтаксис: copyfile источник цель');
    halt (1);
  end;

  (* Открыть входной и выходной файлы *)
  input := fdopen (paramstr(1), Open_RDONLY);
  if input = -1 then
  begin
    writeln(stderr, 'Ошибка при открытии файла ', paramstr(1));
    halt (1);
  end;

  output := fdopen (paramstr(2), Open_RDWR or Open_CREAT or Open_TRUNC,
octal(0666));
  if output = -1 then
  begin
    fdclose (input);
    writeln(stderr, 'Ошибка при открытии файла ', paramstr(2));
    halt (2);
  end;

  (* Создать второй файл с тем же размером, что и первый. *)
  filesize := fdseek (input, 0, SEEK_END);
  fdseek (output, filesize - 1, SEEK_SET);
  fdwrite (output, endchar, 1);

  (* Отобразить в память входной и выходной файлы. *)
  args.fd:=input;
  args.flags:=MAP_SHARED;
  args.prot:=PROT_READ;
  args.size:=filesize;
  args.address:=0;
  args.offset:=0;
  source:=pchar(mmap(args));
  if longint(source) = -1 then
  begin
    writeln(stderr, 'Ошибка отображения файла 1 в память');
    halt (1);
  end;

  args.fd:=output;
  args.flags:=MAP_SHARED;
  args.prot:=PROT_WRITE;
  args.size:=filesize;
  args.address:=0;
  args.offset:=0;
  target:=pchar(mmap(args));
  if longint(target) = -1 then
  begin
    writeln(stderr, 'Ошибка отображения файла 2 в память');
    halt (1);
  end;

  (* Копирование *)
  for i:=0 to filesize-1 do
    poearray(target)^[i] := poearray(source)^[i];

  (* Отменить отображение обоих файлов *)
```

```
  munmap (source, filesize);
  munmap (target, filesize);
  (* Закрыть оба файла *)
  fdclose (input);
  fdclose (output);
  halt (0);
end.
```

Конечно, файлы были бы автоматически закрыты при завершении программы. Вызовы `munmap` включены для полноты изложения.

12.4. Время

В ОС UNIX существует группа процедур для установки и получения системного времени. Время в системе измеряется как число секунд, прошедших с 00:00:00 по Гринвичу с 1 января 1970 г., и размер переменной для хранения этого числа должен быть не меньше формата `longint`.

Основным вызовом этой группы является системный вызов `GetTimeOfDay`, который возвращает текущее время в стандартном формате времени системы UNIX.

Описание

```
uses linux;

Function GetTimeOfDay:longint;
```

Человеку сложно представлять время в виде большого числа секунд, поэтому в ОС UNIX существует набор библиотечных процедур для перевода системного времени в более понятную форму. Наиболее общей является процедура `ctime`, которая преобразует вывод вызова `GetTimeOfDay` в строку из 26 символов, например, выполнение следующей программы

```
uses linux,stdio;

var
  tt:longint;
begin
  tt:=gettimeofday;
  write('Текущее время ', ctime (tt));
  halt(0);
end.
```

дает примерно такой вывод:

```
Текущее время Tue Mar 18 00:17:06 1998
```

С функцией `ctime` связан набор процедур, использующих структуры типа `tm`. Тип структуры `tm` определен в файле `stdio` и содержит следующие элементы:

```
tm=record
  tm_sec:longint;          (* Секунды *)
  tm_min:longint;          (* Минуты *)
  tm_hour:longint;         (* Часы от 0 до 24 *)
  tm_mday:longint;         (* Дни месяца от 1 до 31 *)
  tm_mon:longint;          (* Месяц от 0 до 11 *)
  tm_year:longint;         (* Год минус 1900 *)
  tm_wday:longint;         (* День недели Воскресенье = 0 *)
  tm_yday:longint;         (* День года 0-365 *)
  tm_isdst:longint;        (* Флаг летнего времени только для США *)
end;
ptm=^tm;
```

Назначение всех элементов очевидно. Некоторые из процедур, использующих эту структуру, описаны ниже.

Описание

```
uses stdio;
```

```
function localtime(var _timeval:longint):ptm;

function gmtime(var _timeval:longint):ptm;

function asctime(const tptr:ptm):pchar;

function mktime(tptr:ptm):longint;

function difftime(time1, time2:longint):double;cdecl;external 'c';
```

Процедуры `localtime` и `gmtime` конвертируют значение, полученное в результате вызова `GetTimeOfDay`, в структуру `tm` и возвращают локальное время и стандартное гринвичское время соответственно. Например, программа

```
(* Программа tm - демонстрация структуры tm *)
uses linux,stdio;

var
  t:longint;
  tp:ptm;
begin
  (* Получить системное время *)
  t:=gettimeofday;

  (* Получить структуру tm *)
  tp := localtime (t);
  printf ('Время %02d:%02d:%02d'#$a, [tp^.tm_hour, tp^.tm_min,
                                      tp^.tm_sec]);
  halt (0);
end.
```

выводит сообщение

```
Время 1:13:23
```

Процедура `asctime` преобразует структуру `tm` в строку в формате вывода процедуры `ctime`, а процедура `mktime` преобразует структуру `tm` в соответствующее ей системное время (число секунд). Процедура `difftime` возвращает разность между двумя значениями времени в секундах.

Другие функции для работы со временем:

`EpochToLocal`	Преобразует время от начала эпохи в локальные дату и время
`GetDate`	Возвращает системную дату
`GetDateTime`	Возвращает системные дату и время
`GetEpochTime`	То же, что и `GetTimeOfDay`, но с учетом временной зоны
`GetLocalTimezone`	Возвращает системную временную зону
`GetTime`	Возвращает системное время
`GetTimezoneFile`	Возвращает имя файла временной зоны
`ReadTimezoneFile`	Возвращает содержимое файла временной зоны

Пример использования `GetEpochTime`:

```
Uses linux;

begin
  Write ('Secs past the start of the Epoch (00:00 1/1/1970) : ');
  Writeln (GetEpochTime);
end.
```

Пример использования `EpochToLocal`:

```
Uses linux;

Var Year,month,day,hour,minute,seconds : Word;
```

```
begin
  EpochToLocal (GetEpochTime,Year,month,day,hour,minute,seconds);
  Writeln ('Current date : ',Day:2,'/',Month:2,'/',Year:4);
  Writeln ('Current time : ',Hour:2,':',minute:2,':',seconds:2);
end.
```

Упражнение 12.3. *Напишите свою версию процедуры* `asctime`.

Упражнение 12.4. *Напишите функцию* `weekday`, *возвращающую 1 для рабочих дней и 0 – для выходных. Напишите обратную функцию* `weekend`. *Эти функции также должны предоставлять возможность задания выходных дней.*

Упражнение 12.5. *Напишите процедуры, возвращающие разность между двумя значениями, полученными при вызове* `gettimeofday` *в днях, месяцах, годах и секундах. Не забывайте про високосные годы!*

12.5. Работа со строками и символами

Библиотеки UNIX предоставляют богатые возможности для работы со строчными или символьными данными. Они достаточно полезны и поэтому также заслуживают упоминания.

12.5.1. Семейство процедур strings

Некоторые из этих хорошо известных процедур уже были показаны в предыдущих главах книги, например, процедуры `strcat` и `strcopy`. Ниже следует подробный список процедур этого семейства.

Описание

```
uses strings;

Function StrCat(s1, s2:PChar):PChar;
Function StrLCat(s1, S2:PChar; length:Longint):PChar;

Function StrComp(S1, S2:PChar):Longint;
Function StrLComp(S1,S2:PChar; length:Longint):Longint;
Function StrIComp(S1,S2:PChar):Longint;
Function StrLIComp(S1,S2:PChar; length:Longint):Longint;

Function StrCopy(s1, s2:PChar):PChar;
Function StrLCopy(s1, S2:PChar; length:Longint):PChar;
Function StrMove(s1, S2:PChar; MaxLen:Longint):PChar;
Function StrNew(s1:PChar):PChar;

Function StrLen(s1:PChar):Longint;

Function StrScan(s1: PChar; C:Char):PChar;
Function StrRScan(s1:PChar; C:Char):PChar;
Function StrPos(S1,S2:PChar):PChar;


uses stdio;

function strpbrk(const s1, s2:pchar):pchar;
function strspn(const s1, s2:pchar):longint;
function strcspn(const s1, s2:pchar):longint;

function strtok(s1:pchar; const s2:pchar):pchar; (* Первый вызов *)
function strtok(nil; const s2:pchar):pchar; (* Последующие вызовы *)
```

Процедура `strcat` присоединяет строку `s2` к концу строки `s1`. Процедура `strlcat` делает то же самое, но добавляет при этом не более `length` символов. Обе процедуры

возвращают указатель на строку s1. Пример использования процедуры strcat:

```
strcat(fileprefix, '.dat');
```

Если переменная fileprefix первоначально содержала строку file, то после выполнения процедуры она будет содержать строку file.dat. Следует отметить, что процедура strcat изменяет строку, на которую указывает ее первый аргумент. Таким же свойством обладают и процедуры strlcat, strcopy, strlcopy и strtok. Программист должен убедиться, что размер первого аргумента этих процедур достаточно велик, чтобы вместить результат выполнения соответствующей операции.

Процедура strcomp сравнивает две строки s1 и s2. Если возвращаемое значение положительно, то это означает, что строка s1 лексикографически «больше», чем строка s2, в соответствии с порядком расположения символов в наборе символов ASCII. Если оно отрицательно, то это означает, что строка s1 «меньше», чем строка s2. Если же возвращаемое значение равно нулю, то строки совпадают. Процедура strlcomp аналогична процедуре strcomp, но сравнивает только первые length символов. Процедуры stricomp и strlicomp выполняют те же проверки, но игнорируют регистр символов. Пример использования процедуры strcomp:

```
if strcmop (token, 'print') = 0 then
begin
  (* Обработать ключевое слово print *)
end;
```

Процедура strcopy подобна процедуре strcat. Она копирует содержимое строки s2 в строку s1. Процедура strlcopy копирует в точности length символов, отбрасывая ненужные символы (что означает, что строка s1 может не заканчиваться нулевым символом) или записывая нулевые символы вместо недостающих символов строки s2. Процедура strnew возвращает указатель на копию строки s1. Возвращаемый процедурой strnew указатель может быть передан функции free, так как память выделяется при помощи функции malloc.

Процедура strlen просто возвращает длину строки s1. Другими словами, она возвращает число символов в строке s1 до нулевого символа, обозначающего ее конец.

Процедура strscan возвращает указатель на первое вхождение символа c (который передается в параметре типа char) в строке s1 или nil, если символ в строке не обнаружен. Процедура strpos возвращает адрес первого вхождения подстроки s2 в строке s1 (или nil, если подстрока не найдена). Процедура strrscan точно так же ищет последнее вхождение символа c. В главе 4 процедура strrscan была использована для удаления пути из полного маршрутного имени файла:

```
(* Выделяем имя файла из полного маршрутного имени *)
filename := strscan(pathname, '/');
```

Процедура strpbrk возвращает указатель на первое вхождение в строке s1 любого символа из строки s2 или нулевой указатель, если таких вхождений нет.

Процедура strspn возвращает длину префикса строки s1, который состоит только из символов, содержащихся в строке s2. Процедура strcspn возвращает длину префикса строки s1, который не содержит ни одного символа из строки s2.

И, наконец, процедура strtok позволяет программе разбить строку s1 на лексические единицы (лексемы). В этом случае строка s2 содержит символы, которые могут разделять лексемы (например, пробелы, символы табуляции и перевода строки). Во время первого вызова, для которого первый аргумент равен s1, указатель на строку s1 запоминается, и возвращается указатель на первую лексему. Последующие вызовы, для которых первый аргумент задается равным nil, возвращают следующие лексемы из строки s1. Когда лексем в строке больше не останется, возвращается нулевой указатель.

12.5.2. Преобразование строк в числовые значения

Стандарт ANSI C определяет два набора функций для преобразования строк в числовые величины:

Описание

```
uses sysutils;

(* Преобразование строки в целое число *)
Function StrToInt(const s:string):integer;
Function StrToIntDef(const S:string; Default:integer):integer;

function strtol(const str:pchar; endptr:ppchar; base:longint):longint;
function atoi(const str:pchar):longint;
function atol(const str:pchar):longint;

(* Преобразование строки в вещественное число *)
function strtod(const str:pchar; endptr:ppchar):double;
function atof(const str:pchar):double;
```

Функции `StrToInt` и `StrToIntDef` преобразуют строку в целое число. Если строка содержит нецифровые символы или имеет неверный формат, `StrToInt` генерирует исключение `EConvertError`, а `StrToIntDef` возвращает значение, определенное параметром `Default`.

Функции `atoi`, `atol` и `atof` преобразуют строку числовой константы в число формата `longint` и `double` соответственно. В настоящее время эти функции устарели и заменены функциями `strtol` и `strtod`.

Функции `strtod` и `strtol` намного более надежны. Обе функции удаляют все пробельные символы из начала строки `str` и все нераспознанные символы в конце строки (включая нулевой символ) и записывают указатель на полученную строку в переменную `endptr`, если значение аргумента `endptr` не равно нулю. Последний параметр функции `strtol` – `base` может иметь любое значение между 0 и 36, при этом строка конвертируется в целое число с основанием `base`.

12.5.3. Проверка и преобразование символов

В ОС UNIX существуют два полезных набора макросов и функций для работы с символами, которые определены в файле `stdio`. Первый набор, называемый семейством `ctype`, предназначен для проверки одиночных символов. Эти макросы-предикаты возвращают `true` (*истинно*), если условие выполняется, и `false` (*ложно*) – в противном случае. Например, макрос `isalpha` проверяет, является ли символ буквой, то есть, лежит ли он в диапазонах `a-z` или `A-Z`:

```
uses stdio;
var
  c:integer;
.
.
.
(* Макрос 'isalpha' из набора ctype *)
if isalpha (c) then
begin
  (* Обрабатываем букву *)
end
else
  warn('Символ не является буквой');
```

Обратите внимание на то, что аргумент `c` имеет тип `integer`. Ниже следует полный список макросов `ctype`:

`isalpha(c)`	Является ли `c` буквой?
`isupper(c)`	Является ли `c` прописной буквой?
`islower(c)`	Является ли `c` строчной буквой?
`isdigit(c)`	Является ли `c` цифрой (0–9)?
`isxdigit(c)`	Является ли `c` шестнадцатеричной цифрой?
`isalnum(c)`	Является ли `c` буквой или цифрой?
`isspace(c)`	Является ли `c` пробельным символом; то есть одним из символов: пробел, табуляция, возврат каретки, перевод строки, перевод страницы или вертикальная табуляция?
`ispunct(c)`	Является ли `c` знаком препинания?
`isprint(c)`	Является ли `c` печатаемым знаком? Для набора символов ASCII это означает любой символ в диапазоне от пробела (040) до тильды (~ или 0176)
`isgraph(c)`	Является ли с печатаемым знаком, но не пробелом?
`iscntrl(c)`	Является ли с управляющим символом? В качестве управляющего символа рассматривается символ удаления ASCII и все символы со значением меньше 040
`isascii(c)`	Является ли `c` символом ASCII? Обратите внимание, что для любого целочисленного значения, кроме значения символа `EOF`, определенного в файле `stdio`, которое передается другим процедурам семейства `ctype`, это условие должно выполняться. (Включение символа `EOF` позволяет использовать макрокоманды из семейства `ctype` в процедурах типа `getc`)

Другой набор утилит для работы с символами предназначен для простых преобразований символов. Например, функция `tolower` переводит прописную букву в строчную.

```
uses stdio;
var
  newc, c:integer;
.
.
.
(* Перевод прописной буквы в строчную *)
(* Например, перевод буквы 'A' в 'a' *)
newc := tolower(c);
```

Если `c` является прописной буквой, то она преобразуется в строчную, иначе возвращается исходный символ. Другие функции и макросы (которые могут быть объединены под заголовком `conv` в справочном руководстве системы) включают в себя:

`toupper(c)`	Функция, преобразующая букву `c` в прописную, если она является строчной
`toascii(c)`	Макрос, преобразующий целое число в символ ASCII за счет отбрасывания лишних битов
`_toupper(c)`	Быстрая версия `toupper`, выполненная в виде макроса и не выполняющая проверки того, является ли символ строчной буквой
`_tolower(c)`	Быстрая версия `tolower`, выполненная в виде макроса, аналогичная макросу `_toupper`

12.6. Дополнительные средства

В книге основное внимание было уделено вызовам и процедурам, которые позволяют дать основы системного программирования для ОС UNIX, и читатель к этому времени уже должен располагать средствами для решения большого числа задач. Конечно же, ОС UNIX

является системой с богатыми возможностями, поэтому существует еще много процедур, обеспечивающих выполнение специальных функций. Этот последний раздел просто привлекает внимание к некоторым из них. Попытайтесь найти недостающую информацию в справочном руководстве системы.

12.6.1. Дополнение о сокетах

Сокеты являются мощным и популярным способом взаимодействия между процессами разных компьютеров (в главе 10 они достаточно подробно были описаны). При необходимости получить больше информации по этой теме следует обратиться к специальной литературе. В качестве первого шага можно изучить следующие процедуры, которые позволяют получить информацию о сетевом окружении:[1]

```
gethostent      getservbyname
gethostbyaddr   getservbyport
gethostbyname   getservent
```

12.6.2. Потоки управления

Потоки управления, или *нити* (threads) являются облегченной версией процессов – поддерживающие их системы позволяют выполнять одновременно несколько таких потоков в рамках одного процесса, при этом все потоки могут работать с данными процесса. Их применение может быть очень ценным для повышения производительности и интерактивности определенных классов задач, но техника программирования многопотоковых программ является довольно сложной. Многие версии UNIX поддерживали различные подходы к организации потоков выполнения, но теперь выработана стандартная модель (*POSIX threads*), включенная в стандарт POSIX и пятую версию спецификации XSI. Интерфейс функций работы с потоками управления может быть описан в справочном руководстве системы под заголовком `pthread_`.

Для создания потока может использоваться вызов `Clone`:

Описание

```
uses linux;

TCloneFunc=function(args:pointer):longint;cdecl;
Clone(func:TCloneFunc;sp:pointer;flags:longint;args:pointer):longint;
```

`Clone`, подобно `Fork`, создает дочерний процесс, являющийся копией родительского. Однако, в отличие от `Fork`, процесс-потомок совместно с родителем использует некоторые части контекста, что делает его подходящим для реализации потоков: множества экземпляров программы, разделяющих общую память.

При создании потомка запускается функция `Func`, которой передаются параметры `Args`. Возвращаемым значением `Func` является код завершения потомка.

Указатель `sp` хранит адрес памяти, зарезервированной под стек дочернего процесса.

Параметр `Flags` определяет поведение вызова `Clone`. Младший байт `Flags` содержит номер сигнала, который будет послан родителю при завершении потомка. Он может быть объединен с помощью побитового ИЛИ со следующими константами:

`CLONE_VM`	Родитель и потомок разделяют память, включая отображенную вызовом `mmap`.
`CLONE_FS`	Родитель и потомок разделяют параметры файловой системы; вызовы `chroot`, `chdir` и `umask` действуют на оба процесса.
`CLONE_FILES`	Таблица дескрипторов файлов разделяется родителем и потомком.
`CLONE_SIGHAND`	Таблица обработчиков сигналов разделяется родителем и потомком, однако маски сигналов различаются.
`CLONE_PID`	Родитель и потомок имеют одинаковый `pid`.

[1] Полезно также знать о процедуре `setsockopt`, управляющей параметрами соединения.

`Clone` возвращает идентификатор потомка или -1 в случае ошибки, устанавливая значение `linuxerror` в `Sys_EAGAIN` (слишком много процессов) и `Sys_ENOMEM` (недостаточно памяти для создания дочернего процесса).

Пример использования `Clone`:

```
uses
  Linux, Errors, crt;

const
  Ready : Boolean = false;
  aChar : Char    = 'a';

function CloneProc( Arg: Pointer ): LongInt; Cdecl;
begin
  WriteLn('Hello from the clone ',PChar(Arg));
  repeat
    Write(aChar);
    Select(0,0,0,0,600);
  until Ready;
  WriteLn( 'Clone finished.');
  CloneProc := 1;
end;

var
  PID : LongInt;

procedure MainProc;
begin
  WriteLn('cloned process PID: ', PID );
  WriteLn('Press <ESC> to kill ... ' );
  repeat
    Write('.');
    Select(0,0,0,0,300);
    if KeyPressed then
      case ReadKey of
        #27: Ready := true;
        'a': aChar := 'A';
        'A': aChar := 'a';
        'b': aChar := 'b';
        'B': aChar := 'B';
      end;
  until Ready;
  WriteLn('Ready.');
end;

const
  StackSze = 16384;
  theFlags = CLONE_VM+CLONE_FS+CLONE_FILES+CLONE_SIGHAND;
  aMsg     : PChar = 'Oops !';

var
  theStack : Pointer;
  ExitStat : LongInt;

begin
  GetMem(theStack,StackSze);
  PID := Clone(@CloneProc,
               Pointer( LongInt(theStack)+StackSze),
```

```
              theFlags,
              aMsg);
  if PID < 0 then
    WriteLn('Error : ', LinuxError, ' when cloning.')
  else
    begin
    MainProc;
    case WaitPID(0,@ExitStat,Wait_Untraced or wait_clone) of
      -1: WriteLn('error:',LinuxError,'; ',StrError(LinuxError));
       0: WriteLn('error:',LinuxError,'; ',StrError(LinuxError));
    else
      WriteLn('Clone exited with: ',ExitStat shr 8);
    end;
    end;
  FreeMem( theStack, StackSze );
end.
```

12.6.3. Расширения режима реального времени

В последнее время в стандарт POSIX были введены определенные расширения для режима реального времени. Как и потоки управления, это специализированная и сложная область, и часто ядро UNIX обладает достаточными возможностями для решения большинства задач реального времени. Специфические требования для реализации работы в режиме реального времени включают в себя:

- организацию очереди сигналов и дополнительные средства для работы с сигналами (см. `sigwaitinfo`, `sigtimedwait`, `sigqueue`);
- управление приоритетом и политикой планирования процессов и потоков (см. процедуры, начинающиеся с `sched_`);
- дополнительные средства для работы с таймерами, асинхронным и синхронным вводом/выводом;
- альтернативы рассмотренным процедурам передачи сообщений, интерфейсам семафоров и разделяемой памяти (попробуйте найти процедуры, начинающиеся с `mq_`, `sem_` и `shm_`).

12.6.4. Получение параметров локальной системы

В книге были рассмотрены некоторые процедуры, сообщающие системные ограничения (например, `pathconf`). Есть также и другие процедуры, служащие для той же цели:

`sysconf`	Обеспечивает доступ к конфигурационным параметрам, находящимся в файлах `<limits.h>` и `<unistd.h>`
`SysInfo`	Возвращает информацию о системе
`GetHostName`	Возвращает имя локального компьютера
`uname`	Возвращает указатель на структуру `utsname`, содержащую название операционной системы, имя узла, которое может использоваться системой в сети для установления связи, а также номер версии системы UNIX

`getpwent`	Это семейство процедур обеспечивает доступ к данным из файла паролей `/etc/passwd`. Все следующие вызовы возвращают указатель на структуру `passwd`, определенную в файле `<pwd.h>`:
	`getpwnam(const char *username);`
	`getpwuid(uid_t uid);`
	`getpwent(void);`

`getgrent`	Это семейство процедур связано с доступом к файлу описания групп `/etc/group`
`getrlimit`	Обеспечивает доступ к предельным значениям системных

ресурсов, таких как память или доступное дисковое пространство

`getlogin` `cuserid` Получает имя пользователя для текущего процесса

Пример использования `SysInfo`:

```
Uses Linux;

Function Mb(L : Longint) : longint;

begin
  Mb:=L div (1024*1024);
end;

Var Info : TSysInfo;
    D,M,Secs,H : longint;

begin
  If Not SysInfo(Info) then
    Halt(1);
  With Info do
    begin
    D:=Uptime div (3600*24);
    UpTime:=UpTime mod (3600*24);
    h:=uptime div 3600;
    uptime:=uptime mod 3600;
    m:=uptime div 60;
    secs:=uptime mod 60;
    Writeln('Uptime : ',d,'days, ',h,' hours, ',m,' min, ',secs,' s.');
    Writeln('Loads  : ',Loads[1],'/',Loads[2],'/',Loads[3]);
    Writeln('Total Ram  : ',Mb(totalram),'Mb.');
    Writeln('Free Ram   : ',Mb(freeram),'Mb.');
    Writeln('Shared Ram : ',Mb(sharedram),'Mb.');
    Writeln('Buffer Ram : ',Mb(bufferram),'Mb.');
    Writeln('Total Swap : ',Mb(totalswap),'Mb.');
    Writeln('Free Swap  : ',Mb(freeswap),'Mb.');
    end;
end.
```

Пример использования `uname`:

```
Uses linux;

Var UN : utsname;

begin
  if Uname (UN) then
    With UN do
      begin
      Writeln ('Name       : ',pchar(@sysname[0]));
      Writeln ('Nodename   : ',pchar(@Nodename[0]));
      Writeln ('release    : ',pchar(@Release[0]));
      Writeln ('Version    : ',pchar(@Version[0]));
      Writeln ('Machine    : ',pchar(@Machine[0]));
      Writeln ('Domainname : ',pchar(@domainname[0]));
      end;
end.
```

12.6.5. Интернационализация

Многие версии ОС UNIX могут поддерживать различные национальные среды – позволяют учитывать языковые среды, менять порядок сортировки строк, задавать символы

денежных единиц, форматы чисел и т.д. Попробуйте найти процедуры `setlocale` или `catopen`. Справочное руководство системы может содержать дополнительные сведения по этой теме под заголовком `environ`.

12.6.6. Математические функции

ОС UNIX предоставляет обширную библиотеку математических функций для программирования научных и технических приложений. Для применения некоторых из этих функций необходимо подключать файл `math`, который включает определения функций, некоторых важных констант (таких как `e` или π) и структур, относящихся к обработке ошибок. Для использования большинства функций, которых коснемся ниже, потребуется также подключить математическую библиотеку:

```
uses math;
```

12.6.7. Работа с портами ввода вывода

ОС UNIX предоставляет набор низкоуровневых функций для чтения и записи в порты ввода-вывода:

`IOperm`	Устанавливает права для доступа к порту
`ReadPort`	Читает данные и порта
`ReadPortB`	Читает 1 байт из порта
`ReadPortL`	Читает 4 байта из порта
`ReadPortW`	Читает 2 байта из порта
`WritePort`	Пишет данные в порт
`WritePortB`	Пишет 1 байт в порт
`WritePortL`	Пишет 4 байта в порт
`WritePortW`	Пишет 4 байта в порт

Глава 13. Задачи с решениями

13.1. Введение

В последней главе будут рассмотрены несколько блоков задач, которые соответствуют тематике всех предыдущих глав, с примерами решения.

13.2. Обработка текста

Упражнение 13.1. *Напишите программу, отсекающую n пробелов в начале каждой строки (или n первых любых символов). Учтите, что в файле могут быть строки короче n (например, пустые строки).*

```
Program Tabs;
uses sysutils,linux;
var
  f1,f2:Text;
  TmpS:string;
  n,Code:Integer;
begin
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
  Reset(f1);
  Rewrite(f2);
  if ParamCount=3 then val(ParamStr(3),n,Code)
  else n:=0;
  While Not(eof(f1)) do
    begin
      Readln(f1,TmpS);
      Writeln(f2,Copy(TmpS,n,Ord(TmpS[0])-n));
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.2. *Составьте программу, центрирующую строки файла относительно середины экрана, т.е. добавляющую в начало строки такое количество пробелов, чтобы середина строки печаталась в 40-ой позиции (считаем, что обычный экран имеет ширину 80 символов).*

```
Program Center;
uses crt;
var
  s:string;
  f:text;
procedure writecenter(s:string);
begin
  if ord(s[0])<80 then
  begin
    GotoXY(wherex+(80-ord(s[0]))div 2,wherey);
    writeln(s);
  end
  else writeln(s);
end;

begin
  if paramcount<1 then
    begin
      Writeln('error');
      Halt(1);
```

```
    end;
  Assign(f,paramstr(1));
  Reset(f);
  While not(eof(f)) do
    begin
      Readln(f,s);
      writecenter(s);
    end;
  Close(f);
end.
```

Упражнение 13.3. *Напишите программу, переносящую слишком длинные строки. Слова разбивать нельзя (неумещающееся слово следует перенести целиком). Ширину строки считать равной 60.*

```
Program Tabs;
var
  f1,f2:Text;
  TmpS,StrBuf:string;
  n,Code:Integer;
const
  step=60;
begin
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
  Reset(f1);
  Rewrite(f2);
  StrBuf:='';
  While Not(eof(f1)) do
    begin
      Readln(f1,TmpS);
      TmpS:=StrBuf+TmpS;
      n:=step;
      if ord(Tmps[0])>step then
        begin
          While  (not(TmpS[n]in  [' ',',','(',')',';','.']))do  if  n>=1  then
Dec(n) else Exit;
          While  (TmpS[n]in  [' ',',','(',')',';','.'])do if n>=1 then Dec(n)
else Exit;;
          if n<1 then n:=Pos(',',TmpS);
          Writeln(f2,Copy(TmpS,1,n));
          StrBuf:=Copy(TmpS,n+1,Ord(TmpS[0])-n);
          while Ord(strbuf[0])>step do
            begin
              if Pos(',',Strbuf)>Pos(' ',StrBuf) then
                begin
                  Writeln(f2,Copy(StrBuf,1,Pos(',',StrBuf)));
                  Delete(StrBuf,1,Pos(',',StrBuf));
                end
                else
                begin
                  Writeln(f2,Copy(StrBuf,1,Pos(' ',StrBuf)));
                  Delete(StrBuf,1,Pos(' ',StrBuf));
                end

            end;
        end
      else Writeln(f2,TmpS);
    end;
```

```
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.4. *Напишите программу, разбивающую файл на два по вертикали: в первый файл попадает левая половина исходного файла, во второй – правая. Ширину колонки задавайте из аргументов командной строки. Если же аргумент не указан – 40 позиций.*

```
Program InTwo;
var
  f1,f2,f:Text;
  WidthCol:Byte;
  Code:Integer;
  s:string;
begin
  if ParamCount<3 then
    begin
      Writeln('Wrong parameters');
      Writeln('Format: ./2 <file1> <file2> <file3> n');
      Writeln('<file1> - input file');
      Writeln('<file2> - first output file');
      Writeln('<file3> - first output file');
      Writeln('n - number of column in first file');
      Halt(1);
    end;
  if ParamCount<4 then WidthCol:=40
  else Val(Paramstr(4),WidthCol,Code);
  assign(f,ParamStr(1));
  reset(f);
  assign(f1,ParamStr(2));
  rewrite(f1);
  assign(f2,ParamStr(3));
  rewrite(f2);
    While Not(Eof(f)) do
      begin
        Readln(f,s);
        if Ord(s[0])>WidthCol then
          begin
            Writeln(f1,Copy(s,1,WidthCol));
            Writeln(f2,Copy(s,WidthCol+1,Ord(s[0])-WidthCol));
          end
        else
          begin
            Writeln(f1,s);
            Writeln(f2);
          end;
      end;
  close(f2);
  close(f1);
  close(f);
end.
```

Упражнение 13.5. *Напишите функцию, сжимающую подряд идущие пробелы в табуляции.*

```
Program Tabs;
var
  f1,f2:Text;
```

```
  sin,sou:string;
  i:Byte;
  CheckSpaces:Boolean;
  SetTab:Boolean;
begin
  if ParamCount<2 then
   begin
      Writeln('Wrong parameters');
      Writeln('Format: ./task <inputfile> <outputfile>');
      Halt(1);
   end;
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
  Reset(f1);
  Rewrite(f2);
  While Not(eof(f1)) do
    begin
      Readln(f1,sin);
      CheckSpaces:=False;
      SetTab:=False;
      sou:='';
      For i:=1 to Ord(sin[0]) do
        if      (sin[i]='      ')and(not      CheckSpaces)      then      begin
CheckSpaces:=True;sou:=sou+' ';end
        else  if  not(sin[i]='  ')then  begin  sou:=sou+sin[i];  CheckSpaces:=
False;SetTab:=False;end
        else   if   (sin[i]='   ')and(CheckSpaces)and(not   SetTab)   then   begin
sou[ord(sou[0])]:=#9;SetTab:=True;end
        else if (not (sin[i]=' '))and(CheckSpaces)and (SetTab) then
        begin
          sou:=sou+sin[i];
          SetTab:=False;
      end;
      Writeln(f2,sou);
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.6. *Напишите программу сортировки строк в алфавитном порядке.*

```
uses strings;


type
  achar=array [0..255] of char;
  mas=array [1..1] of achar;

var s:achar;
    t:text;
    a:^mas;
    n,i,j,k:byte;
begin
  if ParamCount<>2 then
  begin
    Writeln('Error!!!');
    Writeln('Format: ./task <sourcefile> <destfile>');
    Halt(1);
  end;
```

```
  s:=ParamStr(1);
  assign(t,s);
  reset(t);
  k:=0;

  while not eof(t) do
  begin
    readln(t,s);
    inc(k)
  end;

  getmem(a,k*256);
  close(t);
  reset(t);
  for i:=1 to k do
    readln(t,a^[i]);
  close(t);
  for i:=1 to k-1 do
  begin
    for j:=i+1 to k do
      if stricomp(a^[i],a^[j])>0 then
      begin
        strcopy(s,a^[j]);
        strcopy(a^[j],a^[i]);
        strcopy(a^[i],s);
      end;
  end;
  s:=ParamStr(2);
  assign(t,s);
  rewrite(t);
  for i:=1 to k do
    writeln(t,a^[i]);
  freemem(a,k*256);
  close(t);
end.
```

Упражнение 13.7. *Напишите функцию, расширяющую табуляции в подряд идущие пробелы.*

```
Program TabsProg;
function Tabs(s:String):String;
begin
  While (Pos(#9,s)>0) do
  s:=Copy(s,1,Pos(#9,s)-1)+'      '+Copy(s,Pos(#9,s)+1,Ord(s[0])-Pos(#9,s));
  Tabs:=s;
end;
var
  f1,f2:Text;
  sin:string;
begin
  if ParamCount<2 then
   begin
      Writeln('Wrong parameters');
      Writeln('Format: ./task <inputfile> <outputfile>');
      Halt(1);
   end;
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
  Reset(f1);
```

```
  Rewrite(f2);
  While Not(eof(f1)) do
    begin
      Readln(f1,sin);
      Writeln(f2,Tabs(sin));
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.8. *Составьте программу, укорачивающую строки исходного файла до заданной величины и помещающую результат в указанный файл.*

```
Program Tabs;
var
  f1,f2:Text;
  TmpS:string;
  n,Code:Integer;
begin
  if ParamCount<2 then
   begin
      Writeln('Wrong parameters');
      Writeln('Format: ./task <inputfile> <outputfile> <column>');
      Writeln('<column> - number of column');
      Halt(1);
   end;
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
  Reset(f1);
  Rewrite(f2);
  if ParamCount=3 then val(ParamStr(3),n,Code)
  else n:=40;
  While Not(eof(f1)) do
    begin
      Readln(f1,TmpS);
      Writeln(f2,Copy(TmpS,1,n));
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.9. *Разработайте программу, укорачивающую строки входного файла до 60 символов без обрубания слов.*

```
Program Tabs;
var
  f1,f2:Text;
  TmpS,StrBuf:string;
  n,Code:Integer;
const
  step=60;
begin
  if ParamCount<2 then
   begin
      Writeln('Wrong parameters');
      Writeln('Format: ./task <inputfile> <outputfile>');
      Halt(1);
   end;
  Assign(f1,Paramstr(1));
```

```
  Assign(f2,Paramstr(2));
  Reset(f1);
  Rewrite(f2);
  StrBuf:='';
  While Not(eof(f1)) do
    begin
      Readln(f1,TmpS);
      TmpS:=StrBuf+TmpS;
      n:=step;
      if ord(Tmps[0])>step then
        begin
          While  (not(TmpS[n]in  [' ',',','(',')',';','.']))do  if  n>=1  then
Dec(n) else Exit;
          While  (TmpS[n]in  [' ',',','(',')',';','.'])do  if  n>=1  then  Dec(n)
else Exit;;
          if n<1 then n:=Pos(',',TmpS);
          Writeln(f2,Copy(TmpS,1,n));
          StrBuf:=Copy(TmpS,n+1,Ord(TmpS[0])-n);
          while Ord(strbuf[0])>step do
            begin
              if Pos(',',Strbuf)>Pos(' ',StrBuf) then
                begin
                  Writeln(f2,Copy(StrBuf,1,Pos(',',StrBuf)));
                  Delete(StrBuf,1,Pos(',',StrBuf));
                end
              else
                begin
                  Writeln(f2,Copy(StrBuf,1,Pos(' ',StrBuf)));
                  Delete(StrBuf,1,Pos(' ',StrBuf));
                end

            end;
        end
      else Writeln(f2,TmpS);
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.10. *Разработайте программу, заполняющую промежутки между словами строки дополнительными пробелами таким образом, чтобы длина строки была равна 60 символам.*

```
Program Tabs;
var
  f1,f2:Text;
  sin:string;
  i:Byte;
  CheckSpaces:Boolean;
  SetTab:Boolean;
begin
  if ParamCount<2 then
   begin
      Writeln('Wrong parameters');
      Writeln('Format: ./task <inputfile> <outputfile>');
      Halt(1);
   end;
  Assign(f1,Paramstr(1));
  Assign(f2,Paramstr(2));
```

```
  Reset(f1);
  Rewrite(f2);
  While Not(eof(f1)) do
    begin
      Readln(f1,sin);
      i:=1;
      While (ord(sin[0])<60) do
        begin
          if Pos(' ',sin)=0 then sin:=' '+sin+' ';
        if sin[i]=' ' then
          begin
             sin:=copy(sin,1,i)+' '+copy(sin,i+1,ord (sin[0])-i);
             Inc(i);
          end;
        Inc(i);
        if i>Ord(sin[0]) then i:=1;
      end;
      Writeln(sin);
      Writeln(f2,sin);
    end;
  Close(f1);
  Close(f2);
end.
```

Упражнение 13.11. *Отсортируйте массив строк по лексикографическому убыванию, игнорируя различия между строчными и прописными буквами.*

```
uses strings;


type
  achar=array [0..255] of char;
  mas=array [1..1] of achar;

var s:achar;
    t:text;
    a:^mas;
    n,i,j,k:byte;
begin
  s:='SRT4.txt';
  assign(t,s);
  reset(t);
  k:=0;

  while not eof(t) do
  begin
    readln(t,s);
    inc(k)
  end;

  getmem(a,k*256);
  close(t);
  reset(t);
  for i:=1 to k do
    readln(t,a^[i]);
  close(t);
  for i:=1 to k-1 do
  begin
    for j:=i+1 to k do
```

```
      if stricomp(a^[j],a^[i])>0 then
      begin
        strcopy(s,a^[j]);
        strcopy(a^[j],a^[i]);
        strcopy(a^[i],s);
      end;
    end;
  for i:=1 to k do
    writeln(a^[i]);
  freemem(a,k*256);
end.
```

Упражнение 13.12. *Простейший строковый редактор. Программа вызывается «*`tsed s/old/new/ имя_файла`*», читая указанный текстовый файл и сканируя каждую строку на первое вхождение подстроки* `old`*. При совпадении выполняется замена* `old` *на* `new`*. Отредактированная строка пишется на стандартный вывод.*

```
var
  s,myold,mynew,s1:string;
  n,l:byte;
  t:text;
begin
 if paramcount<>2 then begin
                        writeln('tsed s/old/new/ имя_файла');
                        exit
                       end;
 s:=paramstr(1);
 delete(s,1,2);
 n:=pos('/',s);
 myold:=copy(s,1,n-1);

 l:=length(myold);
 delete(s,1,n);
 n:=pos('/',s);
 mynew:=copy(s,1,n-1);

 s:=paramstr(2);
 assign(t,s);
 reset(t);
 while not eof(t) do
  begin
    n:=0;
    readln(t,s);
    n:=pos(myold,s);
    if n<>0 then
             begin
              s1:=copy(s,1,n-1)+mynew+copy(s,n+l,length(s));
              writeln(s1);
           end else writeln(s);
  end;
 close(t);
end.
```

Упражнение 13.13. *Составьте аналог команды* `cut`*.*

```
var
  s:string;
  n:byte;
  er:integer;
```

```
  t:text;
begin
 if paramcount<>2 then begin
                        writeln('mycut n имя_файла');
                        exit
                       end;
 s:=paramstr(1);
 val(s,n,er);
 if er<>0 then begin
                writeln('error n ');
                exit
               end;

 s:=paramstr(2);
 assign(t,s);
 reset(t);
 while not eof(t) do
  begin
    readln(t,s);
    if n>length(s) then writeln('')
             else begin
                   s:=copy(s,n,length(s));
                   writeln(s);
                  end;
  end;
 close(t);
end.
```

Упражнение 13.14. *Напишем функцию, которая преобразует строку так, что при ее печати буквы в ней будут подчеркнуты, а цифры – выделены жирно.*

```
function isdigit(c:char):boolean;
begin
  isdigit:=c in ['0'..'9'];
end;

function isalpha(c:char):boolean;
begin
  isalpha:=(c in ['a'..'z']) or (c in ['A'..'Z']) or (c in ['a'..'я']) or (c
in ['A'..'Я']);
end;


function convertor(s:string):string;
var
  i:integer;
const
  res:string='';
begin
  for i:=1 to byte(s[0]) do
    if isdigit(s[i]) then
      res:=res+s[i]+#8+s[i]
    else
      if isalpha(s[i]) then
        res:=res+s[i]+#8+'_'
      else
        res:=res+s[i];
  convertor:=res;
end;
```

```
begin
  writeln(convertor('11 сентября 2001 г. - национальный праздник Афганистана
(до 2003 г.)'));
end.
```

Упражнение 13.15. *Составьте функцию* `expand(s1, s2)`*, которая расширяет сокращенные обозначения вида* `a-z` *строки* `s1` *в эквивалентный полный список* `abcd...xyz` *в строке* `s2`*. Допускаются сокращения для строчных и прописных букв и цифр. Учтите случаи типа* `a-b-c`*,* `a-z0-9` *и* `-a-g` *(соглашение состоит в том, что символ "-", стоящий в начале или в конце, воспринимается буквально).*

```
program demoexpand;
function ex(s1:string; var s2:string):boolean;
var
  i:byte;
begin
  s2:='';
  if (((((s1[1]>='0')and(s1[1]<='9'))and((s1[3]>='0')and(s1[3]<='9')))
  or (((s1[1]>='A')and(s1[1]<='Z'))and((s1[3]>='A')and(s1[3]<='Z')))
  or (((s1[1]>='a')and(s1[1]<='z'))and((s1[3]>='a')and(s1[3]<='z')))))
  and (s1[1]<=s1[3])or(s1[1]='-')
  then
    begin
      for i:=Ord(s1[1]) to Ord(s1[3]) do
        s2:=s2+Chr(i);
      if s1[1]='-'then delete(s[1],1,1);
      ex:=true;
    end
  else ex:=false;
end;

function expand(s1:string; var s2:string):boolean;
var
  i:Byte;
  tmpstr:String;
begin
  s2:=s1;
  while (pos('-',s2)>0) do
    if (pos('-',s2)>1)and(pos('-',s2)<Ord(s2[0])) then
      begin
        if ex(copy(s2,pos('-',s2)-1,3),tmpstr) then
          s2:=Copy(s2,1,pos('-',s2)-2)+tmpstr+Copy(s2,pos('-
',s2)+2,Ord(s2[0])-(pos('-',s2)+2))
        else s2[pos('-',s2)]:='~';
      end
    else s2[pos('-',s2)]:='~';
  While (pos('~',s2)>0) do s2[pos('~',s2)]:='-';
end;
var
  m,n:string;
begin
  repeat
  Readln(m);
  if ex(m,n) then expand(m,n);
  if m='exit' then exit;
  if expand(m,m) then writeln(m)
    else writeln('Error');
```

```
  Until False;
end.
```

Упражнение 13.16. *Составьте программу преобразования прописных букв из файла ввода в строчные, используя при этом функцию, в которой необходимо организовать анализ символа (действительно ли это буква). Строчные буквы выдавать без изменения.*

```
Program demoupcase;

function ischar(c:char):boolean;
begin
  if c in ['a'..'z'] then ischar:=true
    else ischar:=false;

end;

function upcase(s:string):string;
var
  i:Byte;
  tmp:string;
begin
  tmp:=s;
  for i:=1 to ord(tmp[0]) do
    if ischar(tmp[i]) then tmp[i]:=Chr(ord(tmp[i])-Ord('a')+Ord('A'));
  upcase:=tmp;
end;

begin
  writeln(upcase('abcderfFFF123***'));
  readln;
end.
```

Упражнение 13.17. *Составьте программу, распечатывающая слова в строках файла в обратном порядке.*

```
program reverse;
var
  F1,F2:Text;
  tmpstr:string;
function revstr(s:string):string;
var
  tmp:string;
begin
  tmp:='';
  while (pos(' ',s)>0) do
    begin
      tmp:=' '+copy(s,1,pos(' ',s)-1)+tmp;
      delete(s,1,pos(' ',s));
    end;
  tmp:=s+tmp;
  revstr:=tmp;
end;

begin
  if paramcount<2 then
    begin
      writeln('Error: wrong arguments');
      writeln('format: 10 <fileinp> <fileout>');
      Halt(1);
```

```
    end;
  Assign(F1,paramstr(1));
  Reset(F1);
  Assign(F2,paramstr(2));
  Rewrite(F2);
  while not(eof(F1)) do
    begin
      readln(F1,tmpstr);
      writeln(F2,revstr(tmpstr));
    end;
  close(F2);
  close(F1);
end.
```

Упражнение 13.18. *Составьте программу, преобразующую текст, состоящий только из строчных букв в текст, состоящий из прописных и строчных букв. Первая буква и буква после каждой точки – прописные, остальные – строчные.*

```
program check_gram;
var
  F1,F2:text;
  Temp:string;
  Dot:boolean;
  i:byte;
begin
  Dot:=true;
  if paramcount<>2 then halt(1);
  assign(F1,paramstr(1));
  reset(F1);
  assign(F2,paramstr(2));
  rewrite(F2);
  while not(eof(F1)) do
    begin
      readln(F1,Temp);
      for i:=1 to ord(Temp[0]) do
        begin
          if (Temp[i] in ['A'..'Z','a'..'z'])and Dot then
            begin
              Dot:=false;
              if  Temp[i]  in  ['a'..'z']  then  Temp[i]:=chr(ord(Temp[i])-
ord('a')+ord('A'))
            end
          else if (Temp[i] in ['.','!','?']) then Dot:=true
        end;
      writeln(F2,Temp);
    end;
  close(F2);
  close(F1);
end.
```

Упражнение 13.19. *Напишите программу, которая печатает слова из своего файла ввода, расположенные в порядке убывания частоты их появления. Перед каждым словом напечатайте число частоты его появления.*

```
program sortedlist;
type
  TRec = record
          wrd:string[40];
          num:word;
```

```
        end;
var
  List:array [1..1400] of TRec;
  F:text;
  stroka:string;
  nump:byte;
  i,j:word;
  max:word;
  Temp:TRec;
procedure checkadd(sm:string);
var
  x:word;
  T:boolean;
begin
  T:=true;
  if nump>0 then
  for x:=1 to nump do
    if List[x].wrd=sm then
      begin
        T:=false;
        inc(List[x].num);
      end;
  if T then
    begin
      inc(nump);
      List[nump].wrd:=sm;
      List[nump].num:=1;
    end;
end;
function add(s:string):boolean;
var
  m:byte;
  tmp:string;
begin
  if s<>'' then
  begin
    tmp:=s;
    for m:=1 to ord(tmp[0]) do
      if not (tmp[m] in ['A'..'Z','a'..'z','0'..'9']) then tmp[m]:=' ';
    while (pos(' ',tmp)>0) do
      begin
        if (pos(' ',tmp)>1) then checkadd(copy(tmp,1,pos(' ',tmp)-1));
        delete(tmp,1,pos(' ',tmp));
      end;
    if (pos(' ',tmp)=0)and(ord(tmp[0])>0) then checkadd(tmp);
  end;
  add:=true;
end;
begin
  nump:=0;
  writeln('--------------------');
  if paramcount<1 then halt(1);
  assign(F,paramstr(1));
  reset(F);
  while not(eof(F)) do
    begin
      readln(F,stroka);
      add(stroka);
```

```
    end;
  close(F);
  for j:=1 to nump-1 do
    begin
      max:=j;
      for i:=j+1 to nump do
        if List[i].num>List[max].num then max:=i;
      Temp:=List[max];
      List[max]:=List[j];
      List[j]:=Temp;
    end;
  for i:=1 to nump do
    write(List[i].wrd:11,' - ',List[i].num:4,'  ');
  writeln;
end.
```

Упражнение 13.20. *Напишите программу копирования ввода на вывод таким образом, чтобы из каждой группы последовательно одинаковых строк выводилась только одна строка. Это аналог программы* `uniq` *в системе UNIX.*

```
var
  first,second:string;

begin
  readln(first);
  writeln(first);
  while not eof(input) do
  begin
    readln(second);
    if first<>second then
    begin
      writeln(second);
      first:=second;
    end;
  end;
end.
```

Упражнение 13.21. *Составьте аналог команды* `banner`.

```
Uses Crt;
Type
  TArrNum=Array[1..8] of Byte;
Const
  stroki:Array [1..42]of string[6]=
  (('******'),(' *****'),('***** '),('*     '),(' *    '),
   ('  *   '),('   *  '),('    * '),('     *'),('*    *'),
   (' *  * '),('  **  '),('      '),(' **** '),('*   **'),
   ('*   * '),('*  *  '),('***   '),('   ***'),('    * '),
   (' ***  '),(' *   *'),('  * * '),('   *  '),('*  * *'),
   ('**  **'),('* ** *'),('**   *'),('* *  *'),('   ** '),
   (' ** **'),(' **   '),('**** *'),('    **'),('  *** '),
   ('**    '),('*  ***'),('***  *'),('****  '),(' **  *'),
   ('  ****'),('* *** ')
   );

  bukvi:Array ['!'..'_'] of TArrNum=
   (
   (13,12,14,14,12,12,13,12), {!}
   (26,26,11,13,13,13,13,13), {"}
```

```
(13,11,11,1,11,1,11,11), {#}
(6,14,10,18,19,10,14,6), {$}
(13,28,26,12,30,31,26,4), {%}
(21,16,17,32,32,25,16,34), {&}
(12,12,32,13,13,13,13,13), {'}
(13,7,6,5,5,5,6,7), {(}
(13,6,7,8,8,8,7,6), {)}
(13,13,11,12,1,12,11,13), {*}
(13,12,12,1,1,12,12,13), {+}
(13,13,13,13,13,12,12,32), {,}
(13,13,13,1,13,13,13,13), {-}
(13,13,13,13,13,13,13,9), {.}
(13,9,34,30,12,32,36,4), {/}
(13,14,15,37,27,38,28,14), {0}
(13,12,21,39,12,12,12,14), {1}
(13,14,26,30,12,6,39,1), {2}
(13,35,22,9,30,9,22,35), {3}
(13,24,12,11,3,7,7,35), {4}
(13,1,4,4,1,9,10,14), {5}
(13,35,5,4,3,10,10,14), {6}
(13,2,31,34,34,30,12,12), {7}
(13,14,10,10,14,10,10,14), {8}
(13,35,22,22,41,9,22,35), {9}
(13,13,12,12,13,12,12,13), {:}
(13,13,12,12,13,12,12,32), {;}
(13,30,12,32,36,32,12,30), {<}
(13,13,13,1,1,13,13,13), {=}
(13,32,12,30,34,30,12,32), {>}
(13,35,22,9,8,7,13,7), {?}
(13,14,27,26,28,42,4,3), {@}
(13,12,11,10,1,10,10,10), {A}
(13,3,10,10,3,10,10,3),   {B}
(13,2,4,4,4,4,4,2),       {C}
(13,3,10,10,10,10,10,3),  {D}
(13,1,4,4,3,4,4,1),       {E}
(13,1,4,4,3,4,4,4),       {F}
(13,14,10,4,4,15,10,14),  {G}
(13,10,10,10,1,10,10,10), {H}
(13,21,6,6,6,6,6,21),     {I}
(13,19,20,20,20,16,16,21),{J}
(13,10,16,17,18,17,16,10), {K}
(13,4,4,4,4,4,10,1), {L}
(13,10,26,27,10,10,10,10), {M}
(13,10,28,29,25,30,10,10), {N}
(13,14,10,10,10,10,10,14), {O}
(13,3,10,10,3,4,4,4), {P}
(13,14,10,10,10,10,25,2), {Q}
(13,3,10,10,3,17,16,10), {R}
(13,14,10,5,12,8,10,14), {S}
(13,3,6,6,6,6,6,6), {T}
(13,10,10,10,10,10,10,14), {U}
(13,10,10,10,10,10,11,12), {V}
(13,10,10,10,10,27,27,11), {W}
(13,10,10,11,12,11,10,10), {X}
(13,22,22,22,23,24,24,24), {Y}
(13,1,8,7,6,5,4,1),  {Z}
(13,21,5,5,5,5,5,21), {[}
(13,4,36,32,12,30,34,9), {\}
```

```
   (13,35,8,8,8,8,8,35), {]}
   (6,11,10,13,13,13,13,13),{^}
   (13,13,13,13,13,13,13,1) {_}
   );
procedure showchar(ch:Char);
var
  i:Byte;
  PosX,PosY:Byte;
begin
  PosX:=WhereX;
  PosY:=WhereY;
  if ch<>' ' then
  begin
  For i:=1 to 8 do
    begin
      GotoXY(PosX,PosY+i);
      Write(stroki[bukvi[ch][i]]);
    end;
  end
  else GotoXY(PosX+8,PosY);
  GotoXY(PosX+8,PosY);
end;
procedure showword(s:String);
var
  i:Byte;
begin

  For i:=1 to Ord(s[0]) do
    begin
      if WhereX>90 then GotoXY(1,WhereY+8);
      showchar(upcase(s[i]));
    end;
end;
var
  j:Byte;
begin
  clrscr;
  if paramcount<1 then
    begin
      writeln('Incorrect argument');
      halt(1);
    end;
  for j:=1 to paramcount do
    showword(paramstr(j));
  Readln;
end.
```

Упражнение 13.22. *Напишите программу, печатающую файлы, переданные ей в качестве аргументов, останавливающуюся и ожидающую нажатия клавиши через каждые 20 строк.*

```
uses crt,sysutils,linux;
var
    f:text;p:string;i:integer;s:array [0..1000] of char;
BEGIN
    clrscr;
    writeln('Введите имя файла');
    readln(s);
        {$I-}
```

```
    if not access(pchar(s),f_ok or r_ok) then
    begin
      writeln('Файл '+pchar(s)+' не существует или недоступен для чтения');
      halt(1);
    end;

    assign(f,s);
    reset(f);
    clrscr;
    while not eof(f) do
    begin
      for i:=1 to 20 do
      begin
          readln(f,p);
          writeln(p);
          if eof(f) then break;
      end;
      readkey;
    end;
    close(f);
END.
```

Упражнение 13.23. *Составьте программу вывода строк файла в инверсном отображении, причем порядок символов в строках также следует инвертировать.*

```
uses linux,sysutils ;

var f1,f2:text;
    kol_strok,
    i,j,k:longint;
    s:string;
begin
if paramcount<>2 then
  begin
   writeln('Неправильные аргументы командной строки ',paramcount);
   halt(1);
  end;

assign(f1,paramstr(1));
assign(f2,paramstr(2));
reset(f1);
rewrite(f2);
kol_strok:=0;

while  not eof(f1) do  {Подсчет количества строк в файле}
begin
readln(f1);
inc(kol_strok);
end;


for i:=kol_strok downto 1 do
begin
  reset(f1);

  for j:=1 to i do {Установка на нужную строку и соответствнно ее считуем}
   begin
    readln(f1,s);
   end;
```

```
  for k:=length(s) downto 1 do {Запись посимвольно в файл начиная с конца
строчки}
   begin
   write(f2,s[k]);
   end;
  writeln(f2);

end;

writeln(kol_strok);

close(f1);
close(f2);
end.
```

13.3. Бинарные файлы

Упражнение 13.24. *Напишите программу, копирующую бинарный файл в обратном порядке байт.*

```
var
  fb,copyfb:file of byte;
  size,n:longint;
  b:byte;
{$I-}
begin
   if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' входной_файл выходной файл');
    exit;
  end;
  assign(fb,paramstr(1));
  assign(copyfb,paramstr(2));
  reset(fb);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(1),' для чтения');
    exit;
  end;
  rewrite(copyfb);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(2),' для записи');
    exit;
  end;
  n:=filesize(fb);
  while n>=0 do
   begin
     //n:=n-2;
     seek(fb,n-1);
     //blockread(fb,b,1);
     //blockwrite(copyfb,b,1);
     read(fb,b);
     write(copyfb,b);
     n:=n-1;
   end;
  close(fb);
  close(copyfb);
```

```
end.
```

Упражнение 13.25. *Составьте программу кодировки и раскодировки файлов по заданному ключу (строке символов).*

```
{I-}
var
  fin,fout:file of byte;
  pass:string;
  size:longint;
  b:byte;
begin
  if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' входной_файл выходной файл');
    exit;
  end;
  assign(fin,paramstr(1));
  assign(fout,paramstr(2));

  reset(fin);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(1),' для чтения');
    exit;
  end;

  rewrite(fout);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(2),' для записи');
    exit;
  end;

  writeln('Введите кодовое слово: ');
  readln(pass);

  size:=0;
  while not(eof(fin)) do
  begin
    read(fin,b);
    b:=b xor byte(pass[1 + (size mod length(pass))]);
    write(fout,b);
    inc(size);
  end;
  close(fin);
  close(fout);
end.
```

Упражнение 13.26. *Составьте аналог команды* `cmp`.

```
type
  mas=array [1..1] of byte;

var
  fb,fbx:file of byte;
  size,n,nx,i:longint;
  t:boolean;
  b,bx:byte;
```

```
  a:^mas;
{$I-}
begin
   if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' входной_файл выходной файл');
    exit;
  end;
  assign(fb,paramstr(1));
  assign(fbx,paramstr(2));
  reset(fb);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(1));
    exit;
  end;
  reset(fbx);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия файла: ',paramstr(2));
    exit;
  end;
  n:=filesize(fb);
  nx:=filesize(fbx);
 { if n<>nx then  begin
                  writeln('файлы   ',paramstr(1),'   и   ',paramstr(2),'   не
идентичны');
                  exit;
                  end;}
  t:=false;
  if n=nx then
  begin
    getmem(a,nx);
    t:=true;
    while n>=0 do
    begin
      read(fb,b);
      read(fbx,bx);
      if b<>bx then
      begin
        a^[nx-n]:=bx;
        t:=false;
      end;
      n:=n-1;
    end;
    if not t then
    begin
      writeln('различия в файлах:');
      for i:=1 to nx do
        writeln('номер байта в файле ',i,' ,байт',a^[i] );
    end;
    freemem(a,nx);
  end;
  if t then writeln('файлы ',paramstr(1),' и ',paramstr(2),' идентичны');

  if n<>nx then
  begin
    writeln('файлы ',paramstr(1),' и ',paramstr(2),' не идентичны');
```

```
    exit;
  end;
  close(fb);
  close(fbx);
end.
```

Упражнение 13.27. *Создайте упрощенную версию команды* cp*, копирующую один файл в другой с отображением файла на память.*

```
uses crt,sysutils,linux;
type
    smth=array [0..0] of char; (*безразмерный массив*)
    psmth=^smth;               (*то же, что и pchar*)
var
    s1,s2:array [0..1000] of char; (*для имен файлов*)
    t1,t2:integer;                 (*дескрипторы файлов*)
    n,w:longint;                   (*счетчик байт и размер файла*)
    m:tmmapargs;                   (*запись с параметрами mmap*)
    p1,p2:psmth;                   (*указатели, возвращаемые mmap*)
BEGIN
    writeln('Введите имя первого файла');
    readln(s1);
    {$I-}
    (*проверка на существование первого файла и доступность для чтения*)
    if not access(pchar(s1),f_ok or r_ok) then
    begin
      writeln('Файл '+pchar(s1)+' не существует или недоступен для чтения');
      halt(1);
    end;
    writeln('Введите имя второго файла');
    readln(s2);

    (*открытие первого файла для чтения*)
    t1:=fdopen(s1,Open_RDONLY);
    w:=fdseek(t1,0,SEEK_END);  (*определение размера файла*)

    (*заполнение параметров для mmap*)
    m.address:=0;   (*несущественно*)
    m.offset:=0;    (*начальное смещение в файле*)
    m.size:=w;      (*размер файла*)
    m.flags:=MAP_SHARED; (*отображение с занесением изменений в файл*)
    m.prot:=PROT_READ;   (*задание доступа для чтения*)
    m.fd:=t1;            (*дескриптор файла*)

    p1:=psmth(mmap(m));   (*получаем указатель на область памяти, связанную с
файлом*)
    if longint(p1)=-1 then
    begin
      writeln('Ошибка вызова mmap (t1)');
      fdclose(t1);
      halt(0);
    end;
    (*создаем второй файл, существующий усекаем до нулевого размера*)
    t2:=fdopen(s2,Open_WRONLY or Open_TRUNC or Open_CREAT, octal(640));
    if t2=-1 then
    begin
      writeln('Файл ',s2,' не удалось создать или открыть для записи');
      halt(1);
    end;
```

```
    fdseek(t2,w-1,SEEK_SET);(*перемещаемся на w-1 байт от начала*)
    fdwrite(t2,w,1);         (*... и записываем 1 байт для подгоники размеров
файлов*)

    fdclose(t2);
    t2:=fdopen(s2,Open_RDWR);(*переоткрываем второй файл для чтения и записи*)

    m.address:=0;
    m.offset:=0;
    m.size:=w;
    m.flags:=MAP_SHARED;
    m.prot:=PROT_WRITE or PROT_READ;
    m.fd:=t2;
    p2:=psmth(mmap(m));
    if longint(p2)=-1 then
    begin
      writeln('Ошибка вызова mmap (t2)');
      fdclose(t1);
      fdclose(t2);
      munmap(p1,w);
      halt(0);
    end;
    (*копируем данные из одной отображенной области в другую*)
    for n:=0 to w-1 do
      p2^[n]:=p1^[n];

    (*снимаем отображение и закрываем файлы*)
    munmap(p1,w);
    munmap(p2,w);
    fdclose(t1);
    fdclose(t2);
    writeln('Файл ',s1,' скопирован в файл ',s2);
END.
```

13.4. Каталоги

Упражнение 13.28. *Напишите аналог команды* `ls -l`.

```
uses  linux,strings,sysutils;  (*для системных вызовов Linux и работы со
строками PChar*)

function ctime(var time_t:longint):pchar;cdecl;external 'c';


function  gettype(t:word):char;forward;  (*тип объекта ф.с. в формате команды
ls*)

(*тип объекта ф.с. в формате команды ls*)
function gettype(t:word):char;
begin
  if S_ISDIR(t) then            (*проверка на каталог*)
    gettype:='d'
  else
    if S_ISREG(t) then          (*проверка на обычный файл*)
      gettype:='-'
    else
      if S_ISBLK(t) then        (*проверка на блочное устройство*)
        gettype:='b'
      else
```

```
        if S_ISCHR(t) then      (*проверка на символьное устройство*)
          gettype:='c'
        else
          if S_ISFIFO(t) then (*проверка на именованный программный канал*)
            gettype:='p'
          else
            if S_ISLNK(t) then  (*проверка на сиволическую ссылку*)
              gettype:='l'
          else
            gettype:='?';
end;


function getrights(r:word):string;
var
  u,           (*права для владельца*)
  g,           (*права для группы*)
  o,           (*права для всех остальных*)
  s,           (*специальные права*)
  i:integer;
  res:string;  (*права в символьной форме*)
const
  o7777=(1 shl 12)-1; (*восьмеричная константа = все 12 бит прав заданы *)
  o10  =8;            (*010  *)
  o100 =64;           (*0100 *)
  o1000=512;          (*01000*)
  symrights:array [0..7] of string=( (*базовые комбинации прав в символьной
форме*)
       '---', (*0 = 000*)
       '--x', (*1 = 001*)
       '-w-', (*2 = 010*)
       '-wx', (*3 = 011*)
       'r--', (*4 = 100*)
       'r-x', (*5 = 101*)
       'rw-', (*6 = 110*)
       'rwx'  (*7 = 111*)
     );
  spec='tss';          (*массив специальных прав доступа*)
begin
  (*обрезаем старшие биты, не относящиеся к правам доступа (тип файла и
т.п.)*)
  r:=r and o7777;(*восьмеричная константа 10000-1==1*8^4-1==1*(2^3)^4-1==2^12-
1 *)
  (*выделяем числовые права для владельца, группы, остальных + специальные*)
  o:=r mod o10;
  s:=r div o1000;
  u:=(r div o100) mod o10;
  g:=(r mod o100) div o10;
  res:=symrights[u]+symrights[g]+symrights[o];(*формируем символьыне права из
базовых троек*)

  for i:=1 to 3 do   (*цикл проверки наличия чпециальных прав*)
    if s and (1 shl (i-1)) <> 0 then (*если право установлено*)
      if res[12-3*i]='x' then  (*если есть обычное право на выполнение*)
        res[12-3*i]:=spec[i]   (*заносим маленькую букву*)
      else
        res[12-3*i]:=upcase(spec[i]); (*иначе - большую*)
```

```
  getrights:=res;  (*возвращаем результат - 9-символьное представление 12-
битных прав*)
end;


var
  d:^TDir;              (*указатель на запись для работы с каталогом*)
  elem:^Dirent;  (*указатель на запись, хранящую один элекмент каталога*)
  tekkat,                          (*строка для хранения имени каталога*)
  fullpath                         (*полный путь к элементу каталога*)
          :array [0..1000] of char;
  st:stat;          (*для хранения информации о файле или каталоге*)

begin
  if paramcount=0 then      (*если в командной строке не указан каталог*)
    strcopy(tekkat,'.')     (*то в качестве каталога используем текущий*)
  else
    tekkat:=paramstr(1); (*иначе используем каталог из командной строки*)

  if  not  access(pchar(tekkat),F_OK  or  R_OK)  then    (*F_OK  -  проверка
существования объекта ф.с.*)
  begin
    writeln('Каталог ', tekkat, ' не существует или недоступен для чтения');
(*диагностика*)
    halt(1);                     (*возврат в предыдущую программу*)
  end;

  if  not  fstat(pchar(tekkat),st)  then      (*попытка получения информации о
файле или каталоге*)
  begin
    writeln('Ошибка   получения   информации   о   каталоге   ',   tekkat);
(*диагностика*)
    halt(1);                     (*возврат в предыдущую программу*)
  end;

  if not S_ISDIR(st.mode) then    (*проверка на каталог*)
  begin
    writeln(tekkat, ' - не каталог');  (*диагностика*)
    halt(1);                      (*возврат в предыдущую программу*)
  end;

  d:=opendir(tekkat);           (*попытка открытия каталога для чтения*)

  if d=nil then                   (*если попытка не удалась*)
  begin
    writeln('Ошибка вызова opendir для каталога ', tekkat);  (*диагностика*)
    halt(1);                      (*возврат в предыдущую программу*)
  end;

  elem:=readdir(d);               (*попытка чтения элемента каталога*)
  while elem<>nil do
  begin
    (*формирование полного имени элемента каталога*)
    strcopy(fullpath,tekkat);       (*копируем имя текущего каталога в начало
полного имени*)
    if strcomp(tekkat,'/')<>0 then(*если текущий каталог - не корневой*)
    begin
      if fullpath[strlen(fullpath)-1]='/' then (*если в конце имени каталога
```

```
слэш*)
        fullpath[strlen(fullpath)-1]:=#0;       (*заменяем его признаком конца
строки*)
      strcat(fullpath,'/');           (*добавляем после имени каталога слэш-
разделитель*)
    end;
    strcat(fullpath,elem^.name);  (*и имя элемента каталога*)

    if not fstat(pchar(fullpath),st) then     (*попытка получения информации о
файле или каталоге*)
    begin
      writeln('Ошибка получения информации о ', fullpath);  (*диагностика*)
      continue;                 (*возврат в предыдущую программу*)
    end;
    {gmtime_r(st.mtime,mytm);}
    writeln(gettype(st.mode),getrights(st.mode),st.nlink:5,
     '  ',st.size:10,' ',ctime(st.mtime), elem^.name);  (*вывод имени элемента
каталога*)
    elem:=readdir(d);               (*попытка чтения элемента каталога*)
  end;

  closedir(d);                    (*закрытие открытого opendir каталога*)
end.
```

Упражнение 13.29. *Составьте аналог команды* `vdir`.

```
uses linux,strings,sysutils;

function getname(uid:integer):string;
const w='/etc/passwd';
var ts,nam1,namb1:string;
    tx:text;
begin
  assign(tx,w);
  reset(tx);
  while not EOF (tx) do
  begin
       readln(tx,ts);
       uid:=pos(':',ts);
       nam1:=copy(ts,1,uid-1);
       delete(ts,1,uid);
       uid:=pos(':',ts);
       delete(ts,1,uid);
       namb1:=copy(ts,1,uid-1);
       if namb1='500' then
       write(nam1)
  end;
  close(tx);
  getname:=nam1;
end;

function getgroup(gid:integer):string;
const q='/etc/group';
var ts,nam,namb:string;
    t:text;
begin
  assign(t,q);
  reset(t);
  while not EOF (t) do
```

```
  begin
       readln(t,ts);
       gid:=pos(':',ts);
       nam:=copy(ts,1,gid-1);
       delete(ts,1,gid);
       gid:=pos(':',ts);
       delete(ts,1,gid);
       namb:=copy(ts,1,gid-1);
       if namb='500' then
       write(nam);
  end;
  close(t);
  getgroup:=nam;
end;

function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;

function getrights(mode:integer):string;
const
  sympr:array [0..7] of string=(
                                  '---', {0}
                                  '--x', {1}
                                  '-w-', {2}
                                  '-wx', {3}
                                  'r--', {4}
                                  'r-x', {5}
                                  'rw-', {6}
                                  'rwx'  {7}
                                );
  specsympr:array [0..7] of string=(
                                  '---', {0}
                                  '--t', {1}
                                  '-s-', {2}
                                  '-st', {3}
                                  's--', {4}
                                  's-t', {5}
                                  'ss-', {6}
                                  'sst'  {7}
                                );
var
```

```
  s,u,g,o,i:integer;
  res:string;
begin
  mode:=mode and octal(7777);
  u:=(mode div octal(100)) mod octal(10);
  g:=(mode mod octal(100)) div octal(10);
  o:=mode mod octal(10);
  s:=mode div octal(1000);
  res:=sympr[u]+sympr[g]+sympr[o];
  for i:=1 to 3 do
    if specsympr[s][i]<>'-' then
    begin
      if res[3*i]='-' then
        res[3*i]:=upcase(specsympr[s][i])
      else
        res[3*i]:=specsympr[s][i];
    end;
  getrights:=res;
end;

var
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  dt:tdatetime;
  polniypath,name:array [0..2000] of char;

begin
  if paramcount = 0 then
    name:='.'
  else
    name:=paramstr(1);

  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия текущего каталога');
    halt(0);
  end;

  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
      strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      {writeln(polniypath,' ',s.size);}
```

```
      dt:=filedatetodatetime(st.mtime);
      write(gettype(st.mode),getrights(st.mode),st.nlink:5,
            getname(st.uid),'                  ',getgroup(st.gid),st.size:10,'
',datetimetostr(dt),'  ' );
      writeln(el^.name);
    end;
    el:=readdir(d);
  end;
  closedir(d);
end.
```

Упражнение 13.30. *Напишите упрощенный аналог команды* `ls`*, распечатывающий содержимое текущего каталога (файла с именем ".") без сортировки имен по алфавиту. Предусмотрите чтение каталога, чье имя задается как аргумент программы. Имена "." и ".." не выдавать.*

```
uses linux,strings,sysutils,crt;

{$linklib c}

type
  plong=^longint;

function ctime(r:plong):pchar;cdecl;external;


function strchr(s:string;c:char):boolean;
var
  i:integer;
begin
  for i:=1 to length(s) do
    if s[i]=c then
    begin
      strchr:=true;
      exit;
    end;
  strchr:=false;
end;


function getall(w:string;uid:integer):string;
{const w='/etc/passwd';}
var ts,nam1,namb1:string;
    tx:text;
    d:integer;
begin
  assign(tx,w);
  reset(tx);
  while not EOF (tx) do
  begin
       readln(tx,ts);
       d:=pos(':',ts);
       nam1:=copy(ts,1,d-1);
       delete(ts,1,d+2);
       d:=pos(':',ts);
       {delete(ts,1,d);}
       namb1:=copy(ts,1,d-1);
       val(namb1,d);
```

```
      {writeln('имя = ',nam1,', номер=',namb1);}
      if d=uid then
        break;
  end;
  close(tx);
  getall:=nam1;
end;



function getname(uid:integer):string;
begin
  getname:=getall('/etc/passwd',uid);
end;


function getgroup(gid:integer):string;
begin
  getgroup:=getall('/etc/group',gid);
end;


function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;

function getrights(mode:integer):string;
const
  sympr:array [0..7] of string=(
                                '---', {0}
                                '--x', {1}
                                '-w-', {2}
                                '-wx', {3}
                                'r--', {4}
                                'r-x', {5}
                                'rw-', {6}
                                'rwx'  {7}
                               );
  specsympr:array [0..7] of string=(
                                '---', {0}
                                '--t', {1}
```

```
                                  '-s-', {2}
                                  '-st', {3}
                                  's--', {4}
                                  's-t', {5}
                                  'ss-', {6}
                                  'sst'  {7}
                                );
var
  s,u,g,o,i:integer;
  res:string;
begin
  mode:=mode and octal(7777);
  u:=(mode div octal(100)) mod octal(10);
  g:=(mode mod octal(100)) div octal(10);
  o:=mode mod octal(10);
  s:=mode div octal(1000);
  res:=sympr[u]+sympr[g]+sympr[o];
  for i:=1 to 3 do
    if specsympr[s][i]<>'-' then
    begin
      if res[3*i]='-' then
        res[3*i]:=upcase(specsympr[s][i])
      else
        res[3*i]:=specsympr[s][i];
    end;
  getrights:=res;
end;


procedure obhod(name:pchar);
var
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  dt:tdatetime;
  polniypath,datetime:array [0..2000] of char;
  i,k:integer;
begin
  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия каталога ',name);
    exit;
  end;
  i:=0;
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
      strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
```

```
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      (*
      strcopy(datetime,ctime(@st.mtime)+4);
      datetime[12]:=#0;
      write(gettype(st.mode),getrights(st.mode),st.nlink:5,' ',
            getname(st.uid):10,'    ',getgroup(st.gid):10,'    ',st.size:10,'
',datetime,'  ' );
      *)
      if(gettype(st.mode)='d') then
        textcolor(9);
      if(gettype(st.mode)='-') and strchr(getrights(st.mode),'x') then
        textcolor(lightgreen);
      if(gettype(st.mode)='p') then
        textcolor(brown);
      if(gettype(st.mode)='l') then
        textcolor(lightblue);
      if (gettype(st.mode)='c') or (gettype(st.mode)='b') then
        textcolor(yellow);
      write(el^.name);
      for k:=strlen(el^.name) to 15  do
        write(' ');
      textcolor(7);
    end;
    el:=readdir(d);
    inc(i);
    if(i mod 5=0)then writeln;
  end;

  closedir(d);
  if(i mod 5<>0)then writeln;

end;

var
  name:array [0..2000] of char;
begin
  if paramcount = 0 then
    name:='.'
  else
    name:=paramstr(1);

  obhod(name);

end.
```

Упражнение 13.31. *Напишите программу удаления файлов и каталогов, заданных в командной строке. Программа должна удалять каталоги рекурсивно и отказываться удалять файлы устройств.*

```
uses linux,strings,sysutils,crt;

{$linklib c}

type
  plong=^longint;
```

```
function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;



function obhod(name:pchar):boolean;
var
  flag:boolean;
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  polniypath:array [0..2000] of char;
begin
  flag:=true;
  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия каталога ',name);
    exit;
  end;
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      if not (gettype(st.mode) in ['b','c','d']) then
      begin
```

```
        writeln('Стираю файл ',polniypath);
        //unlink(polniypath);
        if not unlink(polniypath) then
        begin
          writeln('невозможно стереть файл ',polniypath);
          flag:=false;(*ошибка удаления файла - нельзя будет стереть каталог*)
        end;
      end;
    end;
    el:=readdir(d);
  end;
  closedir(d);

  d:=opendir(name);
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      if (gettype(st.mode)='d') and
         (strcomp(el^.name,'.')<>0) and
         (strcomp(el^.name,'..')<>0) then
      begin
        writeln('Переход в каталог ',polniypath);
        if not obhod(polniypath) then
          flag:=false;
      end;
    end;
    el:=readdir(d);
  end;
  closedir(d);
  if not flag then
    writeln('Каталог ',name,
    ' не будет стерт, т.к. в нем не удалось стереть часть файлов или
каталогов')
  else
  begin
    {$i-}
    rmdir(name);
    if ioresult <> 0 then
    begin
      writeln('Ошибка удаления каталога ',name);
      flag:=false;
    end;
  end;
  writeln('Для каталога ',name, ' получен ',flag);
  obhod:=flag;
```

```
end;


var
  name:array [0..2000] of char;
begin
  if paramcount<>0 then
  begin
    name:=paramstr(1);
    obhod(name);
  end
  else
    writeln('С особой осторожностью используйте: ',paramstr(0),' удаляемый
каталог');
end.
```

Упражнение 13.32. *Напишите функцию рекурсивного обхода дерева подкаталогов и печати имен всех файлов в нем с выдачей атрибутов в форме команды* `ls -l`.

```
uses linux,strings,sysutils;

{$linklib c}

type
  plong=^longint;

function ctime(r:plong):pchar;cdecl;external;

function getall(w:string;uid:integer):string;
{const w='/etc/passwd';}
var ts,nam1,namb1:string;
    tx:text;
    d:integer;
begin
  assign(tx,w);
  reset(tx);
  while not EOF (tx) do
  begin
       readln(tx,ts);
       d:=pos(':',ts);
       nam1:=copy(ts,1,d-1);
       delete(ts,1,d+2);
       d:=pos(':',ts);
       {delete(ts,1,d);}
       namb1:=copy(ts,1,d-1);
       val(namb1,d);
       {writeln('имя = ',nam1,', номер=',namb1);}
       if d=uid then
         break;
  end;
  close(tx);
  getall:=nam1;
end;




function getname(uid:integer):string;
begin
```

```
  getname:=getall('/etc/passwd',uid);
end;


function getgroup(gid:integer):string;
begin
  getgroup:=getall('/etc/group',gid);
end;


function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;

function getrights(mode:integer):string;
const
  sympr:array [0..7] of string=(
                                  '---', {0}
                                  '--x', {1}
                                  '-w-', {2}
                                  '-wx', {3}
                                  'r--', {4}
                                  'r-x', {5}
                                  'rw-', {6}
                                  'rwx'  {7}
                                );
  specsympr:array [0..7] of string=(
                                  '---', {0}
                                  '--t', {1}
                                  '-s-', {2}
                                  '-st', {3}
                                  's--', {4}
                                  's-t', {5}
                                  'ss-', {6}
                                  'sst'  {7}
                                );
var
  s,u,g,o,i:integer;
  res:string;
begin
  mode:=mode and octal(7777);
  u:=(mode div octal(100)) mod octal(10);
```

```
  g:=(mode mod octal(100)) div octal(10);
  o:=mode mod octal(10);
  s:=mode div octal(1000);
  res:=sympr[u]+sympr[g]+sympr[o];
  for i:=1 to 3 do
    if specsympr[s][i]<>'-' then
    begin
      if res[3*i]='-' then
        res[3*i]:=upcase(specsympr[s][i])
      else
        res[3*i]:=specsympr[s][i];
    end;
  getrights:=res;
end;


procedure obhod(name:pchar);
var
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  dt:tdatetime;
  polniypath,datetime:array [0..2000] of char;

begin
  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия каталога ',name);
    exit;
  end;

  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      strcopy(datetime,ctime(@st.mtime)+4);
      datetime[12]:=#0;
      write(gettype(st.mode),getrights(st.mode),st.nlink:5,' ',
            getname(st.uid):10,'     ',getgroup(st.gid):10,'     ',st.size:10,'
',datetime,'  ' );
      writeln(el^.name);
    end;
    el:=readdir(d);
  end;
```

```
  closedir(d);

  d:=opendir(name);

  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      if S_ISDIR(st.mode) then
      begin
        if (strcomp(el^.name,'.')<>0) and (strcomp(el^.name,'..')<>0) then
        begin
          writeln;
          writeln(polniypath,':');
          obhod(polniypath);
        end;
      end;
    end;
    el:=readdir(d);
  end;

  closedir(d);


end;

var
  name:array [0..2000] of char;
begin
  if paramcount = 0 then
    name:='.'
  else
    name:=paramstr(1);

  obhod(name);

end.
```

Упражнение 13.33. *Напишите программу удаления каталога, которая удаляет все файлы в нем и, рекурсивно, все его подкаталоги.*

```
uses linux,strings,sysutils,crt;

{$linklib c}

type
```

```
  plong=^longint;



function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;



function obhod(name:pchar):boolean;
var
  flag:boolean;
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  polniypath:array [0..2000] of char;
begin
  flag:=true;
  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия каталога ',name);
    exit;
  end;
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
```

```
      if not (gettype(st.mode) = 'd') then
      begin
        writeln('Стираю файл ',polniypath);
        //unlink(polniypath);
        if not unlink(polniypath) then
        begin
          writeln('невозможно стереть файл ',polniypath);
          flag:=false;(*ошибка удаления файла - нельзя будет стереть каталог*)
        end;
      end;
    end;
    el:=readdir(d);
  end;
  closedir(d);

  d:=opendir(name);
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      if (gettype(st.mode)='d') and
         (strcomp(el^.name,'.')<>0) and
         (strcomp(el^.name,'..')<>0) then
      begin
        writeln('Переход в каталог ',polniypath);
        if not obhod(polniypath) then
          flag:=false;
      end;
    end;
    el:=readdir(d);
  end;
  closedir(d);
  if not flag then
    writeln('Каталог ',name,
    ' не будет стерт, т.к. в нем не удалось стереть часть файлов или
каталогов')
  else
  begin
    {$i-}
    rmdir(name);
    if ioresult <> 0 then
    begin
      writeln('Ошибка удаления каталога ',name);
      flag:=false;
    end;
  end;
```

```
  writeln('Для каталога ',name, ' получен ',flag);
  obhod:=flag;
end;


var
  name:array [0..2000] of char;
begin
  if paramcount<>0 then
  begin
    name:=paramstr(1);
    obhod(name);
  end
  else
    writeln('С особой осторожностью используйте: ',paramstr(0),' удаляемый
каталог');
end.
```

13.5. Файловые системы

Упражнение 13.34. *Создайте аналог команды* `df`.

```
uses linux;

var  s:string;
     p:statfs;
     i,k: word;
     fd:text;
     buf:array[1..1000]of char;
     pol:boolean;
var
     size:array[1..3] of double;
     sizepow:array[1..3] of integer;
const
  letters:array [0..3] of char=('b','K','M','G');


function point_Mount(var f:string):string;
var str:string;j,k:integer;flag:boolean;
begin
str:='';
k:=0;
flag:=false;
 for j:=1 to length(f) do
    begin
        if f[j]=' ' then begin flag:=true;inc(k);end;
        if (flag=true)and(k=1)then
       begin
         str:=str+f[j];
       end;
    end;
delete(str,1,1);
point_Mount:=str;
end;


begin
assign(fd,'/etc/mtab');
reset(fd);
```

```
pol:=true;

Writeln('Файловая система     Размер     Испол       Дост    Исп%   Подключено
к');

while  not eof(fd) do
 begin
  readln(fd,s);


  pol:=fsstat(point_Mount(s),p);
 if (pol)and(p.blocks<>0) then
    begin
     for i:=1 to length(s) do
   begin
    write(s[i]);
    if s[i]=' ' then break;
   end;

   for k:=i to 1 do
     write(' ');

    size[1]:=1.*p.blocks*p.bsize;
    size[2]:=1.*(p.blocks-p.bavail)*p.bsize;
    size[3]:=1.*p.bavail*p.bsize;

    for i:=1 to 3 do
    begin
      sizepow[i]:=0;
      while size[i]>1024 do
      begin
        size[i]:=size[i]/1024;
        inc(sizepow[i]);
      end;
    end;

    writeln(size[1]:15:1,letters[sizepow[1]],{Истинный  размер }


    size[2]:10:1,letters[sizepow[2]],         {Используемый размер}



    size[3]:10:1,letters[sizepow[3]],         {Доступно или свободно}


    '     ',((p.blocks-p.bavail)*100/p.blocks):4:0,'%     ',       {Процентное
соотношение используемого места}
    ' ',point_Mount(s));                              {Подключение}
    end
 end;

close(fd);
end.
```

13.6. Файловая система proc

Упражнение 13.35. *Напишите программу, печатающую: свои аргументы, переменные окружения, информацию обо всех открытых ею файлах и используемых каналах.*

```
uses dos,linux,strings;

var
  i,n,pid:integer;
  s,fullname:array [0..200] of char;
  temp:string;
  d:pdir;
  el:pdirent;
begin
  i:=fdopen('/etc/passwd',Open_RDONLY);
  n:=paramcount;
  writeln('В командной строке ',paramcount+1, ' параметров');
  for i:=0 to n do
    writeln('Параметр ',i+1,' - ',paramstr(i));
  writeln('Переменные окружения:');
  n:=envcount;
  for i:=1 to n do
    writeln(envstr(i));
  pid:=getpid;
  strcopy(s,'/proc/');
  str(pid,temp);
  strpcopy(s+strlen(s),temp);
  strcat(s,'/fd/');
  d:=opendir(pchar(s));
  el:=readdir(d);
  writeln('Файлы, открытые процессом:');
  while el<>nil do
  begin
    if (strcomp(el^.name,'.')<>0) and (strcomp(el^.name,'..')<>0) then
    begin
      strcopy(fullname,s);
      strcat(fullname,el^.name);
      writeln('Дескриптор ',el^.name,' соответствует ',readlink(fullname));
    end;
    el:=readdir(d);
  end;
  closedir(d);
end.
```

Упражнение 13.36. *Создайте аналог команды* `ps`.

```
uses linux,strings,sysutils; (*для системных вызовов Linux и работы со
строками PChar*)

var
  d:^TDir;        (*указатель на запись для работы с каталогом*)
  elem:^Dirent;   (*указатель на запись, хранящую один элемент каталога*)
  realname,                    (*имя процесса из файла status*)
  fullpath                     (*полный путь к элементу каталога*)
         :array [0..1000] of char;
  number,coder:integer;    (*номер процесса и код ошибки преобразования*)
  f:text;
  name:string;
begin
```

```
  d:=opendir('/proc');        (*попытка открытия каталога для чтения*)

  if d=nil then                    (*если попытка не удалась*)
  begin
    writeln('Ошибка вызова opendir для каталога /proc');  (*диагностика*)
    halt(1);                        (*возврат в предыдущую программу*)
  end;

  elem:=readdir(d);               (*попытка чтения элемента каталога*)
  while elem<>nil do              (*пока не достигнут конец каталога*)
  begin
    (*пытаемся преобразовать имя элемента катлога в число*)
    val(strpas(elem^.name),number,coder);
    (*если это удается, катлог имеет числовое имя и соответствует процессу*)
    if coder=0 then
    begin
    (*формируем путь к файлу status в виде /proc/номер_процесса/status *)
      strcopy(fullpath,'/proc/');
      strcat(fullpath,elem^.name);
      strcat(fullpath,'/status');
      (*открываем файл и чтиаем из него первую строку*)
      assign(f,fullpath);
      reset(f);
      readln(f,name);
      close(f);
     (*вырезаем из строки ту ее часть, что соответствует имени процесса*)
      strlcopy(realname,@name[7],length(name));
      realname[length(name)-6]:=#0;
      (*выводим номер и имя процесса*)
      writeln(number,#9,realname);
    end;
    elem:=readdir(d);               (*попытка чтения элемента каталога*)
  end;
  closedir(d);                    (*закрытие открытого opendir каталога*)
end.
```

Упражнение 13.37. *Используя файловую систему* /proc*, получите информацию об открытых всеми процессами файлах.*

```
uses linux,strings,sysutils; (*для системных вызовов Linux и работы со
строками PChar*)

var
  d,d1:^TDir;             (*указатель на запись для работы с каталогом*)
  elem,elem1:^Dirent;         (*указатель на запись, хранящую один элекмент
каталога*)
  realname,                       (*имя процесса из файла status*)
  fullpath,fullpath1              (*полный путь к элементу каталога*)
          :array [0..1000] of char;
  number,coder:integer;   (*номер процесса и код ошибки преобразования*)
  f:text;
  name:string;
begin
  d:=opendir('/proc');       (*попытка открытия каталога для чтения*)

  if d=nil then                    (*если попытка не удалась*)
  begin
    writeln('Ошибка вызова opendir для каталога /proc');  (*диагностика*)
    halt(1);                      (*возврат в предыдущую программу*)
```

```
  end;

  elem:=readdir(d);                (*попытка чтения элемента каталога*)
  while elem<>nil do               (*пока не достигнут конец каталога*)
  begin
    (*пытаемся преобразовать имя элемента катлога в число*)
    val(strpas(elem^.name),number,coder);
    (*если это удается, катлог имеет числовое имя и соответствует процессу*)
    if coder=0 then
    begin
      (*формируем путь к файлу status в виде /proc/номер_процесса/fd *)
      strcopy(fullpath,'/proc/');
      strcat(fullpath,elem^.name);
      strcat(fullpath,'/fd');
      d1:=opendir(fullpath);
      if d1=nil then
        writeln('Для  процесса  ',number,'  информация  об  открытых  файлах
недоступна')
      else
      begin
        writeln('Процесс ',number,' открыл следующие файлы:');
        elem1:=readdir(d1);      (*попытка чтения элемента каталога*)
        while elem1<>nil do      (*пока не достигнут конец каталога*)
        begin
          strcopy(fullpath1,fullpath);
          strcat(fullpath1,'/');
          strcat(fullpath1,elem1^.name);
          if (strcomp(elem1^.name,'.')<>0) and (strcomp(elem1^.name,'..')<>0)
then
          begin
            (*realname[readlink(realname,fullpath1,999)]:=#0;*)
              writeln(#9,readlink(strpas(fullpath1)));
          end;
          elem1:=readdir(d1);     (*попытка чтения элемента каталога*)
        end;
      end;
    end;
    elem:=readdir(d);               (*попытка чтения элемента каталога*)
  end;
  closedir(d);                     (*закрытие открытого opendir каталога*)
end.
```

Упражнение 13.38. *Используя файловую* с*истему* /proc*, получите информацию о типе, версии и дате выпуска операционной системы.*

```
uses sysutils,crt,linux;

var
    f:text;
    count:integer;
    ch:char;
begin
assign(f,'/proc/version');
reset(f);
if IOResult<>0 then
 begin
  writeln('Неудалось открыть файл, попробуйте другими средствами');
  halt;
 end;
```

```
textcolor(7);
write('Тип операционной системы - ');
read(f,ch);
textcolor(2);
write(ch);
while ch<>' ' do
begin
read(f,ch);
write(ch);
end;

writeln;
while ch=' ' do
begin
read(f,ch);
end;
while ch<>' 'do
begin
read(f,ch);
end;
while ch=' ' do
begin
read(f,ch);
end;

textcolor(7);
write('Версия операционной системы - ');
textcolor(2);
write(ch);

while ch<>' 'do
begin
read(f,ch);
write(ch);
end;

while ch<>'#' do
begin
read(f,ch);
end;
read(f,ch);

writeln;
textcolor(7);
write('Дата выпуска -');
textcolor(2);
while not eof(f)do
begin
read(f,ch);
write(ch);
end;
textcolor(7);
Close(f);

end.
```

Упражнение 13.39. *Используя файловую систему /proc, получите информацию о*

*процессоре (*vendor_id, cpu family, model, model name, stepping, cpu MHz, cache size, fdiv_bug, hlt_bug, sep_bug, f00f_bug, coma_bug, fpu, fpu_exception, cpuid level, wp, flags, bogomips*).*

```
uses sysutils,linux;
var
    f,count:integer;
    ch:array [0..511] of byte;

begin
  f:=fdopen('/proc/cpuinfo',Open_RDONLY);
  if f=-1 then
  begin
    writeln('Невозможно открыть файл ');
    halt;
  end;
  writeln('Information about CPU');

  count:=fdread(f,ch,512);
  while count>0 do
  begin
    fdwrite(1,ch,count);
    count:=fdread(f,ch,512);
  end;
  fdClose(f);
end.
```

Упражнение 13.40. *Используя файловую систему* /proc*, получите информацию об используемой памяти.*

```
uses sysutils,linux;
var
    f,count:integer;
    ch:array [0..511] of char;

begin
  f:=fdopen('/proc/meminfo',Open_RDONLY);
  if f=-1 then
  begin
    writeln('Невозможно открыть файл для просмотра информации об используемой
памяти');
    halt;
  end;
  writeln('Information about MEMORY');

  count:=fdread(f,ch,512);
  while count>0 do
  begin
    fdwrite(1,ch,count);
    count:=fdread(f,ch,512);
  end;
  fdClose(f);

end.
```

13.7. Управление файлами

Упражнение 13.41. *Составьте аналог команды* rm*.*

```
uses linux,sysutils;
```

```
var
  f:text;
  d:boolean;
  k:char;
  s:string;
begin
  writeln('введите имя файла, который нужно удалить');
  readln(s);
  assign(f,s);

  if s='' then
  begin
    writeln('повторите попытку');
    exit;
  end;
  writeln('подтвердите удаление файла Y/N');
  readln(k);
  if (k='Y') or (k='y') then
  begin
    d:=deletefile(s);
    if d then
      writeln('файл удален')
    else
      writeln('файл не удален');
  end
  else
    writeln('файл не удален');
end.
```

Упражнение 13.42. *Используя системный вызов* `fstat`*, напишите программу, определяющую тип файла: обычный файл, каталог, устройство, FIFO-файл.*

```
uses linux,strings,sysutils;

function gettype(mode:integer):string;
begin
  if S_ISREG(mode) then
    gettype:='файл'
  else
    if S_ISDIR(mode) then
      gettype:='каталог'
    else
      if S_ISCHR(mode) then
        gettype:='байтоориентированное устройство'
      else
        if S_ISBLK(mode) then
          gettype:='блочноориентированное устройство'
        else
          if S_ISFIFO(mode) then
            gettype:='FIFO-файл'
          else
            gettype:='другое';
end;

var
  st:stat;
  name:array[0..255] of char;
begin
  if paramcount = 0 then
```

```
    name:='.'
  else
    name:=fexpand(paramstr(1));

    if not fstat(pchar(name),st) then
      writeln('Ошибка вызова stat для ',name)
    else
      write(gettype(st.mode));
end.
```

Упражнение 13.43. *Составьте аналог команды* `chgrp`.

```
Uses linux;
Var
  UID,GID:Longint;
  F:Text;
  Code:Integer;
begin
  Writeln('This will only work if you are root.');
  if ParamCount<3 then
    begin
      Writeln('Error!!!');
      Writeln('Format: ./task <Filename> <UID> <GID>');
      Halt(1);
    end;
  val(Paramstr(2),UID,Code);
  if Code<>0 then
    begin
      Writeln('Error!!!');
      Writeln('Format: ./task <Filename> <UID> <GID>');
      Halt(1);
    end;
  val(Paramstr(3),GID,Code);
  if Code<>0 then
    begin
      Writeln('Error!!!');
      Writeln('Format: ./task <Filename> <UID> <GID>');
      Halt(1);
    end;
  if not Chown(ParamStr(1),UID,GID)  then
  if LinuxError=Sys_EPERM then
     Writeln('You are not root!')
  else
     Writeln('Chmod failed with exit code: ',LinuxError)
  else
     Writeln('Changed owner successfully!');
end.
```

Упражнение 13.44. *Составьте аналог команды* `mkdir`.

```
Program Tabs;
begin
{$I-}
if ParamCount=1 then
  begin
    MkDir(ParamStr(1));
    if IOResult <> 0 then Writeln('Cannot create directory')
    else Writeln('New directory created');
  end
```

```
else Writeln('Error');
end.
```

Упражнение 13.45. *Составьте аналог команды* chmod.

```
uses linux;

var
  f,ch:string;
  n,i:byte;
  d:integer;
begin
  if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' права_доступа файл/каталог');
    exit;
  end;
  f:=paramstr(2);
  ch:=paramstr(1);
  n:=length(ch);
  d:=0;
  for i:=1 to n do
    if not (ch[i] in ['0'..'7']) then
    begin
      writeln('Права доступа должны быть в восьмеричном формате');
      exit;
    end
    else
      d:=d*8+byte(ch[i])-byte('0');
  if not chmod(f,d) then
    writeln('Ошибка установки прав доступа ',ch,' для ',f);
end.
```

Упражнение 13.46. *Составьте аналог команды* chown.

```
uses linux,strings,sysutils,crt;

type
  plong=^longint;

procedure perror(s:pchar);cdecl;external 'c';


function strchr(s:string;c:char):boolean;
var
  i:integer;
begin
  for i:=1 to length(s) do
    if s[i]=c then
    begin
      strchr:=true;
      exit;
    end;
  strchr:=false;
end;


procedure getall(w:string;name:string;var uid,gid:integer);
var ts,nam1,namb1,namb2:string;
```

```
    tx:text;
    d:integer;
    f:boolean;
begin
  assign(tx,w);
  reset(tx);
  f:=false;
  while not EOF (tx) and not f do
  begin
       readln(tx,ts);
       d:=pos(':',ts);
       nam1:=copy(ts,1,d-1);
       delete(ts,1,d+2);
       d:=pos(':',ts);
       namb1:=copy(ts,1,d-1);
       delete(ts,1,d);
       val(namb1,d);
       uid:=d;
       d:=pos(':',ts);
       namb2:=copy(ts,1,d-1);
       val(namb2,d);
       gid:=d;
       if nam1=name then
         f:=true;
  end;
  if not f then
  begin
    uid:=-1;
    gid:=-1;
  end;
  close(tx);
end;


var
  username,groupname,fname:string;
  uid,gid:integer;
  posit,temp:integer;
begin
  if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' владелец[:группа] файл');
    exit;
  end;
  username:=paramstr(1);
  fname:=paramstr(2);
  posit:=0;
  posit:=pos(':',username);
  if posit<>0 then
  begin
    groupname:=copy(username,posit+1,length(username)-posit);
    username[0]:=char(posit-1);
    getall('/etc/passwd',username,uid,gid);
    getall('/etc/group',groupname,gid,temp);
  end
  else
    getall('/etc/passwd',username,uid,gid);
  if (uid=-1) or (gid=-1) then
```

```
  begin
    writeln('Неверное имя владельца (группы)');
    exit;
  end;
  if not chown(fname,uid,gid) then
    perror('Ошибка вызова chown');
end.
```

Упражнение 13.47. *Создайте программу* `chmodr`*, рекурсивно изменяющую права доступа для всех файлов каталога и вложенных в него подкаталогов. Имя каталога и права указываются в командной строке.*

```
uses linux,strings,sysutils,crt;

function gettype(mode:integer):char;
begin
  if S_ISREG(mode) then
    gettype:='-'
  else
    if S_ISDIR(mode) then
      gettype:='d'
    else
      if S_ISCHR(mode) then
        gettype:='c'
      else
        if S_ISBLK(mode) then
          gettype:='b'
        else
          if S_ISFIFO(mode) then
            gettype:='p'
          else
            gettype:='l';
end;


function obhod(prava:integer;name:pchar):boolean;
var
  flag:boolean;
  d:PDIR;
  el:pdirent;
  st:stat;
  res:integer;
  polniypath:array [0..2000] of char;
  ch:string;
  n,i:byte;
begin
  flag:=true;
  d:=opendir(name);
  if d=nil then
  begin
    writeln('Ошибка открытия каталога ',name);
    exit;
  end;
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
```

```
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
          strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;

    if not fstat(pchar(polniypath),st) then
        writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
          //if not (gettype(st.mode) = 'd') then
      if not chmod(pchar(polniypath),prava) then
        writeln('Ошибка установки прав доступа ',prava,' для ',polniypath);
    end;
    el:=readdir(d);
  end;
  closedir(d);

  d:=opendir(name);
  el:=readdir(d);
  while el<>nil do
  begin
    polniypath:=name;
    if strcomp(name,'/')=0 then
      strcat(polniypath,el^.name)
    else
    begin
      if name[strlen(name)-1]<>'/' then
        strcat(polniypath,'/');
      strcat(polniypath,el^.name);
    end;
    if not fstat(pchar(polniypath),st) then
      writeln('Ошибка вызова stat для ',polniypath)
    else
    begin
      if (gettype(st.mode)='d') and
         (strcomp(el^.name,'.')<>0) and
         (strcomp(el^.name,'..')<>0) then
      begin
        writeln('Переход в каталог ',polniypath);
        if not obhod(prava,polniypath) then
          flag:=false;
      end;
    end;
    el:=readdir(d);
  end;
  closedir(d);
  if not flag then
    writeln(' У каталога ',name, ' не удалось изменить права доступа ');
//  writeln('Для каталога ',name, ' получен ',flag);
  obhod:=flag;
end;


var
  name:array [0..2000] of char;
```

```
  prava,i:integer;
  ch:string;
begin
  if paramcount<>2 then
  begin
    writeln('Используйте: ',paramstr(0),' права_доступа файл/каталог');
    exit;
  end;
  name:=paramstr(2);
  ch:=paramstr(1);
  prava:=0;
  for i:=1 to length(ch) do
    if not (ch[i] in ['0'..'7']) then
    begin
      writeln('Права доступа должны быть в восьмеричном формате');
      exit;
    end
    else
      prava:=prava*8+byte(ch[i])-byte('0');

  obhod(prava,name);
end.
```

Упражнение 13.48. *Напишите программу, совмещающая команды* `mv` *и* `cp` *(в зависимости от своего названия).*

```
uses linux,sysutils;
var
  b:byte;
  s:string;
  f1,f2:file of byte;
begin
  s:=paramstr(0);
  delete(s,1,length(s)-2);
  if s='mv' then
  begin
    if paramcount<2 then
    begin
      writeln('Error: wrong arguments');
      writeln('введите имя файла, который хотите переименовать и новое имя
файла');
      halt(1);
    end;
    Assign(F1,paramstr(1));
    Assign(F2,paramstr(2));
    if not frename(paramstr(1),paramstr(2)) then
    begin
      writeln('невозможно переименовать ');
      halt(1);
    end;
  end
  else
    if s='cp' then
    begin
      if paramcount<2 then
      begin
        writeln('Error: wrong arguments');
        writeln('format: cp <fileinp> <fileout>');
        Halt(1);
```

```
      end;
      Assign(f1,paramstr(1));
      Reset(f1);
      Assign(f2,paramstr(2));
      Rewrite(f2);
      while not eof(f1)do
      begin
        read(f1,b);
        write(f2,b);
      end;
      close(f1);
      close(f2);
    end
    else
      writeln('Переименуйте программу в mv / cp');
end.
```

Упражнение 13.49. *Составьте аналог команды* `sync`.

```
procedure sync;cdecl; external 'c';

begin
  sync;
end.
```

Упражнение 13.50. *Создайте программу, выводящую содержимое символической ссылки, а затем – целевого файла, на который она указывает.*

```
uses linux;

var
  name,temp:array [0..1023] of char;
  kol,fd:integer;

begin
  if paramcount<>1 then
  begin
    writeln('Используйте: ',paramstr(0),' имя_ссылки');
    exit;
  end;
  temp:=paramstr(1);
  kol:=readlink(temp,name,1023);
  if kol=-1 then
  begin
    writeln('Ошибка чтения ссылки ',temp);
    exit;
  end;
  name[kol]:=#0;
  writeln('По ссылке ',paramstr(1), ' найден файл ',name);
  fd:=fdopen(name,Open_RDONLY);
  if fd=-1 then
  begin
    writeln('Ошибка открытия ',name);
    exit;
  end;
  kol:=fdread(fd,name,1024);
  while kol>0 do
  begin
    fdwrite(1,name,kol);
```

```
    kol:=fdread(fd,name,1024);
  end;
  fdclose(fd);
end.
```

13.8. Управление процессами

Упражнение 13.51. *Создайте простейший командный интерпретатор.*

```
uses dos;

var
  cmd:string;
begin
  while true do
  begin
    write('> ');
    readln(cmd);
    if cmd='exit' then
      break
    else
    begin
      cmd:='-c '+cmd;
      writeln('Введена команда ',cmd);
      exec('/bin/sh',cmd);
    end;
  end;
end.
```

Упражнение 13.52. *Создайте программу, выводящую установленные для процесса ограничения.*

```
uses linux;

const
    _SC_ARG_MAX=1;
    _SC_CHILD_MAX=2;
    _SC_CLK_TCK=3;
    _SC_NGROUPS_MAX=4;
    _SC_OPEN_MAX=5;
    _SC_STREAM_MAX=6;
    _SC_TZNAME_MAX=7;
    _SC_JOB_CONTROL=8;
    _SC_SAVED_IDS=9;
    _SC_REALTIME_SIGNALS=10;
    _SC_PRIORITY_SCHEDULING=11;
    _SC_TIMERS=12;
    _SC_ASYNCHRONOUS_IO=13;
    _SC_PRIORITIZED_IO=14;
    _SC_SYNCHRONIZED_IO=15;
    _SC_FSYNC=16;
    _SC_MAPPED_FILES=17;
    _SC_MEMLOCK=18;
    _SC_MEMLOCK_RANGE=19;
    _SC_MEMORY_PROTECTION=20;
    _SC_MESSAGE_PASSING=21;
    _SC_SEMAPHORES=22;
    _SC_SHARED_MEMORY_OBJECTS=23;
    _SC_AIO_LISTIO_MAX=24;
    _SC_AIO_MAX=25;
```

```
_SC_AIO_PRIO_DELTA_MAX=26;
_SC_DELAYTIMER_MAX=27;
_SC_MQ_OPEN_MAX=28;
_SC_MQ_PRIO_MAX=29;
_SC_VERSION=30;
_SC_PAGESIZE=31;
_SC_RTSIG_MAX=32;
_SC_SEM_NSEMS_MAX=33;
_SC_SEM_VALUE_MAX=34;
_SC_SIGQUEUE_MAX=35;
_SC_TIMER_MAX=36;
_SC_BC_BASE_MAX=37;
_SC_BC_DIM_MAX=38;
_SC_BC_SCALE_MAX=39;
_SC_BC_STRING_MAX=40;
_SC_COLL_WEIGHTS_MAX=41;
_SC_EQUIV_CLASS_MAX=42;
_SC_EXPR_NEST_MAX=43;
_SC_LINE_MAX=44;
_SC_RE_DUP_MAX=45;
_SC_CHARCLASS_NAME_MAX=46;
_SC_2_VERSION=47;
_SC_2_C_BIND=48;
_SC_2_C_DEV=49;
_SC_2_FORT_DEV=50;
_SC_2_FORT_RUN=51;
_SC_2_SW_DEV=52;
_SC_2_LOCALEDEF=53;
_SC_PII=54;
_SC_PII_XTI=55;
_SC_PII_SOCKET=56;
_SC_PII_INTERNET=57;
_SC_PII_OSI=58;
_SC_POLL=59;
_SC_SELECT=60;
_SC_UIO_MAXIOV=61;
_SC_IOV_MAX=62;
_SC_PII_INTERNET_STREAM=63;
_SC_PII_INTERNET_DGRAM=64;
_SC_PII_OSI_COTS=65;
_SC_PII_OSI_CLTS=66;
_SC_PII_OSI_M=67;
_SC_T_IOV_MAX=68;
_SC_THREADS=69;
_SC_THREAD_SAFE_FUNCTIONS=70;
_SC_GETGR_R_SIZE_MAX=71;
_SC_GETPW_R_SIZE_MAX=72;
_SC_LOGIN_NAME_MAX=73;
_SC_TTY_NAME_MAX=74;
_SC_THREAD_DESTRUCTOR_ITERATIONS=75;
_SC_THREAD_KEYS_MAX=76;
_SC_THREAD_STACK_MIN=77;
_SC_THREAD_THREADS_MAX=78;
_SC_THREAD_ATTR_STACKADDR=79;
_SC_THREAD_ATTR_STACKSIZE=80;
_SC_THREAD_PRIORITY_SCHEDULING=81;
_SC_THREAD_PRIO_INHERIT=82;
_SC_THREAD_PRIO_PROTECT=83;
```

```
_SC_THREAD_PROCESS_SHARED=84;
_SC_NPROCESSORS_CONF=85;
_SC_NPROCESSORS_ONLN=86;
_SC_PHYS_PAGES=87;
_SC_AVPHYS_PAGES=88;
_SC_ATEXIT_MAX=89;
_SC_PASS_MAX=90;
_SC_XOPEN_VERSION=91;
_SC_XOPEN_XCU_VERSION=92;
_SC_XOPEN_UNIX=93;
_SC_XOPEN_CRYPT=94;
_SC_XOPEN_ENH_I18N=95;
_SC_XOPEN_SHM=96;
_SC_2_CHAR_TERM=97;
_SC_2_C_VERSION=98;
_SC_2_UPE=99;
_SC_XOPEN_XPG2=100;
_SC_XOPEN_XPG3=101;
_SC_XOPEN_XPG4=102;
_SC_CHAR_BIT=103;
_SC_CHAR_MAX=104;
_SC_CHAR_MIN=105;
_SC_INT_MAX=106;
_SC_INT_MIN=107;
_SC_LONG_BIT=108;
_SC_WORD_BIT=109;
_SC_MB_LEN_MAX=110;
_SC_NZERO=111;
_SC_SSIZE_MAX=112;
_SC_SCHAR_MAX=113;
_SC_SCHAR_MIN=114;
_SC_SHRT_MAX=115;
_SC_SHRT_MIN=116;
_SC_UCHAR_MAX=117;
_SC_UINT_MAX=118;
_SC_ULONG_MAX=119;
_SC_USHRT_MAX=120;
_SC_NL_ARGMAX=121;
_SC_NL_LANGMAX=122;
_SC_NL_MSGMAX=123;
_SC_NL_NMAX=124;
_SC_NL_SETMAX=125;
_SC_NL_TEXTMAX=126;
_SC_XBS5_ILP32_OFF32=127;
_SC_XBS5_ILP32_OFFBIG=128;
_SC_XBS5_LP64_OFF64=129;
_SC_XBS5_LPBIG_OFFBIG=130;
_SC_XOPEN_LEGACY=131;
_SC_XOPEN_REALTIME=132;
_SC_XOPEN_REALTIME_THREADS=133;
_SC_ADVISORY_INFO=134;
_SC_BARRIERS=135;
_SC_BASE=136;
_SC_C_LANG_SUPPORT=137;
_SC_C_LANG_SUPPORT_R=138;
_SC_CLOCK_SELECTION=139;
_SC_CPUTIME=140;
_SC_THREAD_CPUTIME=141;
```

```
    _SC_DEVICE_IO=142;
    _SC_DEVICE_SPECIFIC=143;
    _SC_DEVICE_SPECIFIC_R=144;
    _SC_FD_MGMT=145;
    _SC_FIFO=146;
    _SC_PIPE=147;
    _SC_FILE_ATTRIBUTES=148;
    _SC_FILE_LOCKING=149;
    _SC_FILE_SYSTEM=150;
    _SC_MONOTONIC_CLOCK=151;
    _SC_MULTI_PROCESS=152;
    _SC_SINGLE_PROCESS=153;
    _SC_NETWORKING=154;
    _SC_READER_WRITER_LOCKS=155;
    _SC_SPIN_LOCKS=156;
    _SC_REGEXP=157;
    _SC_REGEX_VERSION=158;
    _SC_SHELL=159;
    _SC_SIGNALS=160;
    _SC_SPAWN=161;
    _SC_SPORADIC_SERVER=162;
    _SC_THREAD_SPORADIC_SERVER=163;
    _SC_SYSTEM_DATABASE=164;
    _SC_SYSTEM_DATABASE_R=165;
    _SC_TIMEOUTS=166;
    _SC_TYPED_MEMORY_OBJECTS=167;
    _SC_USER_GROUPS=168;
    _SC_USER_GROUPS_R=169;
    _SC_2_PBS=170;
    _SC_2_PBS_ACCOUNTING=171;
    _SC_2_PBS_LOCATE=172;
    _SC_2_PBS_MESSAGE=173;
    _SC_2_PBS_TRACK=174;
    _SC_SYMLOOP_MAX=175;
    _SC_STREAMS=176;
    _SC_2_PBS_CHECKPOINT=177;
    _SC_V6_ILP32_OFF32=178;
    _SC_V6_ILP32_OFFBIG=179;
    _SC_V6_LP64_OFF64=180;
    _SC_V6_LPBIG_OFFBIG=181;
    _SC_HOST_NAME_MAX=182;
    _SC_TRACE=183;
    _SC_TRACE_EVENT_FILTER=184;
    _SC_TRACE_INHERIT=185;
    _SC_TRACE_LOG=186;


type struct=record
  name:integer;
  value:string;
end;

function sysconf(name:integer):longint;cdecl;external 'c';

const
  count=20;
  mas:array [1..count] of struct=(
```

```
  (name:_SC_ARG_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_CHILD_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name: _SC_CLK_TCK;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_NGROUPS_MAX;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_OPEN_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_STREAM_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_TZNAME_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_JOB_CONTROL;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_ARG_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_SAVED_IDS;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_REALTIME_SIGNALS;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_PRIORITY_SCHEDULING;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_TIMERS;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_ASYNCHRONOUS_IO;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_PRIORITIZED_IO;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_SYNCHRONIZED_IO;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_FSYNC;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_MAPPED_FILES;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_MEMLOCK;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_MEMLOCK_RANGE;value:'максимальная длина аргументов функций
семейства exec')

  (name:_SC_MEMORY_PROTECTION;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_CHILD_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_MESSAGE_PASSING;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_SEMAPHORES;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_SHARED_MEMORY_OBJECTS;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_AIO_LISTIO_MAX;value:'максимальное количество процессов на одного
пользователя'),
```

```
  (name:_SC_AIO_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_AIO_PRIO_DELTA_MAX;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_DELAYTIMER_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_MQ_OPEN_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_MQ_PRIO_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_VERSION;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_PAGESIZE;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_RTSIG_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_SEM_NSEMS_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_SEM_VALUE_MAX;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_SIGQUEUE_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_TIMER_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_BC_BASE_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_BC_DIM_MAX;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_BC_SCALE_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_BC_STRING_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_COLL_WEIGHTS_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_EQUIV_CLASS_MAX;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_EXPR_NEST_MAX;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_LINE_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_RE_DUP_MAX;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_CHARCLASS_NAME_MAX;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_2_VERSION;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_2_C_BIND;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_2_C_DEV;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_2_FORT_DEV;value:'максимальная длина аргументов функций семейства
exec'),
```

```
  (name:_SC_2_FORT_RUN;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_2_SW_DEV;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_2_LOCALEDEF;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_PII;value:'максимальная длина аргументов функций семейства exec'),

  (name:_SC_PII_XTI;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_PII_SOCKET;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_PII_INTERNET;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_PII_OSI;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_POLL;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_SELECT;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_UIO_MAXIOV;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_IOV_MAX;value:'максимальная длина аргументов функций семейства
exec'),

  (name:_SC_PII_INTERNET_STREAM;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_PII_INTERNET_DGRAM;value:'максимальное количество процессов на
одного пользователя'),
  (name:_SC_PII_OSI_COTS;value:'максимальная длина аргументов функций
семейства exec'),
  (name:_SC_PII_OSI_CLTS;value:'максимальная длина аргументов функций
семейства exec'),

  (name:_SC_PII_OSI_M;value:'максимальная длина аргументов функций семейства
exec'),
  (name:_SC_T_IOV_MAX;value:'максимальное количество процессов на одного
пользователя'),
  (name:_SC_THREADS;value:'максимальная длина аргументов функций семейства
exec'),
  (name:;value:'максимальная длина аргументов функций семейства exec'),

  (name:;value:'максимальная длина аргументов функций семейства exec'),
  (name:;value:'максимальное количество процессов на одного пользователя'),
  (name:;value:'максимальная длина аргументов функций семейства exec'),
  (name:;value:'максимальная длина аргументов функций семейства exec'),

 );

var
  i:integer;
begin
  for i:=1 to count do
    writeln(mas[i].value,' = ',sysconf(mas[i].name));
end.
```

Упражнение 13.53. *Напишите программу, печатающую свои параметры и значение первых десяти переменных окружения.*

```
program Parameters;
uses Dos,Crt;
var
  i:Byte;
begin
  ClrScr;
  if ParamCount>1 then
   begin
    Writeln('Owner parameters:');
    For i:=1 to ParamCount do
      Writeln(Paramstr(i));
   end
  else Writeln('No owner parameters.');
  Writeln('System parameters:');
  if envcount<10 then
    for i := 1 to EnvCount do
      Writeln(EnvStr(i))
  else
    for i := 1 to 10 do
      Writeln(EnvStr(i));
end.
```

Упражнение 13.54. *Напишите программу, узнающую у системы и распечатывающую: номер процесса, номер и имя своего владельца, номер группы, название и тип терминала, на котором она работает.*

```
uses linux;

function getnamebyuid(uid:longint):string;
var
  f:text;
  s,name:string;
  n,res:integer;
begin
  assign(f,'/etc/passwd');
  reset(f);
  while not eof(f) do
  begin
    readln(f,s);
    n:=pos(':',s);
    name:=copy(s,1,n-1);
    delete(s,1,n+2);
    n:=pos(':',s);
    delete(s,n,length(s)-n+1);
    val(s,res);
    if uid=res then
    begin
      getnamebyuid:=name;
      exit;
    end;
  end;
  getnamebyuid:='';
end;

var
  pid,uid,gid:longint;
```

```
  name,term:string;
  t:termios;
begin
  pid:=getpid;
  uid:=getuid;
  gid:=getgid;
  name:=getnamebyuid(uid);
  term:=getenv('TERM');
  tcgetattr(0,t);
  writeln('Номер процесса - ',pid);
  writeln('Процесс принадлежит пользователю с номером ',uid, ' по имени
',name);
  writeln('Процесс входит в группу ',gid);
  write('Процесс выполняется на ');
  if (t.c_lflag and ICANON)=0 then
    write('не');
  writeln('каноническом терминале ',term);
end.
```

Упражнение 13.55. *Напишите программу, выборочно выводящую значение переменных окружения и устанавливающую новые значения по желанию пользователя.*

```
uses crt,dos;

var i:integer;
    s,st:array[0..200]of char;

    c:char;

function setenv(s1,s2:pchar;ower:integer):integer;cdecl;external 'c';

begin
  for i:=1 to envcount do
    writeln(envstr(i),'=',getenv(envstr(i)));

  writeln('Введите переменную окружения, но только большими буквами.');
  readln(s);

  if getenv(strpas(s))<>'' then
    writeln(getenv(strpas(s)))
  else
  begin
    writeln('Переменной окружения ',s,' не существует');
    halt;
  end;

  writeln('Хотите изменить значение ',s,' нажмите  "y" ');
  c:=readkey;
  if upcase(c)='Y' then
  begin
    writeln('Введите значение переменной');
    readln(st);
    if setenv(s,st,1)<0 then
      writeln('Значение переменной не поменялось')
    else
      writeln(setenv(s,st,1),' ',getenv(strpas(s)));
  end;
end.
```

Упражнение 13.56. *Напишите функцию* `sleep(n)`*, задерживающую выполнение программы на* `n` *секунд. Воспользуйтесь системным вызовом* `alarm(n)` *(будильник) и вызовом* `pause()`*, который задерживает программу до получения любого сигнала. Предусмотрите рестарт при получении во время ожидания другого сигнала, нежели* `SIGALRM`*. Сохраняйте заказ* `alarm`*, сделанный до вызова* `sleep` *(*`alarm` *выдает число секунд, оставшееся до завершения предыдущего заказа).*

```
uses dos,crt,sysutils,linux;

procedure handler(sig:integer);cdecl;
begin
end;

procedure sleep(count:integer);
var
  oldhandler,newhandler:sigactionrec;
begin

  writeln('count=',count);
  newhandler.handler.sh:=@handler;
  (*newhandler.sa_mask:=$ffffffff;*)
  newhandler.sa_mask:=0;

  sigaction(SIGALRM,@newhandler,@oldhandler);
  alarm(count);
  pause;
  sigaction(SIGALRM,@oldhandler,nil);
end;

begin
repeat
sleep(3);
writeln('Привет');
until keypressed;
end.
```

13.9. Программные каналы

Упражнение 13.57. *Напишите программу, определяющую, возвращает ли* `fstat` *количество байт в FIFO в качестве поля* `size` *структуры* `tstat`*.*

```
uses linux;

var
  s:tstat;

begin
  if paramcount<>1 then
  begin
    writeln('Используйте: ',paramstr(0),' имя_файла');
    exit;
  end;
  if not fstat(paramstr(1),s) then
  begin
    writeln('Ошибка вызова stat для файла ',paramstr(1));
    exit;
  end;
  writeln('Размер файла ',paramstr(1),' равен ',s.size);
end.
```

Упражнение 13.58. *Напишите программу для определения того, что возвращает функция* `select` *при проверке возможности записи в дескриптор канала, у которого закрыт второй конец.*

```
uses linux;

var
  fds:fdset;
  fdin,fdout,ret:longint;

begin
  if not assignpipe(fdin,fdout) then
  begin
    writeln('Ошибка создания программного канала');
    exit;
  end;
  fd_zero(fds);
  fd_set(fdin,fds);
  fd_set(fdout,fds);
  fdwrite(fdout,'Некая достаточно длинная строка',31);
  fdclose(fdin);
  writeln('Вызов select');
  ret:=select(2,nil,@fds,nil,1000);
  writeln('select вернул ',ret);
end.
```

Упражнение 13.59. *Используя канал между родителем (клиентом) и потомком (сервером), создайте программу, в которой клиент считывает имя файла из стандартного ввода и записывает в канал. Если файл существует, сервер считывает его и записывает в канал, в противном случае возвращает клиенту сообщение об ошибке.*

```
uses linux,sysutils;

const
  BLOCKSIZE=1024;

var
  pid,(*идентификатор процесса*)
  filesize,(*размер файла*)
  fd,(*дескриптор файла*)
  kol,(*количество прочитанных байт*)
  i:longint;
  in1,out1,in2,out2:longint;(*дескрипторы программных каналов*)
  filename:string[80];
  st:tstat;(*для получения информации о размере файла*)
  buf:array[0..BLOCKSIZE-1]of char;(*буфер чтения-записи*)

begin
  (*попытка создания двух программных каналов*)
  if not assignpipe(in1,out1) then
  begin
    writeln('Ошибка создания первого программного канала');
    exit;
  end;

  if not assignpipe(in2,out2) then
  begin
    writeln('Ошибка создания второго программного канала');
```

```
    fdclose(in1); (*закрываем ранее созданный канал 1*)
    fdclose(out1);
    exit;
  end;

  pid:=fork; (*клонирование процесса*)

  if pid=-1 then (*ошибка клонирования*)
  begin
    writeln('Ошибка создания потомка');
    exit;
  end;
  if pid=0 (*сервер - потомок*) then (*ветка потомка*)
  begin
    fdclose(out1);(*закрываем для надежности ненужные каналы:
                    первый - для записи, второй - для чтения*)
    fdclose(in2);
    fdread(in1,filename,80); (*читаем из первого канала имя файла*)
    {}writeln('Сервер: получено имя файла - ',filename);
    if access(filename,f_ok or r_ok) then (*если файл существует и доступен
для чтения*)
    begin
      fstat(filename,st);(*получем информацию о файле*)
      filesize:=st.size; (*узнаем размер файла*)
      {}writeln('Сервер: определен размер файла - ',filesize);
      fdwrite(out2,filesize,sizeof(filesize)); (*пишем во второй канал размер
файла*)
      {}writeln('Сервер: размер файла записан в канал');
      fd:=fdopen(filename,Open_RDONLY);(*открываем файл для чтения*)
      i:=1;
      kol:=fdread(fd,buf,BLOCKSIZE); (*читаем блоками по BLOCKSIZE байт*)
      while kol>0 do
      begin
        fdwrite(out2,buf,kol); (*пишем во второй канал столько, сколько прочли
из файла*)
        {}writeln('Сервер: записано в канал ',i*kol/BLOCKSIZE:1:1,' Kb');
        kol:=fdread(fd,buf,BLOCKSIZE);
      end;
      fdclose(fd);(*закрываем файл*)
      {}writeln('Сервер: файл записан в канал');
    end
    else (*если файл не существует или недоступен для чтения*)
    begin
      filesize:=-1; (*записываем в программный канал признак ошибки*)
      fdwrite(out2,filesize,sizeof(filesize));
    end;
    {}writeln('Сервер: работа завершена');
    halt(0); (*завершаем работу потомка*)
  end
  else     (*клиент - родитель*)  (*ветка родителя*)
  begin
    fdclose(in1); (*закрываем ненужные каналы: первый для чтения, второй - для
записи*)
    fdclose(out2);
    write('Введите имя файла: '); (*запрос имени файла*)
    readln(filename);
    fdwrite(out1,filename,80); (*пишем имя в программный канал*)
    {}writeln('Клиент: имя файла записано в канал');
```

```
    fdread(in2,filesize,sizeof(filesize)); (*получем размер файла из канала*)
    {}writeln('Клиент: из канала получен размер файла - ',filesize);
    if in2=-1 then (*при ошибке*)
      writeln('Файл ',filename,' не существует или недоступен для чтения')
    else
    begin
      (*создаем файл*)
      {}writeln('Клиент: файл создан, идет прием');
      for i:=length(filename) downto 1 do
        if filename[i]='/' then
        begin
          delete(filename,1,i);
          break;
        end;
      fd:=fdopen(filename,Open_WRONLY    or    Open_CREAT    or    Open_TRUNC,
octal(644));
      if fd=-1 then
      begin
        writeln('Ошибка создания файла ',filename);
        kill(pid,9);
        halt(1);
      end;

      for i:=1 to filesize do(*пока есть что читать из канала*)
      begin
        fdread(in2,buf,1);
        if i mod BLOCKSIZE=0 then
        {}writeln('Клиент: записано в файл ',i div BLOCKSIZE,' Kb');
        fdwrite(fd,buf,1);(*записываем в файл*)
      end;
      fdclose(fd);(*закрываем созданный файл*)
      {}writeln('Клиент: файл закрыт');
    end;
    waitpid(pid,nil,0); (*ожидаем завершения потомка*)
    {}writeln('Клиент: сервер завершен, конец работы');
  end;
end.
```

Упражнение 13.60. *Используя* `popen`*, создайте канал между* `who` *и* `more`.

```
uses linux;

var
  f1,f2:text;
  s:string;
begin
  popen(f1,'who','r');
  if linuxerror <> 0 then
  begin
    writeln('Ошибка открытия канала с who для чтения');
    exit;
  end;
  popen(f2,'more','w');
  if linuxerror <> 0 then
  begin
    writeln('Ошибка открытия канала с more для записи');
    exit;
  end;
  while not eof(f1) do
```

```
  begin
    readln(f1,s);
    writeln(f2,s);
  end;
  pclose(f1);
  pclose(f2);
end.
```

13.10. Управление терминалом

Упражнение 13.61. *Напишите функции включения и выключения режима эхо-отображения набираемых на клавиатуре символов.*

```
uses linux,crt;

const ECHO=8;

var
t:termios;
s,ms:string;
c:char;
begin
writeln('Для включения эхо введите on, для выключения эхо введите off');
readln(s);
tcgetattr(0,t);

writeln(t.c_lflag);
if s='on' then
    begin
     t.c_lflag:=t.c_lflag or ECHO;
    end;
if s='off' then
    begin
     t.c_lflag:=t.c_lflag and not ECHO;
    end;

tcsetattr(0,tcsanow,t);

readln(ms);
writeln(ms);
end.
```

13.11. Дата и время

Упражнение 13.62. *Составьте аналог команды* `date`.

```
ses crt,dos;
var y,h,d,dw,ch,m,s,ms:word;
BEGIN
getdate(y,h,d,dw);
case dw of
  1:write('Пн');
  2:write('Вт');
  3:write('Ср');
  4:write('Чт');
  5:write('Пт');
  6:write('Сбт');
  7:write('Вс');
end;
write(' ');
```

```
case h of
  1:write('Янв');
  2:write('Фев');
  3:write('Мар');
  4:write('Апр');
  5:write('Май');
  6:write('Июн');
  7:write('Июл');
  8:write('Авг');
  9:write('Сен');
  10:write('Окт');
  11:write('Ноя');
  12:write('Дек');
end;
write(' ');
write(d,' ');
gettime(ch,m,s,ms);
write(ch,':',m,':',s,':',ms,' ',y);
readln;

END.
```

Упражнение 13.63. *Составьте аналог команды* `cal`.

```
uses crt,dos;
procedure cl(year,mes,pol_1:word);
var arr:array [0..7,1..7] of string[2];
    i,j,kdm,n:byte;
    v:boolean;
    kd,k:word;
    s:string[2];
begin
   v:=false;
    for i:=0 to 7 do
       for j:=1 to 7 do
        arr[i,j]:='  ';
   if  ((year  mod  4=0)and(year  mod  100<>0))or((year  mod  4=0)and(year  mod
100=0)and(year mod 400=0))
                                        then v:=true
         else if (year mod 100=0)and(year mod 400<>0)then v:=false;

    case mes of
  1:begin writeln('Янв');
          kdm:=31;
        kd:=0;
    end;
  2:begin writeln('Фев');
         kd:=31;
       if v then kdm:=29 else kdm:=28;
    end;
  3:begin writeln('Мар');
          kdm:=31;
       if v then kd:=60 else kd:=59;
    end;
  4:begin writeln('Апр');
          kdm:=30;
       if v then kd:=91 else kd:=90;
    end;
  5:begin writeln('Май');
```

```
        kdm:=31;
      if v then kd:=121 else kd:=120;
  end;
6:begin writeln('Июн');
        kdm:=30;
      if v then kd:=152 else kd:=151;
  end;
7:begin writeln('Июл');
        kdm:=31;
      if v then kd:=182 else kd:=181;
  end;
8:begin writeln('Авг');
        kdm:=31;
      if v then kd:=213 else kd:=212;
  end;
9:begin writeln('Сен');
        kdm:=30;
      if v then kd:=244 else kd:=243;
  end;
10:begin writeln('Окт');
        kdm:=31;
      if v then kd:=274 else kd:=273;
  end;
11:begin writeln('Ноя');
         kdm:=30;
       if v then kd:=305 else kd:=304;
  end;
12:begin writeln('Дек');
        kdm:=31;
      if v then kd:=335 else kd:=334;
    end;
end;
arr[0,1]:='ПН';
arr[0,2]:='ВТ';
arr[0,3]:='СР';
arr[0,4]:='ЧТ';
arr[0,5]:='ПТ';
arr[0,6]:='СБ';
arr[0,7]:='ВС';

pol_1:=(pol_1+kd) mod 7;
k:=pol_1+1;
n:=1;
for j:=k to 7 do
 begin
  str(n,s);
  arr[1,j]:=s;
  n:=n+1;
 end;
for i:=2 to 7 do
 for j:=1 to 7 do
  begin
  if n<=kdm then begin //n:=n+1;
                       str(n,s);
                  n:=n+1;
             end
              else s:='  ';
  arr[i,j]:=s;
```

```
  end;
  for i:=0 to 7 do
   begin
   for j:=1 to 7 do
    write(arr[i,j]:2,' ');
    writeln;
   end;


end;

var
  y,m,mm,d,dw:word;
  kvy,kd:longint;
  err:integer;
  ii,jj,x1,x2,y1,y2:byte;
begin
  m:=0;
  if paramcount=0 then getdate(y,m,d,dw);
  if paramcount=1 then val(paramstr(1),y,err);
  if paramcount=2 then begin
                       val(paramstr(1),y,err);
                   val(paramstr(2),m,err);
                       end;
  if err<>0 then begin
                  writeln('error');
              exit;
                 end;
  kvy:=((y-1) div 4)-((y-1) div 100)+((y-1) div 400);
  kd:=365*(y-1-kvy)+366*kvy;{kol dney do jen year}
  dw:=kd mod 7;
  gotoxy(1,1);
  mm:=0;

  y1:=1;
  if m=0 then for ii:=0 to 2 do
              begin
              y2:=y1+10;
            x1:=1;
               for jj:=0 to 3 do
                begin
             x2:=x1+20;
             mm:=mm+1;
             window(x1,y1,x2,y2);
             cl(y,mm,dw);
             x1:=x1+24;
            end;
             y1:=y1+10;
              end;
  if (m>0)and (m<=12) then cl(y,m,dw);
  if y<0 then exit;
  window(1,31,25,32);
  writeln;

end.
```

Упражнение 13.64. *Напишите функцию, переводящую год, месяц, день, часы, минуты и секунды в число секунд, прошедшее до указанного момента с 00 часов 00 минут 00*

секунд 1 Января 1970 года.

```
uses sysutils;

var
   s1,s2,s3,s4,s5,s6:integer;

   rezult:real;

function kol_days_y(y,m,d:integer):longint;
var i,god,sum:integer;
begin
sum:=0;

{Подсчет дней при прошедших годах}
god:=1970;
repeat
if y=god then begin break;end;
if (god mod 400<>0)and((god mod 4) =0)and((god mod 100)<>0)then sum:=sum+366
else
sum:=sum+365;
god:=god+1;
until god>y;

{Подсчет дней при определенном количестве месяцев}

god:=1;
repeat
if god=m then begin break;end;
case god of
1,3,5,7,8,10,12:begin sum:=sum+31;end;
4,6,9,11:sum:=sum+30;
2:begin
  if ((y mod 4) =0)and((y mod 100)<>0)then sum:=sum+29 else
  sum:=sum+28;
  end;
end;
god:=god+1;
until god>m;

kol_days_y:=sum+d-1;{Начало дней не с нуля, поэтому и -1}
end;


begin
if paramcount<>6 then
 begin
  writeln('Введите правильно дату и время. Например: 2004 2 20 12 45 55');
  halt(1);
 end else

begin
s1:=strtoint(paramstr(1));
s2:=strtoint(paramstr(2));
s3:=strtoint(paramstr(3));
s4:=strtoint(paramstr(4));
s5:=strtoint(paramstr(5));
s6:=strtoint(paramstr(6));
```

```
 if (s1<0)or(s2<0)or(s3<0)or(s4<0)or(s5<0)or(s6<0)then
 begin
  writeln('Введите положительные значения');
  halt(1);
 end;

 if (s2<1)or(s2>12)then
 begin
  writeln('Введите существующий месяц');
  halt(1);
 end;
 if (s3<1)or(s3>31)then
 begin
  writeln('Введите существующий день');
  halt(1);
 end;
 if (s4<0)or(s4>23)then
 begin
  writeln('Введите существующее количество часов');
  halt(1);
 end;
 if (s5<0)or(s5>59)then
 begin
  writeln('Введите существующее количество минут');
  halt(1);
 end;
 if (s6<0)or(s6>59)then
 begin
  writeln('Введите существующее количество секунд');
  halt(1);
 end;
end;

case s2 of
2:begin
   if ((s1 mod 4)=0)and((s1 mod 4)=0)then
   begin
   if s3>29 then begin
             writeln('В высокосном году в феврале не более 29 дней');
             halt(1);
             end;
   end
   else
   if s3>28 then
        begin
      writeln('В феврале в невысокосном году не более 28 дней');
      halt(1);
      end;

  end;
4,6,9,11: if s3>30 then
          begin
          writeln('В ',s2,' месяце не может быть больше 30 дней');
          halt(1);
          end;
end;

if s1<=1969 then
```

```
 begin
 writeln('Время назад не идет');
 halt(1);
 end;

writeln('Количество секунд прошедшее до указаного момента = ');
writeln(' = ',kol_days_y(s1,s2,s3)*24*3600+s4*3600+s5*60+s6,'c.');

end.
```

Упражнение 13.65. *Напишите "часы", выдающие текущее время каждые 3 секунды.*

```
uses dos,crt,sysutils,linux;
var
  Hour   , Min  , Sec  , HSec     :  word   ;

procedure handler(sig:longint);cdecl;
begin
end;

procedure sleep(count:integer);
var
  old:signalhandler;
begin
  old:=signal(SIGALRM,@handler);
  alarm(count);
  pause;
  signal(SIGALRM,old);
  alarm(0);
end;


function L0  (w : word   ) : string;
               var
                  s   :  string;
               begin
                  Str( w , s );
                  if  w<10    then
                    L0 :=  '0'+ s
                  else
                    L0 :=  s ;
               end  ;

var
  s:string;
  i:integer;
begin
 WriteLn     ( 'По системным часам, через каждые 3 секунды') ;
 repeat
  GetTime     ( Hour   , Min  , Sec  , HSec   ) ;
  s := L0  (Hour    ) + ' : ' + L0 ( Min  )  + ' : ' + L0 ( Sec  )  ;
  write(s);
  sleep(3);
  for i:=1 to length(s) do
    write(#8);
 until keypressed;
end.
```

13.12. Генератор лексических анализаторов lex

Упражнение 13.66. *Составьте вариант программы подсчета служебных слов языка Си, не учитывающий появление этих слов, заключенных в кавычки.*

```
%{
uses lexlib;
const
_AND=1;
_ASM=2;
_ARRAY=3;
_BEGIN=4;
_CASE=5;
_CONST=6;
_CONSTRUCTOR=7;
_DESTRUCTOR=8;
_DIV=9;
_DO=10;
_DOWNTO=11;
_ELSE=12;
_END=13;
_EXPORTS=14;
_FILE=15;
_FOR=16;
_FUNCTION=17;
_GOTO=18;
_IF=19;
_IMPLEMENTATION=20;
_IN=21;
_INHERITED=22;
_INLINE=23;
_INTERFACE=24;
_LABEL=25;
_LIBRARY=26;
_MOD=27;
_NIL=28;
_NOT=29;
_OBJECT=30;
_OF=31;
_OR=32;
_PACKED=33;
_PROCEDURE=34;
_PROGRAM=35;
_RECORD=36;
_REPEAT=37;
_SET=38;
_SHL=39;
_SHR=40;
_STRING=41;
_THEN=42;
_TO=43;
_TYPE=44;
_UNIT=45;
_UNTIL=46;
_USES=47;
_VAR=48;
_WHILE=49;
_WITH=50;
_XOR=51;
```

```
_STR=52;
_IDENT=53;

%}
letter  [a-zA-Z]
digit   [0-9]
%%
'[^']*'            begin yydone:=true; yyretval:=_STR; end;
\"[^\"]\"   begin yydone:=true; yyretval:=_STR; end;
"end."          begin yydone:=true; yyretval:=_END; end;
and             begin yydone:=true; yyretval:=_AND; end;
asm             begin yydone:=true; yyretval:=_ASM; end;
array           begin yydone:=true; yyretval:=_ARRAY; end;
begin           begin yydone:=true; yyretval:=_BEGIN; end;
case            begin yydone:=true; yyretval:=_CASE; end;
const           begin yydone:=true; yyretval:=_CONST; end;
constructor     begin yydone:=true; yyretval:=_CONSTRUCTOR; end;
destructor      begin yydone:=true; yyretval:=_DESTRUCTOR; end;
div             begin yydone:=true; yyretval:=_DIV; end;
do              begin yydone:=true; yyretval:=_DO; end;
downto          begin yydone:=true; yyretval:=_DOWNTO; end;
else            begin yydone:=true; yyretval:=_ELSE; end;
end             begin yydone:=true; yyretval:=_END; end;
exports         begin yydone:=true; yyretval:=_EXPORTS; end;
file            begin yydone:=true; yyretval:=_FILE; end;
for             begin yydone:=true; yyretval:=_FOR; end;
function        begin yydone:=true; yyretval:=_FUNCTION; end;
goto            begin yydone:=true; yyretval:=_GOTO; end;
if              begin yydone:=true; yyretval:=_IF; end;
implementation  begin yydone:=true; yyretval:=_IMPLEMENTATION; end;
in              begin yydone:=true; yyretval:=_IN; end;
inherited       begin yydone:=true; yyretval:=_INHERITED; end;
inline          begin yydone:=true; yyretval:=_INLINE; end;
interface       begin yydone:=true; yyretval:=_INTERFACE; end;
label           begin yydone:=true; yyretval:=_LABEL; end;
library         begin yydone:=true; yyretval:=_LIBRARY; end;
mod             begin yydone:=true; yyretval:=_MOD; end;
nil             begin yydone:=true; yyretval:=_NIL; end;
not             begin yydone:=true; yyretval:=_NOT; end;
object          begin yydone:=true; yyretval:=_OBJECT; end;
of              begin yydone:=true; yyretval:=_OF; end;
or              begin yydone:=true; yyretval:=_OR; end;
packed          begin yydone:=true; yyretval:=_PACKED; end;
procedure       begin yydone:=true; yyretval:=_PROCEDURE; end;
[pP][rR][oO][gG][rR][aA][mM]     begin yydone:=true; yyretval:=_PROGRAM; end;
record          begin yydone:=true; yyretval:=_RECORD; end;
repeat          begin yydone:=true; yyretval:=_REPEAT; end;
set             begin yydone:=true; yyretval:=_SET; end;
shl             begin yydone:=true; yyretval:=_SHL; end;
shr             begin yydone:=true; yyretval:=_SHR; end;
string          begin yydone:=true; yyretval:=_STRING; end;
then            begin yydone:=true; yyretval:=_THEN; end;
to              begin yydone:=true; yyretval:=_TO; end;
type            begin yydone:=true; yyretval:=_TYPE; end;
unit            begin yydone:=true; yyretval:=_UNIT; end;
until           begin yydone:=true; yyretval:=_UNTIL; end;
uses            begin yydone:=true; yyretval:=_USES; end;
var             begin yydone:=true; yyretval:=_VAR; end;
```

```
while           begin yydone:=true; yyretval:=_WHILE; end;
with            begin yydone:=true; yyretval:=_WITH; end;
xor             begin yydone:=true; yyretval:=_XOR; end;

{letter}({letter}|{digit})*     begin yydone:=true; yyretval:=_IDENT; end;
\n      ;
.       ;
%%

function yywrap:integer;
begin
   yywrap:=1;
end;


const
  words:array[_AND.._XOR] of string=(
  'AND',
  'ASM',
  'ARRAY',
  'BEGIN',
  'CASE',
  'CONST',
  'CONSTRUCTOR',
  'DESTRUCTOR',
  'DIV',
  'DO',
  'DOWNTO',
  'ELSE',
  'END',
  'EXPORTS',
  'FILE',
  'FOR',
  'FUNCTION',
  'GOTO',
  'IF',
  'IMPLEMENTATION',
  'IN',
  'INHERITED',
  'INLINE',
  'INTERFACE',
  'LABEL',
  'LIBRARY',
  'MOD',
  'NIL',
  'NOT',
  'OBJECT',
  'OF',
  'OR',
  'PACKED',
  'PROCEDURE',
  'PROGRAM',
  'RECORD',
  'REPEAT',
  'SET',
  'SHL',
  'SHR',
  'STRING',
```

```
  'THEN',
  'TO',
  'TYPE',
  'UNIT',
  'UNTIL',
  'USES',
  'VAR',
  'WHILE',
  'WITH',
  'XOR'  );

var
  kwcount:array [1..51] of integer;
  token,i:integer;

begin
  if paramcount<>1 then
  begin
    writeln('Используйте: ', paramstr(0), ' файл');
    exit;
  end;

  assign(yyinput,paramstr(1));
  {$I-}
  reset(yyinput);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия для чтения ', paramstr(1));
    exit;
  end;

  for i:=_AND to _XOR do
    kwcount[i]:=0;

  token:=yylex;
  while not eof(yyinput) do
  begin
    if token in [_AND.._XOR] then
      inc(kwcount[token]);
    token:=yylex;
  end;

  if token in [_AND.._XOR] then
    inc(kwcount[token]);

  for i:=_AND to _XOR do
    if kwcount[i]<>0 then
      writeln('Слово ', words[i], ' встречается ', kwcount[i], ' раз');
 close(yyinput);
end.
```

Упражнение 13.67. *Составьте программу удаления из программы на языке Си всех комментариев. Обратите внимание на особые случаи со строками в кавычках и символьными константами; так строка* `char s[] = "/*";` *не является началом комментария! Комментарии записывайте в отдельный файл.*

```
%{
uses lexlib;
```

```
  const
    STROKA=1;
    COMMENT=2;
    ANY=3;
    ENTER=4;
%}

%%
'[^']*'           begin yyretval:=STROKA; yydone:=true; end;
\"[^"]*\"   begin yyretval:=STROKA; yydone:=true; end;
"end."            begin yyretval:=STROKA; yydone:=true; end;
\n          begin yyretval:=ENTER;  yydone:=true; end;
.           begin yyretval:=ANY;    yydone:=true; end;
"{"[^"}"]*"}" begin yyretval:=COMMENT;yydone:=true; end;
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/" begin
  yyretval:=COMMENT;yydone:=true;
end;
"(*""("*([^*)]|[^*]")"|"*"[^)])*"*"*"*)" begin
  yyretval:=COMMENT;yydone:=true;
end;
"//"[^\n]*\n      begin yyretval:=COMMENT;yydone:=true; end;
%%

(*
function yywrap:boolean;
begin
  writeln('end?');
  yywrap:=true;
end;
*)

var
  token:integer;
  fresult,fcomment:text;


begin
  if paramcount<>3 then
  begin
    writeln('Используйте: ', paramstr(0), ' исходный_файл результирующий_файл
извлеченные_комментарии');
    exit;
  end;

  assign(yyinput,paramstr(1));
  {$I-}
  reset(yyinput);
  if ioresult<>0 then
  begin
    writeln('Ошибка открытия для чтения ', paramstr(1));
    exit;
  end;

  assign(fresult,paramstr(2));
  rewrite(fresult);
  if ioresult<>0 then
  begin
    writeln('Ошибка создания ', paramstr(2));
```

```
    exit;
  end;

  assign(fcomment,paramstr(3));
  rewrite(fcomment);
  if ioresult<>0 then
  begin
    writeln('Ошибка создания ', paramstr(3));
    exit;
  end;

  token:=yylex;
  while not eof(yyinput) do
  begin
    if token=COMMENT then
      writeln(fcomment,yytext)
    else
      write(fresult,yytext);
    token:=yylex;
  end;
  write(fresult,yytext);
  close(yyinput);
  close(fcomment);
  close(fresult);
end.
```

Приложение 1. Коды ошибок переменной linuxerror и связанные с ними сообщения

Введение

Как уже упоминалось в главе 2, ОС UNIX обеспечивает набор стандартных кодов ошибок и сообщений, описывающих ошибки. Ошибки генерируются при неудачном завершении системных вызовов; С каждым типом ошибки системного вызова связан номер ошибки, мнемонический код (константа, имеющая значение номера ошибки) и строка сообщения. Эти объекты можно использовать в программе, если включить в нее модуль `linux`.

В случае возникновения ошибки системный вызов устанавливает новое значение переменной `linuxerror`. Почти всегда системный вызов сообщает об ошибке, возвращая вызывающему процессу в качестве результата величину –1. После этого можно проверить соответствие значения переменной `linuxerror` мнемоническим кодам, определенным в файле `linux`, например:

```
uses linux;

var
  pid:longint;
.
.
.
pid := fork;
if pid = -1 then
begin
  if linuxerror = Sys_EAGAIN then
    writeln('Превышен предел числа процессов')
  else
  writeln('Другая ошибка');
end;
```

Внешний массив `sys_errlist` является таблицей сообщений об ошибках, выводимых процедурой `perror`. Переменная `linuxerror` может использоваться в качестве индекса этого массива, если нужно вручную получить системное сообщение об ошибке. Внешняя целочисленная переменная `sys_nerr` задает текущий размер таблицы `sys_errlist`. Для получения индекса в массиве следует всегда проверять, что значение переменной `linuxerror` меньше значения `sys_nerr`, так как новые номера ошибок могут вводиться с опережением соответствующего пополнения таблицы системных сообщений.

Список кодов и сообщений об ошибках

Ниже приведен список сообщений об ошибках системных вызовов. Он основан на информации выпуска 4.2 стандарта X/Open System Interfaces Standard (стандарта системных интерфейсов X/Open). Каждый пункт списка озаглавлен мнемоническим сокращением имени ошибки, определенным соответствующим кодом ошибки в файле `linuxerror`, и содержит системное сообщение об ошибке из таблицы `sys_errlist`, а также краткое ее описание. Обратите внимание, что текст сообщения об ошибке может меняться в зависимости от установки параметра `LC_MESSAGES` текущей *локализации* (locale).

`Sys_E2BIG` — *Слишком длинный список аргумента* (Argument list too long). Чаще всего означает, что вызову `exec` передается слишком длинный (согласно полному числу байтов) список аргументов

`Sys_EACCES` — *Нет доступа* (Permission denied). Произошла ошибка, связанная с отсутствием прав доступа. Может

	происходить при выполнении системных вызовов `fdopen`, `link`, `fdcreat` и аналогичных им. Может также генерироваться вызовом `exec`, если отсутствует доступ на выполнение
`Sys_EADDRINUSE`	*Адрес уже используется* (Address in use). Означает, что запрошенный программой адрес уже используется
`Sys_EADDRNOTAVAIL`	*Адрес недоступен* (Address not available). Эта ошибка может возникать, если программа запрашивает адрес, который уже используется процессом
`Sys_Sys_EAFNOSUPPORT`	*Семейство адресов не поддерживается* (Address family not supported). При использовании интерфейса вызова сокетов было задано семейство адресов, которое не поддерживается системой
`Sys_EAGAIN`	*Ресурс временно недоступен, следует повторить попытку позже* (Resource temporarily unavailable, try again later). Обычно означает переполнение некоторой системной таблицы. Эта ошибка может генерироваться вызовом `fork` (если слишком много процессов) и вызовами межпроцессного взаимодействия (если слишком много объектов одного из типов межпроцессного взаимодействия)
`Sys_EALREADY`	*Соединение устанавливается* (Connection already in progress). Означает отказ при попытке установления соединения из-за того, что этот сокет уже находится в состоянии установления соединения
`Sys_EBADF`	*Недопустимый дескриптор файла* (Bad file descriptor). Файловый дескриптор не соответствует открытому файлу или же установленный режим доступа (только для чтения или только для записи) не позволяет выполнить нужную операцию. Генерируется многими вызовами, например, `fdread` и `fdwrite`
`Sys_EBADMSG`	*Недопустимое сообщение* (Bad message). (Данная ошибка связана с архитектурой модулей STREAM.) Генерируется, если системный вызов получает сообщение потока, которое не может прочитать. Может, например, генерироваться, если вызов `fdread` был выполнен для чтения в обход модуля STREAM, и в результате было получено управляющее сообщение STREAM, а не сообщение с данными
`Sys_EBUSY`	*Занято устройство или ресурс* (Device or resource busy). Может генерироваться, например, если вызов `rmdir` пытается удалить каталог, который используется другим процессом
`Sys_ECHILD`	*Нет дочерних процессов* (No child processes). Был выполнен вызов `wait` или `waitpid`, но соответствующий дочерний процесс не существует
`Sys_ECONNABORTED`	*Соединение разорвано* (Connection aborted). Сетевое соединение было разорвано по неопределенной причине
`Sys_ECONNREFUSED`	*Отказ в установке соединения* (Connection refused). Очередь запросов переполнена или ни один процесс не принимает запросы на установку соединения

`Sys_ECONNRESET`	*Сброс соединения* (Connection reset). Соединение было закрыто другим процессом
`Sys_EDEADLK`	*Предотвращен клинч* (Resource deadlock would occur). В случае успешного выполнения вызова произошел бы клинч (то есть ситуация, когда два процесса ожидали бы действия друг от друга). Эта ошибка может возникать в результате вызовов `fcntl` и `lockf`
`Sys_EDESTADDRREQ`	*Требуется адрес назначения* (Destination request required). При выполнении операции с сокетом был опущен адрес назначения
`Sys_EDOM`	*Ошибка диапазона* (Domain error). Ошибка пакета математических процедур. Означает, что аргумент функции выходит за границы области определения этой функции. Может возникать во время выполнения функций семейств `trig`, `exp`, `gamma` и др.
`Sys_EDQUOT`	*Зарезервирован* (Reserved)
`Sys_EEXIST`	*Файл уже существует* (File exists). Файл уже существует, и это препятствует успешному завершению вызова. Может возникать во время выполнения вызовов `link`, `mkdir`, `mkfifo`, `shmget` и `fdopen`
`Sys_EFAULT`	*Недопустимый адрес* (Bad address). Генерируется системой после ошибки защиты памяти. Обычно означает, что был задан некорректный адрес памяти. Не все системы могут отслеживать ошибки памяти и сообщать о них процессам
`Sys_EFBIG`	*Слишком большой файл* (File too large). При попытке увеличить размер файла было превышено максимальное значение размера файла для процесса (установленное вызовом `ulimit`) или общесистемное максимальное значение размера файла
`Sys_EHOSTUNREACH`	*Компьютер недоступен* (Host is unreachable). Генерируется сетью, если компьютер выключен или недоступен для маршрутизатора
`Sys_EIDRM`	*Идентификатор удален* (Identifier removed). Означает, что идентификатор межпроцессного взаимодействия, например, идентификатор разделяемой памяти, был удален из системы при помощи команды `ipcrm`
`Sys_EILSEQ`	*Недопустимая последовательность байтов* (Illegal byte sequence). Недопустимый символ (не все возможные «широкие» символы являются допустимыми). Эта ошибка может возникать во время вызова `fprintf` или `fscanf`
`Sys_EINPROGRESS`	*Соединение в процессе установки* (Connection in progress). Означает, что вызов запроса соединения принят и будете выполнен. Данный код выставляется при вызове `connect` с флагом `Open_NONBLOCK`
`Sys_EINTR`	*Прерванный вызов функции* (Interrupted function call). Возвращается при поступлении сигнала во время выполнения системного вызова. (Возникает только во время выполнения некоторых вызовов – обратитесь к документации системы)

Sys_EINVAL	*Недопустимый аргумент* (Invalid argument). Означает, что системному вызову был передан недопустимый параметр или список параметров. Может генерироваться вызовами fcntl, sigaction, некоторыми процедурами межпроцессного взаимодействия, а также математическими функциями
Sys_EIO	*Ошибка ввода/вывода* (I/O error). Во время ввода/вывода произошла физическая ошибка
Sys_EISCONN	*Сокет подключен* (Socket is connected). Этот сокет уже соединен
Sys_EISDIR	*Это каталог* (Is a directory). Была выполнена попытка открыть каталог для записи. Эта ошибка генерируется вызовами fdopen, fdread или frename
Sys_ELOOP	*Слишком много уровней символьных ссылок* (Too many levels of symbolic links). Возвращается, если системе приходится обойти слишком, много символьных ссылок при попытке найти файл в каталоге. Эта ошибка может генерироваться любым системным вызовом, принимающим в качестве параметра имя файла
Sys_EMFILE	*Слишком много открытых процессом файлов* (Too many open files in a process). Происходит в момент открытия файла и означает, что процессом открыто максимально возможное число файлов, заданное постоянной OPEN_MAX в файле <limits.h>
Sys_EMLINK	*Слишком много ссылок* (Too many links). Генерируется вызовом link, если с файлом связано максимально возможное число жестких ссылок, заданное постоянной LINK_MAX в файле <limits.h>
Sys_EMSGSIZE	*Слишком большое сообщение* (Message too large). Генерируется в сети, если посланное сообщение слишком велико, чтобы поместиться во внутреннем буфере приемника
Sys_EMULTIHOP	*Зарезервирован* (Reserved)
Sys_ENAMETOOLONG	*Слишком длинное имя файла* (Filename too long). Может означать, что имя файла длиннее NAME_MAX или полное маршрутное имя файла превышает значение PATH_MAX. Выставляется любым системным вызовом, принимающим в качестве параметра имя файла или полное маршрутное имя
Sys_ENETDOWN	Сеть не работает (Network is down)
Sys_ENETUNREACH	*Сеть недоступна* (Network unreachable). Путь к указанной сети недоступен
Sys_ENFILE	*Переполнение таблицы файлов* (File table overflow). Генерируется вызовами, которые возвращают дескриптор открытого файла (такими, как fdcreat, fdopen и pipe). Это означает, что внутренняя таблица ядра переполнена, и нельзя открыть новые дескрипторы файлов
Sys_ENOBUFS	*Нет места в буфере* (No buffer space is available). Относится к сокетам. Это сообщение об ошибке выводится, если при выполнении любого из вызовов, работающих с сокетами, система не способна нарастить

	буферы данных
Sys_ENODATA	*Сообщение отсутствует* (No message available). (Данная ошибка связана с архитектурой модулей STREAM.) Возвращается вызовом fdread, если в модуле STREAM нет сообщений
Sys_ENODEV	*Устройство не существует* (No such device). Была сделана попытка выполнить недопустимый системный вызов для устройства (например, чтение для устройства, открытого только для записи)
Sys_ENOENT	*Файл или каталог не существует* (No such file or directory). Эта ошибка происходит, если файл, заданный полным маршрутным именем (например, при выполнении вызова fdopen), не существует или не существует один из каталогов в пути
Sys_ENOEXEC	*Ошибка формата Exec* (Exec format error). Формат запускаемой программы не является допустимым форматом исполняемой программы. Эта ошибка возникает во время вызова exec
Sys_ENOLCK	*Нет свободных блокировок* (No locks available). Больше нет свободных блокировок, которые можно было бы установить при помощи вызова fcntl
Sys_ENOLINK	*Зарезервирован* (Reserved)
Sys_ENOMEM	*Нет места в памяти* (Not enough space). Ошибка нехватки памяти происходит, если процессу требуется больше памяти, чем может обеспечить система. Может генерироваться вызовами exec, fork и процедурами brk и sbrk, которые используются библиотекой управления динамической памятью
Sys_ENOMSG	*Нет сообщений нужного типа* (No message of the desired type). Возвращается, если вызов msgrcv не может найти сообщение нужного типа в очереди сообщений
Sys_ENOPROTOOPT	*Протокол недоступен* (Protocol not available). Запрошенный протокол не поддерживается системным вызовом socket
Sys_ENOSPC	*Исчерпано свободное место на устройстве* (No space left on device). Устройство заполнено, и увеличение размера файла или создание элемента каталога невозможно. Может генерироваться вызовами write, fdcreat, fdopen, mknod или link
Sys_ENOSR	*Нет ресурсов потоков* (No streams resources). (Данная ошибка связана с архитектурой модулей STREAM.) Временное состояние, о котором сообщается, если ресурсы памяти модуля STREAM не позволяют в данный момент передать сообщение
Sys_ENOSTR	*Это не STREAM* (not a STREAM). (Данная ошибка связана с архитектурой модулей STREAM) Возвращается, если функция работы с модулем STREAM, такая как операция «push» функции ioctl, вызывается для устройства, которое не является устройством, представленным модулями STREAM
Sys_ENOSYS	*Функция не реализована* (Function not implemented).

	Означает, что запрошен системный вызов, не реализованный в данной версии системы
`Sys_ENOTCONN`	*Сокет не подключен* (Socket not connected). Эта ошибка генерируется, если для неподключенного сокета выполняется вызов `sendmsg` или `rcvmsg`
`Sys_ENOTDIR`	*Это не каталог* (Not a directory). Возникает, если путь не представляет имя каталога. Может устанавливаться вызовами `chdir`, `mkdir`, `link` и многими другими
`Sys_ENOTEMPTY`	*Каталог не пуст* (Directory not empty). Возвращается, например, вызовом `rmdir`, если делается попытка удалить непустой каталог
`Sys_ENOTSOCK`	*Это не сокет* (Not a socket). Дескриптор файла, используемый в вызове для работы с сетью, например, вызове `connect`, не является дескриптором сокета
`Sys_ENOTTY`	*Не символьное устройство* (Not a character device). Был выполнен вызов `ioctl` для открытого файла, который не является символьным устройством
`Sys_ENXIO`	*Устройство или адрес не существует* (No such device or address). Происходит, если выполняется попытка получить доступ к несуществующему устройству или адресу устройства. Эта ошибка может возникать при доступе к отключенному устройству
`Sys_EOPNOTSUPP`	*Операция не поддерживается сокетом* (Operation not supported on a socket). Связанное с сокетом семейство адресов не поддерживает данной функции
`Sys_EOVERFLOW`	*Значение не может уместиться в типе данных* (Value too large to be stored in the data type)
`Sys_EPERM`	*Запрещенная операция* (Operation not permitted). Означает, что процесс пытался выполнить действие, разрешенное только владельцу файла или суперпользователю (root)
`Sys_EPIPE`	*Разрыв связи в канале* (Broken pipe). Устанавливается вызовом `fdwrite` и означает, что была выполнена попытка осуществить запись в канал, который не открыт на чтение ни одним процессом. Обычно при этом процесс, выполняющий запись в канал, прерывается при помощи сигнала `SIGPIPE`. Код ошибки `EPIPE` устанавливается, только если сигнал `SIGPIPE` перехватывается, игнорируется или блокируется
`Sys_EPROTO`	*Ошибка протокола* (Protocol error). Эта ошибка зависит от устройства и означает, что была получена ошибка протокола
`Sys_EPROTONOSUPPORT`	*Протокол не поддерживается* (Protocol not supported). Возвращается системным вызовом socket, если семейство адресов не поддерживается системой
`Sys_EPROTOTYPE`	*Тип сокета не поддерживается* (Socket type not supported). Возвращается вызовом `socket`, если заданный тип протокола, такой как `SOCK_DGRAM`, не поддерживается системой
`Sys_ERANGE`	*Результат слишком велик или слишком мал* (Result too large or too small). Эта ошибка возникает при вызове

	математических функций и означает, что возвращаемое функцией значение не может быть представлено на процессоре компьютера
`Sys_EROFS`	*Файловая система доступна только для чтения* (Readonly file system). Была выполнена попытка осуществить вызов `fdwrite` или изменить элемент каталога для файловой системы, которая была смонтирована в режиме только для чтения
`Sys_ESPIPE`	*Некорректное позиционирование* (Illegal seek). Для канала была предпринята попытка вызова `fdseek`
`Sys_ESRCH`	*Процесс не существует* (No such process). Задан несуществующий процесс. Генерируется вызовом `kill`
`Sys_ESTALE`	*Зарезервирован* (Reserved)
`Sys_ETIME`	*Таймаут вызова ioctl для модуля STREAM* (ioctl timeout on a STREAM). (Данная ошибка связана с архитектурой модулей STREAM.) Показывает, что истекло время ожидания вызова `ioctl` для модуля ядра STREAM. Это может означать, что период ожидания нужно увеличить
`Sys_ETIMEDOUT`	*Истекло время ожидания соединения* (Connection timed out). Когда процесс пытается установить соединение с другой системой, то заданное время ожидания может истечь, если эта система не включена или перегружена запросами на установку соединения
`Sys_ETXTBSY`	*Файл программного кода занят* (Text file busy). Если ошибка генерируется вызовом `exec`, то это означает, что была выполнена попытка запустить на выполнение исполняемый файл, открытый для записи. Если же она генерируется вызовом, возвращающим дескриптор файла, то была сделана попытка открыть на запись файл программы, которая в данный момент выполняется
`Sys_EWOULDBLOCK`	*Операция привела бы к блокировке* (Operation would block). Эта ошибка возвращается, если дескриптор ввода/вывода был открыт как не блокируемый и был выполнен запрос записи или чтения, который в обычных условиях был бы заблокирован. В соответствии со спецификацией XSI код ошибки `EWOULDBLOCK` должен иметь то же значение, что и `SYS_EAGAIN`
`Sys_EXDEV`	*Попытка создания ссылки между устройствами* (Cross-device link). Возникает, если выполняется попытка связать при помощи вызова `link` файлы в разных файловых системах

Приложение 2. История UNIX

ОС UNIX начала свою историю с того момента, как в 1969 г. Кен Томсон (Ken Thomson) и Деннис Ричи (Dennis Ritchie) с коллегами начали работу над ней на «стоящем в углу свободном компьютере PDP-7». С тех пор ОС UNIX нашла применение в сфере бизнеса и образования, а также стала основой ряда международных стандартов.

Кратко перечислим некоторые вехи истории UNIX:

- *шестая редакция системы*, 1975 (Sixth Edition, или Version 6). Первый широко известный в техническом сообществе коммерческий вариант и основа первой версии Berkeley UNIX;
- *Xenix* (1980). Версия от Microsoft – один из коммерческих вариантов UNIX с измененным названием, возникших в начале восьмидесятых годов;
- *System V* (1983-92). Одна из наиболее важных версий от создателя UNIX – корпорации AT&T. Наследница Version 7 и последовавшей за ней System III;
- *Berkeley UNIX* (версия 4.2 в 1984 г., 4.4 в 1993 г.). Разработанная в университете Berkeley, эта версия также была одной из наиболее важных версий UNIX и ввела в семейство UNIX много новых средств;
- *POSIX* (с 1988 г. и далее). Ключевой набор стандартов комитета IEEE (см. ниже), сыгравший важную роль в развитии UNIX;
- *Open Portability Guides* (XPG3 в 1990 г., XPG4.2 в 1994 г.). Практическая спецификация, объединившая целый ряд основных стандартов и рецептов использования UNIX систем. Консорциум X/Open в конце концов приобрел торговую марку UNIX.

Кроме того, существовало множество коммерческих взаимодействий, таких как передача торговой марки UNIX от компании AT&T к компании Novell, а затем консорциуму X/Open, а также слияние организаций X/Open и Open Software Foundation. Однако это достаточно упрощенный взгляд на историю UNIX. На самом деле дерево семейства UNIX является очень сложным.

Основные стандарты

В книге упомянуты следующие стандарты:

SVID

Название стандарта SVID является сокращением от AT&T System V Interface Definition (определение интерфейса ОС System V). Первоначально был разработан весной 1985 г. для описания интерфейса версии System V операционной системы UNIX. Стандарт SVID имеет ряд версий, последней из которых является третья редакция, вышедшая в 1989 г. Первое издание этой книги основывалось на стандарте SVID.

ANSI C

Комитет ANSI определяет стандарты различных информационных технологий. Наиболее интересным для системных программистов UNIX является стандарт языка ANSI C.

IEEE/POSIX

Институт электротехники и радиоэлектроники (Institute of Electrical and Electronics Engineers, сокращенно IEEE) разрабатывает в числе прочих стандарт интерфейса переносимой операционной системы (Portable Operating Systems Interface, сокращенно POSIX), который непосредственно базируется на ОС UNIX. Этот стандарт был впервые опубликован в 1988 г. и с тех пор несколько раз обновлялся. С темой данной книги наиболее тесно связаны следующие стандарты:

- стандарт IEEE 1003.1-1990, идентичный стандарту ISO POSIX-1 – ISO/IEC 9945-1, 1990, полное название которого: Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C language] (Информационные технологии – стандарт интерфейса переносимой операционной системы – часть 1. Системный интерфейс для прикладных программ [язык C]);

– стандарт IEEE 1003.2-1992, идентичный стандарту ISO POSIX-2 – ISO/IEC 9945-2, 1993, полное название которого: Information Technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities (Информационные технологии – стандарт интерфейса переносимой операционной системы – часть 2. Командный интерпретатор и утилиты).

Позже к стандарту были добавлены расширения и добавления, включая стандарты 1003.1b-1993, 1003.1c-1995, 1003.li-1995, охватывающие такие темы, как потоки управления и расширения реального времени.

X/Open (в настоящее время Open Group)

Группа X/Open Group объединила вышеупомянутые стандарты с другими; эти стандарты, вместе взятые, получили название спецификации Common Application Environment (общей среды приложений, сокращенно CAE). Спецификация CAE охватывает как системные, так и сетевые интерфейсы.

Текст книги лучше всего соответствует Issue 4 Version 2 X/Open CAE Specification, System Interface and Headers (четвертому выпуску второй версии спецификации CAE консорциума X/Open, системные интерфейсы и заголовки), вышедшей в августе 1994 г. Эта спецификация представляет собой базовый документ X/Open. Пятое издание спецификации CAE, вышедшее в 1997 г., включает некоторые из недавно введенных в стандарт POSIX средств работы с потоками управления в режиме реального времени, а также других наработок из практики промышленного использования. Эти обновления также вошли в Version 2 of the Single UNIX Specification (вторую версию единой спецификации ОС UNIX) консорциума Open Group и в описание продукта UNIX98.

Приложение 3. Модуль stdio

Назначение модуля – восполнить недостаток средств прикладного и системного программирования на языке Паскаль аналогичными средствами библиотек языка Си.

```
unit stdio;

{$mode objfpc}
interface
(* A streammarker remembers a position in a buffer. *)

uses linux,sockets;

const
  EOF=-1;          (* признак конца файла для stdio *)

  WCOREFLAG=$80;  (* флаг сброса дампа памяти *)

  (* Размер буфера для чтения и записи *)
  BUFSIZ=8192;

  NULL=nil;

  _IOFBF=0;  (* setvbuf should set fully buffered *)
  _IOLBF=1;  (* setvbuf should set line buffered *)
  _IONBF=2;  (* setvbuf should set unbuffered *)

  MAX_INPUT=255;  (* size of the type-ahead buffer *)

  _SIGSET_NWORDS=1024 div (8 * sizeof (longint)); (* sizeof(sigset_t) *)

  (* флаги для pathconf/fpathconf *)

  (* returns  the maximum number of links to the file.  If filedes or
   * path refer to a directory, then the value applies to  the  whole
   * directory.
   *)
  _PC_LINK_MAX=0;
  (* returns  the  maximum  length  of  a formatted input line, where
   * filedes or path must refer to  a  terminal.
   *)
  _PC_MAX_CANON=1;
  (* returns  the  maximum  length of an input line, where filedes or
   * path must refer to  a  terminal.
   *)
  _PC_MAX_INPUT=2;
  (* returns  the  maximum length of a filename in the directory path
   * or filedes.  the process is allowed to create.
   *)
  _PC_NAME_MAX=3;
  (* returns  the  maximum length of a relative pathname when path or
   * filedes is the current  working  directory.
   *)
  _PC_PATH_MAX=4;
  (* returns the size of the pipe buffer, where filedes must refer to
   * a pipe or FIFO and path must refer to a FIFO.
   *)
```

```
  _PC_PIPE_BUF=5;
  (* returns  nonzero  if  the  chown(2) call may not be used on this
   * file.  If filedes or  path  refer  to  a  directory,  then  this
   * applies to all files in that directory.
   *)
  _PC_CHOWN_RESTRICTED=6;
  (* returns   nonzero   if   accessing   filenames   longer   than
   * _POSIX_NAME_MAX  generates an error.
   *)
  _PC_NO_TRUNC=7;
  (* returns nonzero if special character processing can be disabled,
   * where filedes or path must refer to a terminal.
   *)
  _PC_VDISABLE=8;
  _PC_SYNC_IO=9;
  _PC_ASYNC_IO=10;
  _PC_PRIO_IO=11;
  _PC_SOCK_MAXBUF=12;
  _PC_FILESIZEBITS=13;
  _PC_REC_INCR_XFER_SIZE=14;
  _PC_REC_MAX_XFER_SIZE=15;
  _PC_REC_MIN_XFER_SIZE=16;
  _PC_REC_XFER_ALIGN=17;
  _PC_ALLOC_SIZE_MIN=18;
  _PC_SYMLINK_MAX=19;

  (* For posix fcntl() and `l_type' field of a `struct flock' for lockf().  *)
  F_RDLCK=0;       (* Read lock.  *)
  F_WRLCK=1;       (* Write lock.  *)
  F_UNLCK=2;       (* Remove lock.  *)

  F_DUPFD=0;       (* Duplicate fildes *)

  (* Constants for ulimit *)
  UL_GETFSIZE=1;      (* возвращает установленное ограничение на размер
                         файла (в блоках по 512 байт) *)
  UL_SETFSIZE=2;      (* устанавливает предельный размер файла *)
  __UL_GETMAXBRK=3;   (* возвращает максимально возможный адрес сегмента
данных *)
  __UL_GETOPENMAX=4;  (* возвращает максимальное количество файлов, которое
                         процесс может открыть *)

  FD_SETSIZE=64;

  (* Macros used as `request' argument to `ioctl'.  *)
  __SID=byte('S') shl 8;
  I_NREAD     =__SID or 1;  (* Counts the number of data bytes in the data
                               block in the first message.  *)
  I_PUSH      =__SID or 2;  (* Push STREAMS module onto top of the current
                               STREAM, just below the STREAM head.  *)
  I_POP       =__SID or 3;  (* Remove STREAMS module from just below the
                               STREAM head.  *)
  I_LOOK      =__SID or 4;  (* Retrieve the name of the module just below
                               the STREAM head and place it in a character
                               string.  *)
  I_FLUSH     =__SID or 5;  (* Flush all input and/or output.  *)
  I_SRDOPT    =__SID or 6;  (* Sets the read mode.  *)
  I_GRDOPT    =__SID or 7;  (* Returns the current read mode setting.  *)
```

```
I_STR       =__SID or 8;  (* Construct an internal STREAMS `ioctl'
                             message and send that message downstream. *)
I_SETSIG    =__SID or 9;  (* Inform the STREAM head that the process
                             wants the SIGPOLL signal issued.  *)
I_GETSIG    =__SID or 10; (* Return the events for which the calling
                             process is currently registered to be sent
                             a SIGPOLL signal.  *)
I_FIND      =__SID or 11; (* Compares the names of all modules currently
                             present in the STREAM to the name pointed to
                             by `arg'.  *)
I_LINK      =__SID or 12; (* Connect two STREAMs.  *)
I_UNLINK    =__SID or 13; (* Disconnects the two STREAMs.  *)
I_PEEK      =__SID or 15; (* Allows a process to retrieve the information
                             in the first message on the STREAM head read
                             queue without taking the message off the
                             queue.  *)
I_FDINSERT  =__SID or 16; (* Create a message from the specified
                             buffer(s), adds information about another
                             STREAM, and send the message downstream.  *)
I_SENDFD    =__SID or 17; (* Requests the STREAM associated with `fildes'
                             to send a message, containing a file
                             pointer, to the STREAM head at the other end
                             of a STREAMS pipe.  *)
I_RECVFD    =__SID or 14; (* Non-EFT definition.  *)
I_SWROPT    =__SID or 19; (* Set the write mode.  *)
I_GWROPT    =__SID or 20; (* Return the current write mode setting.  *)
I_LIST      =__SID or 21; (* List all the module names on the STREAM, up
                             to and including the topmost driver name. *)
I_PLINK     =__SID or 22; (* Connect two STREAMs with a persistent
                              link.  *)
I_PUNLINK   =__SID or 23; (* Disconnect the two STREAMs that were
                             connected with a persistent link.  *)
I_FLUSHBAND =__SID or 28; (* Flush only band specified.  *)
I_CKBAND    =__SID or 29; (* Check if the message of a given priority
                             band exists on the STREAM head read
                             queue.  *)
I_GETBAND   =__SID or 30; (* Return the priority band of the first
                             message on the STREAM head read queue.  *)
I_ATMARK    =__SID or 31; (* See if the current message on the STREAM
                             head read queue is "marked" by some module
                             downstream.  *)
I_SETCLTIME =__SID or 32; (* Set the time the STREAM head will delay when
                             a STREAM is closing and there is data on
                             the write queues.  *)
I_GETCLTIME =__SID or 33; (* Get current value for closing timeout.  *)
I_CANPUT    =__SID or 34; (* Check if a certain band is writable.  *)

(* для интерфейса сокетов *)
INADDR_ANY=0;
{ Maximum queue length specificable by listen.   }
SOMAXCONN = 5;
{ Flags we can use with send/ and recv.  }
{ process out-of-band data  }
MSG_OOB = $1;
{ peek at incoming message  }
MSG_PEEK = $2;
{ send without using routing tables  }
MSG_DONTROUTE = $4;
```

```
{ Setsockoptions(2) level. Thanks to BSD these must match IPPROTO_xxx  }
SOL_IP = 0;
SOL_IPX = 256;
SOL_AX25 = 257;
SOL_ATALK = 258;
SOL_NETROM = 259;
SOL_TCP = 6;
SOL_UDP = 17;
{ IP options  }
IPTOS_LOWDELAY = $10;
IPTOS_THROUGHPUT = $08;
IPTOS_RELIABILITY = $04;
{ These need to appear somewhere around here  }
IP_DEFAULT_MULTICAST_TTL = 1;
IP_DEFAULT_MULTICAST_LOOP = 1;
IP_MAX_MEMBERSHIPS = 20;
{ IP options for use with WinSock  }
IP_OPTIONS = 1;
IP_MULTICAST_IF = 2;
IP_MULTICAST_TTL = 3;
IP_MULTICAST_LOOP = 4;
IP_ADD_MEMBERSHIP = 5;
IP_DROP_MEMBERSHIP = 6;
IP_TTL = 7;
IP_TOS = 8;
IP_DONTFRAGMENT = 9;
{ IPX options  }
IPX_TYPE = 1;
{ TCP options }
TCP_NODELAY = $0001;
TCP_MAXSEG = 2;
{ The various priorities.  }
SOPRI_INTERACTIVE = 0;
SOPRI_NORMAL = 1;
SOPRI_BACKGROUND = 2;


(* управление терминалом *)
TCOOFF = 0;
TCOON = 1;
TCIOFF = 2;
TCION = 3;
TCGETA = 5;
TCSETA = 6;
TCSETAW = 7;
TCSETAF = 8;
TCIFLUSH = 0;
TCOFLUSH = 1;
TCIOFLUSH = 2;
TCFLSH = 3;
TCSAFLUSH = 1;
TCSANOW = 2;
TCSADRAIN = 3;
TCSADFLUSH = 4;
TIOCPKT = 6;
TIOCPKT_DATA = 0;
TIOCPKT_FLUSHREAD = 1;
TIOCPKT_FLUSHWRITE = 2;
```

```
TIOCPKT_STOP = 4;
TIOCPKT_START = 8;
TIOCPKT_NOSTOP = 16;
TIOCPKT_DOSTOP = 32;
{ To be compatible with socket version  }
FIONBIO = $8004667e;
{ iflag bits  }
IGNBRK = $00001;
BRKINT = $00002;
IGNPAR = $00004;
IMAXBEL = $00008;
INPCK = $00010;
ISTRIP = $00020;
INLCR = $00040;
IGNCR = $00080;
ICRNL = $00100;
IXON = $00400;
IXOFF = $01000;
IUCLC = $04000;
IXANY = $08000;
PARMRK = $10000;
{ oflag bits  }
OPOST = $00001;
OLCUC = $00002;
OCRNL = $00004;
ONLCR = $00008;
ONOCR = $00010;
ONLRET = $00020;
OFILL = $00040;
CRDLY = $00180;
CR0 = $00000;
CR1 = $00080;
CR2 = $00100;
CR3 = $00180;
NLDLY = $00200;
NL0 = $00000;
NL1 = $00200;
BSDLY = $00400;
BS0 = $00000;
BS1 = $00400;
TABDLY = $01800;
TAB0 = $00000;
TAB1 = $00800;
TAB2 = $01000;
TAB3 = $01800;
XTABS = $01800;
VTDLY = $02000;
VT0 = $00000;
VT1 = $02000;
FFDLY = $04000;
FF0 = $00000;
FF1 = $04000;
OFDEL = $08000;
{ cflag bits  }
{ Baud rate values.  These must fit in speed_t, which is unsigned
char.  See also the extended baud rates below.  These baud rates
set an additional bit.  }
CBAUD = $0100f;
```

```
B0 = $00000;
B50 = $00001;
B75 = $00002;
B110 = $00003;
B134 = $00004;
B150 = $00005;
B200 = $00006;
B300 = $00007;
B600 = $00008;
B1200 = $00009;
B1800 = $0000a;
B2400 = $0000b;
B4800 = $0000c;
B9600 = $0000d;
B19200 = $0000e;
B38400 = $0000f;
CSIZE = $00030;
CS5 = $00000;
CS6 = $00010;
CS7 = $00020;
CS8 = $00030;
CSTOPB = $00040;
CREAD = $00080;
PARENB = $00100;
PARODD = $00200;
HUPCL = $00400;
CLOCAL = $00800;
CBAUDEX = $0100f;
B57600 = $01001;
B115200 = $01002;
B128000 = $01003;
B256000 = $01003;
CRTSXOFF = $04000;
CRTSCTS = $08000;
{ lflag bits  }
ISIG = $0001;
ICANON = $0002;
ECHO = $0004;
ECHOE = $0008;
ECHOK = $0010;
ECHONL = $0020;
NOFLSH = $0040;
TOSTOP = $0080;
IEXTEN = $0100;
FLUSHO = $0200;
ECHOKE = $0400;
ECHOCTL = $0800;
VDISCARD = 1;
VEOL = 2;
VEOL2 = 3;
VEOF = 4;
VERASE = 5;
VINTR = 6;
VKILL = 7;
VLNEXT = 8;
VMIN = 9;
VQUIT = 10;
VREPRINT = 11;
```

```
  VSTART = 12;
  VSTOP = 13;
  VSUSP = 14;
  VSWTC = 15;
  VTIME = 16;
  VWERASE = 17;
  CNUL = 0;
  CDEL = $0007f;
  CESC = byte('\');
  CINTR = byte('C') and $1f;
  CQUIT = $0001c;
  CERASE = byte('H') and $1f;
  CKILL = byte('U') and $1f;
  CEOT = byte('D') and $1f;
  CEOL = 0;
  CEOL2 = 0;
  CEOF = byte('D') and $1f;
  CSTART = byte('Q') and $1f;
  CSTOP = byte('S') and $1f;
  CSWTCH = $0001a;
  NSWTCH = 0;
  CSUSP = byte('Z') and $1f;
  CDSUSP = byte('Y') and $1f;
  CRPRNT = byte('R') and $1f;
  CFLUSH = byte('O') and $1f;
  CWERASE = byte('W') and $1f;
  CLNEXT = byte('V') and $1f;



type PFILE=^TFILE;

_IO_marker=record
  next:^_IO_marker;
  sbuf:PFILE;
  _pos:integer;
end;

P_IO_marker=^_IO_marker;

ppchar=^pchar;
pinteger=^integer;
plongint=^longint;

sigset_t=record
  __val:array [0.._SIGSET_NWORDS-1] of longint;
end;

psigset_t=^sigset_t;

__jmp_buf=array [0..5] of longint;
__jmp_buf_tag=record
  __jmpbuf:__jmp_buf;               (* Calling environment.  *)
  __mask_was_saved:longint;         (* Saved the signal mask?  *)
  __saved_mask:sigset_t;              (* Saved signal mask.  *)
end;
jmp_buf=array [0..0] of __jmp_buf_tag;
sigjmp_buf=jmp_buf;
```

```
flockrec=record
  l_type:word;     (* Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.  *)
  l_whence:word;   (* Where `l_start' is relative to (like `lseek').  *)
  l_start:longint;    (* Offset where the lock begins.  *)
  l_len:longint;      (* Size of the locked area; zero means until EOF.  *)
  l_pid:longint;      (* Process holding the lock.  *)
end;

(* для интерфейса сокетов *)
in_addr=record
  s_addr:cardinal;
end;
pin_addr=^in_addr;
in_addr_t=cardinal;

(* для работы со временем *)
tm=record
  tm_sec:longint;         (* seconds *)
  tm_min:longint;         (* minutes *)
  tm_hour:longint;        (* hours *)
  tm_mday:longint;        (* day of the month *)
  tm_mon:longint;         (* month *)
  tm_year:longint;        (* year *)
  tm_wday:longint;        (* day of the week *)
  tm_yday:longint;        (* day in the year *)
  tm_isdst:longint;       (* daylight saving time *)
end;
ptm=^tm;



TFILE=record
  _flags:integer;      (* High-order word is _IO_MAGIC; rest is flags. *)

  (* The following pointers correspond to the C++ streambuf protocol. *)
  (* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. *)
  _IO_read_ptr:pchar;   (* Current read pointer *)
  _IO_read_end:pchar;   (* End of get area. *)
  _IO_read_base:pchar;  (* Start of putback+get area. *)
  _IO_write_base:pchar; (* Start of put area. *)
  _IO_write_ptr:pchar;  (* Current put pointer. *)
  _IO_write_end:pchar;  (* End of put area. *)
  _IO_buf_base:pchar;   (* Start of reserve area. *)
  _IO_buf_end:pchar;    (* End of reserve area. *)
  (* The following fields are used to support backing up and undo. *)
  _IO_save_base:pchar; (* Pointer to start of non-current get area. *)
  _IO_backup_base:pchar;  (* Pointer to first valid character of backup area
*)
  _IO_save_end:pchar; (* Pointer to end of non-current get area. *)

  _markers:P_IO_marker;

  _chain:^TFILE;

  _fileno:integer;
  _blksize:integer;
  _old_offset:longint; (* This used to be _offset but it's too small.  *)
```

```
  _cur_column:word;
  _vtable_offset:byte;
  _shortbuf:array[1..1]of char;

  _lock:pointer;
end;

var
  stdin,stdout:pfile;

function tmpfile:pfile;cdecl;external 'c';

(* Generate a temporary filename.  *)
function tmpnam(__s:pchar):pchar;cdecl;external 'c';

(* Close STREAM.  *)
function fclose(__stream:pfile):integer;cdecl;external 'c';

(* Flush STREAM, or all streams if STREAM is NULL.  *)
function fflush (__stream:pfile):integer;cdecl;external 'c';

(* Close all streams.  *)
function fcloseall:integer;cdecl;external 'c';

(* Open a file and create a new stream for it.  *)
function fopen(fname:pchar;mode:pchar):pfile;cdecl;external 'c';

(* Open a file by fd and create a new stream for it.  *)
function fdopen(fildes:longint;mode:pchar):pfile;cdecl;external 'c';

(* Open a file, replacing an existing stream with it. *)
function freopen
(_filename:pchar;__modes:pchar;__stream:pfile):pfile;cdecl;external 'c';

(* Create a new stream that refers to a memory buffer.  *)
function
fmemopen(__s:pointer;__len:longint;__modes:pchar):pfile;cdecl;external 'c';

(* Open a stream that writes into a malloc'd buffer that is expanded as
   necessary.  *BUFLOC and *SIZELOC are updated with the buffer's location
   and the number of characters written on fflush or fclose.  *)
function
open_memstream(__bufloc:ppchar;__sizeloc:plongint):pfile;cdecl;external 'c';

(* If BUF is NULL, make STREAM unbuffered.
   Else make it use buffer BUF, of size BUFSIZ.  *)
procedure setbuf(__stream:pfile;__buf:pchar);cdecl;external 'c';

(* Make STREAM use buffering mode MODE.
   If BUF is not NULL, use N bytes of it for buffering;
   else allocate an internal buffer N bytes long.  *)
function setvbuf(__stream:pfile;__buf:pchar;
                 __modes:longint;__n:longint):integer;cdecl;external 'c';

(* If BUF is NULL, make STREAM unbuffered.
   Else make it use SIZE bytes of BUF for buffering.  *)
procedure setbuffer(__stream:pfile;__buf:pchar;
```

```
                __size:longint);cdecl;external 'c';

(* Make STREAM line-buffered.  *)
procedure setlinebuf(__stream:pfile);cdecl;external 'c';

(* Write formatted output to STREAM.  *)
function fprintf(__stream:pfile;fm:pchar;args:array of
const):integer;cdecl;external 'c';

(* Write formatted output to stdout.  *)
function printf(fm:pchar;args:array of const):integer;cdecl;external 'c';

(* Write formatted output to S.  *)
function sprintf(__s:pchar;fm:pchar;args:array of
const):integer;cdecl;external 'c';

(* Read formatted input from STREAM.  *)
function fscanf(__stream:pfile;fm:pchar;args:array of
const):integer;cdecl;external 'c';

(* Read formatted input from stdin.  *)
function scanf(fm:pchar;args:array of const):integer;cdecl;external 'c';

(* Read formatted input from S.  *)
function sscanf(s:pchar;fm:pchar;args:array of const):integer;cdecl;external
'c';

(* Read a character from STREAM.  *)
function fgetc(__stream:pfile):integer;cdecl;external 'c';
function getc(__stream:pfile):integer;cdecl;external 'c';

(* Read a character from stdin.  *)
function getchar:integer;cdecl;external 'c';

(* These are defined in POSIX.1:1996.  *)
function getc_unlocked(__stream:pfile):integer;cdecl;external 'c';
function getchar_unlocked:integer;cdecl;external 'c';

(* Faster version when locking is not necessary.  *)
function fgetc_unlocked(__stream:pfile):integer;cdecl;external 'c';

(* Write a character to STREAM.  *)
function fputc(__c:integer;__stream:pfile):integer;cdecl;external 'c';
function putc(__c:integer;__stream:pfile):integer;cdecl;external 'c';

(* Write a character to stdout.  *)
function putchar(__c:integer):integer;cdecl;external 'c';

(* Faster version when locking is not necessary.  *)
function fputc_unlocked(__c:integer;__stream:pfile):integer;cdecl;external
'c';

(* These are defined in POSIX.1:1996.  *)
function putc_unlocked(__c:integer;__stream:pfile):integer;cdecl;external 'c';
function putchar_unlocked(__c:integer):integer;cdecl;external 'c';

(* Get a word (int) from STREAM.  *)
function getw(__stream:pfile):integer;cdecl;external 'c';
```

```
(* Write a word (int) to STREAM.  *)
function putw(__w:integer;__stream:pfile):integer;cdecl;external 'c';

(* Get a newline-terminated string of finite length from STREAM.  *)
function fgets(__s:pchar;__n:integer;__stream:pfile):pchar;cdecl;external 'c';

(* This function does the same as `fgets' but does not lock the stream.  *)
function
fgets_unlocked(__s:pchar;__n:integer;__stream:pfile):pchar;cdecl;external 'c';

(* Get a newline-terminated string from stdin, removing the newline.
   DO NOT USE THIS FUNCTION!!  There is no limit on how much it will read.  *)
function gets(__s:pchar):pchar;cdecl;external 'c';

(* Read up to (and including) a DELIMITER from STREAM into *LINEPTR
   (and null-terminate it). *LINEPTR is a pointer returned from malloc (or
   NULL), pointing to *N characters of space.  It is realloc'd as
   necessary.  Returns the number of characters read (not including the
   null terminator), or -1 on error or EOF.  *)
function __getdelim(__lineptr:ppchar;__n:plongint;__delimiter:integer;
                    __stream:pfile):integer;cdecl;external 'c';
function getdelim(__lineptr:ppchar;__n:plongint;__delimiter:integer;
                    __stream:pfile):integer;cdecl;external 'c';

(* Like `getdelim', but reads up to a newline.  *)
function getline(__lineptr:ppchar;__n:plongint;
                    __stream:pfile):integer;cdecl;external 'c';

(* Write a string to STREAM.  *)
function fputs(__s:pchar;__stream:pfile):integer;cdecl;external 'c';

(* This function does the same as `fputs' but does not lock the stream.  *)
function fputs_unlocked(__s:pchar;__stream:pfile):integer;cdecl;external 'c';

(* Write a string, followed by a newline, to stdout.  *)
function puts(__s:pchar):integer;cdecl;external 'c';

(* Push a character back onto the input buffer of STREAM.  *)
function ungetc(__c:integer;__stream:pfile):integer;cdecl;external 'c';

(* Read chunks of generic data from STREAM.  *)
function fread(__ptr:pointer;__size, n:longint;
               __stream:pfile):longint;cdecl;external 'c';

(* Write chunks of generic data to STREAM.  *)
function fwrite(__ptr:pointer;__size, n:longint;
               __stream:pfile):longint;cdecl;external 'c';

(* Faster versions when locking is not necessary.  *)
function fread_unlocked(__ptr:pointer;__size, n:longint;
               __stream:pfile):longint;cdecl;external 'c';
function fwrite_unlocked(__ptr:pointer;__size, n:longint;
               __stream:pfile):longint;cdecl;external 'c';

(* Seek to a certain position on STREAM.  *)
function
fseek(__stream:pfile;__off:longint;__whence:integer):longint;cdecl;external
```

```
'c';

(* Return the current position of STREAM.  *)
function ftell(__stream:pfile):longint;cdecl;external 'c';

(* Rewind to the beginning of STREAM.  *)
procedure rewind(__stream:pfile);cdecl;external 'c';

(* Clear the error and EOF indicators for STREAM.  *)
procedure clearerr(__stream:pfile);cdecl;external 'c';

(* Return the EOF indicator for STREAM.  *)
function feof(__stream:pfile):integer;cdecl;external 'c';

(* Return the error indicator for STREAM.  *)
function ferror(__stream:pfile):integer;cdecl;external 'c';

(* Faster versions when locking is not required.  *)
procedure clearerr_unlocked(__stream:pfile);cdecl;external 'c';
function feof_unlocked(__stream:pfile):integer;cdecl;external 'c';
function ferror_unlocked(__stream:pfile):integer;cdecl;external 'c';

(* Print a message describing the meaning of the value of errno.  *)
procedure perror(__s:pchar);cdecl;external 'c';

(* Return the system file descriptor for STREAM.  *)
function fileno(__stream:pfile):longint;cdecl;external 'c';

(* Faster version when locking is not required.  *)
function fileno_unlocked(__stream:pfile):integer;cdecl;external 'c';

(* Create a new stream connected to a pipe running the given command.  *)
function pipeopen(__command, __modes:pchar):pfile;cdecl;external 'c' name
'popen';

(* Close a stream opened by popen and return the status of its child.  *)
function pipeclose (__stream:pfile):integer;cdecl;external 'c' name 'pclose';

(* Return the name of the controlling terminal.  *)
function ctermid(__s:pchar):pchar;cdecl;external 'c';

(* Return the name of the current user.  *)
function cuserid(__s:pchar):pchar;cdecl;external 'c';

(* Acquire ownership of STREAM.  *)
procedure flockfile(__stream:pfile);cdecl;external 'c';

(* Try to acquire ownership of STREAM but do not block if it is not
   possible.  *)
function ftrylockfile(__stream:pfile):integer;cdecl;external 'c';

(* Relinquish the ownership granted for STREAM.  *)
procedure funlockfile(__stream:pfile);cdecl;external 'c';

(* создание файла *)
function fdCreat(PathName:Pchar;mode:longint):longint;cdecl;external 'c' name
'creat';
```

```
(* "перемотка" указателя на первый элемент каталога *)
procedure rewinddir(dp:pdir);cdecl;external 'c';

(* ожидание завершения процесса *)
function wait(__stat_loc:pinteger):longint;cdecl;external 'c';

(* функции для создания нового процесса с переменным числом параметров *)
function linuxexecl(path:pchar;arg0:pchar;args:array of
const):integer;cdecl;external 'c' name 'execl';
function linuxexeclp(filename:pchar;arg0:pchar;args:array of
const):integer;cdecl;external 'c' name 'execlp';

(* функции для определения состояния завершения процесса *)
function WEXITSTATUS(status:integer):integer;
function WSTOPSIG(status:integer):integer;
function WTERMSIG(status:integer):integer;

function WCOREDUMP(status:integer):boolean;
function WIFEXITED(status:integer):boolean;
function WIFSIGNALED(status:integer):boolean;
function WIFSTOPPED(status:integer):boolean;

(* приостанавливает текущий процесс на заданное количество секунд *)
function sleep(seconds:word):word;cdecl;external 'c';

(* удаляет все сигналы из набора сигналов  *)
function sigemptyset(__set:psigset_t):integer;cdecl;external 'c';

(* устанавливает все сигналы в наборе сигналов  *)
function sigfillset(__set:psigset_t):integer;cdecl;external 'c';

(* добавляет сигнал SIGNO в набор сигналов *)
function sigaddset(__set:psigset_t;__signo:integer):integer;cdecl;external
'c';

(* удаляет сигнал SIGNO из набора сигналов *)
function sigdelset(__set:psigset_t;__signo:integer):integer;cdecl;external
'c';

(* возвращает истину, если SIGNO в наборе, и ложь в противном случае *)
function sigismember(__set:psigset_t;__signo:integer):boolean;cdecl;external
'c';

(* возвращает истинное значение, если набор сигналов не пуст *)
function sigisemptyset(__set:psigset_t):boolean;cdecl;external 'c';

(* создает новый набор сигналов из двух входных с помощью логического "И" *)
function sigandset(__set, __left, __right:psigset_t):integer;cdecl;external
'c';

(* создает новый набор сигналов из двух входных с помощью логического "ИЛИ" *)
function sigorset(__set, __left, __right:psigset_t):integer;cdecl;external
'c';

(* набор функций для нелокальных переходов *)

(* Store the calling environment in ENV, also saving the signal mask.
   Return 0.  *)
```

```
function setjmp(var __env:jmp_buf):integer;cdecl;external 'c';

(* Store the calling environment in ENV, not saving the signal mask.
   Return 0.  *)
function _setjmp(var __env:jmp_buf):integer;cdecl;external 'c';

(* Store the calling environment in ENV, also saving the
   signal mask if SAVEMASK is nonzero.  Return 0.  *)
function sigsetjmp(var
__env:jmp_buf;__savemask:longint):integer;cdecl;external 'c' name
'__sigsetjmp';

(* Jump to the environment saved in ENV, making the
   `setjmp' call there return VAL, or 1 if VAL is 0.  *)
procedure longjmp(var __env:jmp_buf;__val:integer);cdecl;external 'c';

(* Jump to the environment saved in ENV, making the
   sigsetjmp call there return VAL, or 1 if VAL is 0.
   Restore the signal mask if that sigsetjmp call saved it.
   This is just an alias `longjmp'.  *)
procedure siglongjmp(var __env:jmp_buf;__val:integer);cdecl;external 'c';

(* fpathconf, pathconf - get configuration values for files *)
function fpathconf(filedes,name:longint):longint;cdecl;external 'c';
function pathconf(path:pchar;name:longint):longint;cdecl;external 'c';
function sysconf(name:integer):longint;cdecl;external 'c';

(* setpgid, getpgid, setpgrp, getpgrp - set/get process group *)
function setpgid(pid, pgid:longint):longint;cdecl;external 'c';
function getpgid(pid:longint):longint;cdecl;external 'c';
function setpgrp:longint;cdecl;external 'c';
function getpgrp:longint;cdecl;external 'c';
function getsid(pid:longint):longint;cdecl;external 'c';
function setsid:longint;cdecl;external 'c';

(* grantpt - grant access to the slave pseudotty*)
function grantpt(fd:longint):longint;cdecl;external 'c';

(* unlockpt - unlock a pseudotty master/slave pair*)
function unlockpt(fd:longint):longint;cdecl;external 'c';

(* ptsname - get the name of the slave pseudotty*)
function ptsname(fd:longint):pchar;cdecl;external 'c';

(* inet_aton() converts the Internet host address  cp  from  the  standard
   numbers-and-dots  notation into binary data and stores it in the structure
   that inp points to. inet_aton returns nonzero if  the  address  is
   valid, zero if not.*)
function inet_aton(cp:pchar; inp:pin_addr):longint;cdecl;external 'c';

(* The  inet_addr()  function  converts  the Internet host address cp from
   numbers-and-dots notation into binary data in network byte  order.   If
   the input is invalid, INADDR_NONE (usually -1) is returned.  This is an
   obsolete interface to inet_aton, described  immediately  above;  it  is
   obsolete because -1 is a valid address (255.255.255.255), and inet_aton
   provides a cleaner way to indicate error return. *)
function inet_addr(cp:pchar):in_addr_t;cdecl;external 'c';
```

```
(* The inet_network() function extracts the network number  in  host  byte
   order  from  the address cp in numbers-and-dots notation.  If the input
   is invalid, -1 is returned. *)
function inet_network(cp:pchar):in_addr_t;cdecl;external 'c';

(* The inet_ntoa() function converts the Internet host address in given in
   network  byte  order to a string in standard numbers-and-dots notation.
   The string is returned in a statically allocated buffer,  which  subse-
   quent calls will overwrite. *)
function inet_ntoa(n:in_addr):pchar;cdecl;external 'c';

(* The  inet_makeaddr() function makes an Internet host address in network
   byte order by combining the network number net with the  local  address
   host in network net, both in local host byte order. *)
function inet_makeaddr(net, host:longint):in_addr;cdecl;external 'c';

(* The  inet_lnaof()  function  returns the local host address part of the
   Internet address in.  The local host address is returned in local  host
   byte order. *)
function inet_lnaof(n:in_addr):in_addr_t;cdecl;external 'c';

(* The inet_netof() function returns the network number part of the Inter-
   net Address in.  The network number is  returned  in  local  host  byte
   order. *)
function inet_netof(n:in_addr):in_addr_t;cdecl;external 'c';

(* sendto - отправить сообщение в сокет.
   send, sendto, и sendmsg используются для пересылки сообщений на  другой
   сокет.    send  можно  использовать,  только  если  сокет находится в
   соединенном состоянии, тогда как sendto и sendmsg можно использовать  в
   любое время. *)
function sendto(sock:longint;var addr;addrlen,flags:longint;
               var sato:tsockaddr;tolen:longint):longint;cdecl;external 'c';

(* recvfrom используется для получения сообщений из сокета, и может
   использоваться для получения данных, независимо от того, является ли
   сокет ориентированным на соединения или нет. *)
function recvfrom(sock:longint;var addr;addrlen,flags:longint;
                 var from:tsockaddr;var
fromlen:longint):longint;cdecl;external 'c';

(* character classification routines *)
function isalnum(c:integer):boolean;cdecl;external 'c';
function isalpha(c:integer):boolean;cdecl;external 'c';
function isascii(c:integer):boolean;cdecl;external 'c';
function isblank(c:integer):boolean;cdecl;external 'c';
function iscntrl(c:integer):boolean;cdecl;external 'c';
function isdigit(c:integer):boolean;cdecl;external 'c';
function isgraph(c:integer):boolean;cdecl;external 'c';
function islower(c:integer):boolean;cdecl;external 'c';
function isprint(c:integer):boolean;cdecl;external 'c';
function ispunct(c:integer):boolean;cdecl;external 'c';
function isupper(c:integer):boolean;cdecl;external 'c';
function isxdigit(c:integer):boolean;cdecl;external 'c';
function isspace(c:integer):boolean;
function tolower(c:integer):integer;cdecl;external 'c';
function _tolower(c:integer):integer;cdecl;external 'c';
function toupper(c:integer):integer;cdecl;external 'c';
```

```
function _toupper(c:integer):integer;cdecl;external 'c';
function toascii(c:integer):integer;cdecl;external 'c';

(* transform date and time to broken-down time or ASCII *)

(* The ctime(), gmtime() and localtime() functions all take an argument of
   data  type  time_t which represents calendar time.  When interpreted as
   an absolute time value, it represents the  number  of  seconds  elapsed
   since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). *)
function ctime(var t:longint):pchar;cdecl;external 'c';
function ctime_r(var timep:longint;buf:pchar):pchar;cdecl;external 'c';
function gmtime(var timep:longint):ptm;cdecl;external 'c';
function gmtime_r(var timep:longint; result:ptm):ptm;cdecl;external 'c';
function localtime(var timep:longint):ptm;cdecl;external 'c';
function localtime_r(var timep:longint; result:ptm):ptm;cdecl;external 'c';

(* The asctime() and mktime() functions both take an argument representing
   broken-down time which is a representation separated into year,  month,
   day, etc. *)
function asctime(const t:ptm):pchar;cdecl;external 'c';
function asctime_r(const t:ptm;buf:pchar):pchar;cdecl;external 'c';
function mktime(t:ptm):longint;cdecl;external 'c';
function difftime(time1, time2:longint):double;cdecl;external 'c';

(* функции для работы с памятью *)
function calloc(nmemb, size:longint):pointer;cdecl;external 'c';
function malloc(size:longint):pointer;cdecl;external 'c';
procedure free(ptr:pointer);cdecl;external 'c';
function realloc(ptr:pointer; size:longint):pointer;cdecl;external 'c';

(* функции для работы с каталогами *)
function mkdir(pathname:pchar;mode:integer):integer;cdecl;external 'c';
function rmdir(pathname:pchar):integer;cdecl;external 'c';
function chdir(path:pchar):integer;cdecl;external 'c';
function getcwd(name:pchar; size:longint):pchar;cdecl;external 'c';

(* работа с буферами *)
procedure sync;cdecl;external 'c';
function fsync(filedes:integer):integer;cdecl;external 'c';

(* выполнение команды shell'a *)
function runshell(__commandline:pchar):longint;cdecl;external 'c' name
'system';

(* уничтожение вызвавшего процесса *)
procedure _exit(status:longint);cdecl;external 'c';

(* изменение или расширение окружения *)
function putenv(str:pchar):longint;cdecl;external 'c';

(* смена корневого каталога *)
function chroot(path:pchar):longint;cdecl;external 'c';

(* установка реальных и действующих идентификаторов пользователя и группы *)
function setuid(uid:longint):longint;cdecl;external 'c';
function setgid(gid:longint):longint;cdecl;external 'c';

(* узнать или изменить ограничения процесса *)
```

```
function ulimit(cmd:longint;args:array of const):longint;cdecl;external 'c';

(* аварийное завершение процесса *)
function abort:longint;cdecl;external 'c';

(* Получить скорость ввода *)
function cfgetispeed(var tdes:TermIOS):longint;cdecl;external 'c';

(* Получить скорость вывода *)
function cfgetospeed(var tdes:TermIOS):longint;cdecl;external 'c';

(* используется для посылки сигнала прерывания сеанса связи *)
Function tcsendbrk(ttyfd, duration:longint):longint;cdecl;external 'c';

(* функции для работы с памятью *)
function memset(buf:pointer; character:longint;
size:longint):pointer;cdecl;external 'c';
function memcpy(buf1:pointer; const buf2:pointer;
size:longint):pointer;cdecl;external 'c';
function memmove(buf1:pointer; const buf2:pointer;
size:longint):pointer;cdecl;external 'c';
function memcmp(const buf1, buf2:pointer; size:longint):longint;cdecl;external
'c';
function memchr(const buf:pointer; character:longint;
size:longint):pointer;cdecl;external 'c';


(* функции для работы со строками *)
function strpbrk(const s1, s2:pchar):pchar;cdecl;external 'c';
function strspn(const s1, s2:pchar):longint;cdecl;external 'c';
function strcspn(const s1, s2:pchar):longint;cdecl;external 'c';
function strtok(s1:pchar; const s2:pchar):pchar;cdecl;external 'c';

function strtol(const str:pchar; endptr:ppchar;
base:longint):longint;cdecl;external 'c';
function atoi(const str:pchar):longint;cdecl;external 'c';
function atol(const str:pchar):longint;cdecl;external 'c';
function strtod(const str:pchar; endptr:ppchar):double;cdecl;external 'c';
function atof(const str:pchar):double;cdecl;external 'c';



implementation

(* If WIFEXITED(STATUS), the low-order 8 bits of the status.  *)
function WEXITSTATUS(status:integer):integer;
begin
  WEXITSTATUS:=(status and $ff00) shr 8;
end;

(* If WIFSTOPPED(STATUS), the signal that stopped the child.  *)
function WSTOPSIG(status:integer):integer;
begin
  WSTOPSIG:=WEXITSTATUS(status);
end;

(* If WIFSIGNALED(STATUS), the terminating signal.  *)
function WTERMSIG(status:integer):integer;
```

```
begin
  WTERMSIG:=status and $7f;
end;

(* Nonzero if STATUS indicates the child dumped core.  *)
function WCOREDUMP(status:integer):boolean;
begin
  WCOREDUMP:=(status and WCOREFLAG) = 0;
end;


(* Nonzero if STATUS indicates normal termination.  *)
function WIFEXITED(status:integer):boolean;
begin
  WIFEXITED:=(status and $7f) = 0;
end;

(* If WIFSIGNALED(STATUS), the terminating signal.  *)
function WIFSIGNALED(status:integer):boolean;
begin
  WIFSIGNALED:=(not WIFSTOPPED(status)) and (not WIFEXITED(status));
end;

(* Nonzero if STATUS indicates the child is stopped.  *)
function WIFSTOPPED(status:integer):boolean;
begin
  WIFSTOPPED:=(status and $ff) = $7f;
end;

(* test on space character: space, CR, LF, Tab, VTab, FF *)
function isspace(c:integer):boolean;
begin
  isspace:=c in [$9,$20,$a,$b,$c,$d];
end;


begin
  stdin:=fdopen(0,'r');
  stdout:=fdopen(1,'w');
end.
```

Приложение 4. Замечания о компиляции во Free Pascal 2.0

Когда работа над книгой подходила к концу, была выпущена новая версия компилятора Free Pascal, в котором де-юре зафиксирована возможность использования интерфейса системных вызовов не только в ОС семейства Linux/BSD, но и других клонов UNIX. Для того, чтобы скомпилировать компилятором версии 2.0 программы, представленные в книге, следует заменить в них подключаемый модуль `linux` на `oldlinux`. Платформенно-независимая функциональность модуля `linux` разделена между `unix`, `unixtype`, `baseunix` и `unixutil`. В модуль `X86` вынесены части, специфичные для X86-архитектуры.

Для удобства перехода на новую версию компилятора приводим таблицу соответствия модулей в разных версиях компилятора.

а) константы:

Название (1.x)	Описание	Название (2.x)	Модуль
	Maximum number of arguments to a program.	`ARG_MAX`	`unix, baseunix, unixtype`
	Number of bits in a word.	`BITSINWORD`	`baseunix`
`CLONE_FILES`	Clone option: open files shared between processes	`CLONE_FILES`	`linux`
`CLONE_FS`	Clone option: fs info shared between processes	`CLONE_FS`	`linux`
`CLONE_PID`	Clone option: PID shared between processes	`CLONE_PID`	`linux`
`CLONE_SIGHAND`	Clone option: signal handlers shared between processes	`CLONE_SIGHAND`	`linux`
`CLONE_VM`	Clone option: VM shared between processes	`CLONE_VM`	`linux`
`CSIGNAL`	Clone option: Signal mask to be sent at exit	`CSIGNAL`	`linux`
	System error: Argument list too long	`ESysE2BIG`	`baseunix`
	System error: Permission denied	`ESysEACCES`	`baseunix`
	System error: Address already in use	`ESysEADDRINUSE`	`baseunix`
	System error: Cannot assign requested address	`ESysEADDRNOTAVAIL`	`baseunix`
	System error: Advertise error	`ESysEADV`	`baseunix`
	System error: Address family not supported by protocol	`ESysEAFNOSUPPORT`	`baseunix`
	System error: Try again	`ESysEAGAIN`	`baseunix`
	System error: Operation already in progress	`ESysEALREADY`	`baseunix`
	System error: Invalid exchange	`ESysEBADE`	`baseunix`
	System error: Bad file number	`ESysEBADF`	`baseunix`
	System error: File descriptor in bad state	`ESysEBADFD`	`baseunix`
	System error: Not a data message	`ESysEBADMSG`	`baseunix`
	System error: Invalid request	`ESysEBADR`	`baseunix`

Название (1.x)	Описание	Название (2.x)	Модуль
	descriptor		
	System error: Invalid request code	ESysEBADRQC	baseunix
	System error: Invalid slot	ESysEBADSLT	baseunix
	System error: Bad font file format	ESysEBFONT	baseunix
	System error: Device or resource busy	ESysEBUSY	baseunix
	System error: No child processes	ESysECHILD	baseunix
	System error: Channel number out of range	ESysECHRNG	baseunix
	System error: Communication error on send	ESysECOMM	baseunix
	System error: Software caused connection abort	ESysECONNABORTED	baseunix
	System error: Connection refused	ESysECONNREFUSED	baseunix
	System error: Connection reset by peer	ESysECONNRESET	baseunix
	System error: Resource deadlock would occur	ESysEDEADLK	baseunix
	System error: File locking deadlock error	ESysEDEADLOCK	baseunix
	System error: Destination address required	ESysEDESTADDRREQ	baseunix
	System error: Math argument out of domain of func	ESysEDOM	baseunix
	System error: RFS specific error	ESysEDOTDOT	baseunix
	System error: Quota exceeded	ESysEDQUOT	baseunix
	System error: File exists	ESysEEXIST	baseunix
	System error: Bad address	ESysEFAULT	baseunix
	System error: File too large	ESysEFBIG	baseunix
	System error: Host is down	ESysEHOSTDOWN	baseunix
	System error: No route to host	ESysEHOSTUNREACH	baseunix
	System error: Identifier removed	ESysEIDRM	baseunix
	System error: Illegal byte sequence	ESysEILSEQ	baseunix
	System error: Operation now in progress	ESysEINPROGRESS	baseunix
	System error: Interrupted system call	ESysEINTR	baseunix
	System error: Invalid argument	ESysEINVAL	baseunix
	System error: I/O error	ESysEIO	baseunix
	System error: Transport endpoint is already connected	ESysEISCONN	baseunix
	System error: Is a directory	ESysEISDIR	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	System error: Is a named type file	ESysEISNAM	baseunix
	System error: Level 2 halted	ESysEL2HLT	baseunix
	System error: Level 2 not synchronized	ESysEL2NSYNC	baseunix
	System error: Level 3 halted	ESysEL3HLT	baseunix
	System error: Level 3 reset	ESysEL3RST	baseunix
	System error: Can not access a needed shared library	ESysELIBACC	baseunix
	System error: Accessing a corrupted shared library	ESysELIBBAD	baseunix
	System error: Cannot exec a shared library directly	ESysELIBEXEC	baseunix
	System error: Attempting to link in too many shared libraries	ESysELIBMAX	baseunix
	System error: .lib section in a.out corrupted	ESysELIBSCN	baseunix
	System error: Link number out of range	ESysELNRNG	baseunix
	System error: Too many symbolic links encountered	ESysELOOP	baseunix
	System error: Too many open files	ESysEMFILE	baseunix
	System error: Too many links	ESysEMLINK	baseunix
	System error: Message too long	ESysEMSGSIZE	baseunix
	System error: Multihop attempted	ESysEMULTIHOP	baseunix
	System error: File name too long	ESysENAMETOOLONG	baseunix
	System error: No XENIX semaphores available	ESysENAVAIL	baseunix
	System error: Network is down	ESysENETDOWN	baseunix
	System error: Network dropped connection because of reset	ESysENETRESET	baseunix
	System error: Network is unreachable	ESysENETUNREACH	baseunix
	System error: File table overflow	ESysENFILE	baseunix
	System error: No anode	ESysENOANO	baseunix
	System error: No buffer space available	ESysENOBUFS	baseunix
	System error: No CSI structure available	ESysENOCSI	baseunix
	System error: No data available	ESysENODATA	baseunix
	System error: No such device	ESysENODEV	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	System error: No such file or directory	ESysENOENT	baseunix
	System error: Exec format error	ESysENOEXEC	baseunix
	System error: No record locks available	ESysENOLCK	baseunix
	System error: Link has been severed	ESysENOLINK	baseunix
	System error: Out of memory	ESysENOMEM	baseunix
	System error: No message of desired type	ESysENOMSG	baseunix
	System error: Machine is not on the network	ESysENONET	baseunix
	System error: Package not installed	ESysENOPKG	baseunix
	System error: Protocol not available	ESysENOPROTOOPT	baseunix
	System error: No space left on device	ESysENOSPC	baseunix
	System error: Out of streams resources	ESysENOSR	baseunix
	System error: Device not a stream	ESysENOSTR	baseunix
	System error: Function not implemented	ESysENOSYS	baseunix
	System error: Block device required	ESysENOTBLK	baseunix
	System error: Transport endpoint is not connected	ESysENOTCONN	baseunix
	System error: Not a directory	ESysENOTDIR	baseunix
	System error: Directory not empty	ESysENOTEMPTY	baseunix
	System error: Not a XENIX named type file	ESysENOTNAM	
	System error: Socket operation on non-socket	ESysENOTSOCK	baseunix
	System error: Not a typewriter	ESysENOTTY	baseunix
	System error: Name not unique on network	ESysENOTUNIQ	baseunix
	System error: No such device or address	ESysENXIO	baseunix
	System error: Operation not supported on transport endpoint		ESysEOPNOTSUPP
	System error: Value too large for defined data type	ESysEOVERFLOW	baseunix
	System error: Operation not permitted.	ESysEPERM	baseunix
	System error: Protocol family	ESysEPFNOSUPPORT	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	not supported		
	System error: Broken pipe	ESysEPIPE	baseunix
	System error: Protocol error	ESysEPROTO	baseunix
	System error: Protocol not supported	ESysEPROTONOSUPPORT	baseunix
	System error: Protocol wrong type for socket	ESysEPROTOTYPE	baseunix
	System error: Math result not representable	ESysERANGE	baseunix
	System error: Remote address changed	ESysEREMCHG	baseunix
	System error: Object is remote	ESysEREMOTE	baseunix
	System error: Remote I/O error	ESysEREMOTEIO	baseunix
	System error: Interrupted system call should be restarted	ESysERESTART	baseunix
	System error: Read-only file system	ESysEROFS	baseunix
	System error: Cannot send after transport endpoint shutdown	ESysESHUTDOWN	baseunix
	System error: Socket type not supported	ESysESOCKTNOSUPPORT	baseunix
	System error: Illegal seek	ESysESPIPE	baseunix
	System error: No such process	ESysESRCH	baseunix
	System error: Srmount error	ESysESRMNT	baseunix
	System error: Stale NFS file handle	ESysESTALE	baseunix
	System error: Streams pipe error	ESysESTRPIPE	baseunix
	System error: Timer expired	ESysETIME	baseunix
	System error: Connection timed out	ESysETIMEDOUT	baseunix
	System error: Too many references: cannot splice	ESysETOOMANYREFS	baseunix
	System error: Text (code segment) file busy	ESysETXTBSY	baseunix
	System error: Structure needs cleaning	ESysEUCLEAN	baseunix
	System error: Protocol driver not attached	ESysEUNATCH	baseunix
	System error: Too many users	ESysEUSERS	baseunix
	System error: Operation would block	ESysEWOULDBLOCK	baseunix
	System error: Cross-device link	ESysEXDEV	baseunix
	System error: Exchange full	ESysEXFULL	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	Maximum elements in a TFDSet array.	FD_MAXFDSET	baseunix
fs_ext	File system type (FSStat): (ext) Extended	fs_ext	unix
fs_ext2	File system type (FSStat): (ext2) Second extended	fs_ext2	unix
fs_iso	File system type (FSStat): ISO 9660	fs_iso	unix
fs_minix	File system type (FSStat): Minix	fs_minix	unix
fs_minix_30	File system type (FSStat): Minix 3.0	fs_minix_30	unix
fs_minux_V2	File system type (FSStat): Minix V2	fs_minux_V2	unix
fs_msdos	File system type (FSStat): MSDOS (FAT)	fs_msdos	unix
fs_nfs	File system type (FSStat): NFS	fs_nfs	unix
fs_old_ext2	File system type (FSStat): (ext2) Old second extended	fs_old_ext2	unix
fs_proc	File system type (FSStat): PROC fs	fs_proc	unix
fs_xia	File system type (FSStat): XIA	fs_xia	unix
F_GetFd	fpFCntl command: Get close-on-exec flag	F_GetFd	baseunix
F_GetFl	fpFCntl command: Get filedescriptor flags	F_GetFl	baseunix
F_GetLk	fpFCntl command: Get lock	F_GetLk	baseunix
F_GetOwn	fpFCntl command: get owner of filedescriptor events	F_GetOwn	baseunix
F_OK	fpAccess call test: file exists.	F_OK	baseunix
F_SetFd	fpFCntl command: Set close-on-exec flag	F_SetFd	baseunix
F_SetFl	fpFCntl command: Set filedescriptor flags	F_SetFl	baseunix
F_SetLk	fpFCntl command: Set lock	F_SetLk	baseunix
F_SetLkW	fpFCntl command: Test lock	F_SetLkW	baseunix
F_SetOwn	FCntl command: Set owner of filedescriptor events F_SetOwn	baseunix	
IOCtl_TCGETS	IOCTL call number: get Terminal Control settings	IOCtl_TCGETS	unix
	Last bit in word.	ln2bitmask	baseunix
	Power of 2 number of bits in word.	ln2bitsinword	baseunix
LOCK_EX	FLock Exclusive lock	LOCK_EX	unix
LOCK_NB	FLock Non-blocking operation	LOCK_NB	unix
LOCK_SH	FLock Shared lock	LOCK_SH	unix

Название (1.x)	Описание	Название (2.x)	Модуль
LOCK_UN	FLock unlock	LOCK_UN	unix
MAP_ANONYMOUS	fpMMap map type: Don't use a file	MAP_ANONYMOUS	baseunix
MAP_DENYWRITE	MMap option: Ignored.	MAP_DENYWRITE	unix
MAP_EXECUTABLE	MMap option: Ignored.	MAP_EXECUTABLE	unix
MAP_FIXED	MMap map type: Interpret addr exactly	MAP_FIXED	unix
MAP_GROWSDOWN	MMap option: Memory grows downward (like a stack)	MAP_GROWSDOWN	unix
MAP_LOCKED	MMap option: lock the pages in memory.	MAP_LOCKED	unix
MAP_NORESERVE	MMap option: Do not reserve swap pages for this memory.	MAP_NORESERVE	unix
MAP_PRIVATE	MMap map type: Changes are private	MAP_PRIVATE	unix, baseunix
MAP_SHARED	MMap map type: Share changes	MAP_SHARED	unix
MAP_TYPE	MMap map type: Bitmask for type of mapping	MAP_TYPE	unix
MINSIGSTKSZ	NCC: Number of control characters in termio record.		
	Maximum filename length.	NAME_MAX	unix, unixtype, baseunix
NCCS	Number of control characters in termios record.		
	fpOpen file open mode: Append to file	O_APPEND	baseunix
	fpOpen file open mode: Create if file does not yet exist.	O_CREAT	baseunix
	fpOpen file open mode: Minimize caching effects	O_DIRECT	baseunix
	fpOpen file open mode: File must be directory.	O_DIRECTORY	baseunix
	fpOpen file open mode: Open exclusively	O_EXCL	baseunix
	fpOpen file open mode: Open for 64-bit I/O	O_LARGEFILE	baseunix
	fpOpen file open mode: Alias for O_NonBlock	O_NDELAY	baseunix
	fpOpen file open mode: No TTY control.	O_NOCTTY	baseunix
	fpOpen file open mode: Fail if file is symbolic link.	O_NOFOLLOW	baseunix
	fpOpen file open mode: Open in non-blocking mode	O_NONBLOCK	baseunix
	fpOpen file open mode: Read only	O_RDONLY	baseunix
	fpOpen file open mode:	O_RDWR	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	Read/Write		
	fpOpen file open mode: Write to disc at once	O_SYNC	baseunix
	fpOpen file open mode: Truncate file to length 0	O_TRUNC	baseunix
	fpOpen file open mode: Write only	O_WRONLY	baseunix
Open_Accmode	Bitmask to determine access mode in open flags.	Open_Accmode	unix
Open_Append	File open mode: Append to file	Open_Append	unix
Open_Creat	File open mode: Create if file does not yet exist.	Open_Creat	unix
Open_Direct	File open mode: Minimize caching effects	Open_Direct	unix
Open_Directory	File open mode: File must be directory.	Open_Directory	unix
Open_Excl	File open mode: Open exclusively	Open_Excl	unix
Open_LargeFile	File open mode: Open for 64-bit I/O	Open_LargeFile	unix
Open_NDelay	File open mode: Alias for Open_NonBlock	Open_NDelay	unix
Open_NoCtty	File open mode: No TTY control.	Open_NoCtty	unix
Open_NoFollow	File open mode: Fail if file is symbolic link.	Open_NoFollow	unix
Open_NonBlock	File open mode: Open in non-blocking mode	Open_NonBlock	unix
Open_RdOnly	File open mode: Read only	Open_RdOnly	unix
Open_RdWr	File open mode: Read/Write	Open_RdWr	unix
Open_Sync	File open mode: Write to disc at once	Open_Sync	unix
Open_Trunc	File open mode: Truncate file to length 0	Open_Trunc	unix
Open_WrOnly	File open mode: Write only	Open_WrOnly	unix
	Maximum pathname length.	PATH_MAX	unix, unixtype, baseunix
Prio_PGrp	Get/set process group priority	Prio_PGrp	unixtype
Prio_Process	Get/Set process priority	Prio_Process	unixtype
Prio_User	Get/set user priority	Prio_User	unixtype
PROT_EXEC	FpMMap memory access: page can be executed	PROT_EXEC	unix
PROT_NONE	FpMMap memory access: page can not be accessed	PROT_NONE	unix
PROT_READ	FpMMap memory access: page can be read	PROT_READ	unix
PROT_WRITE	FpMMap memory access: page can be written	PROT_WRITE	unix

Название (1.x)	Описание	Название (2.x)	Модуль
P_IN	Input file descriptor of pipe pair.	P_IN	unix
P_OUT	Output file descriptor of pipe pair.	P_OUT	unix
R_OK	fpAccess call test: read allowed	R_OK	baseunix
SA_INTERRUPT	Sigaction options: ?	SA_INTERRUPT	baseunix
SA_NOCLDSTOP	Sigaction options: Do not receive notification when child processes stop	SA_NOCLDSTOP	baseunix
	Sigaction options: ?	SA_NOCLDWAIT	baseunix
SA_NOMASK	Sigaction options: Do not prevent the signal from being received when it is handled.	SA_NOMASK	baseunix
SA_ONESHOT	Sigaction options: Restore the signal action to the default state.	SA_ONESHOT	baseunix
SA_ONSTACK	Socket option		
SA_RESTART	Sigaction options: Provide behaviour compatible with BSD signal semantics	SA_RESTART	baseunix
SA_SHIRQ	Sigaction options: ?	SA_SHIRQ	baseunix
	Sigaction options: The signal handler takes 3 arguments, not one.	SA_SIGINFO	baseunix
SA_STACK	Sigaction options: Call the signal handler on an alternate signal stack.	SA_STACK	baseunix
Seek_Cur	fpLSeek option: Set position relative to current position.	Seek_Cur	baseunix
Seek_End	fpLSeek option: Set position relative to end of file.	Seek_End	baseunix
Seek_Set	fpLSeek option: Set absolute position.	Seek_Set	baseunix
SIGABRT	Signal: ABRT (Abort)	SIGABRT	baseunix
SIGALRM	Signal: ALRM (Alarm clock)	SIGALRM	baseunix
SIGBUS	Signal: BUS (bus error)	SIGBUS	baseunix
SIGCHLD	Signal: CHLD (child status changed)	SIGCHLD	baseunix
SIGCONT	Signal: CONT (Continue)	SIGCONT	baseunix
SIGFPE	Signal: FPE (Floating point error)	SIGFPE	baseunix
SIGHUP	Signal: HUP (Hangup)	SIGHUP	baseunix
SIGILL	Signal: ILL (Illegal instruction)	SIGILL	baseunix
SIGINT	Signal: INT (Interrupt)	SIGINT	baseunix
SIGIO	Signal: IO (I/O operation possible)	SIGIO	baseunix
SIGIOT	Signal: IOT (IOT trap)	SIGIOT	baseunix
SIGKILL	Signal: KILL (unblockable)	SIGKILL	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
SIGPIPE	Signal: PIPE (Broken pipe	SIGPIPE	baseunix
SIGPOLL	Signal: POLL (Pollable event)	SIGPOLL	baseunix
SIGPROF	Signal: PROF (Profiling alarm)	SIGPROF	baseunix
SIGPWR	Signal: PWR (power failure restart)	SIGPWR	baseunix
SIGQUIT	Signal: QUIT	SIGQUIT	baseunix
SIGSEGV	Signal: SEGV (Segmentation violation)	SIGSEGV	baseunix
SIGSTKFLT	Signal: STKFLT (Stack Fault)	SIGSTKFLT	baseunix
SIGSTKSZ	Signal Stack size error	SIGSTKSZ	baseunix
SIGSTOP	Signal: STOP (Stop, unblockable)	SIGSTOP	baseunix
SIGTerm	Signal: TERM (Terminate)	SIGTerm	baseunix
SIGTRAP	Signal: TRAP (Trace trap)	SIGTRAP	baseunix
SIGTSTP	Signal: TSTP (keyboard stop)	SIGTSTP	baseunix
SIGTTIN	Signal: TTIN (Terminal input, background)	SIGTTIN	baseunix
SIGTTOU	Signal: TTOU (Terminal output, background)	SIGTTOU	baseunix
SIGUNUSED	Signal: Unused	SIGUNUSED	baseunix
SIGURG	Signal: URG (Socket urgent condition)	SIGURG	baseunix
SIGUSR1	Signal: USR1 (User-defined signal 1)	SIGUSR1	baseunix
SIGUSR2	Signal: USR2 (User-defined signal 2)	SIGUSR2	baseunix
SIGVTALRM	Signal: VTALRM (Virtual alarm clock)	SIGVTALRM	baseunix
SIGWINCH	Signal: WINCH (Window/Terminal size change)	SIGWINCH	baseunix
SIGXCPU	Signal: XCPU (CPU limit exceeded)	SIGXCPU	baseunix
SIGXFSZ	Signal: XFSZ (File size limit exceeded)	SIGXFSZ	baseunix
SIG_BLOCK	Sigprocmask flags: Add signals to the set of blocked signals.	SIG_BLOCK	baseunix
SIG_DFL	Signal handler: Default signal handler	SIG_DFL	baseunix
SIG_ERR	Signal handler: error	SIG_ERR	baseunix
SIG_IGN	Signal handler: Ignore signal	SIG_IGN	baseunix
	Maximum system signal number.	SIG_MAXSIG	unix, unixtype, baseunix
SIG_SETMASK	Sigprocmask flags: Set of blocked signals is given.	SIG_SETMASK	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
SIG_UNBLOCK	Sigprocmask flags: Remove signals from the set set of blocked signals.	SIG_UNBLOCK	baseunix
SI_PAD_SIZE	Signal information record pad bytes size. Do not use.	SI_PAD_SIZE	baseunix
SS_DISABLE	Socket options		
SS_ONSTACK	Socket options		
STAT_IFBLK	File (stat record) mode: Block device	STAT_IFBLK	unix
STAT_IFCHR	File (stat record) mode: Character device	STAT_IFCHR	unix
STAT_IFDIR	File (stat record) mode: Directory	STAT_IFDIR	unix
STAT_IFIFO	File (stat record) mode: FIFO	STAT_IFIFO	unix
STAT_IFLNK	File (stat record) mode: Link	STAT_IFLNK	unix
STAT_IFMT	File (stat record) mode: File type bit mask	STAT_IFMT	unix
STAT_IFREG	File (stat record) mode: Regular file	STAT_IFREG	unix
STAT_IFSOCK	File (stat record) mode: Socket	STAT_IFSOCK	unix
STAT_IRGRP	File (stat record) mode: Group read permission	STAT_IRGRP	unix
STAT_IROTH	File (stat record) mode: Other read permission	STAT_IROTH	unix
STAT_IRUSR	File (stat record) mode: Owner read permission	STAT_IRUSR	unix
STAT_IRWXG	File (stat record) mode: Group permission bits mask	STAT_IRWXG	unix
STAT_IRWXO	File (stat record) mode: Other permission bits mask	STAT_IRWXO	unix
STAT_IRWXU	File (stat record) mode: Owner permission bits mask	STAT_IRWXU	unix
STAT_ISGID	File (stat record) mode: GID bit set	STAT_ISGID	unix
STAT_ISUID	File (stat record) mode: UID bit set	STAT_ISUID	unix
STAT_ISVTX	File (stat record) mode: Sticky bit set	STAT_ISVTX	unix
STAT_IWGRP	File (stat record) mode: Group write permission	STAT_IWGRP	unix
STAT_IWOTH	File (stat record) mode: Other write permission	STAT_IWOTH	unix
STAT_IWUSR	File (stat record) mode: Owner write permission	STAT_IWUSR	unix
STAT_IXGRP	File (stat record) mode: Others execute permission	STAT_IXGRP	unix
STAT_IXOTH	File (stat record) mode: Others execute permission	STAT_IXOTH	unix
STAT_IXUSR	File (stat record) mode:	STAT_IXUSR	unix

Название (1.x)	Описание	Название (2.x)	Модуль
	Others execute permission		
	Max system name length.	SYS_NMLN	unix, unixtype, baseunix
	File mode: Block device	S_IFBLK	baseunix
	File mode: Character device	S_IFCHR	baseunix
	File mode: Directory	S_IFDIR	baseunix
	File mode: FIFO	S_IFIFO	baseunix
	File mode: Link	S_IFLNK	baseunix
	File mode: File type bit mask	S_IFMT	baseunix
	File mode: Regular file	S_IFREG	baseunix
	File mode: Socket	S_IFSOCK	baseunix
	Mode flag: Read by group.	S_IRGRP	baseunix
	Mode flag: Read by others.	S_IROTH	baseunix
	Mode flag: Read by owner.	S_IRUSR	baseunix
	Mode flag: Write by group.	S_IWGRP	baseunix
	Mode flag: Write by others.	S_IWOTH	baseunix
	Mode flag: Write by owner.	S_IWUSR	baseunix
	Mode flag: Execute by group.	S_IXGRP	baseunix
	Mode flag: Execute by others.	S_IXOTH	baseunix
	Mode flag: Execute by owner.	S_IXUSR	baseunix
	Max length of utsname domain name.	UTSNAME_DOMAIN_LENGTH	baseunix
	Max length of utsname system name, release, version, machine.	UTSNAME_LENGTH	baseunix
	Max length of utsname node name.	UTSNAME_NODENAME_LENGTH	baseunix
Wait_Any	WaitPID: Wait on any process	Wait_Any	unix
Wait_Clone	WaitPID: Wait on clone processes only.	Wait_MyPGRP	unix
Wait_MyPGRP	WaitPID: Wait processes from current process group	Wait_MyPGRP	unix
Wait_NoHang	WaitPID: Do not wait	Wait_NoHang	unix
Wait_UnTraced	WaitPID: Also report stopped but untraced processes	Wait_UnTraced	unix
WNOHANG	Waitpid option: Do not wait for processes to terminate.	WNOHANG	baseunix
	Number of words in a TFDSet array	wordsinfdset	baseunix
	Number of words in a signal set.	wordsinsigset	baseunix
WUNTRACED	Waitpid option: Also report children wich were stopped but not yet reported	WUNTRACED	baseunix
W_OK	fpAccess call test: write allowed	W_OK	baseunix
X_OK	fpAccess call test: execute allowed	X_OK	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	Mutex options:	_PTHREAD_MUTEX_ADAPTIVE_NP	unixtype
	Mutex options:	_PTHREAD_MUTEX_DEFAULT	unixtype
	Mutex options:	_PTHREAD_MUTEX_ERRORCHECK	unixtype
	Mutex options: double lock returns an error code.	_PTHREAD_MUTEX_ERRORCHECK_NP	unixtype
	Mutex options: Fast mutex	_PTHREAD_MUTEX_FAST_NP	unixtype
	Mutex options:	_PTHREAD_MUTEX_NORMAL	unixtype
	Mutex options:	_PTHREAD_MUTEX_RECURSIVE	unixtype
	Mutex options: recursive mutex	_PTHREAD_MUTEX_RECURSIVE_NP	unixtype
	Mutex options: ?	_PTHREAD_MUTEX_TIMED_NP	unixtype
__WCLONE	Waitpid option: Wait for clone children only		

б) типы данных:

Название (1.x)	Описание	Название (2.x)	Модуль
	Block count type.	Blkcnt_t	baseunix
	Block size type.	Blksize_t	baseunix
	C type: 8-bit signed integer	cchar	unix, baseunix, unixtype
	Double precision real format.	cDouble	unix, baseunix, unixtype
	Floating-point real format	cFloat	unix, baseunix, unixtype
	C type: integer (natural size)	cInt	unix, baseunix, unixtype
	C type: 16 bits sized, signed integer.	cInt16	unix, baseunix, unixtype
	C type: 32 bits sized, signed integer.	cInt32	unix, baseunix, unixtype
	C type: 64 bits sized, signed integer.	cInt64	unix, baseunix, unixtype
	C type: 8 bits sized, signed integer.	cInt8	unix, baseunix, unixtype
	Long double precision real format (Extended)	clDouble	unix, baseunix, unixtype
	Clock ticks type	clock_t	unix, baseunix, unixtype
	C type: long signed integer (double sized)	cLong	unix, baseunix, unixtype
	C type: 64-bit (double long) signed integer.	clonglong	unix, baseunix, unixtype
	C type: short signed integer (half sized)	cshort	unix, baseunix, unixtype
	C type: 8-bit unsigned integer	cuchar	unix, baseunix, unixtype
	C type: unsigned integer (natural size)	cUInt	unix, baseunix, unixtype
	C type: 16 bits sized, unsigned integer.	cUInt16	unix, baseunix, unixtype
	C type: 32 bits sized, unsigned integer.	cUInt32	unix, baseunix, unixtype

Название (1.x)	Описание	Название (2.x)	Модуль
	C type: 64 bits sized, unsigned integer.	cUInt64	unix, baseunix, unixtype
	C type: 8 bits sized, unsigned integer.	cUInt8	unix, baseunix, unixtype
	C type: long unsigned integer (double sized)	cuLong	unix, baseunix, unixtype
	C type: 64-bit (double long) unsigned integer.	culonglong	unix, baseunix, unixtype
	Alias for cuint	cunsigned	unix, baseunix, unixtype
	C type: short unsigned integer (half sized)	cushort	unix, baseunix, unixtype
ComStr	Command-line string type.	ComStr	unixutil
dev_t	Device descriptor type	dev_t	unix, baseunix, unixtype
	Record used in fpOpenDir and fpReadDir calls	Dir	baseunix
dirent	Record used in the fpReadDir function to return files in a directory.	dirent	baseunix
DirStr	Filename directory part string type.	DirStr	unixutil
ExtStr	Filename extension part string type.	ExtStr	unixutil
fdSet	Array containing file descriptor bitmask for the Select call.		
	Lock description type for fpFCntl lock call.	FLock	baseunix
	Group ID type.	gid_t	unix, baseunix, unixtype
	64-bit inode type.	ino64_t	baseunix
	Inode type.	ino_t	unix, baseunix, unixtype
	Kernel device type	kDev_t	unixtype
	Inode mode type.	mode_t	unix, baseunix, unixtype
NameStr	Filename name part string type.	NameStr	unixutil
	Number of links type.	nlink_t	unix, baseunix, unixtype
	64-bit offset type.	off64_t	baseunix
	Offset type.	off_t	unix, baseunix
PathStr	Filename path part string type.	PathStr	unixutil
	pointer to TBlkCnt type.	PBlkCnt	baseunix
	Pointer to TBlkSize type.	PBlkSize	baseunix
	Alias for cchar	pcchar	unix, baseunix, unixtype
	Pointer to cdouble type.	pcDouble	unix, baseunix, unixtype
	Pointer to cfloat type.	pcFloat	unix, baseunix, unixtype
	Pointer to cInt type.	pcInt	unix, baseunix, unixtype

Название (1.x)	Описание	Название (2.x)	Модуль
	Pointer to cldouble type.	pclDouble	unix, baseunix, unixtype
	Pointer to TClock type.	pClock	unix, baseunix, unixtype, unixtype
	Pointer to cLong type.	pcLong	unix, baseunix, unixtype
	Pointer to cShort type.	pcshort	unix, baseunix, unixtype
	Alias for cuchar	pcuchar	unix, baseunix, unixtype
	Pointer to cUInt type.	pcUInt	unix, baseunix, unixtype
	Pointer to cuLong type.	pculong	unix, baseunix, unixtype
	Alias for cunsigned	pcunsigned	unix, baseunix, unixtype
	Pointer to cuShort type.	pcushort	unix, baseunix, unixtype
	Pointer to TDev type.	pDev	unix, baseunix, unixtype
PDir	Pointer to TDir record	PDir	baseunix
pdirent	Pointer to Dirent record.	pdirent	baseunix
pfdset	Pointer to FDSet array.	pfdset	baseunix
	Pointer to TFilDes type.	pFilDes	baseunix
pfpstate	Pointer to tfpstate record.	pfpstate	baseunix
	Pointer to TGid type.	pGid	unix, baseunix, unixtype
pglob	Pointer to TGlob record.		
	Pointer to TGrpArr array.	pGrpArr	baseunix
	Process ID type.	pid_t	unix, baseunix, unixtype
	Pointer to TIno type.	pIno	unix, baseunix, unixtype
	Pointer to TIno64 type.	pIno64	baseunix
	Pointer to TkDev type.	pkDev	unixtype
	Pointer to TMode type.	pMode	unix, baseunix, unixtype
	Pointer to TnLink type.	pnLink	unix, baseunix, unixtype
	Pointer to TOff type.	pOff	unix, baseunix, unixtype
	Pointer to TOff64 type.	pOff64	baseunix
	Pointer to TPid type.	pPid	unix, baseunix, unixtype
PSigActionRec	Pointer to SigActionRec record.	PSigActionRec	baseunix
PSigAltStack	Pointer to SigAltStack record		
	Pointer to TSigContext record	PSigContext	baseunix
PSigContextRec	Pointer to SigContextRec record		
	Pointer to TSigInfo record type.	psiginfo	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
PSignalHandler	Pointer to SignalHandler type.	PSignalHandler	baseunix
PSignalRestorer	Pointer to SignalRestorer type	PSignalRestorer	baseunix
PSigSet	Pointer to signal set.	PSigSet	baseunix
	Pointer to sigset_t type.	psigset_t	baseunix
	Pointer to TSize type.	pSize	unix, baseunix, unixtype
	Pointer to size_t type.	psize_t	unixtype
	Pointer to TSockLen type.	pSocklen	unix, baseunix, unixtype
	Pointer to TsSize type	psSize	unix, baseunix, unixtype
pstack_t	Pointer to stack_t record		
PStat	Pointer to Stat record.	PStat	baseunix
PStatFS	Pointer to StatFS record.	PStatFS	unixtype
PSysCallRegs	Pointer to SysCallRegs record.		
PSysInfo	Pointer to TSysInfo record.	PSysInfo	linux
	Thread attributes record. Opaque.	pthread_attr_t	unixtype
	Conditional variable attributes type (opaque).	pthread_condattr_t	unixtype
	Thread conditional variable type.	pthread_cond_t	unix, baseunix, unixtype
	Thread local storage key (opaque)	pthread_key_t	unixtype
	Mutex attributes type (opaque).	pthread_mutexattr_t	unixtype
	Thread mutex type.	pthread_mutex_t	unix, baseunix, unixtype
	Thread mutex type (opaque).	pthread_mutex_t	unixtype
	R/W lock attributes (opaque).	pthread_rwlockattr_t	unixtype
	Read/Write lock type (opaque)	pthread_rwlock_t	unixtype
	Posix thread type.	pthread_t	unix, baseunix, unixtype
	Pointer to TTime type.	pTime	unix, baseunix, unixtype
	Pointer to timespec type.	ptimespec	unix, baseunix, unixtype
ptimeval	Pointer to TTimeVal record	ptimeval	unix, baseunix, unixtype
ptimezone	Pointer to TimeZone record.	ptimezone	baseunix
	Pointer to time_t type.	ptime_t	unix, baseunix, unixtype
	Pointer to TTms type.	PTms	baseunix
	Pointer to TUid type.	pUid	unix, baseunix, unixtype
PUTimeBuf	Pointer to TUTimeBuf record	PUTimeBuf	baseunix
PUTSName	Pointer to TUTSName record.	PUTSName	baseunix
	Pointer to wchar_t type.	pwchar_t	unixtype
	Scheduling parameter description record.	sched_param	unixtype
	Semaphore type. (opaque)	sem_t	unixtype
	Callback prototype for a SigActionRec record.	SigActionHandler	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
SigActionRec	Record used in fpSigAction call.	SigActionRec	baseunix
SigAltStack	Alternate stack registers record		
SigContextRec	Record describing the context of the program when it receives a signal		
SignalHandler	Function prototype for the Signal call.	SignalHandler	baseunix
SignalRestorer	Signal restorer function prototype	SignalRestorer	baseunix
SigSet	Signal set type	SigSet	baseunix
	Signal set type	sigset_t	baseunix
Size_T	Size type	Size_T	unix, baseunix, unixtype
	Socket address length type.	socklen_t	unix, baseunix, unixtype
	Small size type.	ssize_t	unix, baseunix, unixtype
stack_t	Alias for SigAltStack type		
Stat	Record describing an inode (file) in the pffstat call.	Stat	baseunix
Statfs	Record describing a file system in the fsstat call.		
SysCallRegs	Register describing system calls.		
	Alias for Blkcnt_t type.	TBlkCnt	baseunix
	Alias for blksize_t type.	TBlkSize	baseunix
	Alias for clock_t type.	TClock	unix, baseunix, unixtype
TCloneFunc	Clone function prototype.	TCloneFunc	linux
	Alias for dev_t type.	TDev	unix, baseunix, unixtype
TDir	Record used in OpenDir and ReadDir calls	TDir	baseunix
TDirEnt	Alias for DirEnt record	TDirEnt	baseunix
Termio	Terminal I/O description record (small)		
Termios	Terminal I/O description record		
TFDSet	Alias for FDSet type.	TFDSet	baseunix
	Array of file descriptors as used in fpPipe call.	TFilDes	baseunix
tfpreg	Record describing floating point register in signal handler.	tfpreg	baseunix
tfpstate	Record describing floating point unit in signal handler.	tfpstate	baseunix
	Describes the search strategy used by FSearch	TFSearchOption	unix
	Alias for gid_t type.	TGid	unix, baseunix, unixtype
	Array of gid_t IDs	TGrpArr	baseunix
tglob	Record containing one entry in		

Название (1.x)	Описание	Название (2.x)	Модуль
	the result of Glob		
timespec	Time interval for the NanoSleep function.	timespec	unix, baseunix, unixtype
timeval	Record specifying a time inteval.	timeval	unix, baseunix, unixtype
timezone	Record describing a timezone	timezone	baseunix
	Time span type	time_t	unix, baseunix, unixtype
tmmapargs	Record containing mmap args.		
	Alias for ino_t type.	TIno	unix, baseunix, unixtype
	Alias for ino64_t type.	TIno64	baseunix
	Alias for kDev_t type.	TkDev	unixtype
	Alias for mode_t type.	TMode	unix, baseunix, unixtype
	Record containing timings for fpTimes call.	tms	baseunix
	Alias for nlink_t type.	TnLink	unix, baseunix, unixtype
	Alias for off_t type.	TOff	unix, baseunix, unixtype
	Alias for off64_t type.	TOff64	baseunix
	Alias for pid_t type.	TPid	unix, baseunix, unixtype
Tpipe	Array describing a pipe pair of filedescriptors.	Tpipe	unix
TSigAction	Function prototype for SigAction call.		
	Alias for SigActionRec record type.	TSigActionRec	baseunix
	Record describing the CPU context when a signal occurs.	TSigContext	baseunix
	Record describing the signal when a signal occurs.	tsiginfo	baseunix
	Alias for SigSet type.	TSigSet	baseunix
	Alias for size_t type	TSize	unix, baseunix, unixtype
	Alias for socklen_t type.	TSocklen	unix, baseunix, unixtype
	Alias for ssize_t type	TsSize	unix, baseunix, unixtype
TStat	Alias for Stat record.	TStat	baseunix
TStatFS	Alias for StatFS type.	TStatFS	unix, baseunix, unixtype
TSysCallRegs	Alias for SysCallRegs record		
TSysinfo	Record with system information, used by the SysInfo call.	TSysinfo	linux
TTermio	Alias for TermIO record		
TTermios	Alias for Termios record.		
	Alias for TTime type.	TTime	unix, baseunix, unixtype

Название (1.x)	Описание	Название (2.x)	Модуль
	Alias for TimeSpec type.	Ttimespec	unix, baseunix, unixtype
TTimeVal	Alias for TimeVal record.	TTimeVal	unix, baseunix, unixtype
TTimeZone	Alias for TimeZone record.	TTimeZone	baseunix
	Alias for Tms record type.	TTms	baseunix
	Alias for uid_t type.	TUid	unix, baseunix, unixtype
TUTimeBuf	Alias for UTimBuf record.	TUTimeBuf	baseunix
TUTSName	Alias for UTSName record.	TUTSName	baseunix
TWinSize	Alias for WinSize record.		
	User ID type	uid_t	unix, baseunix, unixtype
UTimBuf	Record used in Utime to set file access and modificaton times.	UTimBuf	baseunix
UTimeBuf	Alias for UTimBuf record.		
utsname	Record used to return kernel information in UName function.	utsname	baseunix
	Wide character type.	wchar_t	unixtype
	Wide character size type.	wint_t	unixtype
winsize	Record describing terminal window size.		
	Fast lock (mutex) type (opaque).	_pthread_fastlock	unixtype

в) процедуры и функции:

Название (1.x)	Описание	Название (2.x)	Модуль
Access	Check file access	fpAccess	baseunix
Alarm	Schedule an alarm signal to be delivered	fpAlarm	baseunix
	Concert an array of string to an array of null-terminated strings	ArrayStringToPPchar	unixutil
AssignPipe	Create a set of pipe file handlers	AssignPipe	unix
AssignStream	Assign stream for in and output to a program	AssignStream	unix
Basename	Return basename of a file	Basename	unixutil
CFMakeRaw	Sets flags in Termios record.		
CFSetISpeed	Set input baud rate in Termios record		
CFSetOSpeed	Set output baud rate in Termios record		
Chmod	Change file permission bits	fpChmod	baseunix
Chown	Change owner of file	fpChown	baseunix
Clone	Clone current process (create new thread)	Clone	linux
CloseDir	Close directory file descriptor	fpCloseDir	baseunix
CreateShellArgV	Create an array of null-terminated strings		
Dirname	Extract directory part from filename	Dirname	unixutil
Dup	Duplicate a file handle	fpDup	baseunix
Dup2	Duplicate one filehandle to another	fpDup2	baseunix
EpochToLocal	Convert epoch time to local time	EpochToLocal	unixutil
Execl	Execute process (using argument list)	FpExecL	unix
Execle	Execute process (using argument list, environment)	FpExecLE	unix
Execlp	Execute process (using argument list, environment; search path)	FpExecLP	unix

Название (1.x)	Описание	Название (2.x)	Модуль
Execv	Execute process	FpExecV	unix, baseunix
Execve	Execute process using environment	fpExecve	baseunix
Execvp	Execute process, search path	FpExecVP	unix
	Execute process, search path using environment	FpExecVPE	unix
ExitProcess	Exit the current process	fpExit	baseunix
Fcntl	File control operations.	fpFcntl	baseunix
fdClose	Close file descriptor	fpClose	baseunix
fdFlush	Flush kernel file buffer		
fdOpen	Open file and return file descriptor	fpOpen	baseunix
fdRead	Read data from file descriptor	fpRead	baseunix
fdSeek	Set file pointer position.	FpLseek	baseunix
fdTruncate	Truncate file on certain size.	FpFtruncate	baseunix
fdWrite	Write data to file descriptor	fpWrite	baseunix
FD_Clr	Clears a filedescriptor in a set	fpFD_Clr	baseunix
FD_IsSet	Check whether a filedescriptor is set	fpFD_IsSet	baseunix
FD_Set	Set a filedescriptor in a set	fpFD_Set	baseunix
FD_Zero	Clear all file descriptors in set	fpFD_Zero	baseunix
FExpand	Expand filename to fully qualified path		
Flock	Lock a file (advisory lock)	fpFlock	unix
FNMatch	Check whether filename matches wildcard specification	FNMatch	unixutil
Fork	Create child process	fpFork	baseunix
	Change current working directory.	FpChdir	baseunix
	Set all filedescriptors in the set.	fpfdfillset	baseunix
	Retrieve the current working directory.	FpGetcwd	baseunix
	Retrieve extended error information.	fpgeterrno	baseunix
	Get the list of supplementary groups.	FpGetgroups	baseunix
	Get process group ID	FpGetpgrp	baseunix
	Create a new directory	FpMkdir	baseunix
	Create a set of pipe file handlers	FpPipe	baseunix
	Rename file	FpRename	baseunix
	Remove a directory.	FpRmdir	baseunix
	Set extended error information.	fpseterrno	baseunix
	Set the current group ID	FpSetgid	baseunix
	Create a new session	FpSetsid	baseunix
	Set kernel time	fpsettimeofday	baseunix
	Set the current user ID	FpSetuid	baseunix
	Set a signal in a signal set.	FpSigAddSet	baseunix
	Remove a signal from a signal set.	FpSigDelSet	baseunix
	Clear all signals from signal set.	FpsigEmptySet	baseunix
	Set all signals in signal set.	FpSigFillSet	baseunix
	Check whether a signal appears in a signal set.	FpSigIsMember	baseunix
	Suspend process for several seconds	FpSleep	baseunix
	Retrieve file information about a file descriptor.	FpStat	baseunix
	Return execution times for the current process	FpTimes	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
	Wait for a child to exit.	FpWait	baseunix
FReName	Rename file		
FSearch	Search for file in search path.	FSearch	unix
FSplit	Split filename into path, name and extension	FSplit	unixutil
FSStat	Retrieve filesystem information.		
FStat	Retrieve information about a file	fpFStat	baseunix
	Retrieve filesystem information from a file descriptor.	fStatFS	unix
	Synchronize file's kernel data with disk.	fsync	unix
GetDate	Return the system date		
GetDateTime	Return system date and time		
GetDomainName	Return current domain name	GetDomainName	unix
GetEGid	Return effective group ID	fpGetEGid	baseunix
GetEnv	Return value of environment variable.	fpGetEnv	baseunix
GetEpochTime	Return the current unix time	fptime	baseunix
GetEUid	Return effective user ID	fpGetEUid	baseunix
GetFS	Return file selector	GetFS	unixutil
GetGid	Return real group ID	fpGetGid	baseunix
GetHostName	Return host name	GetHostName	unix
GetLocalTimezone	Return local timzeone information	GetLocalTimezone	unix
GetPid	Return current process ID	fpGetPid	baseunix
GetPPid	Return parent process ID	fpGetPPid	baseunix
GetPriority	Return process priority	fpGetPriority	baseunix
GetTime	Return current system time		
GetTimeOfDay	Return kernel time of day in GMT	fpGetTimeOfDay	unix
GetTimezoneFile	Return name of timezone information file	GetTimezoneFile	unix
GetUid	Return current user ID	fpGetUid	baseunix
Glob	Find filenames matching a wildcard pattern		
Globfree	Free result of Glob call		
	Converts a gregorian date to a julian date	GregorianToJulian	unixutil
	Converts a julian date to a gregorian date	JulianToGregorian	unixutil
IOCtl	General kernel IOCTL call.	fpIOCtl	baseunix
IOperm	Set permission on IO ports	fpIOperm	x86
IoPL	Set I/O privilege level	fpIoPL	x86
IsATTY	Check if filehandle is a TTY (terminal)		
Kill	Send a signal to a process	fpKill	baseunix
Link	Create a hard link to a file	fpLink	baseunix
LocalToEpoch	Convert local time to epoch (unix) time	LocalToEpoch	unixutil
Lstat	Return information about symbolic link. Do not follow the link	fpLstat	baseunix
mkFifo	Create FIFO (named pipe) in file system	fpmkFifo	baseunix
MMap	Create memory map of a file	fpMMap	baseunix
MUnMap	Unmap previously mapped memory block	fpMUnMap	baseunix
NanoSleep	Suspend process for a short time	fpNanoSleep	baseunix
Nice	Set process priority	fpNice	baseunix
Octal	Convert octal to decimal value		
OpenDir	Open directory for reading	fpOpenDir	baseunix
Pause	Wait for a signal	fpPause	baseunix
PClose	Close file opened with POpen	PClose	unix

Название (1.x)	Описание	Название (2.x)	Модуль
POpen	Pipe file to standard input/output of program	POpen	unix
ReadDir	Read entry from directory	fpReadDir	baseunix
ReadLink	Read destination of symbolic link	ReadLink	baseunix
	Read data from a PC port	ReadPort	x86
	Read bytes from a PC port	ReadPortB	x86
	Read longints from a PC port	ReadPortL	x86
	Read Words from a PC port	ReadPortW	x86
ReadTimezoneFile	Read the timezone file and initialize time routines	ReadTimezoneFile	unix
SeekDir	Seek to position in directory	SeekDir	unix
Select	Wait for events on file descriptors	fpSelect	baseunix
SelectText	Wait for event on typed ontyped file.	SelectText	unix
SetDate	Set the current system date.		
SetDateTime	Set the current system date and time		
SetPriority	Set process priority	fpSetPriority	baseunix
SetTime	Set the current system time.		
Shell	Execute and feed command to system shell	Shell	unix
SigAction	Install signal handler	fpSigAction	baseunix
Signal	Install signal handler (deprecated)	fpSignal	baseunix
SigPending	Return set of currently pending signals	fpSigPending	baseunix
SigProcMask	Set list of blocked signals	fpSigProcMask	baseunix
SigRaise	Raise a signal (send to current process)	SigRaise	unix
SigSuspend	Set signal mask and suspend process till signal is received	fpSigSuspend	baseunix
	Retrieve filesystem information from a path.	StatFS	unix
StringToPPChar	Split string in list of null-terminated strings	StringToPPChar	unixutil
SymLink	Create a symbolic link	fpSymLink	baseunix
SysCall	Execute system call.		
Sysinfo	Return kernel system information	Sysinfo	linux
	Execute and feed command to system shell	fpSystem	unix
S_ISBLK	Is file a block device	fpS_ISBLK	baseunix
S_ISCHR	Is file a character device	fpS_ISCHR	baseunix
S_ISDIR	Is file a directory	fpS_ISDIR	baseunix
S_ISFIFO	Is file a FIFO	fpS_ISFIFO	baseunix
S_ISLNK	Is file a symbolic link	fpS_ISLNK	baseunix
S_ISREG	Is file a regular file	fpS_ISREG	baseunix
S_ISSOCK	Is file a unix socket	fpS_ISSOCK	baseunix
TCDrain	Terminal control: Wait till all data was transmitted		
TCFlow	Terminal control: Suspend transmission of data		
TCFlush	Terminal control: Discard data buffer		
TCGetAttr	Terminal Control: Get terminal attributes		
TCGetPGrp	Terminal control: Get process group		
TCSendBreak	Terminal control: Send break		
TCSetAttr	Terminal control: Set attributes		
TCSetPGrp	Terminal control: Set process group		
TellDir	Return current location in a directory	TellDir	unix
TTYname	Terminal control: Get terminal name		
Umask	Set file creation mask.	fpUmask	baseunix

Название (1.x)	Описание	Название (2.x)	Модуль
Uname	Return system name.	fpUname	baseunix
UnLink	Unlink (i.e. remove) a file.	fpUnLink	baseunix
Utime	Set access and modification times of a file (touch).	Utime	baseunix
WaitPid	Wait for a process to terminate	fpWaitPid	baseunix
WaitProcess	Wait for process to terminate.	WaitProcess	unix
WEXITSTATUS	Extract the exit status from the WaitPID result.	WEXITSTATUS	baseunix
WIFEXITED	Check whether the process exited normally	WIFEXITED	baseunix
WIFSIGNALED	Check whether the process was exited by a signal.	WIFSIGNALED	baseunix
WIFSTOPPED	Check whether the process is currently stopped.	WIFSTOPPED	unix
	Write data to PC port	WritePort	x86
	Write byte to PC port	WritePortB	x86
	Write longint to PC port.	WritePortl	x86
	Write Word to PC port	WritePortW	x86
WSTOPSIG	Return the exit code from the process.	WSTOPSIG	baseunix
WTERMSIG	Return the signal that caused a process to exit.	WTERMSIG	baseunix
W_EXITCODE	Construct an exit status based on an return code and signal.	W_EXITCODE	unix
W_STOPCODE	Construct an exit status based on a signal.	W_STOPCODE	unix

г) переменные:

Название (1.x)	Описание	Название (2.x)	Модуль
ErrNo	Error number of last operation.		
LinuxError	Last operating system error		
tzdaylight	Indicates whether daylight savings time is active.	tzdaylight	unix
tzname	Timezone name.	tzname	unix
tzseconds	Seconds west of GMT	tzseconds	unixutil, unix

Литература

Бах Морис. Архитектура операционной системы UNIX. (http://lib.ru/BACH/)

Богатырев Андрей. Хрестоматия по программированию на Си в Unix. (http://lib.ru/CTOTOR/book.txt)

Голдт С., ван дер Меер С., Буркетт С., Уэлш М. Руководство программиста для Linux. (http:// www.opennet.ru/docs/RUS/Lpg)

Джонсон Майкл. Ядро ОС Linux. Руководство системного программиста. (www.linuxrsp.ru/docs/books/khg.html)

Керниган Б.В., Пайк Р. UNIX – универсальная среда программирования. – М.: Финансы и статистика, 1992. – 304 с.

Полищук А.П., Семериков С.А. Программирование в X Window средствами Free Pascal. (http://freepascal.ru/article//book/xwin/, http://pascal.sources.ru/graph/xwinfpc/)

Робачевский А.М. Операционная система UNIX. – Спб.: БХВ-Петербург, 2000. – 528 с.

Уолтон Шон. Создание сетевых приложений в среде Linux. Руководство разработчика. – М.: Вильямс, 2001. – 464 с.

Хантер Робин. Основные концепции компиляторов. – М.: Издательский дом «Вильямс», 2002. – 256 с.

Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста по разработке ПО. – М.: ДМК Пресс, 2000. – 368 с.

Эндрюс Грегори Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003. – 512 с.

Draft Standard for Information Technology – Portable Operating System Interface (POSIX) Part 2: Shell and Utilities (IEEE P1003.2 Draft 11.2 – September 1991)

A.D. Marshall. Programming in C. UNIX System Calls and Subroutines using C. (http://www.cm.cf.ac.uk/Dave/C/CE.html)

Mark Mitchell, Jeffrey Oldham, and Alex Samuel. Advanced Linux Programming. – New Riders Publishing, 2001. – 340 p.

Steve D. Pate. UNIX Internals: A Practical Approach. – Addison-Wesley, 1996. – 667 p.

Uresh Vahalia. UNIX Internals: The New Frontiers. – Prentice Hall, 1996. – 601 p.

Kurt Wall, Mark Watson, and Mark Whitis. Linux Programming Unleashed. – Sams, 1999. – 847 p.

`